The algebra of string comparison: Computing with sticky braids

Alexander Tiskin

Draft of December 12, 2022

Contents

Preface

This book is aimed at three overlapping, but distinct audiences:

- mathematicians, particularly those with an interest in abstract algebra, who may discover in this book a surprising application of a simple and elegant algebraic concept;
- computer scientists, particularly those with an interest in fundamental algorithms and data structures, who may find in this book a few ideas missing from their undergraduate textbooks;
- bioinformaticians, particularly those with an interest in sequence alignment algorithms, who may wish to skim this book pondering the opportunities it may give for faster and more flexible error-free sequence alignment.

This book was conceived in a conversation with Gad Landau in Haifa (Israel), continued at the University of Warwick in Coventry (United Kingdom), and completed at St. Petersburg State University in St. Petersburg (Russia). I am deeply grateful to Nikita Mishin, Danya Berezun, and several anonymous referees for their careful reading of different versions of this work, and for their comments that helped me to improve it. I thank Elżbieta Babij, Philip Bille, Paweł Gawrychowski, Tim Griffin, Dima Grigoriev, Gaetan Hains, Peter Krusche, Victor Levandovsky, Yuri Matiyasevich, Bill McColl, Sergei Nechaev, Prakash Panangaden, Dima Pasechnik, Mike Paterson, Luís Russo, Andrew Ryzhikov, Yoshifumi Sakai, Andrei Sobolevski, Nikolai Vassiliev, Nikolai Vavilov, Mikhail Vyalyi, Oren Weimann, and Michal Ziv-Ukelson for fruitful discussions. I thank Graham Cormode for his suggestion of ACM Books as the publisher for this work. The author's research was supported in part by the Centre for Discrete Mathematics and Its Applications (DIMAP), University of Warwick, and by the Royal Society Leverhulme Trust Senior Research Fellowship.

Chapter 1

Introduction

> 'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'
>
> L. Carroll, *Alice's Adventures in Wonderland*, 1865

> Is dynamic programming strictly necessary to solve sequence alignment problems?
>
> D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano, Efficient Algorithms for Sequence Analysis, in R. Capocelli et al., eds., *Sequences II: Methods in Communication, Security, and Computer Science*, 1993

A *monoid* is one of the most basic and ubiquitous structures in mathematics. It is defined by specifying a set of elements, and a binary operation on those elements, called *multiplication*. The monoid multiplication is required to be associative and to have an identity element. A prime example is the *free monoid*, where the elements are *strings* (sequences) of characters over an alphabet, string *concatenation* (writing one string after another) is the monoid multiplication, and the empty string is the monoid identity. Monoids are used as building blocks for richer algebraic structures, such as groups (where each element is also required to have an inverse), rings (where ring addition defines a group, and multiplication a monoid on the same element set), etc.

In this work, we reveal a striking connection between three seemingly unrelated monoids:

- a monoid-like structure obtained as a generalisation of the *longest common subsequence* (*LCS*) problem on a pair of strings, where the monoid's elements are matrices of LCS lengths among certain pairs of substrings, and the monoid multiplication corresponds to string concatenation;

- the monoid of a certain class of integer matrices that we call *unit-Monge matrices*, where the monoid multiplication is given by *distance*

matrix multiplication (also known as *tropical* or *min-plus multiplication*); we call it *the unit-Monge monoid*;

- the monoid of certain braid-like objects, where the monoid multiplication is given by so-called *Demazure multiplication*; this monoid is known as *Hecke monoid*, and its elements as *Hecke words*; however, in order to emphasize its connection with classical braids, we take the liberty to nickname it the *sticky braid monoid*, its elements *sticky braids*, and its operation *sticky multiplication*.

Surprisingly, these three monoids turn out to be not only related, but in fact *isomorphic* to one another. Even more surprisingly, it turns out that their common multiplication operation can be computed by a fast algorithm that is not directly implied by either of the three definitions. The monoids' structure and their fast multiplication algorithm have deep consequences for string comparison and approximate pattern matching, providing a divide-and-conquer alternative to the classical incremental dynamic programming approach that has dominated the field for many decades.

The new divide-and-conquer approach to string comparison problems turns out especially useful when the input strings may not be available all at once: for example, when the input strings change dynamically, or when they are compressed, or when the comparison has to be performed in parallel on a multi-processor machine. Our approach gives efficient solutions to a number of local string comparison problems, and, furthermore, can specialised to permutation strings, providing an efficient solution to local versions of the longest increasing subsequence (LIS) problem, and to the problem of computing a maximum clique in a circle graph.

This work is organised as follows. In Chapter 2, we give the necessary preliminaries. In Chapter 3, we introduce the main building blocks of our structures: permutation matrices and unit-Monge matrices. In **??**, we investigate the algebraic structure underlying the semi-local LCS problem. This is done in two alternative forms: as the monoid of unit-Monge matrices under distance multiplication, and as a formal monoid of sticky braids. In Chapter 5, we establish rigorously the relationship between this structure and the semi-local LCS problem, and use our structural results to obtain efficient algorithms for the semi-local LCS problem. In Chapter 6, we extend these structural and algorithmic results to the weighted alignment and edit distance problems. In Chapter 7, we show a few applications of the techniques developed so far. In Chapter 8, we apply our method to an important special case of comparing permutation strings, or, more generally, to string comparison parameterised by the number of matching characters. In Chapter 10, we develop efficient algorithms for comparing compressed strings. In Chapter 11, we extend our techniques beyond semi-local string comparison to comparing strings locally, towards the ultimate goal of flexible and efficient fully-local string comparison. We also discuss an implementation of our

method, which has found several applications in computational molecular biology. In Chapter 13, we extend our results to comparing strings in parallel, with different models of parallel computation: transposition networks, bit parallelism, vector paralellism, bulk-synchronous parallelism. We also use transposition networks to obtain similarity- and dissimilarity-sensitive string comparison algorithms.

Many results presented in this work appeared incrementally in the author's publications [200, 201, 204, 203, 198, 140, 205, 207, 208, 209]. The aim of this work is to consolidate these results, unifying the terminology and notation. However, a number of results are original to this work.

A string comparison teaser. Since the discussion of string algorithms does not begin until Chapter 5, we offer a small teaser to those primarily interested in this topic. Let a, b be strings, and let $lcs(a, b)$ denote the length of their LCS. Let σ, τ be characters, and consider the LCS between a and the extended strings σb, $b\tau$. Suppose that such an extension increases the LCS length in both cases: $lcs(a, \sigma b) = lcs(a, b\tau) = lcs(a, b) + 1$. We ask ourselves:

1. Does it necessarily follow that $lcs(a, \sigma b\tau) = lcs(a, b) + 2$?
2. What are the consequences of that?

The answer to the first question is trivial. This book is an answer to the second question.

Chapter 2

Preliminaries

2.1 Numbers and intervals

For matrix and vector indices, we will use either integers, or half-integers[1] (sometimes called odd half-integers):

$$\{\ldots,-2,-1,0,1,2,\ldots\} \qquad \left\{\ldots,-\tfrac{5}{2},-\tfrac{3}{2},-\tfrac{1}{2},\tfrac{1}{2},\tfrac{3}{2},\tfrac{5}{2},\ldots\right\}$$

For ease of reading, half-integer variables will be decorated by hats (e.g. $\hat{\imath}$, $\hat{\jmath}$). Undecorated variables (e.g. i, j, with possible subscripts or superscripts) will normally be integer, but can sometimes take values of both types. It will be convenient to denote

$$i^- = i - \tfrac{1}{2} \qquad i^+ = i + \tfrac{1}{2}$$

for any integer or half-integer i.

We will be using integer and half-integer intervals, identifying an interval of either type by its integer endpoints. We denote an integer (respectively, half-integer) interval by the colon notation:

$$[i\colon j] = \{i, i^{++}, \ldots, j^{--}, j\}$$
$$\langle i\colon j\rangle = \left\{i^+, i^{+++}, \ldots, j^{---}, j^-\right\}$$

where i, j are integer. For a finite interval $[i\colon j]$ or $\langle i\colon j\rangle$, we call the difference $j-i$ interval *length*. Note that $[i\colon j]$ and $\langle i\colon j\rangle$, despite having the same length, have a different number of points ($j-i+1$ and $j-i$, respectively).

To denote infinite intervals, we use $-\infty$ and $+\infty$ where appropriate. In particular, $[-\infty\colon+\infty]$ denotes the set of all integers, and $\langle-\infty\colon+\infty\rangle$ the

[1] The intuition behind using both integers and half-integers is that when comparing a pair of strings, we are dealing with a planar grid-like graph; our techniques also require dealing with its dual graph. In this setting, it is natural to use integers for indexing the nodes of the primal grid, and half-integers for indexing the nodes of the dual grid (i.e. the faces of the primal grid).

set of all half-integers. We denote singleton intervals by $[i] = [i\colon i]$ and $\langle \hat{\imath} \rangle = \langle \hat{\imath}^- \colon \hat{\imath}^+ \rangle$. Sometimes, we will abbreviate an endpoint pair with a single letter, e.g. $I = i\colon j$, in such a case, we will also denote $\sqcap I = i$ and $\sqcup I = j$, so e.g. $[I] = [\sqcap I \colon \sqcup I] = [i\colon j]$.

We denote the Cartesian product of two integer or two half-integer intervals by the semicolon notation:

$$[i\colon j; k\colon l] = [i\colon j] \times [k\colon l] \qquad \langle i\colon j; k\colon l \rangle = \langle i\colon j \rangle \times \langle k\colon l \rangle$$

We identify a singleton interval with its only element, extending this notation to Cartesian products. Thus, we have, for example

$$[i; j] = [i\colon i; j\colon j] \qquad \langle \hat{\imath}; j\colon k \rangle = \langle \hat{\imath}^- \colon \hat{\imath}^+; j\colon k \rangle$$

We will also occasionally be using real intervals. We denote a closed (respectively, open) real interval by $[\![x\colon y]\!]$, $(\!(x\colon y)\!)$, respectively, where $x, y \in \mathbb{R}$. We will use the semicolon notation for Cartesian products of real intervals, in the same way as for integer and half-integer ones. We write $\operatorname{cl} X$ to denote the topological closure of a set X.

2.2 Vectors and matrices

We will make extensive use of vectors and matrices with integer (occasionally, also rational or real) elements, and with integer or half-integer indices[2]. We treat a vector or a matrix as a function of one (respectively, two) integer or half-integer arguments. A nondecreasing function of an integer argument will be called *unit-nondecreasing*, if every successive pair of values differ by either 0 or 1. A vector or a matrix is *nonnegative,* if all its elements are nonnegative, and *unit-nondecreasing,* if it is unit-nondecreasing in each of its arguments, with the other argument fixed.

We indicate the index range of a matrix by juxtaposition, e.g. matrix $A[i\colon j; k\colon l]$. The same notation will be used for selecting subvectors and submatrices: for example, given matrix $A[0\colon n; 0\colon n]$, we denote by $A[i\colon j; k\colon l]$ the submatrix defined by the given sub-intervals. When any of the indices coincide with the range boundary, they can be omitted, e.g. $A[i\colon ; \colon j] = A[i\colon n; 0\colon j]$, and $A[\colon ; i\colon j] = A[0\colon n; i\colon j]$. In particular, we write $A[i; j]$ for a single matrix element, $A[i; \colon]$ for row i, and $A[\colon ; j]$ for column j. The indexing of the original vector or a matrix is preserved in any selected subvector or submatrix.

[2]When integers and half-integers are used as matrix indices, one can imagine that the matrices are written on squared paper. The entries of an integer-indexed matrix are at integer points of grid line intersections; the entries of a half-integer-indexed matrix are at half-integer points inside the square cells.

We denote by A^{T} the transpose of matrix A, and by A^{R} the matrix obtained from A by counterclockwise 90-degree rotation. Given matrix $A[0\colon n; 0\colon n]$, we have

$$A^{\mathsf{T}}[i;j] = A[j;i], \qquad A^{\mathsf{R}}[i;j] = A[j;n-i]$$

for all i, j. All the above notation applies to matrices with either integer or half-integer indices.

By default, vectors and matrices will be indexed by integers based at 0, or by half-integers based at $0^+ = \frac{1}{2}$. Where necessary, all our definitions and statements can easily be generalised to indexing over arbitrary integer or half-integer intervals.

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

2.3 Tropical multiplication

Tropical matrix multiplication The $(\min, +)$-semirings of integers and reals play a fundamental role in algorithm design. In these semirings, the ground set (e.g. integers, nonnegative integers, reals, nonnegative reals) is extended by $+\infty$, and the operators min and $+$, denoted by $\oplus$ and $\odot$, play the role of addition and multiplication, respectively. The $(\min, +)$-semiring is often called *tropical* (or *distance*) semiring, and its two operators are called *tropical addition* and *tropical multiplication*. For a detailed introduction into this and related topics, see e.g. Rote [182], Gondran and Minoux [105], Butković [48]. An application of tropical algebra to string comparison has been previously suggested by Comet [61].

Tropical addition and multiplication of numbers extends naturally to matrices and vectors of appropriate shapes. Our focus will be on tropical matrix multiplication.

Definition 2.1 *Let $A[I;J]$, $B[J;K]$, $C[I;K]$ be matrices.* Tropical matrix multiplication $A \odot B = C$ *is defined by*

$$C[i;k] = \bigoplus_{j\in[J]} \big(A[i;j] \odot B[j;k]\big) = \min_{j\in[J]} \big(A[i;j] + B[j;k]\big)$$

for all $i \in [I]$, $k \in [K]$. As usual, matrix product with $|[I]| = 1$ is a vector-matrix product, *and $|[K]| = 1$ a* matrix-vector product. □

Tropical matrix monoid Assuming a fixed index range, the set of all square matrices over $\mathbb{R}^+ = \mathbb{R}^{\geq 0} \cup \{+\infty\}$ forms a monoid under tropical

multiplication. We shall call it the *(multiplicative) tropical matrix monoid*. The identity matrix $\mathbb{I}$ and the zero matrix $\mathbb{O}$ in this monoid are given by

$$\mathbb{I}[i;j] = \begin{cases} 0 & \text{if } i = j \\ +\infty & \text{otherwise} \end{cases} \qquad \mathbb{O}[i;j] = +\infty$$

for all i, j. For any matrix A, we have

$$A \odot \mathbb{I} = \mathbb{I} \odot A = A \qquad A \odot \mathbb{O} = \mathbb{O} \odot A = \mathbb{O}$$

Since the ground matrix set is also closed under tropical matrix addition (taking elementwise minimum), the tropical matrix monoid is in fact the multiplicative monoid of the *tropical matrix semiring*.

Matrix-vector and matrix-matrix multiplication algorithms Consider tropical multiplication of an $n \times n$ matrix by an n-vector. For a generic matrix with no special structure, every matrix element may need to be queried in order to compute the product, therefore the only reasonable method for tropical matrix-vector multiplication of size n is by direct application of Definition 2.1 in time $O(n^2)$.

For a pair of generic matrices with no special structure, direct application of Definition 2.1 gives an algorithm for tropical matrix 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 [53], running in time $O\big(\frac{n^3(\log\log n)^3}{\log^2 n}\big)$.

2.4 Points and orders

Pairs of numbers will often be treated as geometric *points*. We define some natural partial orders on points:

- $\ll$*-dominance*: $(i;j) \ll (k;l)$, if $i \leq k$ and $j \leq l$;
- $\gtrcurlyeq$*-dominance*: $(i;j) \gtrcurlyeq (k;l)$, if $i \geq k$ and $j \leq l$;
- (total) *lexicographic order*: point $(i;j)$ precedes point $(k;l)$ lexicographically, if $i \leq k$, or else $i = k$ and $j \leq l$; this total order is compatible with the $\ll$-dominance order;
- *disjoint precedence order*: point $(i;j)$, $i \leq k$, disjointly precedes point $(k;l)$, $k \leq l$, if $j \leq k$.

When visualising points, we will deviate from the standard Cartesian convention on the direction of the coordinate axes. We will use instead the matrix indexing convention: the first coordinate in a pair increases downwards, and the second coordinate rightwards. Hence, $\ll$- and $\gtrcurlyeq$-dominance

correspond respectively to the "above-left" and "below-left" partial orders. The latter order is usually the default direction for dominance in computational geometry.

We use standard terminology for special elements and subsets in partial orders. In particular, a subset of elements form a *chain*, if they are pairwise comparable, and an *antichain*, if they pairwise incomparable. A subset of elements form an *upset*, if for any element of the subset, all its succeeding elements are also in the subset. An element in a partially ordered set is *minimal* (respectively, *maximal*), if, in terms of the partial order, it is not preceded by (respectively, does not precede) any other element in the set. All minimal (respectively, maximal) elements in a partially ordered set form an antichain.

2.5 Strings

We will consider *strings* (sequences) of *characters* over an *alphabet*. No a priori assumptions are made on the size of the alphabet and on the primitive character operations; we keep the freedom to make specific assumptions in different contexts (e.g. a fixed finite alphabet with only equality comparisons between characters, or alphabet $[1{:}\,n]$ with standard arithmetic operations on characters, etc.)

It will be convenient to index strings by half-integer, rather than integer indices. We will index strings as vectors, writing $a\langle\hat{\imath}\rangle$ for the character of string a at index $\hat{\imath}$, and $a\langle i{:}\,j\rangle$ to indicate the range of string a or to select a substring from it. Both endpoint indices of a string may be chosen freely, and can be either positive or negative. We call interval $\langle i{:}\,j\rangle$ the *supporting interval* of a (sub)string $a\langle i{:}\,j\rangle$.

Given strings $a\langle i{:}\,j\rangle$, $b\langle k{:}\,l\rangle$, their concatenation will be denoted by juxtaposition ab. Unless the endpoints of ab are specified explicitly, supporting interval $\langle i{:}\,j+l-k\rangle$ will be assumed by default.

Given string a, we denote its reverse string by $\bar{a}$.

Character matching Normally, we will say that two alphabet characters α, β *match*, if $\alpha = \beta$, and *mismatch* otherwise. In addition to this, we introduce two special characters outside the default alphabet:

- the *guard character* '$', matching itself but no other characters;
- the *wildcard character* '?', matching itself and all other characters.

(We avoid matching '$' against '?', so we leave the outcome of such an operation undefined.)

Permutation strings A *permutation string* is a string where all the characters are distinct. It will be convenient to assume that a permutation string, in addition to being indexed by half-integers $\langle 0{:}\, n\rangle$, is also over the same half-integer alphabet $\langle 0{:}\, n\rangle$, so that it represents a permutation of this set. The *identity permutation string* is the string $\mathsf{id} = (0^+, 1^+, \ldots, n^-)$. Permutations will play two separate roles in this work: as combinatorial objects (in this role, they will be called just permutations) and as strings of all distinct characters subject to comparison (it is for this role that the above definitions are made).

Substrings and subsequences Given a string, we will 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 string a and a set I of indices, we denote by $a|_I$ a subsequence formed by the characters at the specified indices.

2.6 Range aggregation (TODO)

1D range aggregation Semigroup, operation cost *op*
Usually assumed $op = O(1)$
Will consider variable *op*
Preprocessing; queries
Offline
Window aggregation problem
Interval tree; canonical intervals
Fragment aggregation problem
??

2D range counting Range tree

Chapter 3

Monge matrices

3.1 Totally monotone and Monge matrices

In this section and the next, we introduce several fundamental concepts related to matrices.

Dominance-sum and cross-difference matrices. We define *dominance summation* on a half-integer-indexed matrix: given an integer point, we sum up all matrix elements $\gtrsim$-dominated by this point. We then form an integer-indexed matrix of such dominance sums.

Definition 3.1 *Let $D\langle I;J\rangle$ be a matrix. Its* dominance-sum matrix *(also called* distribution matrix*) $D^{\Sigma}[I;J]$ is defined by*

$$D^{\Sigma}[i;j] = \sum_{\langle i:;:j\rangle} D$$

for all $i \in [I]$, $j \in [J]$. □

Definition 3.1 is based on $\gtrsim$-dominance. Dominance summation in the other directions can be achieved by rotating and/or transposing the matrix:

$$D^{\mathsf{M}} = D^{\mathsf{R\Sigma RRR}} \qquad D^{\mathsf{Z}} = D^{\mathsf{RR\Sigma RR}} = D^{\mathsf{T\Sigma T}} \qquad D^{\mathsf{W}} = D^{\mathsf{RRR\Sigma R}}$$

for $\ll$-, inverse $\gtrsim$-, and inverse $\ll$-dominance, respectively. Given an integer point $[i;j]$, expressions $D^{\Sigma}[i;j]$, $D^{\mathsf{M}}[i;j]$, $D^{\mathsf{Z}}[i;j]$, $D^{\mathsf{W}}[i;j]$ sum up all matrix elements that lie respectively below-left, above-left, above-right, below-right of point $[i;j]$.

Dominance summation can be partially reversed by taking *cross-differences* about every half-integer point in an integer-indexed matrix.

Definition 3.2 *Let $A[I;J]$ be a matrix. Its* cross-difference matrix $A^{\square}\langle I;J\rangle$ *(also called* density matrix*) is defined by*

$$A^{\square}\langle \hat{\imath};\hat{\jmath}\rangle = A[\hat{\imath}^{+};\hat{\jmath}^{-}] - A[\hat{\imath}^{-};\hat{\jmath}^{-}] - A[\hat{\imath}^{+};\hat{\jmath}^{+}] + A[\hat{\imath}^{-};\hat{\jmath}^{+}]$$

for all $\hat{\imath} \in \langle I\rangle$, $\hat{\jmath} \in \langle J\rangle$. □

The operation of taking the cross-difference matrix commutes with matrix transposition: $A^{\square} = A^{\mathsf{T}\square\mathsf{T}}$

Example 3.3 We have

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^{\square} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \blacksquare$$

Note that Definitions 3.1 and 3.2 are applicable even if $\sqcap I = \sqcup I$ (so $|[I]| = 1$, $|\langle I \rangle| = 0$) or $\sqcap J = \sqcup J$ (so $|[J]| = 1$, $|\langle J \rangle| = 0$). In such a case, matrix D of Definition 3.1 is empty, and matrix D^{Σ} is a row (respectively, column) vector consisting of all zeros. Similarly, matrix A of Definition 3.2 is a row (respectively, column) vector, and matrix $A^{\square}$ is empty. Both definitions also extend naturally to matrices over an infinite index range, as long as the sum in Definition 3.1 is defined.

The operations of taking the dominance-sum and the cross-difference matrix are close to be mutually inverse. For any finite matrices D, A as above, and for all i, j, we have

$$D^{\Sigma\square} = D$$
$$A^{\square\Sigma}[i;j] = A[i;j] - A[\sqcup I;j] - A[i;\sqcap J] + A[\sqcup I;\sqcap J]$$

When matrix A is restricted to having all zeros on its bottom-left boundary (i.e. in the leftmost column and the bottom row), the two operations become truly mutually inverse.

Definition 3.4 *Matrix $A[I;J]$ will be called* simple*, if $A[\sqcup I;j] = A[i;\sqcap J] = 0$ for all i, j. Equivalently, A is simple if $A^{\square\Sigma} = A$.* □

More generally, we have the following canonical decomposition lemma, which can be used to represent a matrix implicitly by its cross-difference matrix and a pair of vectors.

Lemma 3.5 (Matrix canonical decomposition) *Let $A[I;J]$ be a matrix. Let*

$$D = A^{\square} = A^{\mathsf{T}\square\mathsf{T}}$$
$$b = A[\sqcup I;:] \quad c = A[:;\sqcap J] \qquad e = A[\sqcap I;:] \quad f = A[:;\sqcup J]$$

We have

$$A[i;j] = D^{\Sigma}[i;j] + b[j] + c[i] - b[\sqcap J] \tag{3.1}$$
$$= D^{\boxplus}[i;j] + e[j] + f[i] - e[\sqcup J] \tag{3.2}$$

for all $i, j \in [I;J]$. □

PROOF Straightforward by Definition 3.8. ∎

We introduce special terminology for matrices with cross-differences satisfying a given property.

Definition 3.6 *Let A be a matrix, and let Π be a matrix property. We say that A has property Σ-Π, if its cross-difference matrix $A^{\square}$ has property Π. We say that A has property* anti-Π*, if its negative $-A$ has property Π.* □

By Lemma 3.5, a simple matrix has property Σ-Π, if is is of the form $A = D^{\Sigma}$, where matrix D has property Π.

Totally monotone and Monge matrices. The following classes of matrices play an important role in optimisation theory (see Burkard et al. [46] and Burkard [47] for an extensive survey), and also arise in graph and string algorithms.

Definition 3.7 *Matrix A is* totally monotone*, if*

$$A[i;j] \geq A[i;l] \text{ implies } A[k;j] \geq A[k;l]$$

for all $i \leq k$, $j \leq l$. □

Definition 3.8 *Matrix A is a* Monge matrix*, if*

$$A[i;j] + A[k;l] \leq A[i;l] + A[k;j]$$

for all $i \leq k$, $j \leq l$. Equivalently, matrix A is a Monge matrix, if it is Σ-nonnegative (i.e. its cross-difference matrix $A^{\square}$ is nonnegative). □

We will sometimes allow elements of a Monge matrix to be equal to $+\infty$, extending Definitions 3.7 and 3.8 by treating $+\infty \geq +\infty$ and $+\infty - \infty \geq 0$ as true, and treating other inequalities and sums involving $+\infty$ in the natural way. Such matrices with infinite elements are sometimes called *partial Monge* or *staircase Monge* matrices (see e.g. [3, 126, 98, 99]).

It is easy to see that Monge matrices form a subclass of totally monotone matrices. The canonical decomposition lemma was introduced in the context of Monge matrices by Burdyuk and Trofimov [45] and Bein and Pathak [28] (see also [46, 47]).

Our particular focus will be on Monge matrices that are also simple (such as e.g. the 4×4 matrix in Example 3.3). A simple Monge matrix is of the form $A = D^{\Sigma}$, where D is nonnegative.

Σ-bistochastic matrices. A special subclass of nonnegative matrices is defined as follows.

Definition 3.9 *A* bistochastic *(respectively,* subbistochastic*)* matrix *is a matrix of nonnegative real elements, in which the sum of elements is exactly one (respectively, at most one) in every row and every column.* □

Typically, (sub)bistochastic matrices will be indexed by half-integers. Note that a bistochastic matrix has to be square, while a subbistochastic matrix need not be. Also note that any submatrix of a (sub)bistochastic matrix is subbistochastic.

Σ-bistochastic and Σ-subbistochastic matrices are both important subclasses of Monge matrices.

3.2 Searching in totally monotone matrices

It is well-known that for a totally monotone matrix, only a small subset of elements needs to be accessed in order to obtain all the row minima. Therefore, row minima searching can be performed efficiently, as long as the unused elements of the matrix do not need to be counted as the algorithm's input (e.g. the whole matrix is pre-stored in random-access memory). More generally, we consider an implicit totally monotone matrix without charging any time cost for its input, and charging a fixed time cost per element query. In this setting, a classical row minima searching algorithm was given by Aggarwal et al. [4] (see also [101, 121]). It is often nicknamed "SMAWK algorithm" by its authors' initials.

Theorem 3.10 ([4]) *Let $A[I;J]$ be an implicit totally monotone matrix with element query time q. The index of the leftmost minimum element in every row of A can be obtained in time $O(qn)$, where $n = \max(|[I]|, |[J]|)$.*□

Proof We give a sketch of the proof; for details, see [4, 101].

Without loss of generality, consider a square matrix $A[0\colon n; 0\colon n]$. Let $B\big[0\colon \frac{n}{2}; 0\colon n\big]$ be an implicit totally monotone matrix obtained by taking every other row of A. We first concentrate on finding the leftmost row minima of B. Clearly, at most $\frac{n}{2}+1$ columns of B contain such a leftmost row minimum, and are therefore "useful". The key idea of the algorithm is to identify and to eliminate $(n+1)-(\frac{n}{2}+1)=\frac{n}{2}$ of the "useless" columns in an efficient process, based on the total monotonicity property.

During the elimination process, we call a matrix element *marked*, if it is already known not to be a leftmost row minimum. We call a column *marked*, if either it is column 0, or it has at least one marked element. A marked column gets eliminated once all its elements become marked, and it stays marked after the elimination.

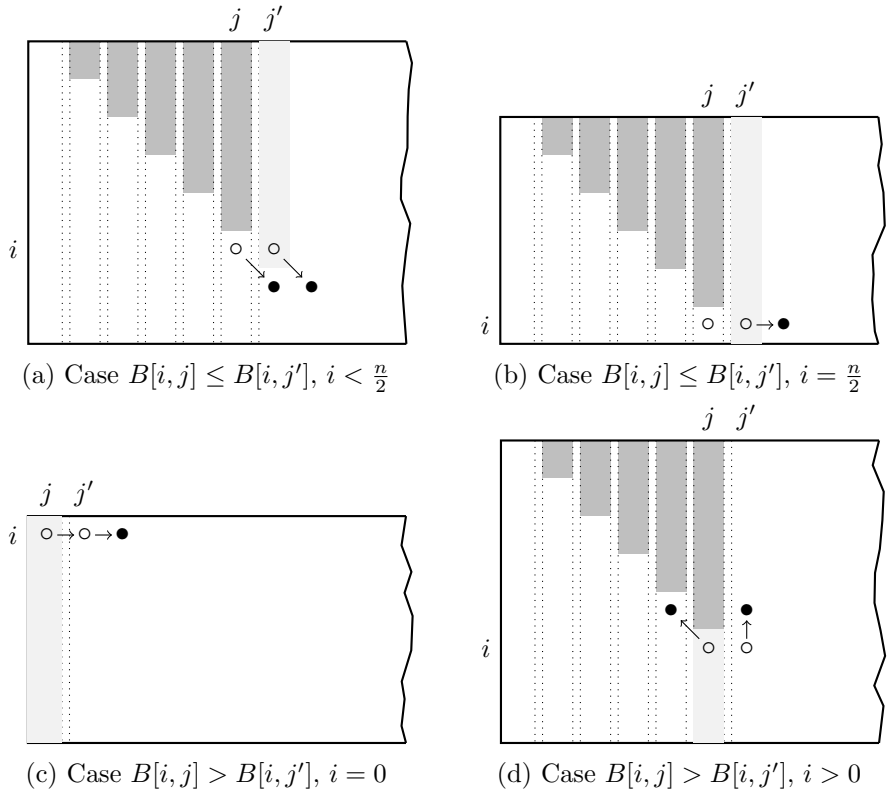
$i \leftarrow 0$; $j \leftarrow 0$; $j' \leftarrow 1$; push 0 onto stack
repeat until $\frac{n}{2}$ columns eliminated:
  case $B[i;j] \leq B[i;j']$:
    case $i < \frac{n}{2}$: push j' onto stack; $i \leftarrow i+1$; $j \leftarrow j'$
    case $i = \frac{n}{2}$: eliminate column j'
    $j' \leftarrow j'+1$
  case $B[i;j] > B[i;j']$:
    eliminate column j; pop j off stack
    case $i = 0$: push j' onto stack; $j' \leftarrow j'+1$
    case $i > 0$: $i \leftarrow i-1$
    $j \leftarrow$ top of stack

Table 3.1: Elimination procedure of Theorem 3.10.

During the elimination, the leftmost i uneliminated columns are marked, where the value of i is initially equal to 1. We maintain the original indexing of the columns, even as some of them become eliminated. The original indices of the marked columns are stored in a stack of height i, with the rightmost marked column at the top of the stack. Within the marked columns, the marked elements form a *staircase*: that is, in a marked column with index i' in the stack, $0 \leq i' < i$, the i' elements in rows from 0 to $i'-1$ are marked, and the rest are unmarked. In every iteration of the algorithm, either the marked column at the top of the stack get eliminated and popped from the stack, decrementing i, or the leftmost unmarked column gets marked and pushed onto the stack, incrementing i. The staircase of marked elements shrinks or expands by one column accordingly.

Let j denote the original index of the rightmost marked column (which is at the top of the stack), and j' the original index of the leftmost unmarked column. The outcome of the current iteration depends on the comparison of element $B[i;j]$, which is the unmarked element with the smallest row index in column j, against element $B[i;j']$, which is the next unmarked element directly to its right. The outcomes of this comparison and the rest of the elimination procedure are given in Table 3.1. A single iteration of this procedure can be implemented to run in time $O(q)$. The whole procedure runs in time $O(qn)$, and eliminates $\frac{n}{2}$ columns.

Let A' be the $(\frac{n}{2}+1) \times (\frac{n}{2}+1)$ matrix obtained from B by deleting the $\frac{n}{2}$ eliminated columns. We now call the algorithm recursively on A'. This recursive call returns the leftmost row minima of A', and therefore also of B. It is then straightforward to fill in the leftmost minima in the remaining rows of A in time $O(qn)$. Thus, the top level of recursion runs in time $O(qn)$. The amount of work gets halved with every recursion level, therefore the overall running time is $O(qn)$. ∎



(a) Case $B[i,j] \le B[i,j']$, $i < \frac{n}{2}$

(b) Case $B[i,j] \le B[i,j']$, $i = \frac{n}{2}$

(c) Case $B[i,j] > B[i,j']$, $i = 0$

(d) Case $B[i,j] > B[i,j']$, $i > 0$

Figure 3.1: Execution of the elimination procedure in Theorem 3.10

Example 3.11 Figure 3.1 gives a snapshot of the four subcases of the elimination algorithm described in the proof of Theorem 3.10. Only uneliminated columns are shown explicitly, delimited by dotted lines. Each inter-column gap represents an arbitrary number of consecutive eliminated columns. Dark-shaded cells represent the staircase of marked elements. The current elements $B[i;j]$, $B[i;j']$ are shown by hollow circles. The light-shaded cells represent the newly marked elements. The elements $B[i;j]$, $B[i;j']$ for the next iteration are shown by filled circles, unless they coincide with one of the hollow circles. Stepping between the iterations is shown by arrows. ■

3.3 Monge and Σ-bistochastic monoids

Monge matrix multiplication. In this section, we explore tropical multiplication of Monge matrices. It is well-known that the Monge property is preserved by tropical matrix multiplication (see e.g. [23]).

Theorem 3.12 *Let $A \odot B = C$. If A, B are Monge matrices, then C is also a Monge matrix.* □

PROOF Consider matrices A, B. Let $i' \leq i''$, $k' \leq k''$. By definition of tropical matrix multiplication, we have

$$C[i';k''] = \min_j \big(A[i';j] + B[j;k'']\big)$$
$$C[i'';k'] = \min_j \big(A[i'';j] + B[j;k']\big)$$

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

$$\begin{aligned}
&C[i';k'] + C[i'';k''] = && \text{(definition of } \odot\text{)}\\
&\min_j \big(A[i';j] + B[j;k']\big) + \min_j \big(A[i'';j] + B[j;k'']\big) \leq &&\\
& && \text{(minimisation over } j\text{)}\\
&\big(A[i';j'] + B[j';k']\big) + \big(A[i'';j''] + B[j'';k'']\big) = && \text{(term rearrangement)}\\
&\big(A[i';j'] + A[i'';j'']\big) + \big(B[j';k'] + B[j'';k'']\big) \leq && (A \text{ is Monge})\\
&\big(A[i';j''] + A[i'';j']\big) + \big(B[j';k'] + B[j'';k'']\big) = && \text{(term rearrangement)}\\
&\big(A[i';j''] + B[j'';k'']\big) + \big(A[i'';j'] + B[j';k']\big) = && \text{(definition of } j', j''\text{)}\\
&C[i';k''] + C[i'';k'] &&
\end{aligned}$$

The case $j' \geq j''$ is treated symmetrically, making use of the Monge property of B. Hence, matrix C is Monge. ∎

Theorem 3.12 implies that, assuming a fixed index range, the set of all square Monge matrices over $\mathbb{R}^+$ forms a monoid under tropical multiplication. We shall call it the *Monge monoid* (another appropriate name would be the *planar distance monoid*).

The Monge monoid forms a submonoid in the tropical matrix monoid. Matrices $\mathbb{I}$ and $\mathbb{O}$ are both Monge matrices by our convention on extending Definition 3.8 to matrices containing elements $+\infty$, and are therefore inherited as the identity and the zero by the Monge monoid.

In contrast with general matrices, the set of Monge matrices is not closed under tropical addition, hence the Monge monoid lacks the additional semiring structure of the tropical matrix monoid.

Matrix-vector multiplication algorithm. We now study algorithms for tropical multiplication of Monge matrices. For a Monge matrix, tropical matrix-vector multiplication can be performed as an application of row minima searching by Theorem 3.10, so that only a small subset of elements needs to be queried.

Theorem 3.13 *Let $A \odot b = c$, where $A[I;J]$ is an implicit Monge matrix with element query time q, and $b[J]$, $c[I]$ are vectors. Given vector b, vector c can be computed in time $O(qn)$, where $n = \max(|[I]|, |[J]|)$.* □

PROOF Let $\tilde{A}[i;j] = A[i;j] + b[j]$ for all i, 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 Theorem 3.10. ∎

In particular, for an explicit matrix A pre-stored in random-access memory, we have $q = O(1)$, therefore Monge tropical matrix-vector multiplication can be performed in time $O(n)$, without even touching most elements of the matrix.

Matrix multiplication algorithms. For Monge matrices, tropical multiplication can easily be performed in quadratic time (see also [23]). For simplicity, we restrict ourselves to square Monge matrices, although the results generalise to rectangular ones.

Theorem 3.14 *Let $A \odot B = C$, where $A[0:n;0:n]$ is a Monge matrix, and $B[0:n;0:n]$, $C[0:n;0:n]$ are arbitrary matrices. Given matrices A, B, matrix C can be computed in time $O(n^2)$.* □

PROOF The problem of computing the product $A \odot B = C$ is equivalent to n instances of the tropical 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 3.13, so the overall running time is $n \cdot O(n) = O(n^2)$.

In the special case where matrix B (and therefore also C) is Monge, an alternative algorithm with the same asymptotic running time can be obtained by the divide-and-conquer technique (see e.g. [**?**]). ∎

Σ-bistochastic matrix multiplication. Similarly to the Monge property, the property of being simple Σ-bistochastic is preserved under tropical matrix multiplication.

Theorem 3.15 *Let $A \odot B = C$. If matrices A, B are simple Σ-bistochastic (respectively, simple Σ-subbistochastic), then matrix C is also simple Σ-bistochastic (respectively, simple Σ-subbistochastic) matrix.* □

PROOF Consider simple regular Monge matrices $A[0:n;0:n]$, $B[0:n;0:n]$. We have $A = D^{\Sigma}$, $B = E^{\Sigma}$, where D, E are bistochastic matrices. It is easy to check that matrix C is simple, therefore $C = F^{\Sigma}$ for some matrix F.

We now have $D^{\Sigma} \odot E^{\Sigma} = F^{\Sigma}$, and we need to show that F is a bistochastic matrix. First, matrix C is Monge by Theorem 3.12, and therefore matrix $C^{\square} = F$ is nonnegative.

Since E is a bistochastic matrix, we have

$$E^{\Sigma}[j;0] = 0 \qquad E^{\Sigma}[j;n] = n - j$$

for all $j \in [0\colon n]$. Hence

$$C[i;0] = \min_j\big(D^{\Sigma}[i;j] + E^{\Sigma}[j;0]\big) = \min_j\big(D^{\Sigma}[i;j] + 0\big) = 0$$
$$C[i;n] = \min_j\big(D^{\Sigma}[i;j] + E^{\Sigma}[j;n]\big) = \min_j\big(D^{\Sigma}[i;j] + n - j\big) = n - i$$

for all $i \in [0\colon n]$, since the minimum is attained respectively at $j = 0$ and $j = n$. Therefore, we have

$$\textstyle\sum_{\hat{k}} F[\hat{\imath};\hat{k}] = \qquad \text{(definition of } \Sigma \text{ and } \square\text{)}$$
$$\textstyle\sum_{\hat{k}}\big(C[\hat{\imath}^+;\hat{k}^-] - C[\hat{\imath}^-;\hat{k}^-] - C[\hat{\imath}^+;\hat{k}^+] + C[\hat{\imath}^-;\hat{k}^+]\big) = \qquad \text{(term cancellation)}$$
$$C[\hat{\imath}^+;0] - C[\hat{\imath}^-;0] - C[\hat{\imath}^+;n] + C[\hat{\imath}^-;n] =$$
$$0 - 0 - (n - \hat{\imath}^+) + (n - \hat{\imath}^-) = 1$$

for all $\hat{\imath} \in \langle 0\colon n\rangle$. Symmetrically, we have

$$\textstyle\sum_{\hat{\imath}} F[\hat{\imath};\hat{k}] = 1$$

for all $\hat{k} \in \langle 0\colon n\rangle$. Thus, matrix F is bistochastic, therefore C is a simple Σ-bistochastic matrix.

Finally, consider simple Σ-subbistochastic matrices A, B. We have $A = D^{\Sigma}$, $B = E^{\Sigma}$, where D, E are subbistochastic matrices. As before, let $C = F^{\Sigma}$ for some matrix F; we have to show that F is a subbistochastic matrix. Let us extend matrices D, E with blocks of compatible sizes, so that the resulting matrices are bistochastic, and the product $D^{\Sigma} \odot E^{\Sigma} = F^{\Sigma}$ is preserved:

$$\begin{bmatrix} \mathsf{I} & \cdot & \cdot \\ \cdot & * & \cdot \\ \cdot & D & * \end{bmatrix}^{\Sigma} \odot \begin{bmatrix} * & \cdot & \cdot \\ E & * & \cdot \\ \cdot & \cdot & \mathsf{I} \end{bmatrix}^{\Sigma} = \begin{bmatrix} * & \cdot & \cdot \\ * & * & \cdot \\ F & * & * \end{bmatrix}^{\Sigma}$$

Here, zero blocks are denoted by dots, and arbitrary blocks by asterisks. It is easy to see that such an extension is always possible (and is generally not unique). As shown before, the matrix on the right-hand side is bistochastic, therefore matrix F is subbistochastic. ∎

Theorem 3.15 implies that, assuming a fixed index range, the set of all simple Σ-bistochastic matrices over $\mathbb{R}^+$ forms a monoid under tropical multiplication. We shall call it the Σ-*bistochastic monoid*.

The Σ-bistochastic monoid forms a submonoid in the Monge monoid. However, matrices $\mathbb{I}$ and $\mathbb{O}$ are neither simple nor regular. Instead, the Σ-bistochastic monoid has an identity matrix I^{Σ} and a zero matrix $\mathsf{I}^{\mathsf{R}\Sigma}$ (recall that I^{R} is the matrix obtained from I by 90-degree rotation):

$$\mathsf{I}^{\Sigma}[i;j] = \max(j - i, 0) \qquad \mathsf{I}^{\mathsf{R}\Sigma}[i;j] = \min(n - i, j)$$

For any bistochastic matrix D, we have

$$D^{\Sigma} \odot \mathsf{I}^{\Sigma} = \mathsf{I}^{\Sigma} \odot D^{\Sigma} = D^{\Sigma} \qquad D^{\Sigma} \odot \mathsf{I}^{\mathsf{R}\Sigma} = \mathsf{I}^{\mathsf{R}\Sigma} \odot D^{\Sigma} = \mathsf{I}^{\mathsf{R}\Sigma}$$

Chapter 4

Unit-Monge matrices and sticky braids

4.1 Unit-Monge matrices

Permutations and permutation matrices. The main building blocks for our structures are (combinatorial) permutations.

Definition 4.1 *A* subpermutation[1] $\pi: \langle I \rangle \to \langle J \rangle$ *is a partial injective function from* $\langle I \rangle$ *to* $\langle J \rangle$*. A* permutation *is a subpermutation that is total and surjective (and therefore bijective).* □

We will be dealing with permutations mostly in matrix form, exploiting the symmetry between the indices and the values of a permutation.

Definition 4.2 *A* permutation *(respectively,* subpermutation*)* matrix *is a zero-one matrix containing exactly one (respectively, at most one) nonzero element in every row and every column. A (sub)permutation* $\pi: \langle I \rangle \to \langle J \rangle$ *corresponds to a (sub)permutation matrix* $P\langle I: J \rangle$*, where* $\pi\langle \hat{\imath} \rangle = \hat{\jmath}$*, iff* $P\langle \hat{\imath}; \hat{\jmath} \rangle = 1$*; such a pair* $\langle \hat{\imath}; \hat{\jmath} \rangle$ *will be called a* nonzero *of* P*.* □

(Sub)permutation matrices are a subclass of (sub)bistochastic matrices. Any submatrix of a (sub)permutation matrix is a subpermutation matrix.

Example 4.3 The 3×3 matrix in Example 3.3 is a permutation matrix.■

An *identity matrix* I is a permutation matrix, in which all the nonzeros lie on the main diagonal. If both index ranges are identical, then $\mathsf{I}\langle \hat{\imath}; \hat{\jmath} \rangle = 1$, iff $\hat{\imath} = \hat{\jmath}$; this definition can be generalised naturally to indexing over arbitrary index ranges.

[1] There is some abuse of terminology in extending the term "(sub)permutation", which is normally reserved for the case $I = J$, to the case of general I, J. Still, we feel that it is justified by the context, and any ambiguity will be avoided.

Due to the extreme sparsity of (sub)permutation matrices, it would obviously be wasteful and inefficient to store them explicitly. Instead, we will normally assume that a permutation matrix P of size n is given implicitly by the underlying permutation and its inverse, i.e. by a pair of vectors π, π^{-1}, such that $P\big\langle \hat{\imath}; \pi\langle\hat{\imath}\rangle\big\rangle = 1$ for all $\hat{\imath}$, and $P\big\langle \pi^{-1}\langle\hat{\jmath}\rangle, \hat{\jmath}\big\rangle = 1$ for all $\hat{\jmath}$. Such a compact representation is linear in the total size of the index ranges, and allows constant-time querying of any nonzero of P by its row index, as well as by its column index. The implicit representation for subpermutation matrices is analogous.

Unit-Monge matrices and their implicit representation. The following subclasses of Monge matrices will play a crucial role in this work.

Definition 4.4 *Matrix A is a* (sub)unit-Monge *matrix, if it is a Σ-(sub)permutation matrix.* □

By the definitions, every unit-Monge matrix is Σ-bistochastic, and every Σ-bistochastic matrix is Monge. Similar inclusions hold for subunit-Monge and Σ-subbistochastic matrices.

Example 4.5 The 4×4 matrix in Example 3.3 is unit-Monge. ■

Lemma 3.5 allows us to represent a (sub)unit-Monge matrix A by its (sub)permutation cross-difference matrix $P = A^{\square}$ and a pair of vectors, reducing the representation size from quadratic to linear. Dominance summation on permutations reduces to *dominance counting*: $P^{\Sigma}[i;j]$ is the number of nonzeros in P that are $\gtrless$-dominated by point $[i;j]$, i.e. lie below-left of it.

An individual element of a (sub)unit-Monge matrix can be queried from its canonical representation (3.1) or (3.2) in time $O(n)$ by a linear sweep of nonzeros in P, counting those that are $\gtrless$-dominated by point $[i;j]$. For faster access to elements of P^{Σ}, matrix P can be preprocessed into an appropriate data structure.

Theorem 4.6 *Given a (sub)permutation matrix P with n nonzeros, there exists a data structure that*

- *has size $O\big(n \log n\big)$;*
- *can be built in time $O\big(n \log n\big)$;*
- *supports queries for any individual element of the simple (sub)unit-Monge matrix P^{Σ} in time $O\big(\log^2 n\big)$;* □

Proof The required data structure is a classical two-dimensional range tree [30] (see also [173]), built on the set of nonzeros in P. There are at most n nonzeros, hence the total number of nodes in the tree is $O\big(n \log n\big)$. A dominance counting query on the set of nonzeros can be answered by accessing $O\big(\log^2 n\big)$ of the tree nodes. ■

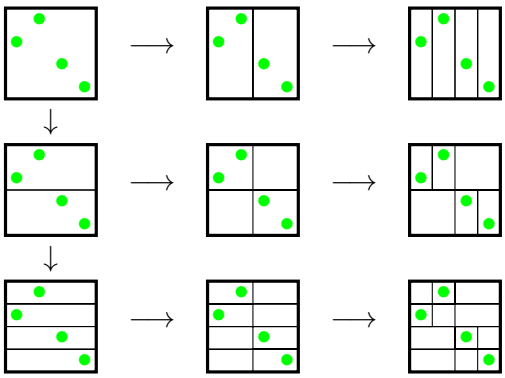


Figure 4.1: A permutation matrix and the corresponding range tree

Example 4.7 Figure 4.1 shows a 4×4 permutation matrix, with nonzeros indicated by green bullets, and the nodes of the corresponding range tree. ■

The bounds given by Theorem 4.6 can be improved upon by employing more advanced data structures. Successive improvements to the efficiency of orthogonal range counting (which includes dominance counting as a special case) were obtained by Chazelle [58], JáJá et al. [122], Chan and Pătraşcu [54]. The currently most efficient data structure of [54] has size $O(n)$, can be built in time $O\big(n(\log n)^{1/2}\big)$, and answers a dominance counting query in time $O\big(\frac{\log n}{\log\log n}\big)$. However, the standard range tree data structure employed by Theorem 4.6 is simpler, requires a less powerful computation model, and is more likely to be practical.

In addition to *random access queries* described by Theorem 4.6, we will have a need for *incremental queries*. Given an element of a canonically represented (sub)unit-Monge matrix, such a query returns the value of a specified adjacent element. Incremental queries can be answered directly from the canonical representation (3.1) or (3.2), without the need for any non-trivial data structures or preprocessing.

Theorem 4.8 *Given a (sub)permutation matrix P and the value $P^{\Sigma}[i;j]$, the values $P^{\Sigma}[i\pm1;j]$, $P^{\Sigma}[i;j\pm1]$, where they exist, can be queried in time $O(1)$.* □

Proof Let P be a permutation matrix; a generalisation to subpermutation matrices is straightforward. Consider a query of the type $P^{\Sigma}[i+1;j]$; the proof for other query types is analogous. Let $\hat{\jmath}$ be such that $P[i^+;\hat{\jmath}] = 1$; value $\hat{\jmath}$ can be obtained from the permutation representation of P in time $O(1)$. We have

$$P^{\Sigma}[i+1;j] = P^{\Sigma}[i;j] - \begin{cases} 1 & \text{if } \hat{\jmath} < j \\ 0 & \text{otherwise} \end{cases}$$

■

We will call the incremental queries of type $P^{\Sigma}[i \pm 1; j]$ *columnwise*, and of type $P^{\Sigma}[i; j \pm 1]$ *rowwise*. Incremental queries described by Theorem 4.8 can be used to answer *batch queries*, returning a set of elements in a row, column or diagonal of a canonically represented (sub)unit-Monge matrix. In particular, given matrix P, all elements in a given row, column or diagonal of P^{Σ} can be obtained by a sequence of incremental queries in time $O(n)$. Given a range tree for matrix P, a subset of r consecutive elements in a given row, column or diagonal of P^{Σ} can be obtained in time $O\big(r + \log^2 n\big)$.

4.2 Searching in unit-Monge matrices

We now specialise our problem to row minima searching in a canonically represented unit-Monge matrix. By Theorem 4.6 and Lemma 3.5, an element of a unit-Monge matrix, represented by an appropriate data structure, can be queried in time $q = O(\log^2 n)$. By plugging this query time into Theorem 3.10, we obtain immediately an algorithm for row minima searching running in time $O(n \log^2 n)$. However, a more careful analysis of the elimination procedure of Theorem 3.10 shows that the required matrix elements can be obtained, instead of the random access queries of Theorem 4.6, by the more efficient incremental queries of Theorem 4.8. At the top level of recursion, each individual query time is $q = O(1)$. In lower recursion levels, an incremental query may take more time, since it may have to traverse one or more eliminated columns. The combined query time in any given recursion level is $O(n)$, therefore the overall running time becomes $O(n \log n)$.

A more efficient algorithm for this problem was given by Gawrychowski [97].

Theorem 4.9 *Let A be a (sub)unit-Monge matrix, canonically represented by the (sub)permutation matrix $P = A^{\square}$ with n nonzeros, and by implicit vectors b, c, where each vector element can be queried in time $O(1)$. Given P, b, c, a data structure can be computed in time $O(n \log\log n)$ in the pointer machine model, and in time $O(n)$ in the unit-cost RAM model, such that the index of the leftmost minimum element in a row of A can be queried in time $O(1)$.* □

Proof First, observe that vector c has no effect on the positions of any row minima (although it may affect their values). Therefore, we assume without loss of generality that $c[i] = 0$ for all i. Further, suppose that some column $P\langle :; \hat{\jmath}\rangle$ is identically zero; then, depending on whether $b[\hat{\jmath}^-] \le b[\hat{\jmath}^+]$ or $b[\hat{\jmath}^-] > b[\hat{\jmath}^+]$, we may ignore respectively column $A[:; \hat{\jmath}^+]$ or $A[:; \hat{\jmath}^-]$ as it cannot contain any leftmost row minima. Also, suppose that some row $P\langle \hat{\imath}; :\rangle$ is identically zero; then the minimum values in rows $A[\hat{\imath}^-; :]$, $A[\hat{\imath}^+; :]$ lie in the same column(s), hence we can safely ignore one of these rows. Therefore, we may consider without loss of generality an implicit

unit-Monge matrix $A[0{:}\,n; 0{:}\,n]$, canonically represented by a permutation matrix P. Also note that we can perturb the elements of vector b slightly, so that each leftmost row minimum becomes the only minimum in its row. Therefore, from now on we will omit the qualifier "leftmost".

Consider vector b, which coincides with the bottom row of matrix A: $b = A[n;:]$. Suppose that for some j, j', $j \leq j'$, we have $A[n;j] \leq A[n;j']$. Then, $A[n;j']$ cannot be the minimum in row n. Furhermore, by the Monge property of matrix A, we have $A[i;j] \leq A[i;j']$ for all i, therefore $A[i;j']$ cannot be the minimum in any row i, and hence column j' can be safely excluded from the search for row minima. After excluding all such columns, the remaining elements in row n form a decreasing subsequence of *prefix minima*. An element $b[j']$ is called a prefix minimum, if we have $b[j] > b[j']$ for all $j \leq j'$. The prefix minima sequence can be found trivially in a single pass of the input vector b in time $O(n)$. The final prefix minimum is the row minimum in row n.

Let $j_0 < j_1 < \ldots < j_r$ be the indices of the prefix minima in row n, so the prefix minima for that row are

$$b[j_0] > b[j_1] > \ldots > b[j_r]$$

Our goal now is to compute prefix minima for every row in matrix A. We will represent the prefix minima sequences implicitly by storing the differences between successive pairs of elements. In particular, the initial prefix minima sequence is represented by the sequence of (all negative) values

$$d_{\hat{k}} = b[j_{\hat{k}^+}] - b[j_{\hat{k}^-}] \qquad \hat{k} \in \langle 0{:}\,r \rangle$$

We now sweep the rows of matrix A from the bottom row n towards the top row 0, updating the implicit prefix minima sequence incrementally for each row. We describe the procedure for updating the sequence from row n to row $n-1$; the other updates are analogous.

Let $P\langle n^-; \hat{\jmath} \rangle = 1$ be the nonzero of matrix P in row n^-. Let

$$k_0 = \max\{k{:}\, j_k < \hat{\jmath}\}$$
$$k_1 = \min\{k{:}\, j_k > \hat{\jmath} \text{ and } b[j_k] + 1 < b[j_{k_0}]\}$$

Recall that $A[i;j] = P^{\Sigma}[i;j] + b[j]$ for all i, j. Assuming that both k_0 and k_1 above are well-defined (i.e. the maximisation and minimisation domains are non-empty), the prefix minima sequence in row $n-1$ is

$$b[j_0] > b[j_1] > \ldots > b[j_{k_0}] > b[j_{k_1}] + 1 > b[j_{k_1+1}] + 1 > \ldots > b[j_r] + 1$$

In other words, we keep all prefix minima in row n from $b[j_0]$ to $b[j_{k_0}]$ inclusive, we delete all prefix minima strictly between $b[j_{k_0}]$ and $b[j_{k_1}]$, and we keep all prefix minima from $b[j_{k_1}]$ to $b[j_r]$, incrementing them by 1.

The described updated prefix minima sequence for row $n-1$ is represented implicitly by the updated difference sequence

$$d_{0^+}, d_{1^+}, \ldots, d_{j_0^-}, b[j_{k_1}] - b[j_{k_0}] + 1, d_{j_1^+}, d_{j_1^+ + 1}, \ldots, d_{r^-}$$

In other words, we keep all the differences for row n from d_{0^+} to $d_{j_0^-}$ inclusive, we delete all the differences from $d_{j_0^+}$ to $d_{j_1^-}$ inclusive, we create a new difference $b[j_{k_1}] - b[j_{k_0}] + 1$, and we keep all the differences from $d_{j_1^+}$ to d_{r^-} inclusive. Assuming that the difference sequence is kept in a linked list, and that indices k_0 and k_1 are known, such an update can be performed in time $O(1)$.

In case k_0 is undefined (this happens whenever $j_0 > \hat{\jmath}$), the updated prefix minima sequence becomes

$$b[j_0] + 1 > b[j_1] + 1 > \ldots > b[j_r] + 1$$

hence the difference sequence for row $n-1$ remains the same as for row n, and need not be updated. In case k_1 is undefined (this happens whenever $b[j_r] + 1 > b[j_{k_0}]$), the updated prefix minima sequence becomes

$$b[j_0] > b[j_1] > \ldots > b[j_{k_0}]$$

hence the difference sequence for row $n-1$ is obtained by keeping the prefix minima for row n from d_{0^+} to $d_{k_0^-}$ inclusive, and deleting the remaining ones. In both above cases, the update can still be performed in time $O(1)$.

Now, assume that only index k_0 is known before the start of the update. Then, index k_1 can be found by linear search through the difference sequence. The size of this linear search is equal to the number of differences deleted from the sequence by the subsequent update. Hence, the amortised running time of the linear search is $O(1)$ per update, and $O(n)$ across all the updates.

It remains to show how to find the index k_0 efficiently. Consider the partitioning of interval $\langle 0 \colon n \rangle$ into a disjoint union of sub-intervals

$$\langle 0 \colon n \rangle = \langle 0 \colon j_0 \rangle \uplus \langle j_0 \colon j_1 \rangle \uplus \cdots \uplus \langle j_{r-1} \colon j_r \rangle$$

The problem of finding k_0 is equivalent to finding the interval $\langle j_{k_0} \colon j_{k_0+1} \rangle$ containing the index $\hat{\jmath}$ of the nonzero $P\langle n^- ; \hat{\jmath} \rangle$. The same problem has to be solved repeatedly for each subsequent row, where we need to find the interval between elements of the current prefix minima sequence, containing the current nonzero of matrix P. As elements get deleted from the prefix minima sequence by the update, pairs of adjacent intervals also have to be merged into one interval.

The described problem fits in the classical setup of the *union-find problem*, in particular its special case called the *interval union-find problem* (see e.g. Italiano and Raman [120]). In the pointer machine model, this problem

can be solved by an algorithm of van Emde Boas [213] (see also [120]) in running time $O(\log\log n)$ per update, hence $O(n \log\log n)$ across all the updates. In the unit-cost RAM model of computation, it can be solved by an algorithm of Gabow and Tarjan [93] (see also [94, 120]) in amortised running time $O(1)$ per update, hence $O(n)$ across all the updates. ■

4.3 Unit-Monge monoid

Unit-Monge matrix multiplication. In this section, we explore tropical multiplication of unit-Monge matrices. It is somewhat surprising that, similarly to the Monge and the simple Σ-bistochastic properties, the simple unit-Monge property is also preserved under tropical multiplication.

Theorem 4.10 *Let $A \odot B = C$. If A, B are simple unit-Monge (respectively, simple subunit-Monge) matrices, then C is also a simple unit-Monge (respectively, simple subunit-Monge) matrix.* □

Proof Matrix C is integer, and is also simple Σ-bistochastic (respectively, Σ-subbistochastic) by Theorem 3.15. Therefore, C is unit-Monge (respectively, subunit-Monge). ■

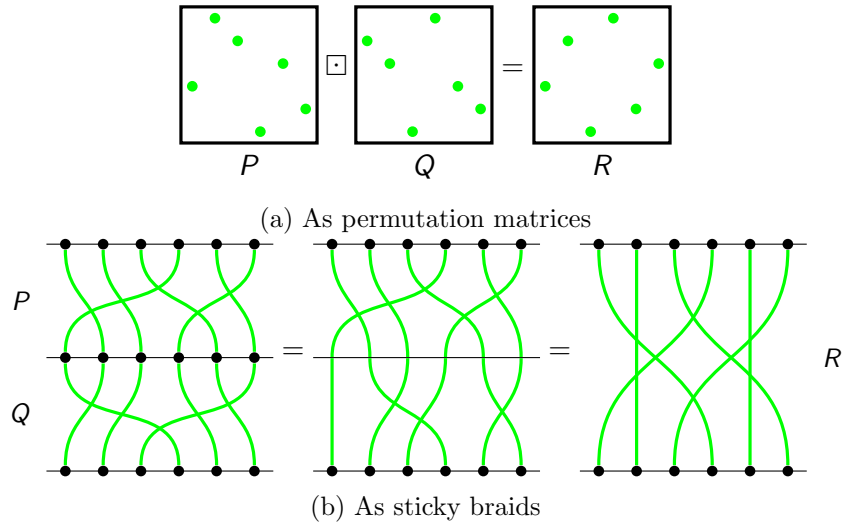
Theorem 4.10 implies that, assuming a fixed index range, the set of all simple unit-Monge matrices over $\mathbb{R}^+$ forms a monoid under tropical matrix multiplication. We shall call it the *unit-Monge monoid* (another appropriate name would be the *planar grid-like distance monoid*).

The unit-Monge monoid forms a submonoid in the Σ-bistochastic monoid. Matrices I^{Σ} and $\mathsf{I}^{\mathsf{R}\Sigma}$ are both simple unit-Monge, and are therefore inherited as the identity and the zero by the unit-Monge monoid.

Sticky matrix multiplication. Theorem 4.10 gives us the basis for performing tropical multiplication of Monge matrices implicitly, using their canonical representation by permutation matrices. It will be convenient to introduce special terminology and notation for such implicit tropical matrix multiplication.

Definition 4.11 *Let D, E, F be matrices. We introduce the following operations:*

- implicit tropical multiplication $D \boxdot E = F$, *also called* sticky multiplication, *defined by* $D^{\Sigma} \odot E^{\Sigma} = F^{\Sigma}$;
- left semi-implicit tropical multiplication $D \ltimes E = F$ *defined by* $D^{\Sigma} \odot E = F$;
- right semi-implicit tropical multiplication $D \rtimes E = F$ *defined by* $D \odot E^{\Sigma} = F$. □



(a) As permutation matrices

(b) As sticky braids

Figure 4.2: Sticky product $P \boxdot Q = R$

By Theorem 4.10, the set of all $n \times n$ permutation matrices forms a monoid with respect to sticky multiplication; we denote this monoid $\mathcal{T}_n$. Monoid $\mathcal{T}_n$ is isomorphic to the unit-Monge monoid, with I as the identity and I^{R} as the zero:

$$P \boxdot \mathsf{I} = \mathsf{I} \boxdot P = P \qquad P \boxdot \mathsf{I}^{\mathsf{R}} = \mathsf{I}^{\mathsf{R}} \boxdot P = \mathsf{I}^{\mathsf{R}}$$

The most familiar monoid on permutation matrices is the symmetric group $\mathcal{S}_n$, defined by standard matrix multiplication. It should be noted that monoid $\mathcal{T}_n$ is substantially different from $\mathcal{S}_n$. While both share the same identity element I, monoid $\mathcal{T}_n$ has a zero element I^{R} and plenty of non-trivial idempotent elements (corresponding to involutive permutations), which are impossible in a group such as $\mathcal{S}_n$.

Example 4.12 Subfigure 4.2a shows a triple of 6×6 permutation matrices P, Q, R, such that $P \boxdot Q = R$. Nonzeros are indicated by green circles. ■

Just like the symmetric group $\mathcal{S}_n$, monoid $\mathcal{T}_n$ is generated by transpositions. Recall that I denotes the identity matrix.

Definition 4.13 *The* transposition matrix G_t *is defined as*

$$G_t\langle \hat{\imath}; \hat{\jmath} \rangle = \begin{cases} 1 - \mathsf{I}\langle \hat{\imath}; \hat{\jmath} \rangle & \text{if } \hat{\imath}, \hat{\jmath} \in \{t^-; t^+\} \\ \mathsf{I}\langle \hat{\imath}; \hat{\jmath} \rangle & \text{otherwise} \end{cases}$$ □

Lemma 4.14 *The sticky multiplication monoid of permutation matrices* $\mathcal{T}_n$ *is generated by the set* $\{G_t\}$, $t \in [1\colon n-1]$. □

PROOF Let P be a permutation matrix. Consider an arbitrary reduced sticky braid corresponding to P, and let t be the position of its first elementary crossing. Consider the truncated sticky braid, obtained by removing this crossing. This braid is still reduced, and such that the pair of strands originating in t^-, t^+ do not cross. Let this pair of strands terminate at indices $\hat{k}_0$, $\hat{k}_1$, where $\hat{k}_0 < \hat{k}_1$. Let Q be the permutation matrix corresponding to the truncated sticky braid. We have

$$P\langle t^-; \hat{k}_1\rangle = P\langle t^+; \hat{k}_0\rangle = 1$$
$$Q\langle t^-; \hat{k}_0\rangle = Q\langle t^+; \hat{k}_1\rangle = 1$$

We will now show that $P = G_t \boxdot Q$ or, equivalently $P^\Sigma = G_t^\Sigma \odot Q^\Sigma$. The lemma statement then follows by induction.

Note that $G_t^\Sigma[i;j] = \mathsf{I}^\Sigma[i;j]$ and $Q^\Sigma[i;j] = P^\Sigma[i;j]$ for all $i \in [0\mathinner{:}n]$, $i \neq t$, and for all $j \in [0\mathinner{:}n]$. Therefore, we have

$$\big(G_t^\Sigma \odot Q^\Sigma\big)[i;k] = \big(\mathsf{I}^\Sigma \odot Q^\Sigma\big)[i;k] = Q^\Sigma[i;k] = P^\Sigma[i;k]$$

for all $i \in [0\mathinner{:}n]$, $i \neq t$, and for all $k \in [0\mathinner{:}n]$.

It remains to consider the case $i = t$. Note that $G_t^\Sigma[t;j] = \mathsf{I}^\Sigma[t;j]$ for all $j \in [0\mathinner{:}n]$, $j \neq t$. Let $k \in [0\mathinner{:}n]$. We have

$$\big(G_t^\Sigma \odot Q^\Sigma\big)[t;k] = \min_{j\in[0:n]}\big(G_t^\Sigma[t;j] + Q^\Sigma[j;k]\big) \tag{4.1}$$

By Definition 3.1 (dominance-sum matrix), we have

$$G_t^\Sigma[t;t-1] = 0$$
$$G_t^\Sigma[t;t] = G_t^\Sigma[t;t+1] = 1$$
$$0 \le Q^\Sigma[t-1;k] - Q^\Sigma[t;k] \le 1$$
$$0 \le Q^\Sigma[t;k] - Q^\Sigma[t+1;k] \le 1$$

Hence, we have

$$G_t^\Sigma[t;t] + Q^\Sigma[t;k] = 1 + Q^\Sigma[t;k] \ge$$
$$0 + Q^\Sigma[t-1;k] = G_t^\Sigma[t;t-1] + Q^\Sigma[t-1;k]$$

and, analogously,

$$G_t^\Sigma[t;t] + Q^\Sigma[t;k] \ge G_t^\Sigma[t;t+1] + Q^\Sigma[t+1;k]$$

We have established that the value under the minimum operator in (4.1) for $j = t$ is always no less than the values for both $j = t-1$ and $j = t+1$. Therefore, the minimum is never attained solely at $j = t$, so we may assume $j \neq t$. We now consider two cases: either $j \in [0\mathinner{:}t-1]$, or $j \in [t+1\mathinner{:}n]$.

For $j \in [0\colon t-1]$, we have $G_t^{\Sigma}[t;j] = 0$. Therefore,

$$\begin{aligned}
&\min_{j\in[0:t-1]}\big(G_t^{\Sigma}[t;j] + Q^{\Sigma}[j;k]\big) = \\
&\min_{j\in[0:t-1]}\big(0 + Q^{\Sigma}[j;k]\big) = && \text{(attained at } j = t-1\text{)}\\
&Q^{\Sigma}[t-1;k] = P^{\Sigma}[t-1;k]
\end{aligned}$$

Similarly, for $j \in [t+1\colon n]$, we have $G_t^{\Sigma}[t;j] = j - t$. Therefore,

$$\begin{aligned}
&\min_{j\in[t+1:n]}\big(G_t^{\Sigma}[t;j] + Q^{\Sigma}[j;k]\big) = \\
&\min_{j\in[t+1:n]}\big(j - t + Q^{\Sigma}[j;k]\big) = && \text{(attained at } j = t+1\text{)}\\
&1 + Q^{\Sigma}[t+1;k] = 1 + P^{\Sigma}[t+1;k]
\end{aligned}$$

Substituting into (4.1), we now have

$$\big(G_t^{\Sigma} \odot Q^{\Sigma}\big)[t;k] = \min\big(P^{\Sigma}[t-1;k], 1 + P^{\Sigma}[t+1;k]\big)$$

Recall that $P\langle t^-; \hat{k}_1\rangle = P\langle t^+; \hat{k}_0\rangle = 1$. We have

$$\begin{aligned}
&P^{\Sigma}[t-1;k] = P^{\Sigma}[t;k] = P^{\Sigma}[t+1;k] && \text{for } k < \hat{k}_0\\
&P^{\Sigma}[t-1;k] = P^{\Sigma}[t;k] = 1 + P^{\Sigma}[t+1;k] && \text{for } \hat{k}_0 < k < \hat{k}_1\\
&P^{\Sigma}[t-1;k] - 1 = P^{\Sigma}[t;k] = 1 + P^{\Sigma}[t+1;k] && \text{for } \hat{k}_1 < k
\end{aligned}$$

In all three above cases, we have

$$\min\big(P^{\Sigma}[t-1;k], 1 + P^{\Sigma}[t+1;k]\big) = P^{\Sigma}[t;k]$$

which completes the proof. ∎

Matrix-vector multiplication algorithm. As in Theorem 3.13, an efficient algorithm for semi-implicit distance multiplication of a simple (sub)unit-Monge matrix by a vector can be obtained as an application of row minima searching.

Theorem 4.15 *Let $P \ltimes b = c$, where $P\langle I; J\rangle$ is a (sub)permutation matrix, and $b[J]$, $c[I]$ are vectors. Given the nonzeros of P and the full vector b, vector c can be computed in time $O(n \log\log n)$ in the pointer machine model, and in time $O(n)$ in the unit-cost RAM model, where $n = \max(|\langle I\rangle|, |\langle J\rangle|)$.*□

Proof Analogous to Theorem 3.13, but using Theorem 4.9 for finding row minima. ∎

Matrix multiplication algorithm. While the quadratic running time of tropical matrix multiplication by Theorem 3.14 is trivially optimal for explicit Monge matrices, it turns out to be possible to multiply matrices in the unit-Monge monoid faster, representing them implicitly.

We now present our algorithm of [209] for implicit distance multiplication of canonically represented simple (sub)unit-Monge matrices (equivalently, sticky multiplication of (sub)permutation matrices); a related algorithm for the same problem was proposed independently by Sakai [188]. For simplicity, we restrict ourselves once again to square matrices (a simplifying assumption for subunit-Monge matrices, which holds trivially for unit-Monge ones).

Theorem 4.16 (The Steady Ant algorithm) *Let $P \boxdot Q = R$, where P, Q, R are (sub)permutation matrices with at most n nonzeros each. Given the nonzeros of P, Q, the nonzeros of R can be computed in time $O(n \log n)$.* □

Proof Without loss of generality, consider permutation matrices $P\langle 0{:}\,n; 0{:}\,n\rangle$, $Q\langle 0{:}\,n; 0{:}\,n\rangle$, $R\langle 0{:}\,n; 0{:}\,n\rangle$; the generalisation to subpermutation matrices is as in Theorem 3.15. The algorithm proceeds 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. We first give a brief informal sketch of the algorithm. The algorithm works by divide-and-conquer. In the *divide phase*, we split the range of index j into two sub-intervals of size $\frac{n}{2}$. For each of these half-sized sub-intervals of j, we use the sparsity of the input permutation matrix P (respectively, Q) to condense the range of index i (respectively, k) to a (not necessarily contiguous) subset of size $\frac{n}{2}$, obtaining two independent half-sized subproblems. We perform two recursive calls on the subproblems, and then restore each subproblem's solution to its original index range. In the *conquer phase*, we reconstruct the output permutation matrix R from the two subproblem solutions by a non-trivial and fairly elegant linear-time procedure.

We now describe each phase of divide-and-conquer in more detail.

Divide phase. Matrices P, Q are partitioned into rectangular blocks

$$P = \begin{bmatrix} P_{lo} & P_{hi} \end{bmatrix} \qquad Q = \begin{bmatrix} Q_{lo} \\ Q_{hi} \end{bmatrix}$$

where $P_{lo} = P\big\langle :; 0{:}\,\frac{n}{2}\big\rangle$, $P_{hi} = P\big\langle :; \frac{n}{2}{:}\,n\big\rangle$, $Q_{lo} = Q\big\langle 0{:}\,\frac{n}{2}; :\big\rangle$, $Q_{hi} = Q\big\langle \frac{n}{2}{:}\,n; :\big\rangle$ are subpermutation matrices with $\frac{n}{2}$ nonzeros each. We now have two sticky matrix multiplication subproblems

$$P_{lo} \boxdot Q_{lo} = R_{lo} \qquad P_{hi} \boxdot Q_{hi} = R_{hi}$$

where the product matrices $R_{lo}\langle 0{:}\,n; 0{:}\,n\rangle$, $R_{hi}\langle 0{:}\,n; 0{:}\,n\rangle$ are subpermutation matrices with $\frac{n}{2}$ nonzeros each. Note that the multiplicands' nonzeros in the

two subproblems have disjoint index ranges in both coordinates. Therefore, matrix $R_{lo} + R_{hi}$, which represents the union of product nonzeros across both subproblems, is a permutation matrix.

It is easy to see that a zero row in P_{lo} (respectively, a zero column in Q_{lo}) corresponds to a zero row (respectively, column) in their implicit tropical product R_{lo}. Therefore, we can delete all zero rows and columns from P_{lo}, Q_{lo}, R_{lo}, obtaining from each, after appropriate index remapping, a condensed $\frac{n}{2} \times \frac{n}{2}$ permutation matrix. 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_{lo}, Q_{lo}), then making a recursive call on the resulting condensed half-sized problem, and then performing an inverse index remapping (corresponding to the reinsertion of the zero rows and columns into R_{lo}). The second subproblem can be solved analogously.

Conquer phase. We now need to combine the solutions for the two subproblems to a solution for the original problem. It might be tempting to expect that the permutation matrix $R_{lo} + R_{hi}$ is itself such a solution. However, this is not the case, since the original problem's solution depends on the subproblems' solutions in a more subtle way: some elements of P^{Σ} depend on elements of both P_{lo} and P_{hi}, and therefore may not be accounted for by the solution to either subproblem on its own. A similar observation holds for elements of Q^{Σ}.

In order to combine correctly the solutions of the two subproblems, let us consider carefully their relationship. By Definition 4.11 (sticky matrix multiplication), we have

$$P^{\Sigma} \odot Q^{\Sigma} = R^{\Sigma} \qquad P^{\Sigma}_{lo} \odot Q^{\Sigma}_{lo} = R^{\Sigma}_{lo} \qquad P^{\Sigma}_{hi} \odot Q^{\Sigma}_{hi} = R^{\Sigma}_{hi}$$

We now apply Definition 2.1 (tropical matrix multiplication), splitting the range of index j into a "low" and a "high" sub-interval, each of size $\frac{n}{2}$:

$$R^{\Sigma}[i;k] = \min_{j \in [0:n]} \big(P^{\Sigma}[i;j] + Q^{\Sigma}[j;k]\big) =$$
$$\min\Big(\min_{j \in \left[0:\frac{n}{2}\right]} \big(P^{\Sigma}[i;j] + Q^{\Sigma}[j;k]\big), \min_{j \in \left[\frac{n}{2}:n\right]} \big(P^{\Sigma}[i;j] + Q^{\Sigma}[j;k]\big)\Big) \tag{4.2}$$

for all $i, k \in [0:n]$. Let us denote the two arguments in (4.2) by $M_{lo}[i;k]$ and $M_{hi}[i;k]$, respectively:

$$R^{\Sigma}[i;k] = \min\big(M_{lo}[i;k], M_{hi}[i;k]\big) \tag{4.3}$$

for all $i, k \in [0:n]$. The first argument in (4.2), (4.3) can be expressed via the solutions of the two subproblems as follows:

$$M_{lo}[i;k] = \min_{j \in \left[0:\frac{n}{2}\right]} \big(P^{\Sigma}[i;j] + Q^{\Sigma}[j;k]\big) = \qquad \text{(definition of } \Sigma\text{)}$$

$$\min_{j\in\left[0:\frac{n}{2}\right]}\left(P^{\Sigma}_{lo}[i;j]+Q^{\Sigma}_{lo}[j;k]+Q^{\Sigma}_{hi}[\tfrac{n}{2},k]\right)= \qquad \text{(term rearrangement)}$$

$$\min_{j\in\left[0:\frac{n}{2}\right]}\left(P^{\Sigma}_{lo}[i;j]+Q^{\Sigma}_{lo}[j;k]\right)+Q^{\Sigma}_{hi}[\tfrac{n}{2},k]= \qquad \text{(definition of } \odot)$$

$$R^{\Sigma}_{lo}[i;k]+R^{\Sigma}_{hi}[0;k] \qquad (4.4)$$

Here, the final equality is due to

$$R^{\Sigma}_{hi}[0;k]=\min_{j\in\left[\frac{n}{2}:n\right]}\left(P^{\Sigma}_{hi}[0;j]+Q^{\Sigma}_{hi}[j;k]\right)=$$
$$\min_{j\in\left[\frac{n}{2}:n\right]}\left(j-\tfrac{n}{2}+Q^{\Sigma}_{hi}[j;k]\right)=Q^{\Sigma}_{hi}[\tfrac{n}{2},k]$$

since the minimum is attained at $j=\frac{n}{2}$. The second argument in (4.2), (4.3) can be expressed analogously to (4.4) as

$$M_{hi}[i;k]=\min_{j\in\left[\frac{n}{2}:n\right]}\left(P^{\Sigma}[i;j]+Q^{\Sigma}[j;k]\right)=R^{\Sigma}_{hi}[i;k]+R^{\Sigma}_{lo}[i;n] \qquad (4.5)$$

To evaluate the minimisation operator in (4.2), (4.3), we establish the sign of the difference of its two arguments. Let $\Delta=M_{lo}-M_{hi}$. We have

$$\Delta[i;k]=M_{lo}[i;k]-M_{hi}[i;k]= \qquad \text{(by (4.4), (4.5))}$$
$$\left(R^{\Sigma}_{lo}[i;k]+R^{\Sigma}_{hi}[0;k]\right)-\left(R^{\Sigma}_{hi}[i;k]+R^{\Sigma}_{lo}[i;n]\right)= \qquad \text{(term rearrangement)}$$
$$\left(R^{\Sigma}_{hi}[0;k]-R^{\Sigma}_{hi}[i;k]\right)-\left(R^{\Sigma}_{lo}[i;n]-R^{\Sigma}_{lo}[i;k]\right)= \qquad \text{(def of } \Sigma, \mathsf{M}, \mathsf{W})$$
$$R^{\mathsf{M}}_{hi}[i;k]-R^{\mathsf{W}}_{lo}[i;k] \qquad (4.6)$$

Since R_{lo}, R_{hi} are subpermutation matrices, and $R_{lo}+R_{hi}$ a permutation matrix, it follows that matrix Δ is unit-nondecreasing both by rows and by columns.

Nonzeros in R are determined by the signs of elements in Δ as follows. Let us fix some half-integer point $\hat{\imath};\hat{k}\in\langle 0:n\rangle$ in R, and consider the sign of Δ at the four points $[\hat{\imath}^{\pm};\hat{k}^{\pm}]$, where the signs in each coordinate are chosen independently. Due to the unit-nondecreasing property of Δ, only three cases are possible.

Case $\Delta[\hat{\imath}^{\pm};\hat{k}^{\pm}]\le 0$ *for all four sign combinations.* We have

$$M_{lo}[\hat{\imath}^{\pm};\hat{k}^{\pm}]\le M_{hi}[\hat{\imath}^{\pm};\hat{k}^{\pm}]$$

for each sign combination taken consistently on both sides of the inequality, and, by (4.3),

$$R^{\Sigma}[\hat{\imath}^{\pm};\hat{k}^{\pm}]=M_{lo}[\hat{\imath}^{\pm};\hat{k}^{\pm}]$$

Hence, we have

$$R\langle\hat{\imath};\hat{k}\rangle = R^{\Sigma\square}\langle\hat{\imath};\hat{k}\rangle = M^{\square}_{lo}\langle\hat{\imath};\hat{k}\rangle = R_{lo}\langle\hat{\imath};\hat{k}\rangle \qquad \text{(def of } \Sigma, \square, (4.4), (4.5))$$

Thus, in this case $R\langle\hat{\imath};\hat{k}\rangle = 1$, if and only if $R_{lo}\langle\hat{\imath};\hat{k}\rangle = 1$. Note that this also implies $\Delta[\hat{\imath}^-;\hat{k}^-] < 0$, since otherwise we would have $\Delta[\hat{\imath}^\pm;\hat{k}^\pm] = 0$ for all four sign combinations, and hence, by symmetry, also $R_{hi}\langle\hat{\imath};\hat{k}\rangle = 1$. However, that would imply $R_{lo}\langle\hat{\imath};\hat{k}\rangle + R_{hi}\langle\hat{\imath};\hat{k}\rangle = 1 + 1 = 2$, which is a contradiction to $R_{lo} + R_{hi}$ being a permutation matrix.

Case $\Delta[\hat{\imath}^\pm;\hat{k}^\pm] \geq 0$ *for all four sign combinations.* Symmetrically to the previous case, we have

$$R\langle\hat{\imath};\hat{k}\rangle = R_{hi}\langle\hat{\imath};\hat{k}\rangle$$

Thus, in this case $R\langle\hat{\imath};\hat{k}\rangle = 1$ if and only if $R_{hi}\langle\hat{\imath};\hat{k}\rangle = 1$, which implies $\Delta[\hat{\imath}^+;\hat{k}^+] > 0$.

Case $\Delta[\hat{\imath}^-;\hat{k}^-] < 0$, $\Delta[\hat{\imath}^-;\hat{k}^+] = \Delta[\hat{\imath}^+;\hat{k}^-] = 0$, $\Delta[\hat{\imath}^+;\hat{k}^+] > 0$. By (4.3), we have

$$\begin{aligned}
R^{\Sigma}[\hat{\imath}^-;\hat{k}^-] &= M_{lo}[\hat{\imath}^-;\hat{k}^-]\\
R^{\Sigma}[\hat{\imath}^+;\hat{k}^-] &= M_{lo}[\hat{\imath}^+;\hat{k}^-]\\
R^{\Sigma}[\hat{\imath}^-;\hat{k}^+] &= M_{lo}[\hat{\imath}^-;\hat{k}^+]\\
R^{\Sigma}[\hat{\imath}^+;\hat{k}^+] &= M_{hi}[\hat{\imath}^+;\hat{k}^+] < M_{lo}[\hat{\imath}^+;\hat{k}^+]
\end{aligned}$$

Hence,

$$R\langle\hat{\imath};\hat{k}\rangle = R^{\Sigma\square}\langle\hat{\imath};\hat{k}\rangle > M^{\square}_{lo}\langle\hat{\imath};\hat{k}\rangle = R_{lo}\langle\hat{\imath};\hat{k}\rangle \qquad \text{(def of } \Sigma, \square, (4.4), (4.5))$$

Since both R and R_{lo} are zero-one matrices, the above strict inequality implies that $R\langle\hat{\imath};\hat{k}\rangle = 1$ and $R_{lo}\langle\hat{\imath};\hat{k}\rangle = 0$. Symmetrically, also $R_{hi}\langle\hat{\imath};\hat{k}\rangle = 0$.

Summarising the above three cases, we have $R\langle\hat{\imath};\hat{k}\rangle = 1$, if and only if one of the following conditions holds:

$$R_{lo}\langle\hat{\imath};\hat{k}\rangle = 1 \text{ and } \Delta[\hat{\imath}^-;\hat{k}^-] < 0 \tag{4.7}$$

$$R_{hi}\langle\hat{\imath};\hat{k}\rangle = 1 \text{ and } \Delta[\hat{\imath}^+;\hat{k}^+] > 0 \tag{4.8}$$

$$\Delta[\hat{\imath}^-;\hat{k}^-] < 0 \text{ and } \Delta[\hat{\imath}^+;\hat{k}^+] > 0 \tag{4.9}$$

By the argument above, conditions (4.7)–(4.9) are mutually exclusive. A nonzero of R_{lo} (respectively, R_{hi}) will be called *good*, if it satisfies (4.7) (respectively, (4.8)), and otherwise *bad*. A nonzero of R that satisfies (4.9) will be called *fresh*. Every nonzero in R is either a good nonzero of R_{lo}, or a good nonzero of R_{hi}, or a fresh nonzero.

In order to check the conditions (4.7)–(4.9), we need an efficient procedure for determining the sign of $\Delta[i;k]$ in an arbitrary point of $[0\colon n; 0\colon n]$. Informally speaking, low (respectively, high) values of both i and k correspond to negative (respectively, positive) values of Δ. In particular, we have

$$\begin{aligned}
\Delta[0;0] &= R^{\blacksquare}_{hi}[0;0] - R^{\blacksquare}_{lo}[0;0] = 0 - \tfrac{n}{2} < 0\\
\Delta[0;n] &= R^{\blacksquare}_{hi}[0;n] - R^{\blacksquare}_{lo}[0;n] = 0 - 0 = 0\\
\Delta[n;0] &= R^{\blacksquare}_{hi}[n;0] - R^{\blacksquare}_{lo}[n;0] = 0 - 0 = 0\\
\Delta[n;n] &= R^{\blacksquare}_{hi}[n;n] - R^{\blacksquare}_{lo}[n;n] = \tfrac{n}{2} - 0 > 0
\end{aligned}$$

By the unit-nondecreasing property of Δ, there must exist a unique monotone rectilinear path through half-integer points, connecting points $\langle n^+; 0^-\rangle$ and $\langle 0^-; n^+\rangle$ (the two half-integer points just beyond the bottom-left and the top-right corners of $[0\colon n; 0\colon n]$), such that all integer points where Δ is negative (respectively, nonnegative) lie above-left (respectively, below-right) of that path. We call it the *left border path*. Symmetrically, there must also exist a unique *right border path* connecting the same endpoints, such that all integer points where Δ is nonpositive (respectively, positive) lie above-left (respectively, below-right) of that path. The integer points where Δ equals zero lie in the *border area* between the two border paths, and the half-integer points where condition (4.9) is satisfied are precisely the points where the two border paths meet (excluding their endpoints).

We now give an efficient procedure for finding both border paths. By symmetry, we only need to consider the left border path. For all integer points $[i;k]$ above-left (respectively, below-right) of this path, we have $\Delta[i;k] < 0$ (respectively, $\Delta[i;k] \geq 0$).

The left border path will be traversed in unit steps, beginning at $\langle \hat{\imath}; \hat{k}\rangle = \langle n^+; 0^-\rangle$ as the initial point. We have $\Delta[\hat{\imath}^-; \hat{k}^+] = \Delta[n;0] = 0$. Let $\langle \hat{\imath}; \hat{k}\rangle$ now denote a current point on the left border path. The sign of $\Delta[\hat{\imath}^-; \hat{k}^+]$ determines whether the next point on the path should be immediately to the right or immediately above the current point:

$$\begin{aligned}
&\langle \hat{\imath}; \hat{k}+1\rangle \qquad \text{if } \hat{\imath} = 0^- \text{ or } \Delta[\hat{\imath}^-; \hat{k}^+] < 0\\
&\langle \hat{\imath}-1; \hat{k}\rangle \qquad \text{if } \Delta[\hat{\imath}^-; \hat{k}^+] = 0
\end{aligned}$$

In each step of the path traversal, an incremental update to $\Delta = R^{\blacksquare}_{hi} - R^{\blacksquare}_{lo}$ is performed, unless $\hat{\imath} = 0^-$, by updating $R^{\blacksquare}_{hi}$, $R^{\blacksquare}_{lo}$ via Theorem 4.8, each in time $O(1)$. The computation is then repeated with the new point taking the role of the current point $\langle \hat{\imath}; \hat{k}\rangle$. The described path traversal procedure runs until it eventually reaches the final point $\langle \hat{\imath}; \hat{k}\rangle = \langle 0^-; n^+\rangle$. Note that since $\Delta[n;0] = \Delta[0;n] = 0$, the initial step of the traversal is necessarily from $\langle n^+; 0^-\rangle$ upwards to $\langle n^-; 0^-\rangle$, and the final step from $\langle 0^-; n^-\rangle$ rightwards to $\langle 0^-; n^+\rangle$.

The traversal of the left border path runs in time $O(n)$. The traversal of the right border path is symmetric and also runs in time $O(n)$.

Using the two border paths, the nonzeros of matrix R can now be obtained as follows. Given a value $d \in [-n+1:n-1]$, let $lo[d]$ (respectively, $hi[d]$) denote the unique index $\hat{\imath}$, such that point $\langle \hat{\imath}; \hat{\imath}+d\rangle$ lies exactly on the left (respectively, the right) border path. Conditions (4.7)–(4.9) can now be rewritten to say that we have $R\langle \hat{\imath}; \hat{k}\rangle = 1$, if and only if one of the following conditions holds:

$$R_{lo}\langle \hat{\imath}; \hat{k}\rangle = 1 \text{ and } \hat{\imath} \le lo[\hat{k}-\hat{\imath}] \tag{4.10}$$

$$R_{hi}\langle \hat{\imath}; \hat{k}\rangle = 1 \text{ and } \hat{\imath} \ge hi[\hat{k}-\hat{\imath}] \tag{4.11}$$

$$\hat{\imath} = lo[\hat{k}-\hat{\imath}] = hi[\hat{k}-\hat{\imath}] \tag{4.12}$$

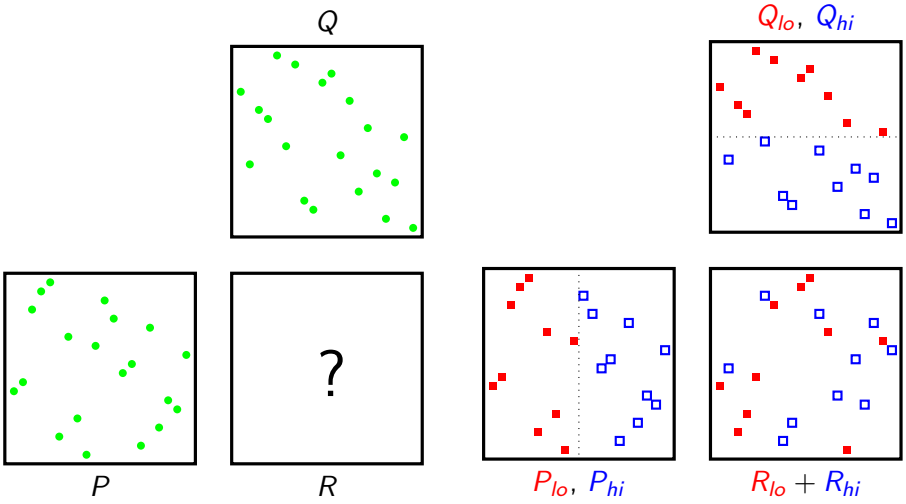
Good nonzeros of R_{lo}, R_{hi} can now be determined in time $O(n)$ by checking (4.10)–(4.11) on each nonzero in these matrices. Fresh nonzeros of R can be obtained in time $O(n)$ by checking (4.12) on each point in either of the two border paths. Alternatively, we can obtain the fresh nonzeros of R first; then, any nonzero in R_{lo}, R_{hi} that shares a row or a column index with a fresh nonzero of R can be eliminated as a bad one, and the remaining ones accepted as good ones.

We have now obtained all the nonzeros of matrix R.

(End of recursive step)

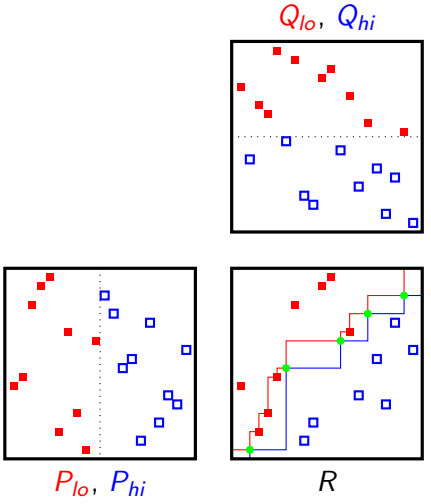
Time analysis. The recursion tree is a balanced binary tree of height $\log n$. In the root node, the computation runs in time $O(n)$. In each subsequent level, the number of nodes doubles, and the running time per node decreases by a factor of 2. Therefore, the overall running time is $O(n \log n)$.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

Informally, the conquer phase in Theorem 4.16 can be illustrated as follows. Imagine a small ant that can position itself at arbitrary real points within the square $[0:n;0:n]$. The ant has two eyes, which are on the lookout for bad nonzeros of matrices R_{lo}, R_{hi}. One eye observes the upper-left quadrant of matrix R_{hi} with the origin at the ant's current location $[i;k]$; the $R^{\blacksquare}_{hi}[i;k]$ nonzeros visible by that eye are deemed to be bad. Symmetrically, the other eye observes the lower-right quadrant of matrix R_{lo} with the origin at the ant's current location $[i;k]$; the $R^{\blacksquare}_{lo}[i;k]$ nonzeros visible by that eye are also deemed to be bad. Furthermore, the ant's view must be *balanced*: the numbers of bad nonzeros visible by each eye must be equal. This condition holds whenever $\Delta[i;k] = R^{\blacksquare}_{hi}[i;k] - R^{\blacksquare}_{lo}[i;k] = 0$, and therefore the set of locations making the ant's view balanced corresponds to the (topologically open) border area defined in the proof of Theorem 4.16. Fresh nonzeros form single-point "bridges" connecting disjoint connected components of the border area. Traversing the left border path corresponds to the ant crawling "just inside" the border area from the bottom-left to the

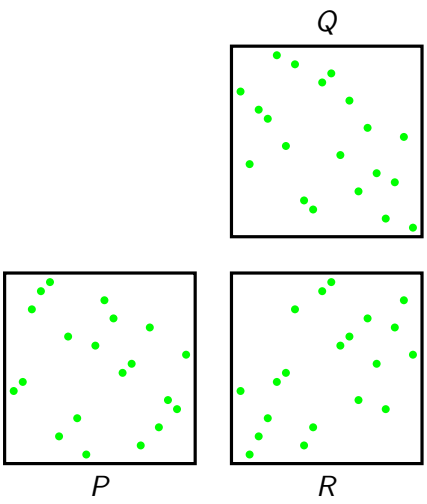


(a) Input matrices P, Q

(b) Subproblems $P_{lo} \boxdot Q_{lo} = R_{lo}$ and $P_{hi} \boxdot Q_{hi} = R_{hi}$



(c) Steady Ant's trail



(d) Output matrix R

Figure 4.3: Proof of Theorem 4.16 (the Steady Ant algorithm): $P \boxdot Q = R$

top-right corner of the square, using the fresh nonzero "bridges" to cross from a component to the adjacent one on its above-left.

Example 4.17 Figure 4.3 illustrates the proof of Theorem 4.16 (the Steady Ant algorithm) on a problem instance with a solution generated by the Wolfram Mathematica software. Subfigure 4.3a shows a pair of input 20×20 permutation matrices P, Q, with nonzeros indicated by green circles. Subfigure 4.3b shows the partitioning of the implicit 20×20 matrix distance multiplication problem into two 10×10 subproblems. The nonzeros in the two subproblems are shown respectively by filled red squares and hollow blue squares. Subfigure 4.3c shows a recursive step. The left and the right border paths are shown respectively in red and in blue. Good nonzeros of R_{lo}, good nonzeros of R_{hi} and fresh nonzeros of R are shown respectively by filled red squares, hollow blue squares, and green circles. Note that overall, there are 20 such nonzeros, and that they define a permutation matrix. This matrix is the output matrix R shown in Subfigure 4.3d. ■

4.4 Sticky braids

Classical braid group. Further understanding of the unit-Monge monoid can be gained via an algebraic formalism closely related to braid theory. Its main definition is as follows.

Definition 4.18 *The* classical braid group $\mathcal{B}_n$ *is a finitely presented group with the identity element* ι *and* $n-1$ *generators* $g_1, g_2, \ldots, g_{n-1}$*, along with their inverses* $g_1^{-1}, g_2^{-1}, \ldots, g_{n-1}^{-1}$*. The (monoid) presentation of group* $\mathcal{B}_n$ *consists of the* inversion relations

$$g_i g_i^{-1} = \iota \qquad i \in [1\!:n-1] \tag{4.13}$$

the far commutativity relations

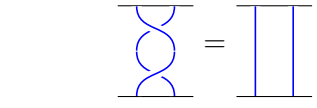
$$g_i g_j = g_j g_i \qquad i, j \in [1\!:n-1], j - i \geq 2 \tag{4.14}$$

and the braid relations

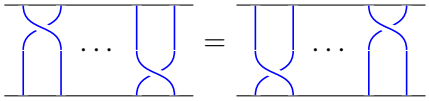
$$g_i g_j g_i = g_j g_i g_j \qquad i, j \in [1\!:n-1], j - i = 1 \tag{4.15}$$

□

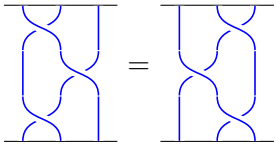
An element of $\mathcal{B}_n$ can be represented geometrically. Given a natural number n, we draw two parallel horizontal lines in the Euclidean plane, and select a set of n points, called *nodes*, on each line. The two node sets can be put into one-to-one correspondence by connecting them pairwise, in some order, with continuous monotone curves called *strands*. (Here, a curve is called



(a) Inversion relations (4.13)



(b) Far commutativity relations (4.14)



(c) Braid relations (4.15)

Figure 4.4: Defining relations of the classical braid group

monotone, if its vertical projection is always directed downwards.) We call the resulting configuration a *classical braid* of order n.

The strands' crossings correspond to the group's generators and their inverses, and are therefore signed; every crossing is represented as a "bridge", with the crossing's sign indicated by one of the strands going on top of the other. We will assume that all crossings are between exactly two strands, i.e. three or more strands can never meet at a single point. In a classical braid, a given pair of strands may cross an arbitrary number of times, with crossings of either sign.

Two classical braids of the same order can be *multiplied*. The product braid is obtained as follows. First, we draw one braid above the other, identifying the bottom nodes of the top braid with the top nodes of the bottom braid. Then, we join up each pair of strands that became incident in the previous step.

Group $\mathcal{B}_n$ is represented by classical braids of order n as follows. Group multiplication (i.e. concatenation of words in the generators) corresponds to multiplication of classical braids. The identity element ι corresponds to a classical braid where the top nodes are connected to the bottom nodes consistently in the left-to-right order, without any crossings. Each of the remaining generators g_i corresponds to an *elementary crossing*, i.e. to a classical braid where the only crossing is between a pair of neighbouring strands in half-integer positions i^- and i^+, the former going on top of the latter. Figure 4.4 shows the defining relations of the classical braid group (4.13)–(4.15) in terms of classical braids.

A classical braid can be associated with a permutation matrix $P\langle 0\colon n; 0\colon n\rangle$

by *canonical projection*: the top and the bottom nodes of the braid, ordered from left to right, correspond respectively to the row and the column indices of the matrix; a strand connecting top node $\hat{\imath}$ and bottom node $\hat{\jmath}$ in the braid corresponds to a nonzero $P\langle\hat{\imath};\hat{\jmath}\rangle = 1$ in the matrix. The correspondence between a classical braid and a permutation matrix obtained from it by canonical projection is not one-to-one: different braids (in fact, an infinite number of braids) may correspond to the same permutation matrix. However, this correspondence is preserved by braid multiplication: a canonical projection of classical braid product is equal to the (ordinary) matrix product of the canonical projections of the multiplicand braids.

We refer the reader to [129] for further background on braid theory.

Sticky braid monoid. Normally, the inversion relations are not written down explicitly as part of a group's presentation, since they hold in all groups by definition. However, we have defined group $\mathcal{B}_n$ as a monoid, including explicitly its inversion relations in Definition 4.18, in order to contrast it with another important monoid.

Definition 4.19 *The* sticky braid monoid[2] $\mathcal{T}_n$ *is a finitely presented monoid with the identity element* ι *and* $n-1$ *generators* g_1, g_2, …, g_{n-1}. *The presentation of monoid* $\mathcal{T}_n$ *consists of the* idempotence relations

$$g_i^2 = g_i \qquad i \in [1\colon n-1] \tag{4.16}$$

the far commutativity relations

$$g_i g_j = g_j g_i \qquad i,j \in [1\colon n-1], j-i \geq 2 \tag{4.17}$$

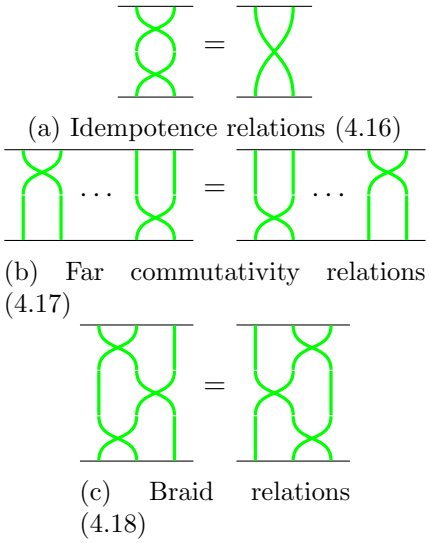
and the braid relations

$$g_i g_j g_i = g_j g_i g_j \qquad i,j \in [1\colon n-1], j-i = 1 \tag{4.18}$$

□

Traditionally, monoid $\mathcal{T}_n$ is known as the 0-*Hecke monoid of the symmetric group* $H_0(\mathcal{S}_n)$, the *Hecke monoid*, or the *Richardson–Springer monoid*. Its elements are sometimes known as *Hecke words*, and its multiplication as *Demazure multiplication* (for details, see e.g. Denton et al. [76], Mazorchuk and Steinberg [158], Deng et al. [75], Gorsky [106]).

[2]Notation $\mathcal{T}_n$ that we are using for the sticky braid monoid has already been used in **??** for the sticky multiplication monoid of permutation matrices (which is isomorphic to the unit-Monge monoid). We will see shortly that this seeming abuse of notation is well-justified.



(a) Idempotence relations (4.16)

(b) Far commutativity relations (4.17)

(c) Braid relations (4.18)

Figure 4.5: Defining relations of the sticky braid monoid

Just as with the classical braid group, an element of $\mathcal{T}_n$ can be represented geometrically by a *sticky braid*[3] of order n. There is remarkable similarity between classical and sticky braids. However, one crucial difference is that all strand crossings are now unsigned, and drawn as "crossroads", rather than "bridges", passing directly through one another. Just as in a classical braid, a given pair of strands in a sticky braid may cross an arbitrary number of times.

Definition 4.20 *A sticky braid is called* reduced*, if every pair of its strands cross at most once (i.e. either once, or not at all).* □

Two sticky braids of the same order can be *multiplied* in the same way as classical braids. Note that, even if both multiplicand sticky braids are reduced, their product may not in general be reduced.

Monoid $\mathcal{T}_n$ is represented by sticky braids of order n in a way similar to representing group $\mathcal{B}_n$ by classical braids. Figure 4.5 shows the defining relations of the sticky braid monoid (4.16)–(4.18) in terms of sticky braids.

Example 4.21 In Figure 4.2, Subfigure 4.2b shows three different sticky braids. The sticky braid on the left-hand side is a product of two reduced sticky braids. This product braid has some pairs of strands crossing twice, hence it is not reduced. ■

[3] A tongue-in-cheek justification for this term is that these braids are like ordinary braids, except for their "stickiness": due to the idempotence property, a pair of strands that cross any number of times can only be partially untangled by removing all crossings except one; this last crossing stays there forever and cannot be removed. In earlier versions of this work, the strands of a sticky braid were called "seaweeds", a term coined by Yu. V. Matiyasevich during a presentation once given by the author.

Definition 4.22 *Two sticky braids of order n are called* equivalent*, if they represent the same element of the sticky braid monoid $\mathcal{T}_n$.* □

Theorem 4.23 *Every sticky braid has an equivalent reduced sticky braid.* □

Proof Given a sticky braid, we *comb* the strands by running through all their crossings, respecting the top-to-bottom partial order of the crossings. For each crossing, we check whether the two crossing strands have previously crossed above the current point. If this is the case, then we undo the current crossing by removing it from the braid and patching it with non-crossing strand pieces. The correctness of this combing procedure is easy to prove by relations (4.16)–(4.18) of Definition 4.19. When all the double crossings have been combed away, we are left a reduced sticky braid. ■

Example 4.24 Recall from Example 4.21 that in Subfigure 4.2b, the product braid on the left-hand side is unreduced. Combing this product braid results in a reduced sticky braid shown in the middle. Another equivalent reduced sticky braid in shown in the right-hand side. ■

A sticky braid can be associated with a permutation matrix $P\langle 0{:}\,n; 0{:}\,n\rangle$ by *canonical projection* in the same way as a classical braid. As with classical braids, the correspondence between a sticky braid and a permutation matrix obtained from it by canonical projection is not one-to-one: different braids, even if they are reduced, may correspond to the same permutation matrix. However, it turns out that all the reduced braids corresponding to the same permutation matrix are equivalent. We formalise this observation by the following lemma.

Lemma 4.25 *The sticky braid monoid $\mathcal{T}_n$ consists of at most $n!$ distinct elements.* □

Proof It is straightforward to see that any sticky braid can be transformed into a reduced one, using relations (4.16)–(4.18). Then, any two reduced sticky braids corresponding to the same permutation can be transformed into one another, using far commutativity relations (4.17) and braid relations (4.18). Therefore, each permutation corresponds to a single element of $\mathcal{T}_n$. This mapping is surjective, therefore the number of elements in $\mathcal{T}_n$ is at most the total number of permutations $n!$. ■

Recall that for classical braids, canonical projection is preserved by braid multiplication: a canonical projection of classical braid product is equal to the (ordinary) matrix product of the canonical projections of the multiplicand braids. We are now able to obtain an analogue of this property for sticky braids, replacing the ordinary product of permutation matrices with their sticky product.

	idem?	f/c?	br?
sticky braid monoid	yes	yes	yes
positive braid monoid (e.g. [129, Section 6.5])	no	yes	yes
locally free idempotent monoid [214] (also [85])	yes	yes	no
nil-Hecke monoid [106]	$g_i^2 = 0$	yes	yes
symmetric group (e.g. [186, Section 2.12])	$g_i^2 = 1$	yes	yes
classical braid group (e.g. [129, Section 1.1])	$g_i g_i^{-1} = 1$	yes	yes

Table 4.1: Some structures related to the sticky braid monoid ("idem": idempotence; "f/c": far commutativity; "br": braid relations)

Theorem 4.26 *For a fixed n, the sticky multiplication monoid of permutation matrices is isomorphic to the sticky braid monoid (hence we are justified in denoting both monoids identically by $\mathcal{T}_n$).* □

Proof We have a straightforward bijection between the generators of both monoids: a transposition matrix G_t corresponds to a generator g_t of the sticky braid monoid. It is straightforward to check that relations (4.16)–(4.18) are verified by matrices G_t, therefore the bijection on the generators defines a homomorphism from the sticky braid monoid to the sticky multiplication monoid of permutation matrices. By Lemma 4.14, this homomorphism is surjective, hence the cardinality of the sticky braid monoid is at least the number of all permutation matrices of size n, equal to $n!$. However, by Lemma 4.25, the cardinality of the sticky braid monoid is at most $n!$. Thus, the cardinality of the sticky braid monoid is exactly $n!$, and the two monoids are isomorphic. ■

Example 4.27 In Figure 4.2, the sticky braids shown in Subfigure 4.2b correspond to the sticky matrix product $P \boxdot Q = R$ in Subfigure 4.2a. ■

Confluent rewriting A traditional approach to computation in semigroups and monoids is via *confluent rewriting systems* (see e.g. [194]). For example, Definition 4.19 yields the following confluent rewriting system for $\mathcal{T}_4$, where the generators are denoted by a, b, c:

$$aa \to a \quad bb \to b \quad cc \to c \quad ca \to ac$$
$$bab \to aba \quad cbc \to bcb \quad cbac \to bcba \quad abacba \to 0$$

In general, such a rewriting system can be used to perform multiplication on monoid elements, represented as words in the generators of $\mathcal{T}_n$ for any n. However, this approach lacks any strong efficiency guarantees. In contrast, our method presented in this chapter allows multiplication in $\mathcal{T}_n$ in time $O(n \log n)$ on monoid elements represented as permutations of size n.

Variations on the sticky braid monoid. The sticky braid monoid is closely related to a number of other well-known algebraic structures, that can be obtained by removing or replacing some of the relations (4.16)–(4.18) in Definition 4.19. These structures are listed in Table 4.1. A generalisation of the sticky braid monoid is given by 0-Hecke monoids of general Coxeter groups, also known as *Coxeter monoids*. These monoids arise naturally as subgroup monoids in groups. The theory of Coxeter monoids can be traced back to Bourbaki [40], and was subsequently developed in [210, 179, 91, 42, 130]. A further generalisation to *$\mathcal{J}$-trivial monoids* has been studied by Denton et al. [76]. The contents of this chapter can be regarded as a first step in the algorithmic study of such general classes of monoids.

4.5 Bruhat order

Given a permutation, it is natural to ask how well-sorted it is. In particular, a permutation may either be fully sorted (the identity permutation), or fully anti-sorted (the reverse identity permutation), or anything in between. More generally, given two permutations, it is natural to ask whether, in some sense, one is "more sorted" than the other.

Let P, Q be permutation matrices over index range $\langle 0{:}\,n; 0{:}\,n\rangle$. A classical "degree-of-sortedness" comparison is given by the following partial order (see e.g. Bóna [39], Hammett and Pittel [110], and references therein).

Definition 4.28 *Matrix P is lower than matrix Q in the* Bruhat order*, $P \preceq Q$, if P can be transformed to Q by a sequence of* anti-sorting *steps. Each such step substitutes a (not necessarily contiguous) submatrix of the form $\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$ by a submatrix of the form $\left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right]$.* □

Informally, $P \preceq Q$, if P defines a "more sorted" permutation than Q. More precisely, $P \preceq Q$, if the permutation defined by P can be transformed into the one defined by Q by successive pairwise anti-sorting between arbitrary pairs of elements. Symmetrically, the permutation defined by Q can be transformed into the one defined by P by successive pairwise sorting (or, equivalently, by an application of a comparison network; see e.g. Knuth [135]).

Bruhat order is an important group-theoretic concept, which can be generalised to arbitrary Coxeter groups (see Björner and Brenti [37], Denton et al. [76] for more details and further references).

Many equivalent definitions of the Bruhat order on permutations are known; see e.g. [82, 174, 108, 37, 80, 221, 124] and references therein. A classical combinatorial characterisation of the Bruhat order, known as *Ehresmann's criterion* or *dot criterion*, is as follows.

Theorem 4.29 *We have $P \preceq Q$, if and only if $P^{\Sigma} \leq Q^{\Sigma}$ elementwise.* □

PROOF Straightforward from the definitions; see [37]. ∎

Example 4.30 We have

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^{\Sigma} = \begin{bmatrix}0&1&2&3\\0&0&1&2\\0&0&1&1\\0&0&0&0\end{bmatrix} \leq \begin{bmatrix}0&1&2&3\\0&1&2&2\\0&0&1&1\\0&0&0&0\end{bmatrix} = \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}^{\Sigma}$$

where the inequality is elementwise. Hence, the permutation matrix on the left is dominated by the one on the right in the Bruhat order. Note that the right-hand matrix can be obtained from the left-hand one by anti-sorting the 2×2 submatrix at the intersection of the top two rows with the leftmost and rightmost columns.

We also have

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^{\Sigma} = \begin{bmatrix}0&1&2&3\\0&0&1&2\\0&0&1&1\\0&0&0&0\end{bmatrix} \not\leq \begin{bmatrix}0&1&2&3\\0&1&1&2\\0&0&0&1\\0&0&0&0\end{bmatrix} = \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}^{\Sigma}$$

Hence, the permutation matrix on the left is not dominated by the one on the right in the Bruhat order. It is also easy to check exhaustively that the right-hand matrix cannot be obtained from the left-hand one by anti-sorting any of its 2×2 submatrices. ∎

Theorem 4.29 immediately gives one an algorithm for deciding whether two permutations are Bruhat-comparable in time $O(n^2)$. To the author's knowledge, no asymptotically faster algorithm for deciding Bruhat comparability has been known so far.

To demonstrate an application of our techniques, we now give a new characterisation of the Bruhat order in terms of the unit-Monge monoid (or, equivalently, the sticky braid monoid). This characterisation will give us a substantially faster algorithm for deciding Bruhat comparability.

Intuitively, the connection between the Bruhat order and sticky braids is as follows. Consider matrix P and the rotated matrix P^{R}. The matrix rotation induces a one-to-one correspondence between the nonzeros of P^{R} and P, and therefore also between individual strands in their reduced sticky braids. A pair of strands cross in a reduced braid of P^{R}, if and only if the corresponding pair of strands do not cross in a reduced braid of P. Now consider the product braid $P^{\mathsf{R}} \boxdot P$, where each strand is made up of two corresponding strands from P^{R} and P. Every pair of strands in braid $P^{\mathsf{R}} \boxdot P$ either cross in the top subbraid P^{R}, or in the bottom subbraid P, but not in both. Therefore, the product braid is a reduced sticky braid, in which every pair of strands cross exactly once. Thus, we have $P^{\mathsf{R}} \boxdot P = \mathsf{I}^{\mathsf{R}}$.

Now suppose $P \preceq Q$. By Theorem 4.29, we have $P^{\Sigma} \le Q^{\Sigma}$ elementwise. Therefore, by **??**, $P^{\mathsf{R}\Sigma} \odot P^{\Sigma} \le P^{\mathsf{R}\Sigma} \odot Q^{\Sigma}$ elementwise, hence by Theorem 4.29, we have $P^{\mathsf{R}} \boxdot P \preceq P^{\mathsf{R}} \boxdot Q$. However, as argued above, $P^{\mathsf{R}} \boxdot P = \mathsf{I}^{\mathsf{R}}$, which is the highest possible permutation matrix in the Bruhat order, corresponding to the reverse identity permutation. Therefore, $P^{\mathsf{R}} \boxdot Q = \mathsf{I}^{\mathsf{R}}$. We thus have a necessary condition for $P \preceq Q$. It turns out that this condition is also sufficient, giving us a new, computationally efficient criterion for Bruhat comparability.

Theorem 4.31 *We have $P \preceq Q$, if and only if $P^{\mathsf{R}} \boxdot Q = \mathsf{I}^{\mathsf{R}}$.* □

Proof Let $i, j \in [0\mathinner{:} n]$. We have

$$\begin{aligned}
&P^{\mathsf{R}\Sigma}[i;j] + Q^{\Sigma}[j;n-i] = && \text{(definition of } \mathsf{R}\text{)}\\
&\big(n-i-P^{\Sigma}[j;n-i]\big) + Q^{\Sigma}[j;n-i] = && \text{(term rearrangement)}\\
&\big(Q^{\Sigma}[j;n-i] - P^{\Sigma}[j;n-i]\big) + n - i && (4.19)
\end{aligned}$$

We now prove the implication separately in each direction.

Necessity. Let $P \preceq Q$. By (4.19) and Theorem 4.29, we have

$$P^{\mathsf{R}\Sigma}[i;j] + Q^{\Sigma}[j;n-i] \ge n - i$$

This lower bound is attained at $j = 0$ (and, symmetrically, $j = n$): we have $P^{\mathsf{R}\Sigma}[i;0] + Q^{\Sigma}[0;n-i] = 0 + (n-i) = n - i$. Therefore,

$$\begin{aligned}
&(P^{\mathsf{R}} \boxdot Q)^{\Sigma}[i;n-i] = && \text{(definition of } \boxdot\text{)}\\
&\min_j \big(P^{\mathsf{R}\Sigma}[i;j] + Q^{\Sigma}[j;n-i]\big) = n-i && \text{(attained at } j = 0\text{)}
\end{aligned}$$

It is now easy to prove (e.g. by induction on n) that $P^{\mathsf{R}} \boxdot Q = \mathsf{I}^{\mathsf{R}}$ is the only permutation matrix satisfying the above equation for all i.

Sufficiency. Let $P^{\mathsf{R}} \boxdot Q = \mathsf{I}^{\mathsf{R}}$. By **??**, we have

$$\min_j \big(P^{\mathsf{R}\Sigma}[i;j] + Q^{\Sigma}[j;n-i]\big) = \mathsf{I}^{\mathsf{R}\Sigma}[i;n-i] = n - i$$

for all i. Therefore, for all i, j, $P^{\mathsf{R}\Sigma}[i;j] + Q^{\Sigma}[j;n-i] \ge n - i$. By (4.19), this is equivalent to $Q^{\Sigma}[j;n-i] - P^{\Sigma}[j;n-i] \ge 0$, therefore $P^{\Sigma}[j;n-i] \le Q^{\Sigma}[j;n-i]$, hence by Theorem 4.29, we have $P \preceq Q$. ∎

The combination of Theorems 4.16 and 4.31 gives us a fast algorithm for deciding Bruhat comparability of permutations.

Theorem 4.32 *Given permutation matrices P, Q, it is possible to determine whether $P \preceq Q$ in time $O(n \log n)$.* □

Proof Immediately from Theorems 4.16 and 4.31. ∎

Gawrychowski [96] has subsequently improved the algorithm of Theorems 4.31 and 4.32 to run in time $O\big(\frac{n \log n}{\log \log n}\big)$.

Chapter 5

Longest common subsequence problems

5.1 Longest common subsequence (LCS)

Longest common subsequence. Unless indicated otherwise, a string comparison problem will take strings $a\langle 0\colon m\rangle$ and $b\langle 0\colon n\rangle$ as input.

Definition 5.1 *Let a, b be strings. Their* longest common subsequence (LCS) score*, denoted $\mathsf{lcs}(a, b)$, is the length of the longest string that is a subsequence of both a and b. Given strings a, b, the* LCS problem *asks for their LCS score.* □

Example 5.2 Let

$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

This example, borrowed from Alves et al. [14], will serve as a running example for this chapter and the next. String b of length 13 contains the whole string a as a subsequence, therefore we have

$$\mathsf{lcs}(a, b) = \mathit{length}(a) = 8$$

The LCS score of string a against substring $b\langle 4\colon 11\rangle =$ "CABCABA" is realised by a common subsequence "ABCBA", therefore we have

$$\mathsf{lcs}(a, b\langle 4\colon 11\rangle) = \mathit{length}(\text{"ABCBA"}) = 5$$ ■

LCS grid. The LCS problem can be represented by a graph as follows.

Definition 5.3 *A* triangulated grid $G[0\colon m; 0\colon n]$ *is a directed acyclic graph, defined on the set of nodes $[0\colon m; 0\colon n]$. For all $i \in [0\colon m]$, $\hat{\imath} \in \langle 0\colon m\rangle$, $j \in [0\colon n]$, $\hat{\jmath} \in \langle 0\colon n\rangle$, the triangulated grid contains:*

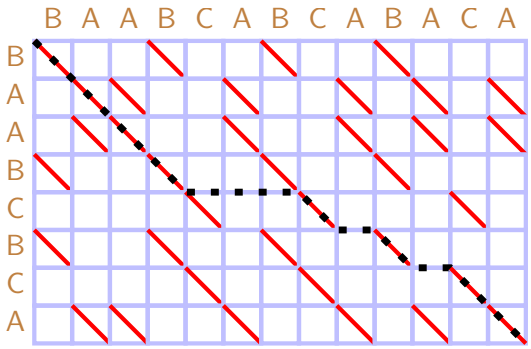


Figure 5.1: LCS grid $\mathsf{G}_{a,b}$; highest-scoring path for $lcs(a,b) = 8$

- *the horizontal edge* $[i;\hat{\jmath}^-] \to [i;\hat{\jmath}^+]$*;*
- *the vertical edge* $[\hat{\imath}^-;j] \to [\hat{\imath}^+;j]$*;*
- *the diagonal edge* $[\hat{\imath}^-;\hat{\jmath}^-] \to [\hat{\imath}^+;\hat{\jmath}^+]$*.* □

We can also view such a triangulated grid as an $m \times n$ grid of *cells*, each induced by four neigbouring nodes. A cell contains a pair of vertical edges, a pair of horizontal edges, and a diagonal edge.

Triangulated grids representing string comparison will have weights on the edges. These weights will be called *scores*. The score of a path is defined as the sum of the scores along its edges.

Definition 5.4 *The* LCS grid *for strings a, b is a weighted triangulated grid $\mathsf{G}_{a,b}[0\colon m; 0\colon n]$, defined as follows. A cell induced by nodes $[\hat{\imath}^{\pm};\hat{\jmath}^{\pm}]$, $\hat{\imath} \in \langle 0\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$, is called*

- *a* match cell*, if $a\langle\hat{\imath}\rangle$ matches $b\langle\hat{\jmath}\rangle$;*
- *a* mismatch cell*, otherwise*

An edge is called

- *a* match edge*, if it is a diagonal edge in a match cell;*
- *a* mismatch edge*, if it is a diagonal edge in a mismatch cell;*
- *a* gap edge*, if it is a horizontal or vertical edge.*

In the LCS grid, match edges have score 1*; mismatch and gap edges have score* 0*.* □

Clearly, mismatch edges do not affect maximum path scores between any pair given endpoints, and can therefore be ignored.

Example 5.5 Figures 5.1 and 5.2 show the LCS grid for strings $a =$ "BAABCBCA", $b =$ "BAABCABCABACA". All edges are directed left-to-right and top-to-bottom. Match and gap edges are coloured red and blue, respectively; mismatch edges are not shown. ∎

A particular special case of an LCS grid is the *full-mismatch grid*, which consists entirely of mismatch cells. This grid can be obtained as the LCS grid of a pair strings that have no characters in common. Another special case is the *full-match grid*, which consists entirely of match cells. This grid can be obtained as the LCS grid of a pair of strings over an alphabet of a single character, or, alternatively, of a pair of strings, one of which consists entirely of wildcard characters.

Given a pair of strings a, b, their common subsequences correspond to paths in the LCS grid $\mathsf{G}_{a,b}$ from node $[0;0]$ to node $[m;n]$. The length of a subsequence is equal to the total score of the corresponding path. The LCS problem is therefore equivalent to finding the maximum path score:

$$\mathsf{lcs}(a,b) = \max \mathsf{score}([0;0] \rightsquigarrow [m;n]) \tag{5.1}$$

where the score maximum is taken across all paths connecting the given endpoints in $\mathsf{G}_{a,b}$; we adopt this notation for path score maximisation from here onwards.

Example 5.6 In Figure 5.1, the highlighted path corresponds to $\mathsf{lcs}\big(a,b\big) = 8$. ∎

Finding the maximum path score between the given endpoints in the LCS grid is equivalent to finding the corresponding distances in an undirected graph, obtained from the LCS grid 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. The problem thus becomes a special case of the single-source, single-destination shortest path problem in an undirected planar graph.

Prefix LCS. The classical approach to the LCS problem solves in fact the following more general problem.

Definition 5.7 *Given strings a, b, the* prefix LCS problem *asks for the LCS score of every prefix of a against every prefix of b. In terms of the LCS grid, the problem asks for the* prefix LCS matrix $\mathsf{L}_{a,b}[0\colon m; 0\colon n]$ *of single-source maximum scores*

$$\mathsf{L}_{a,b}[i;j] = \mathsf{lcs}\big(a\langle\colon i\rangle, b\langle\colon j\rangle\big) = \max \mathsf{score}([0;0] \rightsquigarrow [i;j])$$

where $i \in [0\colon m]$, $j \in [0\colon n]$. □

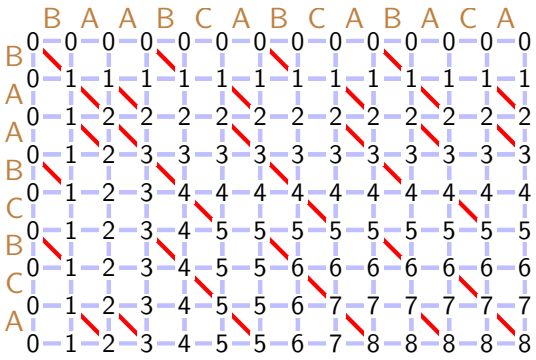


Figure 5.2: LCS grid $\mathsf{G}_{a,b}$; execution of Algorithm 5.8 (Prefix LCS)

Algorithms for the prefix LCS problem were discovered independently by Needleman and Wunsch [167] (without an explicit analysis), and by Wagner and Fischer [215] (see also [121]).

Algorithm 5.8 (Prefix LCS by classical dynamic programming)

Input: strings a, b.

Output: prefix LCS matrix $\mathsf{L}_{a,b}$.

Description. We construct matrix L incrementally, iterating through the cells of the LCS grid $\mathsf{G}_{a,b}$. We initialise

$$L[0;j] \leftarrow 0 \qquad L[l;0] \leftarrow 0$$

for all $l \in [0\colon m]$, $j \in [0\colon n]$. We then iterate through the cells of $\mathsf{G}_{a,b}$ for all $\hat{l} \in \langle 0\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$ in lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the cells. In each iteration, we assign

$$L[\hat{l}^+;\hat{\jmath}^+] \leftarrow \max \begin{cases} L[\hat{l}^-;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ L[\hat{l}^-;\hat{\jmath}^+] \\ L[\hat{l}^+;\hat{\jmath}^-] \end{cases}$$

The final state of the matrix is returned as the algorithm's output: $\mathsf{L}_{a,b} \leftarrow L$. ■

Theorem 5.9 *The prefix LCS problem can be solved in time $O(mn)$.* □

Proof In Algorithm 5.8, each iteration runs in constant time. The overall running time is $mn \cdot O(1) = O(mn)$. ■

Example 5.10 Figure 5.2 shows the execution trace of Algorithm 5.8 (Prefix LCS) on the LCS grid $\mathsf{G}_{a,b}$ for strings $a =$ "BAABCBCA", $b =$ "BAABCABCABACA". ■

The solution to the prefix LCS problem can be used to *trace back* (i.e. to obtain character by character) the actual LCS of strings a and b in time proportional to the size of the output (i.e. the length of the output subsequence). A memory-saving recomputation technique by Hirschberg [111] can be applied to achieve LCS traceback in the same asymptotic time, but in a linear amount of memory.

Running time optimality. Assuming an unordered alphabet that only allows equality testing between characters, Aho et al. [5] gave a lower bound of $\Omega(mn)$ on the running time for the LCS problem (see also a survey by Bergroth et al. [31]). This lower bound is matched by Algorithm 5.8 (Prefix LCS by classical dynamic programming), which is therefore asymptotically optimal for an unordered alphabet.

Under a stronger model of a totally ordered alphabet allowing relative order comparison between characters, Abboud et al. [2] and Bringmann and Künnemann [41] gave a lower bound of $\omega(n^{2-\epsilon})$ for all $\epsilon > 0$, $m = n$, assuming the Strong Exponential Time Hypothesis. The exponent 2 is matched by Algorithm 5.8 (Prefix LCS by classical dynamic programming); therefore, this exponent is optimal for a totally ordered alphabet. This does not preclude speeding up the algorithm by polylogarithmic factors; indeed, such speedups have been obtained in [155, 67, 218, 35].

5.2 Semi-local LCS

The comparison of strings by a single similarity score, such as the LCS score, is usually termed global. A more detailed local string comparison is achieved by obtaining a similarity score between pairs of substrings within the input strings (either all substring pairs, or a subset of the pairs that is considered relevant for a particular application). Local comparison is particularly useful for biological applications; we will consider it in Chapter 11.

Although global comparison (whole string against whole string) and fully-local comparison (all substrings against all substrings in each string) are the two most common approaches to comparing strings, another important type of string comparison lies "in between": comparing one whole string against all substrings of the other (as, for example, in classical pattern matching, where the former string is the pattern, and the latter string is the text). This type of comparison turns out to be fundamental for developing efficient and flexible string comparison algorithms. A key step in development of such algorithms is to consider the following definition, that generalises string-substring comparison.

Definition 5.11 *Given strings a, b, the* semi-local LCS problem *asks for the LCS scores of*

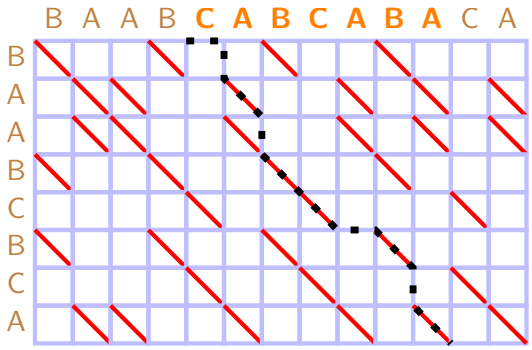


Figure 5.3: LCS grid $\mathsf{G}_{a,b}$; highest-scoring path for $lcs(a, b\langle 4\colon 11\rangle) = 5$

- *the whole a against every substring of b* (string-substring LCS)*;*
- *every prefix of a against every suffix of b* (prefix-suffix LCS)*;*
- *every suffix of a against every prefix of b* (suffix-prefix LCS)*;*
- *every substring of a against the whole b* (substring-string LCS)*.*

In terms of the LCS grid $\mathsf{G}_{a,b}$, the problem asks for boundary-to-boundary maximal scores

$$\mathsf{lcs}\big(a, b\langle i\colon j\rangle\big) = \max \mathsf{score}([0; i] \rightsquigarrow [m; j]) \tag{5.2}$$
$$\mathsf{lcs}\big(a\langle k\colon m\rangle, b\langle 0\colon j\rangle\big) = \max \mathsf{score}([k; 0] \rightsquigarrow [m; j]) \tag{5.3}$$
$$\mathsf{lcs}\big(a\langle 0\colon l\rangle, b\langle i\colon n\rangle\big) = \max \mathsf{score}([0; i] \rightsquigarrow [l; n]) \tag{5.4}$$
$$\mathsf{lcs}\big(a\langle k\colon l\rangle, b\big) = \max \mathsf{score}([k; 0] \rightsquigarrow [l; n]) \tag{5.5}$$

where $k, l \in [0\colon m]$, $i, j \in [0\colon n]$. □

Example 5.12 In Figure 5.3, the highlighted top-to-bottom path corresponds to the string-substring $\mathsf{lcs}\big(a, b\langle 4\colon 11\rangle\big) = \mathsf{lcs}(a, \text{“CABCABA”}) = 5$. ■

Note that Definition 5.11 is symmetric with respect to exchanging the two strings, and also with respect to the left and the right directions within each of the strings. There is a further, more subtle symmetry between comparing strings against substrings on one hand, and prefixes against suffixes on the other; this will be treated in detail in subsequent sections. The essence of our approach lies in preserving and exploiting such symmetries.

Some alternative terms for semi-local comparison, used especially in biological texts, are “end-free alignment” [121], [**?**, Subsection 11.6.4] or “semi-global alignment” [121], [125, Problem 6.24], [103, Section 8.4]. The string-substring (and its symmetric substring-string) component of semi-local string comparison is also called “fitting alignment” [125, Problem 6.23].

String-substring LCS is an important problem in its own right, closely related to approximate pattern matching, where a short fixed pattern string is compared to various substrings of a long text string. We will consider approximate pattern matching in Chapter 9. The prefix-suffix (and the symmetric suffix-prefix) LCS problem, sometimes called "overlap alignment" [125, Problem 6.22], [103, Section 8.4], also occurs independently in some applications.

Often, string comparison problems ask for 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 5.11 asks for all the locally optimal comparison scores. This approach is more flexible, and will be useful for various algorithmic applications described later in this work.

Semi-local LCS matrix. The analysis of boundary-to-boundary paths in an LCS grid can be simplified by padding string b with wildcard characters on both sides. Let $b^{pad}\langle -m\colon m+n\rangle = ?^m\underline{b}?^m$. Semi-local common subsequences correspond to top-to-bottom paths in the $m\times(2m+n)$ padded LCS grid $\mathsf{G}_{a,b^{pad}}[0\colon m; -m\colon m+n]$.

Definition 5.13 *The* semi-local LCS matrix[1] *for strings a, b is a matrix $\mathsf{H}_{a,b}[-m\colon n; 0\colon m+n]$, defined by*

$$\mathsf{H}_{a,b}[i;j] = \begin{cases} \mathsf{lcs}(a, b^{pad}\langle i\colon j\rangle) = \max \mathsf{score}([0;i] \rightsquigarrow [m;j]) & \text{if } i \le j \\ j-i & \text{if } i \ge j \end{cases}$$

where $i \in [-m\colon n]$, $j \in [0\colon m+n]$. In particular, we have $\mathsf{H}_{a,b}[i;j] = 0$ if $i = j$. □

We will abbreviate "semi-local LCS matrix" to just *LCS matrix*, where justified by the context.

The solution for each of the four components of the semi-local LCS problem in Definition 5.11 can now be obtained from (5.2)–(5.5) as follows:

$$\begin{aligned} \mathsf{lcs}\big(a, b\langle i\colon j\rangle\big) &= \mathsf{H}_{a,b}[i+m; j] \\ \mathsf{lcs}\big(a\langle k\colon m\rangle, b\langle 0\colon j\rangle\big) &= \mathsf{H}_{a,b}[-k+m; j] - k \\ \mathsf{lcs}\big(a\langle 0\colon l\rangle, b\langle i\colon n\rangle\big) &= \mathsf{H}_{a,b}[i+m; m+n-l] + l - m \\ \mathsf{lcs}\big(a\langle k\colon l\rangle, b\big) &= \mathsf{H}_{a,b}[-k+m; m+n-l] - k + l - m \end{aligned}$$

where $k, l \in [0\colon m]$, $i, j \in [0\colon n]$.

Example 5.14 Figure 5.4 shows matrix $\mathsf{H}_{a,b}$. The entry $\mathsf{H}_{a,b}[4+8; 11] = \mathsf{H}_{a,b}[12; 11] = 5$ is circled. ■

[1] These matrices are called "DIST matrices" e.g. in [190, 66]. Our terminology is chosen to reflect the semi-local score-maximising nature of the matrix elements. These matrices should not be confused with pairwise substitution score matrices used in comparative genomics (see e.g. [125]).

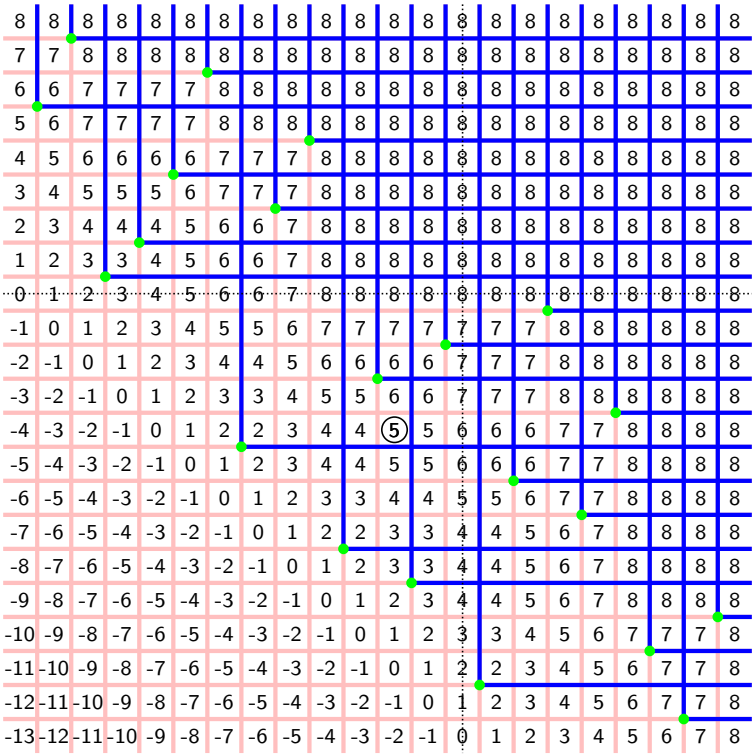


Figure 5.4: LCS matrix $\mathsf{H}_{a,b}$; LCS kernel $\mathsf{P}_{a,b}$

Planar graph-based algorithms. As described by Definition 5.11, the semi-local LCS problem is equivalent to the problem of obtaining maximum boundary-to-boundary path scores in a triangulated grid with zero-one weights. This problem is itself equivalent to finding boundary-to-boundary distances in a weighted graph (which may be assumed either directed, or undirected), which is obtained from the LCS grid by assigning length 1 to vertical and horizontal edges, 0 to diagonal match edges, and 2 to diagonal mismatch edges.

For reference, we introduce the necessary background on finding maximum distances in general weighted planar graphs.

Definition 5.15 *Let G be a directed plane graph with nonnegative edge lengths. The* multiple-source shortest paths (MSSP) *problem asks for distances from every node on the external boundary of G to every node of G.*□

In the case of a triangulated $m \times n$ grid with arbitrary real weights, where $m \leq n$, the MSSP problem was studied by Schmidt [190], who gave an algorithm for constructing an MSSP oracle in time $O(mn \log m)$, such that an individual distance can be queried in time $O(\log m)$. This implies a semi-local LCS algorithm, running in time $O(mn \log m)$. Alves et al. [14] improved on this running time, obtaining an algorithm for the string-substring

LCS problem, running in time $O(mn)$.

In the case of an arbitrary real-weighted directed plane graph with N nodes, the MSSP problem was studied by Klein [133].

Theorem 5.16 *An MSSP oracle with query time $O(\log N)$ can be obtained in time $O(N \log N)$.* □

PROOF See [190] for grids, and [133] for general planar graphs. ■

For undirected graphs with constant integer weights, Eisenstat and Klein [83] improved the construction running time of Theorem 5.16 to $O(N)$. These results, again, imply semi-local LCS algorithms, running in time $O(mn \log m)$ and $O(mn)$, respectively.

5.3 LCS kernel

Canonical decomposition. The key property of semi-local LCS matrices is captured by the following theorem.

Theorem 5.17 (LCS matrix canonical decomposition) *Let a, b be strings. The semi-local LCS matrix $\mathsf{H}_{a,b}$ is unit-anti-Monge:*

$$\mathsf{H}_{a,b}[i;j] = -i + j + m - \mathsf{P}^{\Sigma}_{a,b}[i;j] = m - \mathsf{P}^{\Im}_{a,b}[i;j]$$

for all i, j, where $\mathsf{P}_{a,b} = -\mathsf{H}^{\square}_{a,b}$ is a permutation matrix. □

PROOF Consider the LCS grid $\mathsf{G}_{a,b^{pad}}[0\colon m; -m\colon m+n]$. For any crossing pair of highest-scoring paths

$$[0;\hat{\imath}^+] \rightsquigarrow [m;\hat{\jmath}^- + m] \qquad [0;\hat{\imath}^-] \rightsquigarrow [m;\hat{\jmath}^+ + m]$$

where $\hat{\imath} \in \langle -m\colon n\rangle$, $\hat{\jmath} \in \langle 0\colon m+n\rangle$, there exists a non-crossing pair of paths

$$[0;\hat{\imath}^-] \rightsquigarrow [m;\hat{\jmath}^- + m] \qquad [0;\hat{\imath}^+] \rightsquigarrow [m;\hat{\jmath}^+ + m]$$

of at least the same total score, which can be obtained by rearranging the edges among the paths. Therefore, we have

$$\mathsf{H}_{a,b}[\hat{\imath}^+;\hat{\jmath}^-] + \mathsf{H}_{a,b}[\hat{\imath}^-;\hat{\jmath}^+] \le \mathsf{H}_{a,b}[\hat{\imath}^-;\hat{\jmath}^-] + \mathsf{H}_{a,b}[\hat{\imath}^+;\hat{\jmath}^+]$$

for all $\hat{\imath}, \hat{\jmath}$, hence $\mathsf{H}^{\square}_{a,b}\langle \hat{\imath}, \hat{\jmath}\rangle \le 0$, and matrix $\mathsf{H}_{a,b}$ is anti-Monge.

Let $A[i;j] = -i + j + m - \mathsf{H}_{a,b}[i;j]$. From the above, matrix A is Monge. By Definition 5.13, we have

$$A[m+n;j] = -m - n + j + m - \mathsf{H}_{a,b}[m+n;j] = j - n - (j-n) = 0$$
$$A[i;0] = -i + 0 + m - \mathsf{H}_{a,b}[i;0] = m - i - (m-i) = 0$$

for all $i, j \in [0{:}\, m+n]$, hence A is simple. We also have

$$A[0; j] = -0 + j + j - (-m) - \mathsf{H}_{a,b}[-m; j] = j - (-m) - m = j$$
$$A[i; m+n] = (m+n) - i - \mathsf{H}_{a,b}[i; m+n] = (m+n) - i - m = n - i$$

for all $i \in [-m{:}\, n]$, $j \in [0{:}\, m+n]$, hence the cross-difference matrix $A^{\square}$ is bistochastic, so A is regular Monge. Thus, A is integer and simple regular Monge, therefore it is simple unit-Monge. The theorem statement is now straightforward from the definitions, with $\mathsf{P}_{a,b} = A^{\square}$. ∎

Corollary 5.18 *String a is a subsequence of substring $b\langle i{:}\, j\rangle$ for some $i, j \in [0{:}\, n]$, if and only if $\mathsf{P}^{\intercal}_{a,b}[i; j] = 0$.* □

The intuition behind Theorem 5.17 is as follows. Let b' be a substring of b. If substring b' is extended on either the left or the right by one character, then its LCS score against string a either is unchanged, or increases by 1, depending on whether or not the new character of b' can be matched to a character of a. The unit-anti-Monge property of matrix $\mathsf{H}_{a,b}$ reflects the fact that, as substring b' grows while a is fixed, obtaining a matching for each new character becomes relatively harder. Each nonzero in the permutation matrix $\mathsf{P}_{a,b}$ represents a unit obstacle to character matching: at such a point, extending b' independently on the left or on the right gives a match, and a corresponding increase by 1 in the LCS score; however, these two matches are incompatible, so extending b' simultaneously on the left or on the right gives a total increase in the LCS score by 1 and not by 2. The above is only very general intuition, since"obtaining a match" for a given character is not absolute, but depends on the choice of a particular highest-scoring path through the LCS grid. Even more generally, one can think of the nonzeros in $\mathsf{P}_{a,b}$ as "the places where string-substring LCS score, as a function of the substring's endpoints, is nonlinear".

Theorem 5.17 holds, with a similar proof, not only for an LCS grid of a pair of strings, but more generally for any triangulated grid with zero weights on the horizontal and vertical edges, and zero-one weights on the diagonal edges. However, the theorem in its given form will be sufficient for the rest of this work.

Definition 5.19 *The* semi-local LCS kernel *for strings a, b is the permutation matrix $\mathsf{P}_{a,b}\langle -m{:}\, n; 0{:}\, m+n\rangle$, determined by Theorem 5.17.* □

We will abbreviate "semi-local LCS kernel" to just *LCS kernel*, where that is justified by the context.

The key idea of our approach is to regard Theorem 5.17 as defining an implicit solution to the semi-local LCS problem.

Definition 5.20 *Let X be an algorithmic problem asking for comparison scores between certain substrings of input string(s). An* oracle *for X with*

query time T is a data structure obtained from the input strings, that supports the queries specified by X on substrings of length at most s in time $T(s)$. □

Theorem 5.21 *Given strings a, b, there exists a semi-local LCS oracle with*

- *size $O(m+n)$ and query time $O(s)$;*
- *size $O\big((m+n)\log(m+n)\big)$ and query time $O\big((\log s)^2\big)$;*

where s is the length of the shorter string or substring in the query. □

PROOF The first oracle is given by (the nonzeros of) kernel $\mathsf{P}_{a,b}$. By Theorem 5.17, the LCS score $\mathsf{H}_{a,b}[i;j]$ for string a against substring $b\langle i\colon j\rangle$ is determined by the length $j-i$ of string $b\langle i\colon j\rangle$, and by the number $\mathsf{P}^{\Sigma}_{a,b}[i;j]$ of nonzeros in $\mathsf{P}_{a,b}$ that are $\gtrcurlyeq$-dominated by the point $[i;j]$. Alternatively, the same LCS score is also determined by the length m of string a, and by the number $\mathsf{P}^{\Sigma}_{a,b}[i;j]$ of nonzeros in $\mathsf{P}_{a,b}$ that $\gtrcurlyeq$-dominate the point $[i;j]$.

The second oracle is obtained from the first one by Theorem 4.6. ■

Example 5.22 Figure 5.4 illustrates the unit-anti-Monge property of matrix $\mathsf{H}_{a,b}$ by the coloured grid pattern: red (respectively, blue) unit-length grid lines separate matrix elements that differ by 1 (respectively, by 0). The same figure also shows kernel $\mathsf{P}_{a,b}\langle -8\colon 13; 0\colon 8+13\rangle$: its nonzeros are indicated by green bullets.

Nonzeros of $\mathsf{P}_{a,b}$ that are $\gtrcurlyeq$-dominated by the point $[4;11]$ correspond to the green bullets lying below-left of the circled entry in $\mathsf{H}_{a,b}$. There are exactly two such nonzeros, therefore $\mathsf{P}^{\Sigma}_{a,b}[4;11]=2$, and the circled entry is $\mathsf{H}_{a,b}[4;11]=11-4-\mathsf{P}^{\Sigma}_{a,b}[4;11]=11-4-2=5$.

Nonzeros of $\mathsf{P}_{a,b}$ that $\gtrcurlyeq$-dominate the point $[4;11]$ correspond to the green bullets lying above-right of the circled entry in $\mathsf{H}_{a,b}$. There are exactly three such nonzeros, therefore $\mathsf{P}^{\Sigma}_{a,b}[4;11]=3$, and the circled entry is again $\mathsf{H}_{a,b}[4;11]=8-\mathsf{P}^{\Sigma}_{a,b}[4;11]=8-3=5$. ■

An LCS kernel can naturally be identified with a reduced sticky braid of order $m+n$ (or, more precisely, with a family of equivalent reduced sticky braids). As we will show in subsequent sections, sticky multiplication of LCS kernels (or, equivalently, multiplication of the corresponding sticky braids) can be used to obtain efficient algorithms for the semi-local LCS problem.

Definition 5.23 *Let G be an $m\times n$ triangulated grid, and B a sticky braid of order $m+n$. The corresponding* embedded sticky braid *is a set of $m+n$ monotone paths in the dual graph of G, each path originating at a different face on the top-left boundary of G and terminating at a different face on the bottom-right boundary of G, such that pairwise crossings between the paths realise strand crossings in B.* □

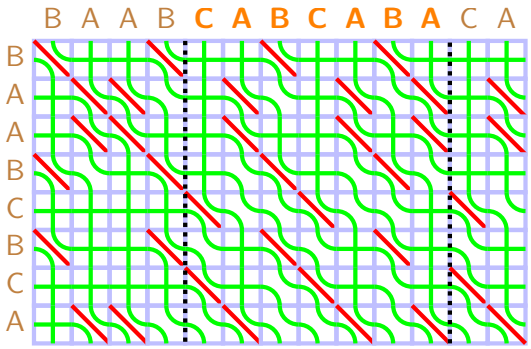


Figure 5.5: LCS grid $\mathsf{G}_{a,b}$; LCS kernel $\mathsf{P}_{a,b}$ as a sticky braid

Example 5.24 Figure 5.5 shows LCS kernel $\mathsf{P}_{a,b}$ as a sticky braid of order $8 + 13 = 21$, embedded in the LCS grid $\mathsf{G}_{a,b}$. The choice of a specific embedding is not important for this example; the significance of this particular embedding is that it can be obtained by Algorithm 5.36 that will be introduced in Section 5.5.

Note that the two nonzeros that are $\geqslant$-dominated by the point $[4; 11]$ in Figure 5.4 correspond to the two strands that fit in the rectangle between two dashed vertical lines $i = 4$ and $j = 11$ in Figure 5.5. Also, the three nonzeros that $\geqslant$-dominate the point $[4; 11]$ in Figure 5.4 correspond to the three strands that pierce both dashed vertical lines in Figure 5.5. ■

The definitions of semi-local LCS matrix and kernel are not symmetric with respect to the order of the input strings. The precise relationship between matrices $\mathsf{H}_{a,b}$, $\mathsf{H}_{b,a}$, and between the corresponding kernels $\mathsf{P}_{a,b}$, $\mathsf{P}_{b,a}$ is as follows.

Theorem 5.25 *Let a, b be strings. We have*

$$\mathsf{H}_{b,a}[i; j] = \mathsf{H}_{a,b}[-i, m + n - j] - i + j - n$$
$$\mathsf{P}_{b,a}\langle \hat{\imath}, \hat{\jmath}\rangle = \mathsf{P}_{a,b}\langle -\hat{\imath}, m + n - \hat{\jmath}\rangle$$

for all $i \in [-n\!:\!m]$, $j \in [0\!:\!m + n]$, $\hat{\imath} \in \langle -n\!:\!m\rangle$, $\hat{\jmath} \in \langle 0\!:\!m + n\rangle$. □

Proof Straightforward by the definitions. ■

Submatrix notation. Just as the semi-local LCS problem gives rise to the LCS matrix $\mathsf{H}_{a,b}$ and the LCS kernel $\mathsf{P}_{a,b}$, the four individual components of the semi-local LCS problem correspond to their submatrices. For example, $\mathsf{H}_{a,b}[0\!:\!|\!:\!n]$ is the submatrix of all string-substring LCS scores for strings a, b, and similarly for the other three components of the semi-local LCS problem. To deal with each component individually, we introduce superscript notation as follows.

Definition 5.26 *The* suffix-prefix, substring-string, string-substring and prefix-suffix LCS submatrices *for strings a, b are the matrices*

$$\mathsf{H}^{\searrow}_{a,b} = \mathsf{H}_{a,b}[:0;:n] \quad \mathsf{H}^{\swarrow}_{a,b} = \mathsf{H}_{a,b}[:0;n:]$$
$$\mathsf{H}^{\nearrow}_{a,b} = \mathsf{H}_{a,b}[0::;:n] \quad \mathsf{H}^{\nwarrow}_{a,b} = \mathsf{H}_{a,b}[0::;n:]$$

The corresponding LCS subkernels *of each kind are the subpermutation matrices*

$$\mathsf{P}^{\searrow}_{a,b} = \mathsf{P}_{a,b}\langle:0;:n\rangle \quad \mathsf{P}^{\swarrow}_{a,b} = \mathsf{P}_{a,b}\langle:0;n:\rangle$$
$$\mathsf{P}^{\nearrow}_{a,b} = \mathsf{P}_{a,b}\langle0::;:n\rangle \quad \mathsf{P}^{\nwarrow}_{a,b} = \mathsf{P}_{a,b}\langle0::;n:\rangle \qquad \square$$

Note that the four component submatrices of $\mathsf{H}_{a,b}$ do not form a proper partitioning, since they overlap by row 0 and column n. In particular, the global LCS score $lcs(a,b) = \mathsf{H}_{a,b}[0;n]$ belongs to all four components of the semi-local LCS problem, and therefore to all four submatrices. In contrast, the four component subkernels of $\mathsf{P}_{a,b}$ are pairwise disjoint, and form a proper partitioning:

$$\mathsf{P}_{a,b} = \begin{bmatrix} \mathsf{P}^{\searrow}_{a,b} & \mathsf{P}^{\swarrow}_{a,b} \\ \mathsf{P}^{\nearrow}_{a,b} & \mathsf{P}^{\nwarrow}_{a,b} \end{bmatrix}$$

Example 5.27 Figure 5.4 illustrates Definition 5.26, showing by thin dotted lines the partitioning of $\mathsf{H}_{a,b}$ and $\mathsf{P}_{a,b}$ into component submatrices (respectively, subkernels). The suffix-prefix, substring-string, string-substring and prefix-suffix submatrices (subkernels) are respectively on the top-left, top-right, bottom-left and bottom-right. The elements of $\mathsf{H}_{a,b}$ lying directly on the dotted lines are shared by the bordering LCS submatrices. Note that the substring-string LCS submatrix $\mathsf{H}^{\swarrow}_{a,b}$ and subkernel $\mathsf{P}^{\swarrow}_{a,b}$ are both filled with a constant value (8 and 0, respectively); this is due to the fact that the whole string a is a subsequence of b. ■

The nonzeros of each LCS subkernel introduced in Definition 5.26 can be regarded as an implicit solution to the corresponding component of the semi-local LCS problem. Similarly to the full kernel, its component subkernels can be processed into an efficient data structure of Theorem 4.6 for efficient random access to explicit semi-local LCS scores.

Recall that the LCS kernel $\mathsf{P}_{a,b}$ is a permutation matrix with $m+n$ nonzeros. Sometimes, we would like to avoid storing all these nonzeros, resorting to a more limited kind of string comparison with substantially fewer nonzeros. Such an approach is warranted when the input string sizes are substantially skewed, so that either $m \gg n$, or $m \ll n$. We will introduce the following terminology and notation in order to deal with just the relevant submatrices.

Definition 5.28 *An* extended string-substring *(respectively,* substring-string*)* LCS matrix *for strings a, b and a parameter $k \in [0\colon m]$ (respectively, $k \in [0\colon n]$) is the submatrix*

$$\mathsf{H}^{\nearrow(k)}_{a,b} = \mathsf{H}_{a,b}[-k\colon;\colon m+n-k] \qquad \mathsf{H}^{\swarrow(k)}_{a,b} = \mathsf{H}_{a,b}[\colon k;k\colon]$$

The corresponding extended LCS kernel *is the submatrix*

$$\mathsf{P}^{\nearrow(k)}_{a,b} = \mathsf{P}_{a,b}\langle -k\colon;\colon m+n-k\rangle \qquad \mathsf{P}^{\swarrow(k)}_{a,b} = \mathsf{P}_{a,b}\langle \colon k;k\colon\rangle \qquad \square$$

In particular, we have

$$\mathsf{P}^{\swarrow(0)}_{a,b} = \begin{bmatrix}\mathsf{P}^{\searrow}_{a,b} & \mathsf{P}^{\swarrow}_{a,b}\end{bmatrix} \qquad \mathsf{P}^{\swarrow(m)}_{a,b} = \begin{bmatrix}\mathsf{P}^{\swarrow}_{a,b}\\ \mathsf{P}^{\nwarrow}_{a,b}\end{bmatrix}$$

$$\mathsf{P}^{\nearrow(m)}_{a,b} = \begin{bmatrix}\mathsf{P}^{\searrow}_{a,b}\\ \mathsf{P}^{\nearrow}_{a,b}\end{bmatrix} \qquad \mathsf{P}^{\nearrow(0)}_{a,b} = \begin{bmatrix}\mathsf{P}^{\nearrow}_{a,b} & \mathsf{P}^{\nwarrow}_{a,b}\end{bmatrix}$$

For every $k \in [0\colon m]$, the extended string-substring LCS kernel $\mathsf{P}^{\nearrow(k)}_{a,b}$ has n nonzeros; such kernels are useful whenever $m \gg n$. Analogously, for every $k \in [0\colon n]$, the extended string-substring LCS kernel $\mathsf{P}^{\swarrow(k)}_{a,b}$ has m nonzeros; such kernels are useful whenever $m \ll n$.

Let string a of length $m = m' + m''$ be a concatenation of two fixed strings: $a = a'a''$, where a', a'' are nonempty strings of length m', m'' respectively. A substring of the form $a\langle i'\colon i''\rangle$ with $i' \in [0\colon m']$, $i'' \in [m'\colon m]$ will be called a *cross-substring*. A cross-substring that is either a suffix of a', or a prefix of a'' (i.e. with either $i' = m'$, or $i'' = m'$), will be called *degenerate*. In other words, a cross-substring of a consists of a suffix of a' and a prefix of a''; a cross-substring is non-degenerate, if and only if both of these parts are non-empty. A cross-substring that is a prefix or a suffix of a will be called a *cross-prefix* and a *cross-suffix*, respectively. Given string b of length n that is a concatenation of two fixed strings, $b = b'b''$, cross-substrings of b are defined analogously.

Definition 5.29 *Given a fixed decomposition of string $a = a'a''$, and a string b, the* cross-semi-local LCS matrix *and* kernel *are the extended string-substring LCS matrix $\mathsf{H}^{\nearrow(m')}_{a,b}$ and kernel $\mathsf{P}^{\nearrow(m')}_{a,b}$. Analogously, given a string a, and a fixed decomposition of string $b = b'b''$, the* cross-semi-local LCS matrix *and* kernel *are the extended string-substring LCS matrix $\mathsf{H}^{\swarrow(n')}_{a,b}$ and kernel $\mathsf{P}^{\swarrow(n')}_{a,b}$.* $\square$

A cross-semi-local score matrix represents the solution of a restricted version of the semi-local LCS problem, where the set of all substrings (prefixes, suffixes) of either string is replaced by just the cross-substrings (cross-prefixes, cross-suffixes).

Cross-substrings of string $a = a'a''$ degenerate to suffixes of a' and prefixes of a'' at the submatrix boundaries $\mathsf{H}^{\nearrow(m')}_{a,b}[:; m''+n]$ and $\mathsf{H}^{\nearrow(m')}_{a,b}[-m';:]$. In particular, cross-prefixes and cross-suffixes of a degenerate respectively to the whole a' and the whole a''. Similarly, cross-substrings of string $b = b'b''$ degenerate to suffixes of b' and prefixes of b'' at the submatrix boundaries $\mathsf{H}^{\swarrow(n')}_{a,b}[:; n']$ and $\mathsf{H}^{\swarrow(n')}_{a,b}[n';:]$. Submatrix elements on these boundaries will be called *degenerate*.

5.4 LCS kernel composition

We now combine previously introduced concepts and techniques in order to provide a divide-and-conquer framework for string comparison.

Let a', a'' be nonempty strings of length m', m'' respectively. We consider the semi-local LCS problem on the concatenated string $a = a'a''$ of length $m = m' + m''$ against a fixed string b of length n.

The LCS grid $\mathsf{G}_{a,b}$ consists of subgrids $\mathsf{G}_{a',b}$, $\mathsf{G}_{a'',b}$, sharing a horizontal row of n nodes and $n-1$ edges, which is simultaneously the bottom row of $\mathsf{G}_{a',b}$ and the top row of $\mathsf{G}_{a'',b}$. We will say that the grid $\mathsf{G}_{a,b}$ is the *composite* of grids $\mathsf{G}_{a',b}$ and $\mathsf{G}_{a'',b}$.

Our goal is, given the LCS kernels $\mathsf{P}_{a',b}$, $\mathsf{P}_{a'',b}$, to compute the *composite* LCS kernel $\mathsf{P}_{a,b}$. It turns out that this problem can be solved efficiently by sticky matrix multiplication. Since the grids $\mathsf{G}_{a',b}$, $\mathsf{G}_{a'',b}$ share only a part of their boundary, we need to apply sticky matrix multiplication in such a way that the corresponding sticky braids are multiplied where they touch along the common boundary, and are left intact otherwise.

Theorem 5.30 (LCS kernel composition) *The string-substring and the semi-local LCS kernels for strings $a = a'a''$, b can be obtained as sticky products*

$$\mathsf{P}^{\nearrow}_{a,b} = \mathsf{P}^{\nearrow}_{a',b} \boxdot \mathsf{P}^{\nearrow}_{a'',b} \tag{5.6}$$

$$\mathsf{P}_{a,b} = \mathsf{diag}\big(\mathsf{I}_{m''}, \mathsf{P}_{a',b}\big) \boxdot \mathsf{diag}\big(\mathsf{P}_{a'',b}, \mathsf{I}_{m'}\big) \tag{5.7}$$

in time $O(n \log N)$ (respectively, $O(m+n \log N)$), where $N = \min(m', m'', n)$.□

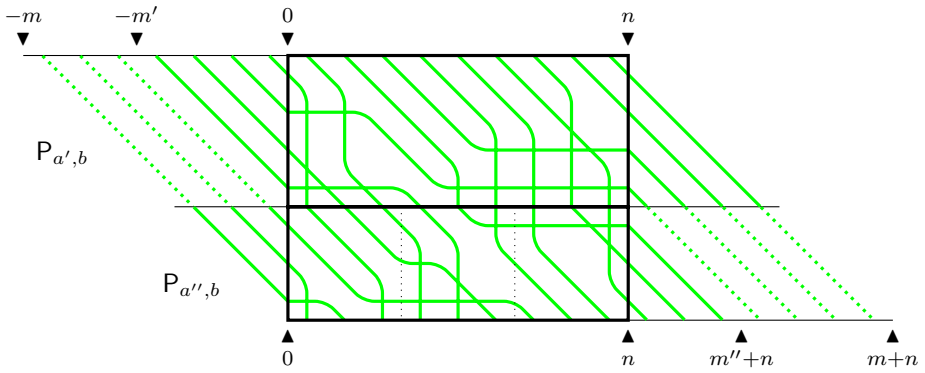
Proof By Theorem 5.17, we have

$$\mathsf{H}_{a,b}[i;k] = k - i - \mathsf{P}^{\Sigma}_{a,b}[i;k]$$

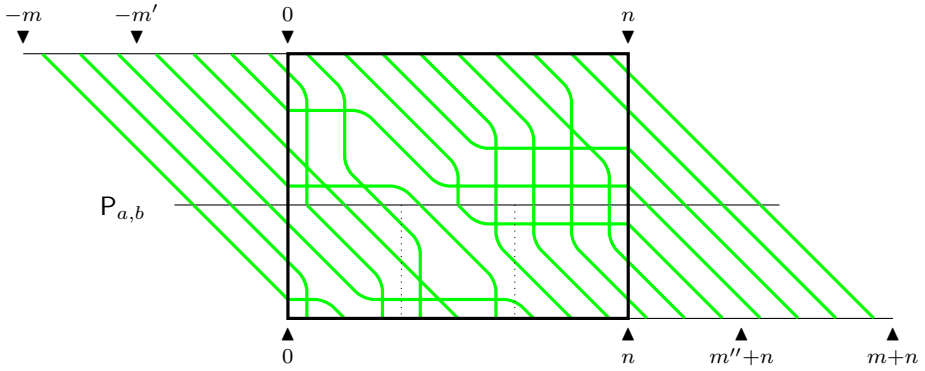
for all $i \in [-m:n]$, $k \in [0:m+n]$. Three cases are possible, based on the partitioning of the index ranges.

Case $i \in [-m':n]$, $k \in [0:m''+n]$. By Definition 5.13 and Theorem 5.17, we have

$$\mathsf{H}_{a,b}[i;k] = \max_{j \in [0:n]} \big(\mathsf{H}_{a',b}[i;j] + \mathsf{H}_{a'',b}[j,k]\big) =$$



(a) Input LCS kernels $\mathsf{P}_{a',b}$, $\mathsf{P}_{a'',b}$



(b) Output LCS kernel $\mathsf{P}_{a,b}$

Figure 5.6: Theorem 5.30 (LCS kernel composition)

$$\max_{j\in[0:n]}\big(j-i-\mathsf{P}^{\Sigma}_{a',b}[i;j]+k-j-\mathsf{P}^{\Sigma}_{a'',b}[j,k]\big)=$$
$$k-i-\min_{j\in[0:n]}\big(\mathsf{P}^{\Sigma}_{a',b}[i;j]+\mathsf{P}^{\Sigma}_{a'',b}[j,k]\big)$$

Therefore,

$$\mathsf{P}^{\Sigma}_{a,b}[i;k]=\min_{j\in[0:n]}\big(\mathsf{P}^{\Sigma}_{a',b}[i;j]+\mathsf{P}^{\Sigma}_{a'',b}[j,k]\big)=\big(\mathsf{P}^{\Sigma}_{a',b}\odot\mathsf{P}^{\Sigma}_{a'',b}\big)[i;k]$$

In particular, this holds for $i,k\in[0:n]$. Hence, we have (5.6).

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

$$\mathsf{H}_{a,b}[i;k]=m'+\mathsf{H}_{a'',b}[i+m';k]=$$
$$m'+k-(i+m')-\mathsf{P}^{\Sigma}_{a'',b}[i+m';k]=$$
$$k-i-\mathsf{P}^{\Sigma}_{a'',b}[i+m';k]$$

Therefore,

$$\mathsf{P}^{\Sigma}_{a,b}[i;k]=\mathsf{P}^{\Sigma}_{a'',b}[i+m';k]=(\mathsf{I}_{m'}\boxdot\mathsf{P}_{a'',b})^{\Sigma}[i;k]$$

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

$$\mathsf{P}^{\Sigma}_{a,b}[i;k] = (\mathsf{P}_{a',b} \boxdot \mathsf{I}_{m''})^{\Sigma}[i;k]$$

Summarising the above three cases, we have (5.7).

We now prove the running time for (5.7); the proof for (5.6) is analogous. The non-trivial part of the product (5.7) is between an $(m+n) \times n$ subpermutation matrix and an $n \times (m+n)$ subpermutation matrix, both of which have at most n nonzeros. Therefore, this non-trivial part of the product can be computed in time $O(n \log n)$, and the whole product in time $O(m + n \log n)$, which is the claimed running time if $m' \le n$, $m'' \le n$, in which case we have $N = n$.

We may now assume without loss of generality that $N = m''$. Assume, also without loss of generality, that $\frac{n}{m''} \ge 1$ is an integer. Consider the LCS grid $\mathsf{G}_{a'',b}$, and partition it into $\frac{n}{m''}$ square blocks of size $m'' \times m''$. It is straightforward to decompose the $(m''+n) \times (m''+n)$ LCS kernel $\mathsf{P}_{a'',b}$ into a sequence of staggered products of $\frac{n}{m''}$ LCS kernels, each of size $2m'' \times 2m''$, one kernel per block of the grid $\mathsf{G}_{a'',b}$. Each product in the sequence has overlap m''.

We now compute the product (5.7) by $\frac{n}{m''}$ successive staggered multiplications of LCS kernels $\mathsf{P}_{a',b}$ by each of the LCS kernels in the decomposition of $\mathsf{P}_{a'',b}$. The resulting running time is $O\big(m + \frac{n}{m''} \cdot m'' \log m''\big) = O(m + n \log m'')$. ∎

Example 5.31 Figure 5.6 shows an instance of LCS kernel composition, as obtained by Theorem 5.30, as a product of sticky braids. Figure 5.6a shows the input LCS kernels $\mathsf{P}_{a',b}$, $\mathsf{P}_{a'',b}$ as sticky braids with solid strands, and the auxiliary identity matrices as sticky braids with dotted strands. String-substring subkernels $\mathsf{P}^{\nearrow}_{a',b}$, $\mathsf{P}^{\nearrow}_{a'',b}$, $\mathsf{P}^{\nearrow}_{a,b}$, correspond to their respective subbraids. Decomposition of the sticky braid corresponding to LCS kernel $\mathsf{P}_{a'',b}$ into blocks is shown in Figure 5.6a by thin dotted lines. Clearly, the whole sticky braid for $\mathsf{P}_{a,b}$ can be obtained from $\mathsf{P}_{a',b}$ by a sequence of staggered multiplications with sticky braids for each block of $\mathsf{P}_{a'',b}$. Figure 5.6b shows the output LCS kernel $\mathsf{P}_{a,b}$ as a sticky braid. ∎

Corollary 5.32 *The extended string-substring LCS kernels (in particular, the cross-semi-local LCS kernel) for strings $a = a'a''$, b can be obtained as sticky products*

$$\mathsf{P}^{\nearrow(0)}_{a,b} = \mathsf{P}^{\nearrow(0)}_{a',b} \boxdot \mathsf{diag}\big(\mathsf{P}^{\nearrow(0)}_{a'',b}, \mathsf{I}_{m'}\big) \tag{5.8}$$

$$\mathsf{P}^{\nearrow(m')}_{a,b} = \mathsf{P}^{\nearrow(m')}_{a',b} \boxdot \mathsf{P}^{\nearrow(0)}_{a'',b} \tag{5.9}$$

$$\mathsf{P}^{\nearrow(m)}_{a,b} = \mathsf{diag}\big(\mathsf{I}_{m''}, \mathsf{P}^{\nearrow(m)}_{a',b}\big) \boxdot \mathsf{P}^{\nearrow(m)}_{a'',b} \tag{5.10}$$

each in time $O(n \log N)$. □

PROOF Straightforward by considering subkernels in (5.7). The proof of the running time is analogous to Theorem 5.30. ∎

5.5 Algorithms for semi-local LCS

We have already established that the semi-local LCS problem can be solved by partitioning the LCS grid, solving the problem in each subgrid, and then composing the subgrid solutions into a solution for the whole grid. There are two natural approaches to the partitioning: we can either split the grid recursively into approximately half-sized parts (*recursive combing*), or split off individual cells, one cell at a time (*iterative combing*). In both approaches, the $(m+n)\times(m+n)$ LCS kernel for the LCS grid is ultimately composed by multiple application of staggered sticky multiplication from 2×2 *elementary* LCS kernels in individual cells. In a cell corresponding to a pair of characters α, β, this is defined as

$$\mathsf{P}_{\alpha,\beta} = \begin{cases} \mathsf{I} = \left[\begin{smallmatrix} 1 & \cdot \\ \cdot & 1 \end{smallmatrix}\right] & \text{if } \alpha \text{ matches } \beta \\ \mathsf{I}^{\mathsf{R}} = \left[\begin{smallmatrix} \cdot & 1 \\ 1 & \cdot \end{smallmatrix}\right] & \text{otherwise} \end{cases}$$

Recursive combing. In this approach, we partition the LCS grid into two approximately equal subgrids, we obtain the LCS kernel for each subgrid recursively, and then we compose these two kernels to obtain the LCS kernel for the whole grid.

Algorithm 5.33 (Semi-local LCS by recursive combing)

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

Output: a semi-local LCS oracle for a against b.

Description. Recursion on the input strings. The LCS kernel $\mathsf{P}_{a,b}$ is obtained by partitioning one of the strings a, b into two substrings, calling the algorithm recursively to obtain LCS kernels for each substring against the other string, and then composing the two resulting LCS kernels.

Recursion base: $m = n = 1$. The LCS grid $\mathsf{G}_{a,b}$ consists of a single cell. Its LCS kernel is the 2×2 elementary kernel $\mathsf{P}_{a,b} \in \{\mathsf{I}, \mathsf{I}^{\mathsf{R}}\}$.

Recursive step: $m > 1$ or $n > 1$. We choose string a (if $m > 1$) or b (if $n > 1$) for recursive partitioning. If both $m > 1$ and $n > 1$, then we can choose either string.

Assume without loss of generality that string a is chosen for recursive partitioning, and that its length $m > 1$ is even. Let $a = a'a''$, where strings a', a'' are of length $\frac{m}{2}$. The LCS grid $\mathsf{G}_{a,b}$ gets partitioned into two half-sized subgrids $\mathsf{G}_{a',b}$, $\mathsf{G}_{a'',b}$. We call the algorithm recursively on strings a', a'' against b, obtaining LCS kernels $\mathsf{P}_{a',b}$, $\mathsf{P}_{a'',b}$. Given these kernels, the LCS kernel $\mathsf{P}_{a,b}$ is obtained by the algorithm Theorem 5.30 (via Theorem 5.25 if

the recursion is on string b), which calls the algorithm of Theorem 4.16 (the Steady Ant algorithm) as a subroutine. Thus, we have two nested recursions: the outer recursion of the current algorithm, and the inner recursion of Theorem 4.16. (End of recursive step.) ■

Example 5.34 Figure 5.7 (split over two pages) shows an execution of Algorithm 5.33 as the construction of an embedded sticky braid for the LCS kernel $\mathsf{P}_{a,b}$:

- Subfigure 5.7a shows the initial state of the sticky braid, composed from elementary braids of order 2 in individual grid cells; we assume that both strings are padded implicitly to the closest highest power of 2 (i.e. string a stays at length 8, and string b is padded to length 16);
- Subfigures 5.7b–5.7e show the intermediate states of the sticky braid at different recursion levels;
- Subfigure 5.7f shows the final state of the sticky braid, identical to the one shown in Figure 5.5.

The resulting reduced sticky braid corresponds to the output LCS kernel $\mathsf{P}_{a,b}$. ■

Theorem 5.35 *The semi-local LCS oracle of Theorem 5.21 can be obtained in time $O(mn)$ by Algorithm 5.33.* □

Proof In Algorithm 5.33, the recursion tree is dominated by the bottom level, which consists of $O(mn)$ instances of sticky matrix multiplication of size $O(1)$. Therefore, the total running time is $O(mn)$. ■

Iterative combing. In this approach, we iterate through the LCS grid, building the solution incrementally from elementary kernels of each cell.

Algorithm 5.36 (Semi-local LCS by iterative combing)

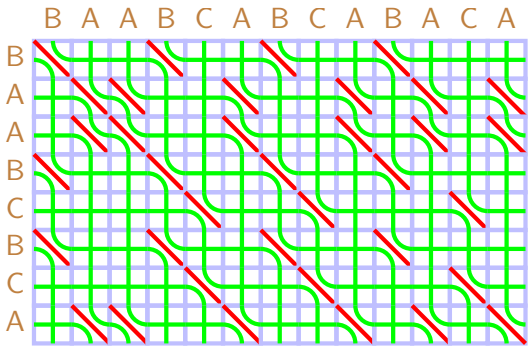
Input: strings a, b of length m, n, respectively.

Output: a semi-local LCS oracle for a against b.

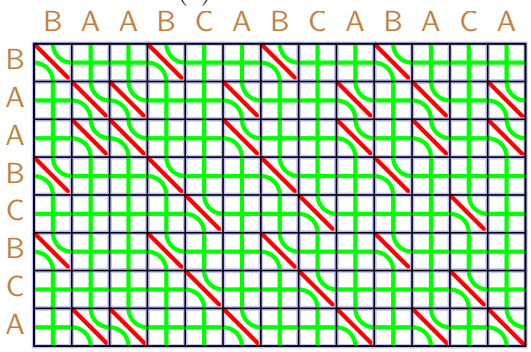
Description. We obtain kernel P of order $m+n$ incrementally, iterating through the cells of the LCS grid $\mathsf{G}_{a,b}$. This kernel is composed from elementary kernels of order 2 of individual cells. We initialise

$$P \leftarrow \mathsf{I}_{m+n}$$

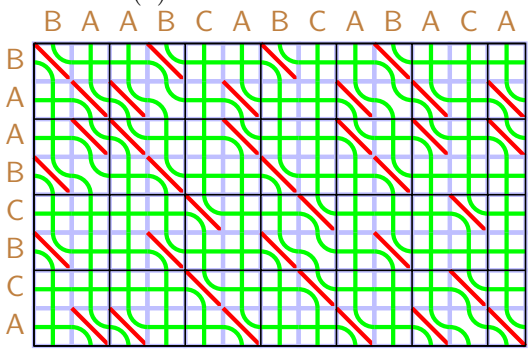
We then iterate through the cells of $\mathsf{G}_{a,b}$ for all $\hat{\imath} \in \langle 0\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$ in lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the cells. We call the union of the cells that



(a) Initial state

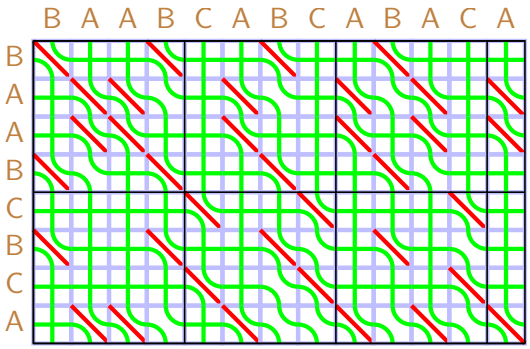


(b) State at $m = n = 1$

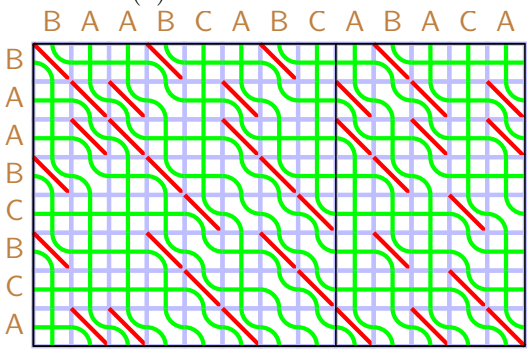


(c) State at $m = n = 2$

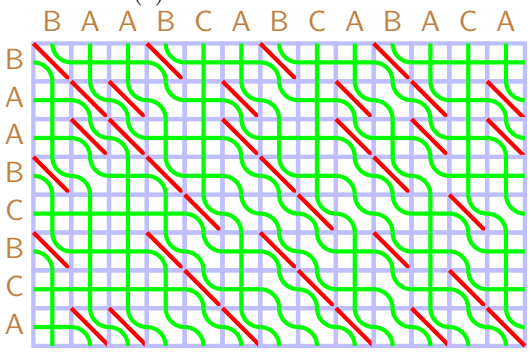
Figure 5.7: Execution of Algorithm 5.33 (semi-local LCS by recursive combing)



(d) State at $m = n = 4$



(e) State at $m = n = 8$



(f) Final state

Figure 5.7: Execution of Algorithm 5.33 (semi-local LCS by recursive combing), continued

have been iterated through the *swept area* of the grid, and its bottom-right boundary the *frontier*. Initially, the swept area is empty, and the frontier coincides with the top-left boundary of the grid. Every iteration extends the swept area by one cell at the frontier. In each iteration, we update kernel P by

$$P \leftarrow P \boxdot \mathsf{diag}(\mathsf{I}_{-\hat{\imath}+\hat{\jmath}+m-1}, \mathsf{P}_{a\langle\hat{\imath}\rangle,b\langle\hat{\imath}\rangle}, \mathsf{I}_{\hat{\imath}-\hat{\jmath}+n-1})$$

where $\mathsf{P}_{a\langle\hat{\imath}\rangle,b\langle\hat{\jmath}\rangle}$ is the elementary kernel for the current cell. By Theorem 5.30, the updated matrix P corresponds to the swept area extended by the current cell.

The final state of kernel P provides the algorithm's output: $\mathsf{P}_{a,b} \leftarrow P$.■

Algorithm 5.36 can be interpreted visually as an incremental construction of a reduced sticky braid of order $m+n$ for the LCS kernel $\mathsf{P}_{a,b}$, embedded in the LCS grid $\mathsf{G}_{a,b}$. Individual strands within the braid are identified by their starting index at the top boundary of the padded LCS grid $\mathsf{G}_{a,b^{pad}}$.

We begin with a (generally unreduced) sticky braid of order $m+n$ embedded in the LCS grid $\mathsf{G}_{a,b}$, which is composed of mn *elementary* braids of order 2. In each cell, the elementary braid corresponds to the cell's elementary kernel $\mathsf{P}_{a\langle\hat{l}\rangle,b\langle\hat{\jmath}\rangle}$: it is the identity braid (a pair of non-crossing strands) in a match cell, and the zero braid (a pair of crossing strands) in a mismatch cell.

The iteration procedure of Algorithm 5.36 transforms this initial unreduced sticky braid to an equivalent reduced one. This transformation is similar to the combing procedure of Theorem 4.23, leaving intact the elementary braids within the cells along the top-left boundary of the LCS grid $\mathsf{G}_{a,b}$, and updating the elementary braids within the remaining cells. The cells are iterated through from left to right and from top to bottom, either in the lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the cells.

For each cell, we consider its embedded elementary braid. The two strands of this braid will now extend a pair of strands in the embedded braid of the swept area of the grid. The update operates on the cell's embedded braid as follows. In a match cell, we keep the identity braid embedded in the current cell, extending the two strands across the frontier without creating a new crossing. In a mismatch cell, we first determine whether the two strands being extended have ever crossed previously anywhere in the swept subgrid. This check can be performed efficiently by comparing the starting indices of both strands. If the two strands have never crossed previously, we keep the zero braid embedded in the current cell, extending the two strands across the frontier and letting them cross in the current cell. However, if the two strands have crossed previously, we replace the zero braid with the identity braid, extending the two strands across the frontier to avoid a second crossing. This completes the update for the current cell.

By Definition 4.19 (the sticky braid monoid), each update is an equivalence transformation on the embedded sticky braid for the whole LCS grid $\mathsf{G}_{a,b}$. Hence, the resulting sticky braid is equivalent to the initial one. Furthermore, since our procedure combs away all double crossings for any pair of strands, the resulting sticky braid is reduced. Therefore, it provides the nonzeros of the LCS kernel $\mathsf{P}_{a,b}$.

Example 5.37 Figure 5.8 shows an execution of of Algorithm 5.36 in terms of an embedded sticky braid construction for the LCS kernel $\mathsf{P}_{a,b}$.

Figure 5.8a shows the initial state of the sticky braid, composed from elementary braids of order 2 in individual grid cells. The LCS grid $\mathsf{G}_{a,b}$ is then swept in the top-to-bottom, left-to-right lexicographic cell order, which is compatible with $\ll$-dominance of the cells.

Figure 5.8b shows a snapshot of some intermediate state of the sticky braid. The grid area that has already been swept is shown by the dark border; the current cell is shaded in yellow. Since the two strands passing through the current cell have previously crossed, their crossing has been undone (combed away) in the current cell.

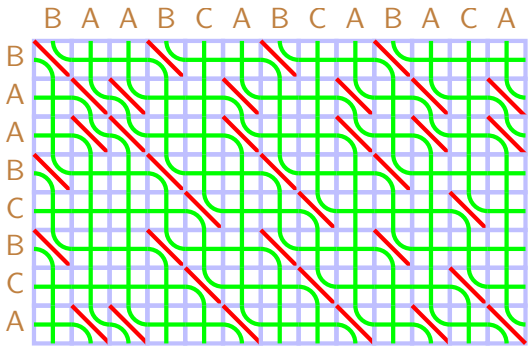
Figure 5.8c shows the final state of the sticky braid, identical to the one shown in Figure 5.5. It is a reduced sticky braid corresponding to the output LCS kernel $\mathsf{P}_{a,b}$. ■

Theorem 5.38 *The semi-local LCS oracle of Theorem 5.21 can be obtained in time $O(mn)$ by Algorithm 5.36.* □

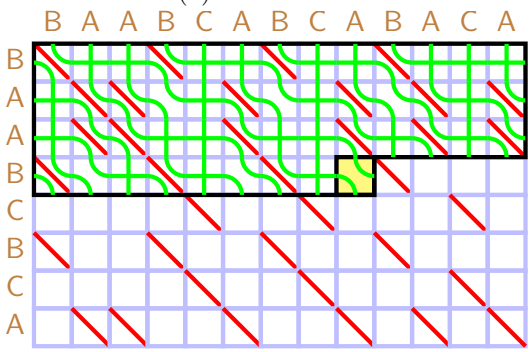
Proof In Algorithm 5.36, the update for each cell involves sticky multiplication of two matrices with two nonzeros each, and can therefore be performed in time $O(1)$. The overall running time of the algorithm is $O(mn)$.■

We observe that the intermediate values from the execution of Algorithm 5.36 provide an oracle not only to the semi-local LCS problem, but also to the more general *prefix semi-local LCS problem,* that asks for the LCS scores of all prefixes of a against all substrings of b, and of all substrings of a against all prefixes of b. Similarly to a solution to the prefix LCS problem by Algorithm 5.8 (Prefix LCS by classical dynamic programming), the prefix semi-local LCS oracle obtained by Algorithm 5.36 (Semi-local LCS by iterative combing) can be used to trace back the actual LCS for any prefix semi-local (i.e. prefix-substring or substring-prefix) LCS query, in time proportional to the size of the output. Just as for prefix LCS, the memory-saving recomputation technique by Hirschberg [111] can be applied to achieve prefix semi-local LCS traceback in the same asymptotic time, but in a linear amount of memory.

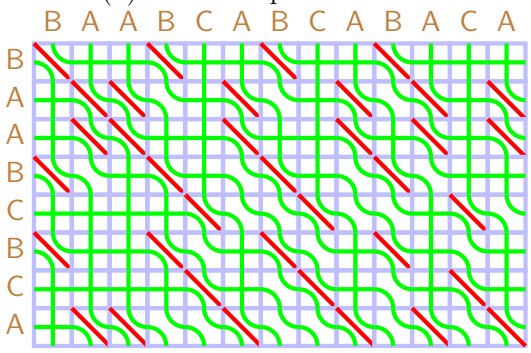
Running time optimality. Both Algorithm 5.33 (Semi-local LCS by recursive combing) and Algorithm 5.36 (Semi-local LCS by iterative combing)



(a) Initial state



(b) State at a particular cell



(c) Final state

Figure 5.8: Execution of Algorithm 5.36 (semi-local LCS by iterative combing)

match the lower bound of $\Omega(mn)$ by Aho et al. [5] on the running time for the (global) LCS problem. Therefore, these algorithms are asymptotically optimal for an unordered alphabet. However, assuming a totally ordered alphabet, the algorithms' running times can be improved by a (model-dependent) polylogarithmic factor, as will be shown in the next section.

5.6 Micro-block speedup

Assume that the alphabet is a totally ordered set, allowing relative order comparison between characters. In such a case, we no longer need to perform all the mn pairwise comparisons of characters from each string: the "missing" comparisons can be obtained by transitivity. Furthermore, we no longer need to process every cell of the LCS grid individually, so string comparison algorithms with running time $o(mn)$ become possible[2].

A classical speedup for the LCS problem originates from a matrix multiplication method by Arlazarov et al. [19], often nicknamed the "four Russians method". In this work, we call it the *micro-block speedup (MBS),* adopting the terminology of Bille and Gørtz [33]. The main idea of this method is to partition the LCS grid into regular micro-blocks of a small, suitably chosen size, such that running time can be saved by precomputing the problem's solution for every possible micro-block in advance.

Without loss of generality, we assume that $m \leq n$. By applying MBS to the classical dynamic programming algorithm, Masek and Paterson [155] gave an algorithm for the (global) LCS problem running in time $O\big(\frac{mn}{\log^2 n}+n\big)$ for a constant-size alphabet[3]. An alternative precomputation-based approach to subquadratic LCS computation was developed by Crochemore et al. [67] (see also [121]).

An extension of MBS to an alphabet of unbounded size, running in time $O\big(\frac{mn(\log\log n)^2}{\log^2 n}+n\big)$, was suggested by Paterson and Dančík [171], and fully developed by Bille and Farach-Colton [35]. In this extension, a second, coarser level of LCS grid 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.

[2]This 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.

[3]The original algorithm by Masek and Paterson [155] runs in time $O\big(\frac{mn}{\log n}+n\big)$ for a constant-size alphabet in the log-cost RAM model. The unit-cost RAM version of the algorithm was given in [218, 35].

Precomputing micro-block kernels. In order to speed up the solution of the semi-local LCS problem, Algorithm 5.36 can be adapted to take advantage of MBS as follows. For simplicity, we assume that strings a, b are of equal length.

Lemma 5.39 *Assuming $m = n$, the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O\big(\frac{n^2(\log\log n)^2}{\log n}\big)$.* □

PROOF We call two strings of equal length (or two string pairs, etc.) *isomorphic*, if one can obtained from the other by a permutation of the alphabet. Given a pair of strings, each of length r, over an arbitrary ordered alphabet, it is straightforward to obtain an isomorphic pair of strings over a reduced alphabet of size $2r$ by sorting the proper alphabet of each string in time $O(r \log r)$, and then merging the two sorted alphabets in time $O(r)$. Therefore there are, as a conservative over-estimate, at most $(2r)^{2r}$ non-isomorphic pairs of strings of length r.

The LCS grid $\mathsf{G}_{a,b}$ is partitioned into regular $t \times t$ *micro-blocks*, where $t < n$ will be specified later. There are $\frac{n^2}{t^2}$ micro-blocks in total. Let $\mathsf{G}_{a',b'}$ be a micro-block, where a' is a substring of a and b' a substring of b, both of length t. For every one of the $(2t)^{2t}$ non-isomorphic pairs a', b', we can precompute the $2t$ nonzeros of the kernel $\mathsf{P}_{a',b'}$ in time $O(t^2)$ by Algorithm 5.33 or Algorithm 5.36. Therefore, the total precomputation time for micro-block kernels, as a conservative over-estimate, is at most $(2t)^{2t} \cdot O(t^2) = (2t)^{2t} \cdot O((2t)^t) = O\big((2t)^{3t}\big)$.

The semi-local LCS computation can now be performed on input strings a, b, keeping the overall structure of Algorithm 5.36 (semi-local LCS by iterative combing), but operating on micro-blocks instead of individual cells as a basic unit. We construct a semi-local LCS kernel P of order $m + n$ incrementally, iterating through the micro-blocks of the LCS grid $\mathsf{G}_{a,b}$. This kernel is composed from micro-block kernels of order $2t$, that are associated with each micro-block by the alphabet sorting/merging procedure described above in time $O(t \log t)$, reducing the micro-block's alphabet to one of size $2t$.

As in Algorithm 5.36, we iterate through the micro-blocks in lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the micro-blocks. In each iteration, we update kernel P by subkernel assignment

$$P' \leftarrow P' \boxdot \mathsf{P}_{a\langle t\hat{l}^-:t\hat{l}^+\rangle, b\langle t\hat{\jmath}^-:t\hat{\jmath}^+\rangle}$$

where $P' = P\langle :; t\cdot(\hat{\jmath}+m-\hat{l}-1) : t\cdot(\hat{\jmath}+m-\hat{l}+1)\rangle$ is a $2t$-column submatrix of P, and $\mathsf{P}_{a\langle t\hat{l}^-:t\hat{l}^+\rangle, b\langle t\hat{\jmath}^-:t\hat{\jmath}^+\rangle}$ is the precomputed $2t \times 2t$ kernel for the current micro-block. By Theorem 5.30, the updated matrix P corresponds to the swept area extended by the current micro-block. The update is performed by sticky matrix multiplication, running in time $O(t \log t)$. Therefore, the main

algorithm's running time (after precomputation) becomes $\frac{n^2}{t^2} \cdot O(t \log t) = O(\frac{n^2 \log t}{t})$.

Taking $t = \frac{\log n}{3 \log \log n}$, we find the total running time of the precomputation followed by the main algorithm, as a conservative over-estimate, to be at most

$$O\big((2t)^{3t}\big) + O\big(\tfrac{n^2 \log t}{t}\big) = O\big(2^{3t \log(2t)} + \tfrac{n^2 \log t}{t}\big) =$$
$$O\big(2^{\frac{\log n}{\log \log n} \cdot \log \log n} + \tfrac{n^2 (\log \log n)^2}{\log n}\big) = O\big(n + \tfrac{n^2 (\log \log n)^2}{\log n}\big) =$$
$$O\big(\tfrac{n^2 (\log \log n)^2}{\log n}\big)$$

A similar approach can be applied to speed up Algorithm 5.33 (Semi-local LCS by divide-and-conquer). We follow the recursive structure of the algorithm, with the precomputed micro-block kernels forming the recursion base. The main algorithm is dominated by the recursion base, running in time $O\big(\frac{n^2 (\log \log n)^2}{\log n}\big)$. ∎

Precomputing main kernel updates. We now describe a further speedup that performs extra precomputation in order to reduce the main algorithm's micro-block processing time from $O(t \log t)$ to $O(t)$.

Lemma 5.40 *Assuming $m = n$, the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O\big(\frac{n^2 \log \log n}{\log n}\big)$.* □

Proof We keep the terminology and the notation of Lemma 5.39. Consider a micro-block $\mathsf{G}_{a',b'}$, where a' is a substring of a and b' a substring of b, both of length t. Observe that the result of alphabet sorting of each of strings a', b' can be reused for different micro-blocks. Each substring need only be sorted once, therefore the total time required for alphabet sorting across all micro-blocks is $\frac{2n}{t} \cdot O(t \log t) = O(n \log t)$. The merging of two sorted substring alphabets can be performed in time $O(t)$ per micro-block, reducing the micro-block's alphabet to one of size $2t$.

In order to speed up the frontier matrix updates, we introduce a second level of blocking, similarly to the global LCS algorithm of [35]. The LCS grid $\mathsf{G}_{a,b}$ is partitioned into coarser regular $s \times s$ *macro-blocks*, where s, $t < s < n$, will be specified later. There are $\frac{n^2}{s^2}$ macro-blocks in total. Alphabet reduction will now be performed in macro-blocks. Consider a macro-block $\mathsf{G}_{a',b'}$, where a' is a substring of a and b' a substring of b, both of length s. Each substring need only be sorted once, therefore the total time required for alphabet sorting across all macro-blocks is $\frac{2n}{s} \cdot O(s \log s) = O(n \log s)$. The merging of two sorted substring alphabets can be performed in time $O(s)$ per macro-block, reducing the macro-block's alphabet to one of size $2s$.

Let $\mathsf{G}_{a'',b''}$ be a micro-block, where a'' is a substring of a' and b'' a substring of b', both of length t. More precomputation will now be required: for every one of the $(2s)^{2t}$ non-isomorphic pairs a'', b'', we will precompute the $2t$ nonzeros of the LCS kernel $\mathsf{P}_{a'',b''}$ in time $O(t^2)$ by Algorithm 5.33 or Algorithm 5.36. Therefore, the total precomputation time for micro-block kernels, as a conservative over-estimate, is at most $(2s)^{2t} \cdot O(t^2) = O\big((2s)^{3t}\big)$.

We refine the main algorithm of Lemma 5.39 as follows. We now perform an outer iteration through the LCS grid in macro-blocks, and an inner iteration though each of those macro-blocks in micro-blocks. Since the micro-block alphabet has already been reduced at the outer iteration level, the appropriate precomputed micro-block kernel can be identified in time $O(t)$ by a straightforward lookup during the inner iteration.

During the inner iteration, we maintain a $2s \times 2s$ kernel Q for the current macro-block. In each iteration, we update kernel Q by subkernel assignment

$$Q' \leftarrow Q' \boxdot \mathsf{P}_{a\langle t\hat{l}^- : t\hat{l}^+\rangle, b\langle t\hat{\jmath}^- : t\hat{\jmath}^+\rangle}$$

where $Q' = Q\langle : ; t \cdot (\hat{\jmath} + m - \hat{l} - 1) : t \cdot (\hat{\jmath} + m - \hat{l} + 1)\rangle$ is a $2t$-column submatrix of Q, and $\mathsf{P}_{a\langle t\hat{l}^- : t\hat{l}^+\rangle, b\langle t\hat{\jmath}^- : t\hat{\jmath}^+\rangle}$ is the precomputed $2t \times 2t$ kernel for the current micro-block.

Observe that the $2s \times 2t$ product $Q' \boxdot \mathsf{P}_{a\langle t\hat{l}^- : t\hat{l}^+\rangle, b\langle t\hat{\jmath}^- : t\hat{\jmath}^+\rangle}$ can itself be precomputed for every possible pair of multiplicands. The multiplicand matrices have $2t$ nonzeros each, therefore, as a conservative estimate, there are at most $(2s)^{2t} \cdot (2t)^{2t} < (2s)^{4t}$ possible multiplicand pairs. For every such pair, we can precompute the $2t$ nonzeros of the product in time $O(t \log t)$ by Theorem 4.16. Therefore, the total precomputation time for micro-block sticky products, as a conservative over-estimate, is at most $(2s)^{4t} \cdot O(t \log t) = O\big((2s)^{5t}\big)$.

The kernel update is now performed in time $O(t)$ by a straightforward lookup of a precomputed matrix product. Therefore, the main algorithm's running time (after precomputation) becomes $\frac{n^2}{t^2} \cdot O(t) = O(\frac{n^2}{t})$.

Taking $t = \frac{\log n}{10 \log\log n}$, $s = \frac{(\log n)^2}{2}$, we obtain the total running time of the precomputation followed by the main algorithm, as a conservative over-estimate, to be at most

$$O\big((2s)^{5t}\big) + O\big(\tfrac{n^2}{t}\big) = O\big(2^{5t \log(2s)} + \tfrac{n^2}{t}\big) =$$
$$O\big(2^{\frac{\log n}{2 \log\log n} \cdot 2 \log\log n} + \tfrac{n^2 \log\log n}{\log n}\big) = O\big(n + \tfrac{n^2 \log\log n}{\log n}\big) =$$
$$O\big(\tfrac{n^2 \log\log n}{\log n}\big)$$

In contrast to the micro-block kernel precomputation speedup, this extra speedup exploits in a substantial way the frontier structure of Algorithm 5.36 (Semi-local LCS by incremental sweeping), and does not appear to be applicable to Algorithm 5.33 (Semi-local LCS by divide-and-conquer). ∎

Packing micro-block data. So far, the application of MBS has allowed us effectively to replace $O(t^2)$ operations within a micro-block by $O(t)$ operations along the micro-block's boundary, on values ranging up to $2s$. Still further speedup can be achieved by taking advantage of the unit-cost RAM model, where we can pack several such values in a single integer ranging up to n. The resulting algorithm is as follows.

Algorithm 5.41 (Semi-local LCS: iterative combing with MBS)

Input: strings a, b of length m, n, respectively; we assume $m \le n$.

Output: a semi-local LCS oracle for a against b.

Description. Let $t = \frac{\log n}{10 \log\log n}$, $s = \frac{(\log n)^2}{2}$ as before. The LCS grid $\mathsf{G}_{a,b}$ is partitioned into regular $s \times s$ macro-blocks. (If $m < s$, then the macro-blocks are $m \times s$, and the grid is only partitioned along string b.) There are $\frac{n^2}{s^2}$ macro-blocks in total. Let $\mathsf{G}_{a',b'}$ be a macro-block, where a' is a substring of a and b' a substring of b, both of length s. As described previously, we translate the pair of strings a', b' to an isomorphic pair of strings over an alphabet of size $2s$.

Every macro-block is now partitioned into regular $t \times t$ micro-blocks. (If $m < t$, then the micro-blocks are $m \times t$, and the macro-block is only partitioned along string b.) Let $\mathsf{G}_{a'',b''}$ be a micro-block, where a'' is a substring of a' and b'' a substring of b', both of length t. Each of strings a', b' can be packed into an integer in the range up to $(2s)^t$. For every such pair, we then precompute the kernel $\mathsf{P}_{a'',b''}$. This kernel has $2t$ nonzeros, therefore it can be represented by a permutation of $2t$ integers, each in the range up to $2t$. Therefore, the whole kernel can be packed into an integer in the range up to $(2t)^{2t}$. Furthermore, for every $2s \times 2t$ subpermutation matrix P', and every $2t \times 2t$ permutation matrix Q, we precompute the sticky product $P' \boxdot Q$. Matrices P', Q, $P' \boxdot Q$ can each be packed into an integer in the range up to $(2s)^{2t}$, $(2t)^{2t}$, $(2s)^{2t}$, respectively.

As described previously, we perform an outer iteration through the LCS grid in macro-blocks, and an inner iteration though each of those macro-blocks in micro-blocks. All the data required by a single macro-block kernel update are now stored in a constant number of integers, therefore the update can be performed by a constant number of lookups.

In the outer iteration, the main kernel P for the whole LCS grid $\mathsf{G}_{a,b}$ is initialised and updated as in Algorithm 5.36. The final state of kernel P provides the algorithm's output: $\mathsf{P}_{a,b} \leftarrow P$. ■

Theorem 5.42 *Assuming $m \le n$, the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O\big(\frac{mn(\log\log n)^2}{(\log n)^2} + n\big)$.* □

Proof As shown in Lemmas 5.39 and 5.40, as well as the description of Algorithm 5.41, the precomputation time is negligible, compared to the main algorithm.

In the main algorithm, operations are performed on integers in the range up to $(2s)^{2t} = 2^{2t\log(2s)} = o(n)$. Every such operation can be performed in time $O(1)$ in the unit-cost RAM model. There are $\frac{mn}{t^2}$ micro-block updates, each running in time $O(1)$, and $\frac{mn}{s^2}$ macro-block updates, each running in time $O(s \log s)$. There is also an additive term $O(n)$, dominating when m is small. Therefore, the main algorithm runs in time

$$\frac{mn}{t^2} \cdot O(1) + \frac{mn}{s^2} \cdot O(s \log s) + O(n) = mn \cdot O\big(\frac{1}{t^2} + \frac{\log s}{s}\big) + O(n) =$$
$$mn \cdot O\big(\frac{(\log\log n)^2}{(\log n)^2} + \frac{\log\log n}{(\log n)^2}\big) + O(n) = O\big(\frac{mn(\log\log n)^2}{(\log n)^2} + n\big)$$ ■

Example 5.43 Figure 5.9 shows the execution of Algorithm 5.41, using the same conventions as Figure 5.8. For simplicity, the macro-blocks are not shown, and the micro-blocks are shown to be of size 2. As in Algorithm 5.36, the final layout of the sticky braid is identical to the one given in Figure 5.5.■

5.7 Subsequence matching

Subsequence recognition. Apart from string comparison, where both input strings play symmetric roles, we also consider an asymmetric setting, where the inputs are a pattern string p of length m and a text string t of length $n \geq m$. One natural question to ask is whether the pattern is included in the text as a subsequence.

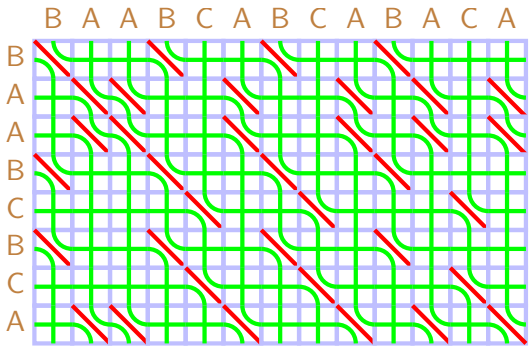
Definition 5.44 *Given a pattern p and a text t, the* subsequence recognition (SR) problem *asks whether t contains p as a subsequence.* □

The SR problem is a special case of the LCS problem: it is equivalent to asking whether the LCS score of t against p is exactly m. This problem has been considered e.g. by Aho et al. [6, Section 9.3], who describe a straightforward finite-automaton based algorithm running in time $O(n)$. Various extensions of this problem have been explored by Crochemore et al. [68].

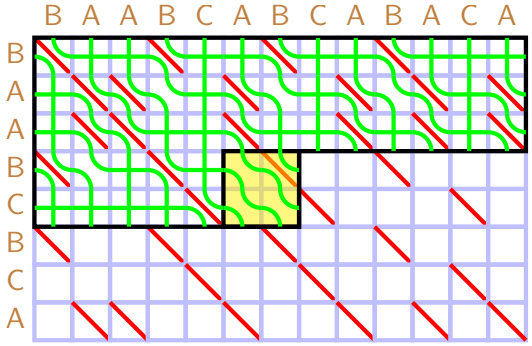
Subsequence matching. We also introduce a local version of subsequence recognition.

Definition 5.45 *Given a pattern p and a text t, the* subsequence matching (SM) problem *(also known as the* episode matching problem*) asks for all substrings in t containing p as a subsequence. Such substrings in t will be called* matching substrings*.* □

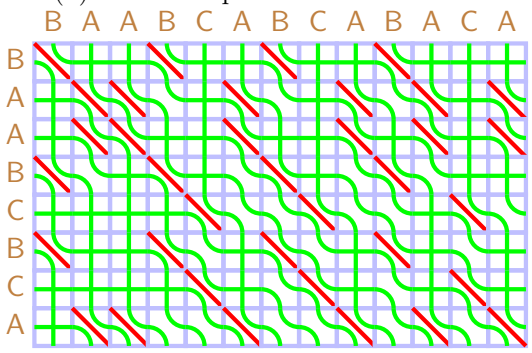
The SM problem can be regarded as a basic form of approximate pattern matching. It is a special case of the string-substring LCS problem, since it asks for the substrings of t, having LCS score against p that is exactly equal to m.



(a) Initial state



(b) State at a particular micro-block



(c) Final state

Figure 5.9: Execution of Algorithm 5.41 (Semi-local LCS by iterative combing with MBS)

Definition 5.46 *Given a pattern p and a text t, the* minimal-substring SM problem *asks for all inclusion-minimal matching substrings. Given a window length w, the* window SM problem *asks for all matching substrings of length w.* □

We refer to the problems in Definition 5.46 as SM with *output filtering*. A superstring of a matching substring is also matching, therefore a solution to the minimal-substring SM problem provides a general SM oracle answering a query in time $O(1)$. In terms of point dominance, the set of all points $[i; j]$ corresponding to matching substrings $t\langle i: j\rangle$ forms a $\geqslant$-upset, and a solution to the minimal-substring SM problem is the set of its $\geqslant$-minimal points.

SM problems were considered by Das et al. [70], who gave an algorithm solving both the minimal-substring and the window SM problems in time $O(mn)$. Using MBS, this algorithm can be modified to run in time $O\big(\frac{mn}{\log^2 n} + n\big)$ for a constant-size alphabet, and in time $O\big(\frac{mn(\log\log n)^2}{\log^2 n} + n\big)$ for an unbounded-size one. A multi-pattern version of SM problems was considered by Cégielski et al. [51].

An algorithm for minimal-substring SM can be obtained by the LCS kernel method as follows.

Algorithm 5.47 (Minimal-substring SM)

Input: pattern p of length m; text t of length n.

Output: endpoints of inclusion-minimal matching substrings in t.

Description. The algorithm runs in two phases.

Phase 1. We obtain the nonzeros of the LCS kernel $\mathsf{P}_{p,t}$ by either of Algorithms 5.33 and 5.36.

Phase 2. Consider the wildcard-padded string $t^{pad}\langle -m: m+n\rangle$. Let $i \in [-m: n]$, $j \in [0: m+n]$. By Theorem 5.17 (LCS matrix canonical decomposition), a substring $t^{pad}\langle i: j\rangle$ is matching, if and only if $\mathsf{P}^{\Sigma}_{p,t}[i; j] = 0$, i.e. point $[i; j]$ is not dominated by any of the nonzeros in $\mathsf{P}_{p,t}$. Consider the nonzeros that are $\geqslant$-maximal among all nonzeros in $\mathsf{P}_{p,t}$. These nonzeros form a $\ll$-chain in $\langle -m: n; 0: m+n\rangle$:

$$L = \Big\{\big\langle \hat{\imath}_{0^+}; \hat{\jmath}_{0^+}\big\rangle \ll \big\langle \hat{\imath}_{1^+}; \hat{\jmath}_{1^+}\big\rangle \ll \cdots \ll \big\langle \hat{\imath}_{s^-}; \hat{\jmath}_{s^-}\big\rangle\Big\}$$

where $\hat{\imath}_{0^+} = (-m)^+$, $\hat{\jmath}_{s^-} = (m+n)^-$, $s = |L| \leq m+n$. If a point is $\geqslant$-dominated by any nonzero in $\mathsf{P}_{p,t}$, then it is dominated by some $\geqslant$-maximal nonzero, i.e. by a point in L. Therefore, a substring $t^{pad}\langle i: j\rangle$ is matching, if and only if point $[i, j]$ is not $\geqslant$-dominated by any point in L. Such points form a $\geqslant$-upset, where inclusion-minimal matching substrings correspond to $\geqslant$-minimal points, forming a *skeleton $\ll$-chain* in $[-m: n; 0: m+n]$ with respect to L:

$$M = \Big\{\big[\hat{\imath}^-_{0^+}; \hat{\jmath}^+_{0^-}\big] \ll \big[\hat{\imath}^-_{1^+}; \hat{\jmath}^+_{1^-}\big] \ll \cdots \ll \big[\hat{\imath}^-_{s^+}; \hat{\jmath}^+_{s^-}\big]\Big\}$$

where $\hat{\jmath}_{0^-} = 0^-$, $\hat{\imath}_{s^+} = 0^+$. All inclusion-minimal matching substrings of t can now be obtained as the elements of subchain $M \cap [0\colon n; 0\colon n]$. ■

Theorem 5.48 *The minimum-substring and the window SM problems can both be solved in time $O(mn)$.* □

PROOF In Phase 1 of Algorithm 5.47, the LCS kernel $\mathsf{P}_{p,t}$ can be obtained in time $O(mn)$ by Theorem 5.35 or Theorem 5.38.

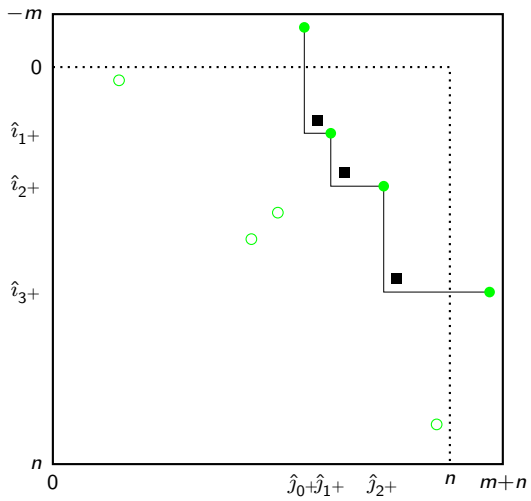
In Phase 2, the $\ll$-chains L and M can be obtained in time $O(m+n) = O(n)$, using incremental queries of Theorem 4.8. From the $\ll$-chain M, it is also straightforward to obtain all matching substrings of length w.

The total running time is dominated by Phase 1, and is therefore as claimed. ■

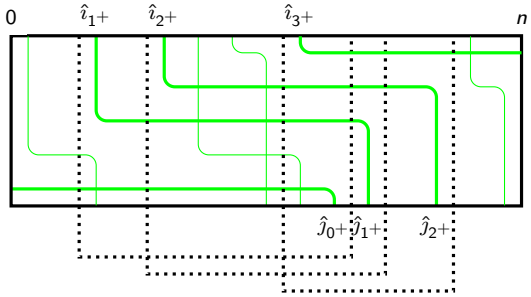
Using MBS, the running time in Theorem 5.48 can be reduced to $O\big(\frac{mn(\log\log n)^2}{\log^2 n} + n\big)$.

Example 5.49 Figure 5.10 illustrates the execution of Phase 2 of Algorithm 5.47. Subfigure 5.10a shows a sketch of the LCS kernel $\mathsf{P}_{p,t}$ with eight nonzeros, represented by green circles. Subfigure 5.10b shows the corresponding reduced sticky braid, with the nonzeros represented by green strands. The $\gtrsim$-maximal nonzeros forming the $\ll$-chain L are shown by filled circles in Subfigure 5.10a, and by thicker strands in Subfigure 5.10b. Any point that is not $\gtrsim$-dominated by any nonzero in $\mathsf{P}_{p;t}$ is located above-right of the $\ll$-chain L. The $\gtrsim$-minimal such points form the $\ll$-chain M. The points of $M \cap [0\colon n; 0\colon n]$ are shown in Subfigure 5.10a by black squares, and in Subfigure 5.10b by dotted brackets; each of these points corresponds to a minimally matching substring in t.

By Corollary 5.18, a substring in t is matching, if and only if the corresponding rectangle in the LCS grid is not pierced by a left-to-right strand in the sticky braid corresponding to $\mathsf{P}_{p,t}$. Notice that the bracketed substrings in Figure 5.10b are exactly the inclusion-minimal substrings satisfying this property. ■



(a) LCS kernel $\mathsf{P}_{p,t}$; $\gtrdot$-chains L, $M \cap [0\colon n; 0\colon n]$



(b) Corresponding sticky braid

Figure 5.10: Execution of Algorithm 5.47 (minimal-substring SM)

Chapter 6

Alignment and edit distance problems

6.1 Alignment and edit distance

Alignment. The concept of LCS score is generalised by that of *alignment score* (see e.g. [121]). An *alignment* of strings a, b is obtained by putting a subsequence of a into one-to-one correspondence with a subsequence of b, respecting the index order. In contrast with an LCS, the two subsequences need not be identical. A pair of corresponding characters, one from a and the other from b, are said to be *aligned*. A character in one string that is not aligned against a character of the other string is said to be aligned against a *gap* in that string. An aligned character-character or character-gap pair is given a real-valued *score*:

- $\mathrm{w}_+ \geq 0$ for a pair of matching characters;
- $\mathrm{w}_0 < \mathrm{w}_+$ for a pair of mismatching characters;
- $\mathrm{w}_- \leq \frac{1}{2}\mathrm{w}_0$ for a gap-character or character-gap pair; it is also normally assumed that $\mathrm{w}_- \leq 0$.

Any particular triple of character alignment scores $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$ will be called a *scoring scheme*. A scoring scheme will be called *rational*, if all its components are rational numbers.

The intuition behind the score inequalities is as follows: aligning a matching pair of characters is always better than aligning a mismatching pair of characters, while the latter is at least as good as aligning each of the two characters against a gap.

Definition 6.1 *Let a, b be strings. Assuming a fixed scoring scheme, the* score *of an alignment is*

$$\mathrm{w}_+ \cdot k_+ + \mathrm{w}_0 \cdot k_0 + \mathrm{w}_- \cdot (m + n - 2k_+ - 2k_0) =$$

$$(\mathrm{w}_+ - 2\mathrm{w}_-) \cdot k_+ + (\mathrm{w}_0 - 2\mathrm{w}_-) \cdot k_0 + \mathrm{w}_- \cdot (m+n)$$

where k_+, k_0 is the number of matching (respectively, mismatching) character pairs in the alignment. □

Definition 6.2 *Let a, b be strings. Assuming a fixed scoring scheme, the* alignment score*, denoted* $\mathsf{score}(a,b)$*, is the maximum score across all possible alignments of a, b. Given strings a, b, the* alignment problem *asks for their alignment score.* □

Example 6.3 The LCS score is a special case of alignment score, given by the scoring scheme $(1,0,0)$. Slightly more sophisticated scoring schemes, intended to penalise gaps in DNA sequence alignment, are given by $(1,0,-0.5)$ and $(2,-1,-1.5)$. The latter scheme is suggested e.g. in [56, Section 1.3].

The gaps in an alignment can be prohibited altogether by making the gap penalty very large in absolute value. The classical *Hamming score* can be thought of as a degenerate scoring scheme $(1,0,-\infty)$ with infinite gap penalty. Hamming score gives the count of matching character pairs under a rigid alignment model, where every character must be aligned (resulting in either a match or a mismatch), and no gaps are allowed. ■

Alignment grid. The concept of LCS grid can be naturally extended to the alignment problem. In notation related to alignment under a general scoring scheme, we will use an upright serif typeface (e.g. G), in order to distinguish it from the LCS scoring scheme, for which we keep using an upright sans-serif typeface (e.g. G).

Definition 6.4 *Assuming a fixed scoring scheme, the* alignment grid $\mathrm{G}_{a,b}$ *is defined analogously to Definition 5.4, replacing the scores of match, mismatch and gap edges with* w_+, w_0, w_-*, respectively. The* prefix alignment problem*, the* prefix alignment matrix $\mathrm{L}_{a,b}$*, the* semi-local alignment problem *and its component subproblems (*string-substring alignment problem*, etc.), the* semi-local alignment matrix $\mathrm{H}_{a,b}$ *are defined analogously to Definitions 5.7, 5.11 and 5.13, replacing the LCS score by the alignment score.* □

Prefix alignment. It is straightforward to generalise Theorem 5.9 (Prefix LCS by classical dynamic programming) to alignment with a fixed scoring scheme.

Algorithm 6.5 (Prefix alignment by classical dynamic programming)

Parameters: scoring scheme $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$.

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

Output: prefix alignment matrix $\mathrm{L}_{a,b}$.

Description. The algorithm is analogous to Algorithm 5.8, replacing the LCS grid with the alignment grid $\mathrm{G}_{a,b}$, the prefix LCS matrix with the prefix alignment matrix $\mathrm{L}_{a,b}$, and the inner loop assignment with

$$L[\hat{l}^+;\hat{\jmath}^+] \leftarrow \max \begin{cases} L[\hat{l}^-;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ L[\hat{l}^-;\hat{\jmath}^+] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^+] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ L[\hat{l}^+;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^+;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \end{cases}$$

■

Theorem 6.6 *The prefix alignment problem can be solved in time $O(mn)$.*□

Proof Analogous to Theorem 5.9. ■

Assuming a rational scoring scheme, the running time can be improved by MBS to $O\big(\frac{mn}{\log^2 n}+n\big)$ for a constant-size alphabet, and to $O\big(\frac{mn(\log\log n)^2}{\log^2 n}+n\big)$ for an arbitrary alphabet.

Regular scoring. Instead of dealing with the three parameters of a general scoring scheme, it will be convenient to reduce them to just a single parameter.

Definition 6.7 *A scoring scheme $(1, \mathrm{w}_0, 0)$, where $0 \le \mathrm{w}_0 < 1$, will be called* regular. □

An arbitrary (finite) scoring scheme can be reduced to a regular one as follows (a similar method is used by Rice et al. [178]; see also [109, 125]). Let us consider first the global alignment of strings a, b, with an arbitrary scoring scheme $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$. This scheme can be replaced by a scheme $(\mathrm{w}_+ + 2x, \mathrm{w}_0 + 2x, \mathrm{w}_- + x)$ for any real x. Indeed, consider an arbitrary path $[0;0] \rightsquigarrow [m;n]$ in $\mathrm{G}_{a,b}$. Such a transformation increases the path's score by $(m+n)x$, which is independent of the path; therefore, the relative scores of different paths between the given endpoints do not change. The desired regular scoring scheme can be obtained by taking $x = -\mathrm{w}_-$, and then dividing each of the resulting scores by $\mathrm{w}_+ - 2\mathrm{w}_- > 0$:

$$(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-) \mapsto \big(1, \mathrm{w}_0^* = \tfrac{\mathrm{w}_0 - 2\mathrm{w}_-}{\mathrm{w}_+ - 2\mathrm{w}_-}, 0\big) \tag{6.1}$$

Let $\mathsf{score}^*(a,b)$ denote the alignment score under the regularised scheme, while $\mathsf{score}(a,b)$ will denote the score under the original scheme. The two scores are related by the following identities:

$$\mathsf{score}^*(a,b) = \tfrac{\mathsf{score}(a,b) - (m+n)\mathrm{w}_-}{\mathrm{w}_+ - 2\mathrm{w}_-} \tag{6.2}$$

$$\mathsf{score}(a,b) = \mathsf{score}^*(a,b) \cdot (\mathrm{w}_+ - 2\mathrm{w}_-) + (m+n) \cdot \mathrm{w}_- \tag{6.3}$$

Example 6.8 In Example 6.3, the LCS scoring scheme is already regular. The other two schemes can be regularised as follows:

$$(1, 0, -0.5) \mapsto (1, 0.5, 0) \qquad (2, -1, -1.5) \mapsto (1, 0.4, 0)$$

■

Edit distance. An important type of a scoring scheme is one that corresponds to the *edit distance problem*. In such a scheme, character matches are regarded as "neutral" with score zero, while the mismatches and the gaps are penalised.

Definition 6.9 *Let a, b be strings. Assuming a fixed scoring scheme $(0, \mathrm{w}_0, \mathrm{w}_-)$, where $2\mathrm{w}_- \leq \mathrm{w}_0 < 0$, their* edit distance *is $-\mathsf{score}(a, b) \geq 0$. Given strings a, b, the* edit distance problem *asks for their edit distance.* □

Equivalently, edit distance can be defined by considering a transformation of string a into string b by a sequence of *character edits*, each with an associated *cost*:

- $-\mathrm{w}_- > 0$ for a character *insertion* or *deletion* (*indel*);
- $-\mathrm{w}_0 > 0$ for a character *substitution*.

The edit distance between strings a, b is the minimum total cost of a sequence of character edits transforming a into b. Edit distance is a *metric:* it is nonnegative (zero on equal strings and positive otherwise), symmetric, and satisfies the triangle inequality.

A detailed treatment of the edit distance problem as a special case of the alignment problem is given e.g. in [121] and references therein.

Example 6.10 The *indel distance* (also called the *LCS distance* or *simple distance*) [167, 18, 31] has indel cost $-\mathrm{w}_- = 1$ and substitution cost $-\mathrm{w}_0 = 2$, making any substitution redundant, since it is no better than an insertion-deletion pair. The corresponding *indel scoring scheme* can be regularised to the LCS scoring scheme:

$$(0, -2, -1) \mapsto (1, 0, 0)$$

The *indelsub distance* (also called the *Levenshtein distance*) [147] has $-\mathrm{w}_- = -\mathrm{w}_0 = 1$. The corresponding *indelsub scoring scheme* can be regularised to the *Levenshtein scoring scheme*:

$$(0, -1, -1) \mapsto (1, 0.5, 0)$$

The classical *Hamming distance* counts the number of mismatches, while prohibiting any gaps: formally, it has $-\mathrm{w}_- = +\infty$, $-\mathrm{w}_0 = 1$, and corresponds to the scoring scheme $(0, -1, -\infty)$. It is equivalent to the Hamming scoring scheme $(1, 0, -\infty)$. ■

Definition 6.9 could be further generalised by allowing insertions and deletions to have two separate, distinct costs. For example, the asymmetric *episode distance* [70] corresponds to insertion cost 0, and strictly positive deletion and substitution costs.

All the results on string alignment presented in the rest of this work can be specialised to edit distance. We won't dwell on this fact, using the alignment terminology throughout.

6.2 Semi-local alignment

We will abbreviate "semi-local alignment matrix" to just *alignment matrix*, where that is justified by the context. The alignment matrix $\mathrm{H}_{a,b}$ is anti-Monge, but, in contrast with the LCS matrix, it may not be unit-anti-Monge.

We note that score regularisation, as described in the previous section, is compatible with the semi-local alignment problem: given strings a, b, and assuming an arbitrary scoring scheme, semi-local alignment scores can be obtained by regularising the scheme via (6.1), solving the problem assuming the regular scheme, and then reversing the regularisation of scores via (6.2). This is because for a fixed string-substring (prefix-suffix, etc.) pair, score regularisation does not affect the relative scores of different common subsequences. However, more care is required when comparing alignment scores across different string-substring (prefix-suffix, etc.) pairs. In such cases, the comparison's outcome may be affected by score regularisation, therefore the regularisation must be reversed prior to the comparison.

6.3 Alignment kernel

Canonical decomposition. The techniques that we developed in Sections 5.3 and 5.4 for the semi-local LCS problem can be partially extended to the semi-local alignment problem.

Theorem 6.11 (Alignment matrix canonical decomposition) *Let a, b be strings. Assuming a fixed scoring scheme, the semi-local alignment matrix $\mathrm{H}_{a,b}$ is anti-Monge. Furthermore, for a regular scoring scheme, $\mathrm{H}_{a,b}$ is anti-Σ-bistochastic:*

$$\mathrm{H}_{a,b}[i;j] = j - i - \mathrm{S}^{\Sigma}_{a,b}[i;j] = m - \mathrm{S}^{\exists}_{a,b}[i;j]$$

for all $i \in [-m\colon n]$, $j \in [0\colon m+n]$, where $\mathrm{S}_{a,b} = -\mathrm{H}^{\square}_{a,b}$ is a bistochastic matrix. □

Proof The anti-Monge property of $\mathrm{H}_{a,b}$ is proved analogously to Theorem 5.17.

Assume a regular scoring scheme $(1, \mathrm{w}_0, 0)$. First, consider a rational scheme: $\mathrm{w}_0 = \frac{\mu}{\nu}$. Then, the semi-local alignment problem on strings a, b can be reduced to the semi-local LCS problem by the following *blow-up* technique. We transform input strings a, b of lengths m, n into *blown-up* strings $\mathfrak{a}$, $\mathfrak{b}$ of lengths $\mathfrak{m} = \nu m$, $\mathfrak{n} = \nu n$. The transformation replaces every character γ by a substring $\$^{\mu}\gamma^{\nu-\mu}$ of length ν (recall that $\$$ is a special guard character, not present in the original strings). We have

$$\mathrm{H}_{a,b}[i;j] = \tfrac{1}{\nu} \cdot \mathsf{H}_{\mathfrak{a},\mathfrak{b}}[\nu i; \nu j] \tag{6.4}$$

for all $i \in [-m\colon n]$, $j \in [0\colon m+n]$, where the alignment matrix $\mathrm{H}_{a,b}$ is defined by the given scoring scheme on the original strings a, b, and the LCS matrix $\mathsf{H}_{\mathfrak{a},\mathfrak{b}}$ by the LCS scoring scheme on the blown-up strings $\mathfrak{a}$, $\mathfrak{b}$.

Every element of $\mathrm{S}_{a,b}$ can now be obtained as a scaled sum of a $\nu \times \nu$ block of $\mathsf{P}_{\mathfrak{a},\mathfrak{b}}$:

$$\mathrm{S}_{a,b}\langle \hat{\imath}; \hat{\jmath} \rangle = \tfrac{1}{\nu} \sum_{\langle \nu\hat{\imath}^- : \nu\hat{\imath}^+ ; \nu\hat{\jmath}^- : \nu\hat{\jmath}^+ \rangle} \mathsf{P}_{\mathfrak{a},\mathfrak{b}} \tag{6.5}$$

for all $\hat{\imath} \in \langle -m\colon n \rangle$, $\hat{\jmath} \in \langle 0\colon m+n \rangle$. By construction, matrix $\mathrm{S}_{a,b}$ is bistochastic.

A general regular scoring scheme can be approximated by a rational regular scheme to arbitrary precision. The property of matrix $\mathrm{S}_{a,b}$ to be bistochastic is preserved in the limit, therefore the theorem statement follows by compactness. ∎

We generalise Definition 5.19 as follows.

Definition 6.12 *Assuming a regular scoring scheme, the* semi-local alignment kernel *for strings a, b is the bistochastic matrix $\mathrm{S}_{a,b}\langle -m\colon n; 0\colon m+n \rangle$, determined by Theorem 6.11.* □

We will abbreviate "semi-local alignment kernel" to just *alignment kernel*, where that is justified by the context.

The alignment matrix $\mathrm{H}_{a,b}$ can be represented implicitly by (the nonzeros of) the alignment kernel $\mathrm{S}_{a,b}$. However, in contrast with the LCS kernel, the alignment kernel may not be a permutation matrix. In fact, it is typically dense, and therefore in general cannot provide a small semi-local alignment oracle that would be analogous to Theorem 5.21. Loosely speaking, an alignment kernel can still be represented by a sticky braid embedded in a grid, but both the braid and the grid will be finer and more fragmented than in the LCS case, and, when the scores are irrational, may not correspond to the character/cell boundaries in the LCS grid.

Rational alignment We now assume that the scoring scheme is rational, and use this to obtain an efficient semi-local alignment oracle.

Definition 6.13 *Let $\nu > 0$ be a natural number. A (sub)bistochastic matrix will be called* ν-(sub)bistochastic*, if every its element is an integer multiple of $1/\nu$.* □

A (sub)permutation matrix is 1-(sub)bistochastic, and a (sub)unit-Monge matrix is Σ-1-(sub)bistochastic. Note that in a ν-(sub)bistochastic matrix, there can be at most ν nonzeros in every row and every column.

Theorem 6.14 *Let a, b be strings. Assuming a regular rational scoring scheme with denominator ν, the alignment kernel $\mathrm{S}_{a,b}$ is ν-bistochastic, and the alignment matrix $\mathrm{H}_{a,b}$ is anti-Σ-ν-bistochastic.* □

PROOF Every element of $\mathrm{S}_{a,b}$ is the sum of a $\nu \times \nu$ block of the blown-up matrix $\frac{1}{\nu}\mathsf{P}_{\mathfrak{a},\mathfrak{b}}$ by (6.5), and is therefore a multiple of $1/\nu$. ∎

Theorem 6.15 *Assuming a regular rational scoring scheme with denominator ν, there exists a semi-local alignment oracle with*

- *size $O\big(\nu(m+n)\big)$ and query time $O(\nu s)$;*
- *size $O\big(\nu(m+n)\log(m+n)\big)$ and query time $O(\nu s)$*

where s is the size of the shorter string or substring in the query. □

PROOF Analogous to Theorem 5.21, replacing the LCS kernel with the alignment kernel. ∎

Example 6.16 Figure 6.1 shows semi-local alignment of strings a, b, using the regular indelsub scoring scheme $\mathrm{w}_+ = 1$, $\mathrm{w}_0 = 0.5$, $\mathrm{w}_- = 0$ (see Example 6.10).

Figure 6.1a shows the alignment grid $\mathrm{G}_{a,b}$. Match edges of score $\mathrm{w}_+ = 1$ and mismatch edges of score $\mathrm{w}_+ = 0.5$ are shown respectively by solid and dotted red lines. The highlighted path of score 5.5 corresponds to the string-substring alignment score for string a against substring $b\langle 4{:}\,11\rangle =$ "CABCABA".

Figure 6.1b shows the LCS grid $\mathsf{G}_{\mathfrak{a},\mathfrak{b}}$ for the blown-up strings $\mathfrak{a}$, $\mathfrak{b}$. We have $\mu = 1$, $\nu = 2$,

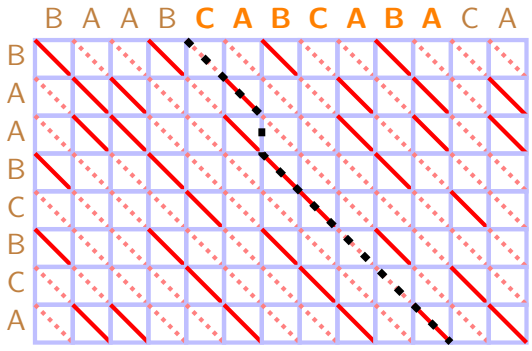
$\mathfrak{a} =$ "BAABCBCA"
$\mathfrak{b} =$ "BAABCABCABAC$A"

The highlighted path has the the same meaning as in Figure 5.3.

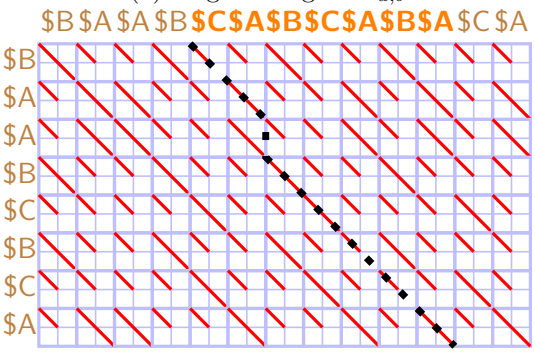
Figure 6.1c shows the LCS kernel $\mathsf{P}_{\mathfrak{a},\mathfrak{b}}$ for the blown-up strings as a reduced sticky braid of order $(8 + 13) \cdot 2 = 42$. Compared to the LCS kernel $\mathsf{P}_{a,b}$ for the original strings (Figure 5.5), the kernel's complexity has increased by a factor of $\nu^2 = 4$. ∎

Submatrix notation. We extend the submatrix definitions and notation of Section 5.3 to alignment matrices and kernels.

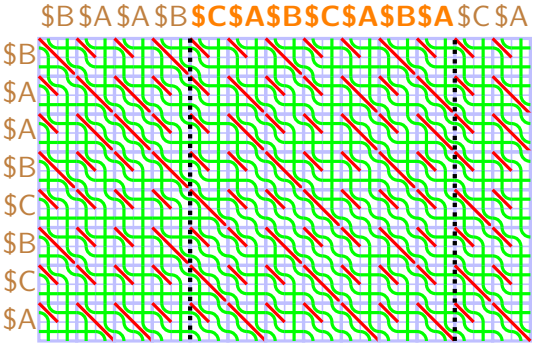
Definition 6.17 *The* suffix-prefix, (extended) substring-string, (extended) string-substring, prefix-suffix, and cross-semi-local alignment matrices, *as well as the respective* alignment kernels, *are defined and will be denoted analogously to Definition 5.26.* □



(a) Alignment grid $G_{a,b}$



(b) LCS grid $\mathsf{G}_{\mathfrak{a},\mathfrak{b}}$ for the blown-up strings



(c) LCS grid $\mathsf{G}_{\mathfrak{a},\mathfrak{b}}$; LCS kernel $\mathsf{P}_{\mathfrak{a},\mathfrak{b}}$ as a sticky braid

Figure 6.1: Semi-local alignment

6.4 Alignment kernel composition

In this section, we extend the kernel composition method of Section 5.4 from LCS kernels to alignment kernels. In the case of a rational scoring scheme with a constant denominator, we also show that the running time of the composition algorithms will not increase asymptotically, relative to the LCS scoring scheme.

We generalise Theorems 3.15 and 4.10 as follows.

Theorem 6.18 *Let $A \odot B = C$. If A, B are simple Σ-ν-bistochastic (respectively, Σ-ν-subbistochastic) matrices, then C is also a simple Σ-ν-bistochastic (respectively, Σ-ν-subbistochastic) matrix.* □

Proof Let $A = D^{\Sigma}$, $B = E^{\Sigma}$, where D, E are ν-(sub)bistochastic. We apply the blow-up technique as follows. It is straightforward to define a $\nu n \times \nu n$ (sub)permutation matrix P, such that

$$D\langle \hat{\imath}; \hat{\jmath} \rangle = \tfrac{1}{\nu} \sum_{\langle \nu\hat{\imath}^-:\nu\hat{\imath}^+;\nu\hat{\jmath}^-:\nu\hat{\jmath}^+ \rangle} P$$

for all $\hat{\imath}, \hat{\jmath} \in \langle 0\colon n; 0\colon n \rangle$, and a $\nu n \times \nu n$ (sub)permutation matrix Q, defined analogously with respect to E. Let $P \boxdot Q = R$, and let

$$F\langle \hat{\imath}; \hat{\jmath} \rangle = \tfrac{1}{\nu} \sum_{\langle \nu\hat{\imath}^-:\nu\hat{\imath}^+;\nu\hat{\jmath}^-:\nu\hat{\jmath}^+ \rangle} R$$

for all $\hat{\imath}, \hat{\jmath} \in \langle 0\colon n; 0\colon n \rangle$. It is straightforward to check that $D \boxdot E = F$, and that F is ν-(sub)bistochastic. ■

Theorem 6.19 *Let D, E, F be ν-(sub)bistochastic matrices with $O(n)$ nonzeros each. Let $D \boxdot E = F$. Given the nonzeros of D, E, the nonzeros of F can be computed in time $O(\nu n \log n)$.* □

Proof In the proof of Theorem 6.18, the sticky matrix multiplication can be performed by an algorithm similar to that of Theorem 4.16 (the Steady Ant algorithm), with $\log n$ recursion levels and sticky multiplication of $\nu \times \nu$ identity matrices as recursion base. Therefore, the running time is as claimed. ■

The LCS kernel composition method of Theorem 5.30 and Corollary 5.32 can now be generalised to alignment kernels, assuming a regular scoring scheme.

Theorem 6.20 *Assuming a regular scoring scheme, the string-substring (respectively, the semi-local) alignment kernel for strings $a = a'a''$, b can be obtained as a product*

$$\mathrm{S}^{\nearrow}_{a,b} = \mathrm{S}^{\nearrow}_{a',b} \boxdot \mathrm{S}^{\nearrow}_{a'',b} \tag{6.6}$$

$$\mathrm{S}_{a,b} = \mathrm{S}_{a',b} \boxdot_n \mathrm{S}_{a'',b} \tag{6.7}$$

Assuming furthermore a rational scoring scheme with denominator ν, this can be done in time $O(\nu n \log N)$ (respectively, $O\big(\nu(m + n \log N)\big)$), where $N = \min(m', m'', n)$. □

Proof Straightforward by Theorems 5.30, 6.14 and 6.19. ■

Corollary 6.21 *Assuming a regular scoring scheme, the extended string-substring alignment kernels (in particular, the cross-semi-local alignment kernel) for strings $a = a'a''$, b can be obtained as a product*

$$\mathrm{S}^{\nearrow(0)}_{a,b} = \mathrm{S}^{\nearrow(0)}_{a',b} \boxdot_n \mathrm{S}^{\nearrow(0)}_{a'',b} \tag{6.8}$$

$$\mathrm{S}^{\nearrow(m')}_{a,b} = \mathrm{S}^{\nearrow(m')}_{a',b} \boxdot \mathrm{S}^{\nearrow(0)}_{a'',b} \tag{6.9}$$

$$\mathrm{S}^{\nearrow(m)}_{a,b} = \mathrm{S}^{\nearrow(m)}_{a',b} \boxdot_n \mathrm{S}^{\nearrow(m)}_{a'',b} \tag{6.10}$$

Assuming furthermore a rational scoring scheme with denominator ν, each of these can be done in time $O(\nu n \log N)$. □

Proof Straightforward by Corollary 5.32 and Theorems 6.14 and 6.19. ■

6.5 Algorithms for semi-local alignment

The techniques developed in the previous sections allow us to extend semi-local LCS algorithms of Chapter 5 to semi-local alignment. It is sufficient to consider a regular scoring scheme, since for an arbitrary scheme, the semi-local alignment scores can be recovered by (6.2).

Explicit semi-local alignment. As discussed in Section 6.3, for a general scoring scheme an alignment kernel is typically not sparse. As such, there is no advantage in using it as an implicit representation of semi-local alignment scores, instead of an explicit alignment matrix.

For explicit semi-local alignment, an efficient solution is provided by treating it as a special case of the MSSP problem in a planar graph. An alternative divide-and-conquer solution can be obtained based on efficient distance multiplication of Monge matrices.

Theorem 6.22 *Assuming $m \le n$, the semi-local alignment problem can be solved in time $O(mn \log n)$.* □

Proof The semi-local alignment matrix $\mathrm{H}_{a,b}$ can be obtained in the claimed running time by applying Theorem 5.16 (MSSP in a planar graph) to the alignment grid $\mathrm{G}_{a,b}$.

Alternatively, matrix $\mathrm{H}_{a,b}$ can be obtained by following the recursive structure of Algorithm 5.33. Substring LCS kernels are replaced by explicit

alignment matrices, and sticky multiplication of LCS kernels is replaced by explicit distance multiplication of alignment matrices by Theorem 3.14.

In the algorithm's recursion tree, a given recursion level $\log r$ consists of $\frac{mn}{r^2}$ instances of $r \times r$ anti-Monge matrix distance multiplication. By Theorem 3.14, every such instance can be solved in time $O(r^2)$, therefore each recursion level runs in time $\frac{mn}{r^2} \cdot O(r^2) = O(mn)$. There are $O(\log n)$ recursion levels, therefore the total running time is $O(mn \log n)$. ∎

Implicit semi-local alignment. Assuming a rational scoring scheme with denominator ν, implicit semi-local alignment has been considered by Krusche [138, Section 6.2] and Matarazzo et al. [157], both giving an algorithm running in time $O(\nu mn)$. Their algorithms essentially are based on alignment kernel composition.

Theorem 6.23 *Assuming a rational scoring scheme with denominator ν, the semi-local alignment oracle of Theorem 6.15 can be obtained in time $O(\nu mn)$.* □

Proof Analogously to Theorems 5.35 and 5.38, replacing the LCS grid $\mathsf{G}_{a,b}$ with the alignment grid $\mathrm{G}_{a,b}$, and LCS kernels with alignment kernels. LCS kernel composition by Theorem 5.30 is replaced with alignment kernel composition by Theorem 6.20. ∎

Similarly to semi-local LCS, semi-local alignment can also benefit from micro-block speedup described in Section 5.6; however, this comes at a price of increasing the slowdown factor from ν to ν^2.

Theorem 6.24 *Assuming $m \leq n$ and a rational scoring scheme with denominator ν, the semi-local alignment oracle of Theorem 6.15 can be obtained in time $O\big(\frac{\nu^2 mn(\log\log n)^2}{(\log n)^2} + n\big)$.* □

Proof Analogously the proof of Theorem 6.11, we reduce the semi-local alignment problem on strings a, b of lengths m, n to the semi-local LCS problem on a pair of blown-up strings of lengths νm, νn. We then obtain a semi-local LCS oracle by Theorem 5.42. ∎

6.6 Approximate matching

Approximate matching is a natural generalisation of classical (exact) pattern matching, allowing for some character differences between the pattern and a matching substring of the text. Given a pattern string p of length m and a text string t of length $n \geq m$, approximate pattern matching asks for all the substrings of the text that are close to the pattern, i.e. those that have sufficiently high alignment score (or, equivalently, sufficiently low edit distance) against the pattern. Such substrings of the text will be called

matching substrings. The precise definition of "sufficiently high alignment score" (or "sufficiently low edit distance"), and therefore of a matching substring, may vary in different versions of the problem. Typically, matching substrings will correspond to a certain set of maxima (global, local, row, column etc.) in the string-substring score matrix $\mathrm{H}^{\nearrow}_{p,t}$.

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

Definition 6.25 *Given a pattern p and a text t, and assuming a fixed scoring scheme, the* complete AM problem *is defined as follows. For every prefix $t\langle 0\colon j\rangle$, the problem asks for the maximum alignment score of p against all possible choices of a suffix $t\langle i\colon j\rangle$ from this prefix. In terms of the LCS grid, the problem asks for multiple-destination maximum scores*

$$\begin{aligned}\mathrm{h}_{p,t}[j] &= \max_{i\in[0:j]} \mathrm{H}^{\nearrow}_{p,t}[i;j] = \max_{i\in[0:j]} \mathsf{score}\big(p, t\langle i\colon j\rangle\big)\\ &= \max_{i\in[0:j]} \mathsf{score}([0;i] \rightsquigarrow [l;j])\end{aligned}$$

where $j \in [0\colon n]$, and the inner score maxima are taken across all paths in $\mathsf{G}_{a,b}$ between the given endpoints. □

The complete AM problem can be solved by a classical dynamic programming algorithm due to Sellers [192] (see also [121]).

Algorithm 6.26 (Complete AM)

Parameters: scoring scheme $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$.

Input: pattern p of length m; text t of length n.

Output: complete AM scores for p against t.

Description. We solve the complete AM problem for every prefix of p against t. We construct matrix L incrementally, iterating through the cells of the alignment grid $\mathrm{G}_{p,t}$. We initialise

$$L[0;j] \leftarrow 0 \qquad L[l;0] \leftarrow l \cdot \mathrm{w}_-$$

for all $l \in [0\colon m]$, $j \in [0\colon n]$. We then iterate through the cells of $\mathrm{G}_{p,t}$ for all $\hat{l} \in \langle 0\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$ in lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the cells. In each iteration, we obtain a new element of matrix L by evaluating

$$L[\hat{l}^+;\hat{\jmath}^+] \leftarrow \max \begin{cases} L[\hat{l}^-;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big)\\ L[\hat{l}^-;\hat{\jmath}^+]\\ L[\hat{l}^+;\hat{\jmath}^-]\end{cases}$$

The final state of the matrix is returned as the algorithm's output:

$$\mathrm{h}_{a,b} \leftarrow L[l;:]$$

■

Theorem 6.27 *The complete AM problem can be solved in time $O(mn)$.* □

PROOF In Algorithm 6.26, each iteration runs in constant time. The overall running time is $mn \cdot O(1) = O(mn)$. ■

Assuming a rational scoring scheme with constant denominator, the running time of Algorithm 6.26 can be reduced by MBS to $O\big(\frac{mn}{\log^2 n} + n\big)$ for a constant-size alphabet, and to $O\big(\frac{mn(\log\log n)^2}{\log^2 n} + n\big)$ for an arbitrary alphabet. Various extensions of the problem have been considered by Cormode and Muthukrishnan [63] and many others (see e.g. a survey by Navarro [166] and references therein).

Assuming a rational scoring scheme, an algorithm for complete AM can also be obtained using the alignment kernel approach. We first generalise Theorem 4.9 as follows

Theorem 6.28 *Let $A[0\colon n_1; 0\colon n_2]$ be a Σ-ν-(sub)bistochastic matrix, canonically represented by the ν-(sub)bistochastic matrix $S = A^{\square}$ with $n \leq \nu \cdot \min(n_1, n_2)$ nonzeros, and implicit vectors $b = A[n_1; :]$, $c = A[:; 0]$, where random access to an element can be performed in time $O(1)$. Given S, b, c, the index of the leftmost minimum element in every row of A can be obtained in time $O(\nu n \log\log n)$ in the pointer machine model, and in time $O(\nu n)$ in the unit-cost RAM model.* □

PROOF The proof is similar to that of Theorem 4.9, replacing each update of the difference sequence by an increment of 1 with a sequence of ν updates by an increment of $1/\nu$. ■

For complete AM, we now have the following.

Theorem 6.29 *Assuming a rational scoring scheme with denominator ν, the complete AM problem can be solved in time $O(\nu mn)$.* □

PROOF Let $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$ be the scoring scheme. Without loss of generality, assume $\mathrm{w}_+ - 2\mathrm{w}_- = 1$ (otherwise, the scoring scheme can be scaled to satisfy this condition without affecting the correctness of the algorithm).

Consider the corresponding regular scoring scheme $(1, \mathrm{w}_0^* = \mathrm{w}_0 - 2\mathrm{w}_-, 0)$. The algorithm runs in two phases.

Phase 1. The alignment kernel $\mathrm{S}^*_{p,t}$ for the regular scheme is obtained in time $O(\nu mn)$ by Theorem 6.23. The kernel $\mathrm{S}^*_{p,t}$ is ν-bistochastic, and represents implicitly the alignment matrix $\mathrm{H}^*_{p,t}$, which is anti-Σ-ν-bistochastic.

Phase 2. Since complete AM compares alignment scores across substrings of different lengths, we now need to reverse score regularisation via Equation (6.2). We have

$$\mathrm{H}_{p,t}(i,j) = \mathrm{H}^*_{p,t}(i,j) + (m + j - i) \cdot \mathrm{w}_-$$

for all i, j. Furthermore, we have $\mathrm{H}_{p,t}^{\square} = (\mathrm{H}_{p,t}^{*})^{\square} = \mathrm{S}_{p,t}^{*}$, therefore the alignment matrix $\mathrm{H}_{p,t}$ is also anti-Σ-ν-bistochastic. Its string-substring submatrix $\mathrm{H}_{p,t}^{\nearrow}$ is anti-Σ-ν-subbistochastic, hence column maxima in this submatrix can be obtained by Theorem 6.28.

The total running time is dominated by the first phase of the algorithm, and is therefore as claimed. ∎

Theorem 6.30 *Assuming a rational scoring scheme with denominator ν, the complete AM problem can be solved in time $O\big(\frac{\nu^2 mn(\log\log n)^2}{\log^2 n} + n\big)$.* □

Proof Similarly to Theorem 6.29, replacing in Phase 1 the invocation of Theorem 6.23 with Theorem 6.24. ∎

Threshold AM. An alternative to complete AM is to identify the text substrings that match the pattern above a given similarity threshold.

Definition 6.31 *Given pattern p, text t, threshold h, and assuming a fixed scoring scheme, the* threshold AM problem *(often called simply "approximate matching") asks for all substrings of t that have alignment score at least h against p. Such substrings in t will be called* matching. □

The subsequence matching (SM) problem, considered in Section 5.7, is a special case of the threshold AM problem with the LCS scoring scheme $(1, 0, 0)$ and threshold $h = m$.

Another important special case of the threshold AM problem is the *threshold edit distance matching (EDM) problem*, which assumes an edit distance scoring scheme $(0, \mathrm{w}_0, \mathrm{w}_-)$, where $2\mathrm{w}_- \leq \mathrm{w}_0 < 0$, has a threshold $h \leq 0$, and asks for all substrings of t that have edit distance at most $k = -h > 0$ against p. The most well-studied case of the threshold EDM problem is *threshold Levenshtein matching* (also known as *matching with k differences*), with $\mathrm{w}_0 = \mathrm{w}_- = -1$. When the threshold k is low, the best known algorithm for threshold Levenshtein matching is by Cole and Hariharan [60], running in time $O\big(\frac{nk^4}{m} + n\big)$. For higher values of k, the best known algorithm is by Landau and Vishkin [145], running in time $O(nk)$.

In contrast with the SM problem, points corresponding to matching substrings in the threshold AM problem may not form an upset in the $\geqslant$-dominance partial order. Nevertheless, the problem's output is still typically redundant, and needs to be filtered analogously to Definition 5.46.

Definition 6.32 *Given pattern p, text t, threshold h, and assuming a fixed scoring scheme, the* minimal-substring threshold AM problem *asks for all inclusion-minimal matching substrings. Given a window length w, the* window threshold AM problem *asks for all matching substrings of length w.* □

Theorem 6.33 *Assuming a rational scoring scheme with denominator ν, the minimal-substring and the window threshold AM problems can both be solved in time $O(\nu mn)$.* □

PROOF The algorithm runs in two phases.

Phase 1. The shortest matching prefix for every suffix of t is obtained by Theorem 6.29.

Phase 2. Inclusion-minimal matching substrings of t are obtained by running through the list of these prefixes backwards, filtering out any that overlaps with the preceding one (obtained for a shorter suffix). All matching substrings of length w are obtained in time $O(m+n) = O(n)$, by running through the diagonal $j - i = w$, using incremental queries of Theorem 4.8 to verify whether the alignment score for p against $t\langle i\colon j\rangle$ equals the alignment score of p against the highest-scoring prefix of $t\langle i\colon\rangle$, in which case the substring $t\langle i\colon j\rangle$ is matching.

The total running time is dominated by the first phase of the algorithm, and is therefore as claimed. ∎

Theorem 6.34 *Assuming a rational scoring scheme with denominator ν, the minimal-substring and the window threshold AM problems can both be solved in time $O\big(\frac{\nu^2 mn(\log\log n)^2}{(\log n)^2} + n\big)$.* □

PROOF Similarly to Theorem 6.33, replacing in Phase 1 the invocation of Theorem 6.29 with Theorem 6.30. ∎

Chapter 7

Further string comparison problems

7.1 Incremental LCS

We now begin to explore the applications of the sticky braid method. First, we consider on-line string comparison. In this framework, rather than dealing with a pair of fixed input strings, we consider either one or both strings to be variable. These variable string(s) are subject to updates; several different models of string updating may be considered. An online string comparison problem asks for a comparison oracle that is both efficient for the current pair of input strings, and can be updated efficiently under input string updates.

We denote by a, b the current state of each input string, and by m, n their respective current size.

Appending characters. The simplest possible model of string updating is by appending a character at the end of an input string. The *incremental append LCS problem* asks for the current LCS score, subject to updates to both input strings. Algorithm 5.8 (prefix LCS) maintains an array of all prefix LCS scores, which can be efficiently updated whenever a character is appended (but not prepended) to either of the input strings. This provides an incremental append LCS oracle of size $O(m+n)$ with query time $O(1)$, and update time $O(n)$ (respectively, $O(m)$) per update of string a (respectively, b).

Appending/prepending characters. Another possible model for string updating is to allow both appending a character at the end, and prepending a character at the beginning of a string. The *semi-incremental append/prepend LCS problem* asks for the current LCS score, subject to updates to string b at either end. Landau et al. [144], and then Kim and Park

[185] gave oracles for this problem with query time $O(1)$, and update time $O(m)$ per update of string b.

The more general *incremental append/prepend LCS problem* asks for the current LCS score, subject to updates to both input strings at either end. Ishida et al. [119] gave an oracle for this problem with query time $O(1)$, and update time $O(n)$ (respectively, $O(m)$) per update of string a (respectively, b).

[[[check if parameterised]]]

[[[[57]: arbitrary insertions/deletions, update $O(n(\log n)^2)$. query $O(1)$

[[[Update rest]]]

We generalise the problem still further and consider the *incremental append/prepend semi-local LCS problem*, that asks for the current semi-local LCS scores, subject to updates to both input strings at either end. An efficient oracle for this problem can be obtained by the LCS kernel method as follows.

Theorem 7.1 *There exists an incremental append/prepend semi-local LCS oracle with size $O(m+n)$, query time $O(s)$, where s is the size of the shorter string or substring in the query, and update time $O(n)$ (respectively, $O(m)$) per update of string a (respectively, b).* □

Proof We extend Algorithm 5.36 (semi-local LCS by iterative combing) as follows. Whenever a new character is prepended or appended to string a (respectively, b), the LCS kernel $\mathsf{P}_{a,b}$ is updated by composing it with the LCS kernel for the new character against the opposite string. Such a composition can be computed by the algorithm of Theorem 5.30, modified so that the kernel update can be performed in-place in time $O(n)$ (respectively, $O(m)$). Having computed the product, it is also straightforward to update the value of the LCS in the same asymptotic running time. Therefore, the overall running time per update is $O(n)$ (respectively, $O(m)$) per update of string a (respectively, b). ■

7.2 Semi-local LCS on block strings

To observe another natural application of the LCS kernel method, let us consider the following generalisation of string comparison. Suppose that string a is a *block string*, composed of substrings taken from a pre-specified set of *admissible blocks* (for example, a list of frequently occurring words in a text). The set of admissible blocks is known in advance, and off-line preprocessing of the blocks against input string b is allowed.

Block decomposition of strings is very common in string compression. In this section, we are only dealing with the most general case of such decomposition; we will consider specific string compression models in Chapter 10.

We denote by $\bar{m}$ the number of blocks in a block string a, while keeping the notation m for its total number of characters. A block may consist of a

single character, therefore this setup generalises ordinary string comparison. On the other hand, a block may be long; in such a case, our goal is to process the block quickly, ideally in the same time as it takes to process a single character in string a.

We consider the LCS and the semi-local LCS problems on a block on a block string a against an ordinary string b. We can solve both problems naively, ignoring the block structure of string a. By Theorems 5.9, 5.35 and 5.38, the LCS score and a semi-local LCS oracle for strings a, b can be obtained in time $O(mn)$.

Landau and Ziv-Ukelson [146] (see also [66]) considered a special case of the LCS problem on a block string a against a plain string b, where there is only a single non-trivial admissible block of length l, called *common substring*. Their algorithm preprocesses the common substring against string b in time $O(nl)$, and then solves the LCS problem for a, b in time $O(\bar{m}n)$.

An efficient solution to the LCS and the semi-local LCS problems on a block string can be obtained by the LCS kernel method as follows.

Theorem 7.2 *Given a set S of admissible blocks of total length u for string a, and a plain string b, this input can be preprocessed in time $O(nu)$, so that the LCS problem can be solved in time $O(\bar{m}n)$, and the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O(\bar{m}n \log v + m)$, where v is maximum block length.* □

Proof The preprocessing and the processing phases of the algorithm are as follows.

Preprocessing. Let c be an admissible block of length l. We preprocess c by computing the semi-local LCS kernel $\mathsf{P}_{c,b}$; by Theorems 5.35 and 5.38, this can be done in time $O(nl)$. The overall preprocessing time is therefore $O(nu)$.

Obtaining the LCS score. Let $h_{a,b}[0{:}\,n] = \mathsf{H}^{\nearrow}_{a,b}[0;:]$. We compute this vector incrementally as follows. Suppose we have computed the vector $h_{a',b}$, and let c be an admissible block. Then we have $h_{a'c,b} = h_{a',b} \rtimes \mathsf{P}_{c,b}$, where matrix $\mathsf{P}_{c,b}$ has been obtained in the preprocessing phase. This vector-matrix sticky product can be computed by Theorem 4.15 in time $O(n)$. The overall processing time is $O(\bar{m}n)$.

Obtaining the semi-local LCS oracle. We obtain the LCS kernel $\mathsf{P}_{a,b}$ incrementally as follows. Suppose we have computed the LCS kernel $\mathsf{P}_{a',b}$, and let c be an admissible block. Then we have $\mathsf{P}_{a'c,b} = \mathsf{P}_{a',b} \boxdot \mathsf{P}_{c,b}$ and $\mathsf{P}_{ca',b} = \mathsf{P}_{c,b} \boxdot \mathsf{P}_{a',b}$, where, as before, the LCS kernel $\mathsf{P}_{c,b}$ has been obtained in the preprocessing phase. This kernel composition can be computed by Theorem 5.30 in time $O(n \log l + l)$, where l is the length of block c. The overall processing time is $O(\bar{m}n \log v + m)$. ■

Theorem 7.2 can be generalised in a straightforward way to a setup where both input strings are block strings.

Theorem 7.3 *Given sets S_1, S_2 of admissible blocks of total lengths u_1, u_2 for strings a, b respectively, this input can be preprocessed in time $O(u_1u_2)$, so that the LCS problem can be solved in time $O(\bar{m}n + m\bar{n})$, and the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O\big((\bar{m}n + m\bar{n}) \cdot \log v\big)$, where v is maximum block length.* □

Proof Analogous to Theorem 7.2. ■

7.3 Window and cyclic LCS

Window LCS. We consider the following natural problem.

Definition 7.4 *Given a string and a fixed parameter w, we call a substring of length w a w-*window *in the string. Given strings a, b, and a window length w, the* window LCS problem *asks for the LCS score of the whole string a against every w-window in string b.* □

The LCS kernel method provides a simple efficient algorithm for the window LCS problem.

Theorem 7.5 *The window LCS problem can be solved in time $O(mn)$; using MBS, the running time can be reduced to $O\big(\frac{mn(\log\log n)^2}{\log^2 n} + n\big)$.* □

Proof The algorithm runs in two phases.

Phase 1. We run Algorithm 5.36 (semi-local LCS by iterative combing) or Algorithm 5.41 (semi-local LCS by iterative combing with MBS) on strings a, b, obtaining the LCS kernel $\mathsf{P}_{a,b}$.

Phase 2. We perform $m-w+1$ string-substring LCS score queries for whole a against every w-window of b. This can be done efficiently by Theorem 4.8 as a diagonal batch query.

Running time analysis. The overall running time is dominated by Phase 1. By Theorems 5.35, 5.38 and 5.42, the running time is as claimed. ■

Cyclic LCS. The window LCS problem is closely related to the following problem.

Definition 7.6 *Given strings a, b of length m, n respectively, the* cyclic LCS problem *asks for the maximum LCS score of a against all cyclic shifts of b (equivalently, all cyclic shifts of a against b, or all cyclic shifts of both strings against each other).* □

Cyclic string comparison has been considered by Maes [151], Bunke and Bühler [44], Landau et al. [144], Schmidt [190], Marzal and Barrachina [154]. The algorithms of [144, 190] solve the cyclic LCS problem in worst-case time $O(mn)$.

An algorithm for the cyclic LCS problem can be obtained by the LCS kernel method based on Theorem 7.5, improving on the existing algorithms by the micro-block speedup.

Theorem 7.7 *The cyclic LCS problem can be solved in time $O(n^2)$; using MBS, the running time can be reduced to $O\big(\frac{n^2(\log\log n)^2}{\log^2 n}\big)$.* □

Proof The algorithm runs in two phases.

Phase 1. We solve the window LCS problem by Theorem 7.5 on string a against string bb (a concatenation of string b with itself), which is of length $2n$, with window size $w = n$.

Phase 2. We take the maximum LCS score across all the windows.

Running time analysis. The running time is dominated by Phase 1, which runs in time $O(mn)$. Using micro-block speedup, the running time can be reduced to $O\big(\frac{mn(\log\log n)^2}{\log^2 n} + n\big)$. ■

7.4 String-substring LCS on periodic strings

Strings with periodic (or approximately periodic) structure play an important role in both the theory and the applications of string algorithms. In particular, a variant of periodic string comparison is a key subroutine in threshold Levenshtein matching algorithm by Cole and Hariharan [60], which is the fastest known algorithm for low values of the threshold. In computational molecular biology, approximately periodic substrings of a genome are known as *tandem repeats*, and are crucial for efficient genome analysis (see e.g. Schmidt [190] and references therein).

Definition 7.8 *Let b be a string of length n. A* (finite) k-repeat *of b is the string $b^k = bb\ldots b$ (b repeated k times) of length nk. An* infinite repeat *of b is the string $b^\infty = \ldots bbb\ldots$, infinite in both directions. A string that is a k-repeat of b for any k is called* periodic*, and string b is called its* period*. A 2-repeat is called a* square*.* □

LCS on plain pattern vs periodic text. Consider the LCS problem on a plain pattern p against a finite periodic text t^k. The problem can be solved naïvely as the LCS problem on string p of length m against string t^k of length nk, ignoring the periodic structure of the latter. By Theorem 5.9, the resulting algorithm runs in time $O(mnk)$; this running time can be reduced slightly by MBS.

This problem can also be regarded as a special case of the common-substring LCS problem. By Theorem 7.2, it can be solved in time $O\big(m(k+n)\big)$. Landau et al. [66, 143] gave an algorithm parameterised by the LCS score of the input strings; however, the worst-case running time of this

algorithm is still $O\big(m(k+n)\big)$. Landau [**?**] asked if the running time can be improved to $O\big(m(\log k+n)\big)$. In the rest of this section, we show that by generalising the problem to string-substring LCS, and by considering an infinite repeat text, it is possible not only to improve on these algorithms, but also to match and even to exceed Landau's conjecture.

String-substring LCS on plain pattern vs periodic text. Consider the string-substring LCS problem on a plain pattern p against a finite or infinite periodic text t^k (respectively, t^∞). For a finite k, the problem can be solved naïvely as the string-substring LCS problem on string p of length m against string t^k of length nk, ignoring the periodic structure of the latter. By Theorems 5.35 and 5.38, an oracle of size nk for this problem can be obtained in time $O(mnk)$; this running time can be reduced slightly by MBS.

Without loss of generality, we may assume that every character of p occurs in the text period t at least once. Furthermore, we only need to consider text substrings of length at most mn: any rotation of string t^m contains the whole p as a subsequence, since every character of p could be matched to a different period. Therefore, the problem for any finite and infinite k is equivalent to the string-substring LCS problem on plain string p against block string t^m, with a single admissible block t of length n. By Theorem 7.2, an oracle of size mn for this problem can be obtained in time $O(mn+m^2\log n)$.

By exploiting more carefully the periodic structure of the text, it is possible to construct a simpler, smaller oracle with a more efficient algorithm.

Theorem 7.9 *Given a pattern p and a text period t, there exists a string-substring LCS oracle for p against the infinite repeat t^∞ (and therefore also against any finite repeat t^k) with*

- *size $O(n)$ and query time $O(s)$;*
- *size $O\big(n\log n\big)$ and query time $O(\log^2 s)$*

where s is the length of the substring in the query. □

Proof The definition of the LCS grid (Definition 5.4) extends naturally to the periodic LCS problem. The LCS grid G_{p,t^∞} is itself infinite and *periodic*: all vertical and horizontal edges have score 0, and each pair of diagonal edges

$$[\hat{l}^-;\hat{\imath}^-]\to[\hat{l}^+;\hat{\imath}^+] \qquad [\hat{l}^-;\hat{\imath}^-+n]\to[\hat{l}^+;\hat{\imath}^++n]$$

have equal scores for all $\hat{l}\in\langle 0{:}\,m\rangle$, $\hat{\imath}\in\langle -\infty{:}\,+\infty\rangle$. Such an infinite LCS grid can also be regarded as a horizontal composition of an infinite sequence of *period subgrids*, each isomorphic to the $m\times n$ LCS grid $\mathsf{G}_{p,t}$.

String t^∞ is infinite, and has no finite prefixes or suffixes. Therefore, the semi-local LCS matrix and kernel can be understood as just the respective infinite string-substring LCS matrix and kernel:

$$\mathsf{H}_{p,t^\infty}[-\infty{:}+\infty] = \mathsf{H}^{\nearrow}_{p,t^\infty}[-\infty{:}+\infty]$$
$$\mathsf{P}_{p,t^\infty}\langle-\infty{:}+\infty\rangle = \mathsf{P}^{\nearrow}_{p,t^\infty}\langle-\infty{:}+\infty\rangle$$

Furthermore, these matrices are themselves *periodic*: we have

$$\mathsf{H}_{p,t^\infty}[i;j] = \mathsf{H}_{p,t^\infty}[i+n;j+n]$$
$$\mathsf{P}_{p,t^\infty}\langle\hat{\imath};\hat{\jmath}\rangle = \mathsf{P}_{p,t^\infty}\langle\hat{\imath}+n;\hat{\jmath}+n\rangle$$

for all $i,j \in [-\infty{:}\infty]$, $\hat{\imath},\hat{\jmath} \in \langle-\infty{:}\infty\rangle$. To represent such matrices, it is sufficient to store n nonzeros of the LCS kernel P_{p,t^∞}, each representing an infinite family of its periodic translates. In particular, we can store the n nonzeros of either

- the $n \times \infty$ *row-period subkernel* $\mathsf{P}_{p,t^\infty}\langle 0{:}\,n;:\rangle$
- the $\infty \times n$ *column-period subkernel* $\mathsf{P}_{p,t^\infty}\langle:;0{:}\,n\rangle$

Either of these subkernels can be used as the first oracle claimed by the theorem, by adapting the dominance counting procedure so that each nonzero of the subkernel is counted with an appropriate multiplicity. The two subkernels can also be obtained from one another in time $O(n)$.

The second oracle is obtained from the first analogously to Theorem 5.21.■

The string-substring LCS problem on a plain pattern against a periodic text can now be solved by a variant of Algorithm 5.36 (Semi-local LCS by iterative combing), where an infinitely wide periodic sticky braid is traced within a single period subgrid, wrapping around at the subgrid's boundaries.

Algorithm 7.10 (String-substring LCS: Plain pattern vs periodic text)

Input: plain pattern p; text period t.

Output: a string-substring LCS oracle for p against t^∞.

Description. Similarly to Algorithm 5.36 (semi-local LCS by iterative combing), we start with a generally unreduced sticky braid on the LCS grid G_{p,t^∞}, obtained by composition of elementary sticky braids for individual cells. We now need to comb this braid in order to obtain an equivalent reduced sticky braid. Note that we only need to maintain n strands, corresponding to the row-period subkernel $\mathsf{P}_{p,t^\infty}\langle 0{:}\,n;:\rangle$; every such strand will represent an infinite periodic family of strands.

In contrast with Algorithm 5.36, we are no longer able to iterate over the cells of the LCS grid G_{p,t^∞} in an order compatible with $\ll$-dominance,

since there are no $\ll$-minimal cells to begin from. Instead, we iterate over the cells in the following special order. In the outer loop, we iterate over the rows of cells top-to-bottom. For each current row $\hat{l} \in \langle 0\colon m\rangle$, we start the inner loop at an arbitrary match cell, i.e. at an index $\hat{\imath}_0 \in \langle 0\colon n\rangle$, such that $a\langle\hat{l}\rangle = b\langle\hat{\imath}\rangle$. Such an index $\hat{\imath}_0$ is guaranteed to exist by the assumption that every character of a occurs in b at least once. Then, starting from $\hat{\imath} = \hat{\imath}_0$, we iterate over the cells of the current row left-to-right, wrapping around from $\hat{\imath} = n^-$ to $\hat{\imath} = 0^+$, and continuing over the same row left-to-right up to $\hat{\imath} = \hat{\imath}_0$.

For each cell, we consider the two strands passing through it. The new layout for the two strands within the current cell is decided similarly to Algorithm 5.36. In a match cell, we always leave the strands uncrossed. In a mismatch cell, we leave the strands crossed, if and only if this pair of strands have never crossed previously (in terms of the $\ll$-dominance order) in the LCS grid. However, if the two strands have crossed previously, we undo ("comb away") their crossing in the current cell. We then move on to the next cell in the iteration order.

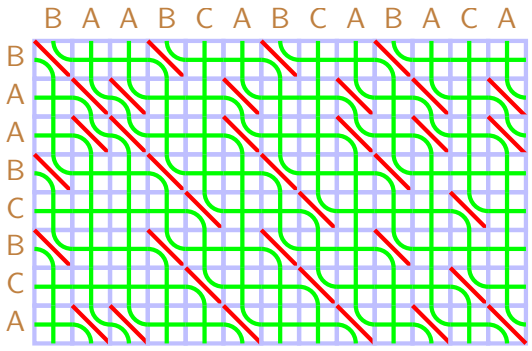
In order to perform the crossing check efficiently, we maintain, similarly to Algorithm 5.36, the starting index of every strand. This starting index will always be at the (infinite) top boundary of the grid G_{p,t^∞}. However, since the strands enter and leave the current period subgrid during the iteration, a strand's starting index may not necessarily belong to the current subgrid. The starting indices of all the strands in the current period subgrid can still be maintained efficiently as follows. As the iteration over the current row wraps around, a strand leaves the current period subgrid at the right boundary of the cell with $\hat{\imath} = n^-$. This strand is replaced by a strand from the same periodic family entering the current period subgrid at the left boundary of the cell with $\hat{\imath} = 0^+$. The starting index of the entering strand can be obtained by subtracting the period length n from the starting index of the leaving strand. The crossing check for a pair of strands is performed by comparing their starting indices, similarly to Algorithm 5.36.

The correctness of the combing procedure is implied, similarly to Algorithm 5.36, by Theorem 5.30. The resulting sticky braid is reduced, and provides us with the nonzeros of the column-period subkernel $\mathsf{P}_{p,t^\infty}\langle :\,; 0\colon n\rangle$. ■

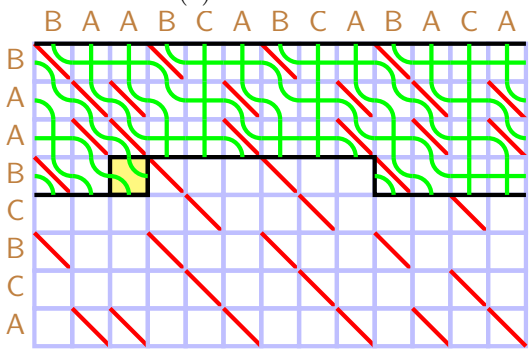
Theorem 7.11 *The string-substring LCS oracle of Theorem 7.9 can be obtained in time $O(mn)$ by Algorithm 7.10.* □

Proof In Algorithm 7.10, similarly to Algorithm 5.36, a strand crossing check and a cell update both run in time $O(1)$. Therefore, the overall running time of the algorithm is $O(mn)$ as claimed. ■

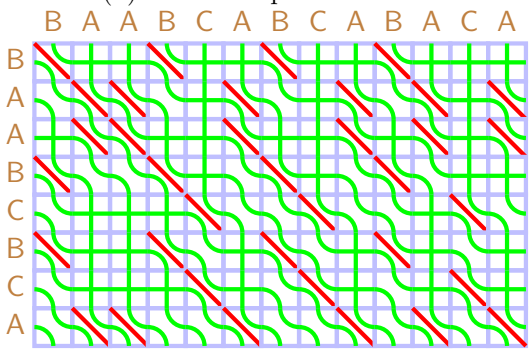
Example 7.12 Figure 7.1 shows the execution of Algorithm 7.10, using the same conventions as Figure 5.8. The sticky braid is embedded in a period



(a) Initial state



(b) State at a particular cell



(c) Final state

Figure 7.1: Execution of Algorithm 7.10 (String-substring LCS: plain pattern vs periodic text)

subgrid $\mathsf{G}_{p,t}$; each strand leaving the subgrid on the right is replaced by a strand from the same periodic family entering into the same horizontal row of the subgrid on the left. Note that, although the two strands meeting in the highlighted current cell do not cross in the current period subgrid, they have both arrived from the previous period subgrid, where they did cross. Therefore, their crossing has been undone in the current cell. Also note the difference between the final states in Figure 7.1 on one hand, and **??** and Figures 5.8 and 5.9 on the other hand. ■

In contrast with Algorithm 5.36, the extra data dependencies caused by the wraparound and the resulting restrictions of the cell iteration order seem to rule out the possibility of a micro-block speedup for Algorithm 7.10.

Approximate matching on plain pattern vs periodic text. A family of approximate matching problems for a periodic text was introduced by Benson [29].

Definition 7.13 *Given a plain pattern p and a text period t, the* periodic ApxMatch *problem asks for a substring of text t^∞ that is closest to t in terms of alignment score, under different restrictions on the substring. In particular:*

- *the* strict periodic ApxMatch problem *restricts the substring of t^∞ to consist of a whole number k of copies of t, for an arbitrary integer $k \geq 0$; hence, the substring is of the form t^k;*
- *the* window periodic ApxMatch problem *restricts the substring of t^∞ to be of length kn for an arbitrary integer $k \geq 0$; hence, the substring is of the form u^k, where u is a cyclic shift of t;*
- *the* free periodic ApxMatch problem *leaves the substring of t^∞ unrestricted.* □

All three versions of the periodic ApxMatch problem can be approached naïvely as special cases of Definition 6.25 (the complete ApxMatch problem) on pattern p of length m against text t^m of length mn, ignoring the periodic structure of the latter. Given an arbitrary scoring scheme, Algorithm 6.26 solves the problem in time $O(m \cdot mn) = O(m^2 n)$. Assuming a rational scoring scheme, the running time can be improved slightly by MBS.

The strict and the free periodic ApxMatch problems can be solved more efficiently by the technique of *wraparound dynamic programming* [162, 90] (see also [29]) in time $O(mn)$. For the window periodic ApxMatch problem, Benson [29] modified this technique to give an algorithm running in time $O(mn \log n)$.

Assuming a rational scoring scheme, the periodic ApxMatch problem can be solved by the alignment kernel method as follows.

Theorem 7.14 *Assuming a rational scoring scheme with denominator ν, the strict, window and free periodic ApxMatch problems can each be solved in time $O(\nu mn)$.* □

PROOF A straightforward generalisation of Theorems 6.29 and 7.11. ■

The algorithm of Theorem 7.14 matches the running time of the respective algorithms of [29] for strict and free periodic matching, and improves on them by a factor of $O(\log n)$ for window periodic matching.

7.5 Longest square subsequence

Definition 7.15 *Given a string a of length n, the* longest square subsequence problem *asks for the length of the longest subsequence of a that is a square.* □

This problem has been considered (under a different name) by Kosowski [136], who gave an algorithm running in time $O(n^2)$.

Using the LCS kernel method, we obtain a new algorithm for the longest square subsequence problem, improving on the existing algorithm by the micro-block speedup.

Theorem 7.16 *The longest square subsequence problem can be solved in time $O(n^2)$; using MBS, the running time can be reduced to $O\big(\frac{n^2(\log\log n)^2}{\log^2 n}\big)$.* □

PROOF The algorithm runs in two phases.

Phase 1. We run Algorithm 5.36 (semi-local LCS by iterative combing) or Algorithm 5.41 (semi-local LCS by iterative combing with MBS) on string a against itself, obtaining the semi-local LCS kernel $\mathsf{P}_{a,a}$.

Phase 2. we perform $n-1$ prefix-suffix LCS queries for every possible non-trivial prefix-suffix decomposition of a. This can be done efficiently by Theorem 4.8 as a diagonal batch query. We take the maximum score among these queries.

Running time analysis. The overall running time is dominated by Phase 1. By Theorems 5.35, 5.38 and 5.42, the running time is as claimed. ■

Using a similar approach, Theorem 7.16 can be generalised to finding the longest periodic subsequence that is a q-repeat for any rational q, $1 \le q \le 2$.

Chapter 8

Sparse string comparison

8.1 LCS on permutation strings (LCSP)

Recall that a permutation string is a string where all the characters are distinct. Let both input strings a, b be permutation strings. We assume that the strings' characters are elements of a given totally ordered alphabet of size n. Without loss of generality, we also assume that $m = n$.

Definition 8.1 *The* LCS problem on permutation strings *(the* LCSP problem*) is the LCS problem, specialised to inputs that are both permutation strings. The* prefix LCSP problem *and the* semi-local LCSP problem *are defined analogously.* □

The LCSP problem is equivalent to the following classical problem.

Definition 8.2 *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. We will call this length the* LIS score *of string a.* □

In general, Definition 8.2 does not require the input string a to be a permutation string. The LIS problem on string a can be solved as the LCSP problem on string a against the identity string id, while the LCSP problem can be reduced to the LIS problem by first sorting the alphabet and removing any repeated characters from a.

Prefix LCSP. The LCSP/LIS problems are closely related to geometric dominance on the plane. In this chapter, we assume the $\ll$-dominance relation by default, writing "chain" for "$\ll$-chain", etc. The prefix LCSP problem corresponds to the following classical problem.

Definition 8.3 *Let X be a set of points in the plane. The* height *of a point $x \in X$ is the cardinality of the longest chain in the subset of points in X that are dominated by x. Given X, the* canonical antichain partition (CAP) problem *asks for the height of every point in X.* □

In particular, every minimal point in X has height 0; the set X_0 of all such points forms an antichain in X. Furthermore, every minimal point in $X \setminus X_0$ has height 1 in X; the set X_1 of all such points forms another antichain in X. It is straightforward to show that the solution to the CAP problem provides a partitioning of X into the minimum possible number of antichains. By Dilworth's theorem, this number is equal to the length of the longest chain in X.

The LIS problem has a long history, going back to Erdös and Szekeres [84] and Robinson [181]. Based on their ideas, a classical LIS algorithm running in time $O(n \log n)$ was made explicit by Knuth [134], Fredman [92] and Dijkstra [79]. The problem was studied further by Chang and Wang [55], Felsner and Wernisch [88], Bespamyatnikh and Segal [32]. Crochemore and Porat [69] gave a LIS algorithm on an integer alphabet, running in time $O(n \log\log \lambda)$, where λ is the input strings' LIS score. The CAP problem is also known in literature as *greedy cover* [**?**, 78], *patience sorting* [32], or *nondominated sorting* [**?**]; the antichains of the canonical partitioning are also known as *layers of minima/maxima* [43, 38], *Pareto fronts* [49, 50], or *terraces* [153]. This problem is a special case of the *Robinson–Schensted–Knuth correspondence* between permutations and ordered pairs of Young tableaux.

Based on the classical LIS algorithm, the CAP problem can be solved as follows.

Algorithm 8.4 (CAP)

Input: set X of points of cardinality n.

Output: the height of each point in X.

Description. Without loss of generality, we assume that in each dimension, the points' coordinates are unique. We iterate through points of X in order of their first coordinate, assigning the height to every point in turn. Let i be the number of points processed so far, and let r be the number of different heights (in the range $[0\colon r-1]$) assigned so far. For every $h \in [0\colon r-1]$, let $\mathit{tail}[h]$ be the second coordinate of the most recently processed point of height h. Note that the array tail must necessarily be in increasing order, containing the longest chain of points among all processed so far; each value $\mathit{tail}[h]$ represents the $\geqslant$-maximal point in the canonical $\ll$-antichain of height h (hence the array's name). We initialise

$$i \leftarrow 0 \qquad r \leftarrow 0 \qquad \mathit{tail}[0] \leftarrow +\infty$$

In each iteration, let t be the current point's second coordinate. We assign

$$\mathit{height}[i] \leftarrow \min\{h \in [0\colon r], t < \mathit{tail}[h]\}$$
$$\mathit{tail}[\mathit{height}[i]] \leftarrow t$$
$$\text{if } \mathit{height}[i] = r \text{ then } \{r \leftarrow r+1;\ \mathit{tail}[r] \leftarrow +\infty\}$$

$i \leftarrow i + 1$ ■

Theorem 8.5 *The CAP and the LIS problems can both be solved in time $O(n \log n)$ by Algorithm 8.4.* □

Proof Since array *tail* is maintained in increasing order, operator min can be evaluated by binary search in time $O(\log r) = O(\log n)$. The overall running time is $n \cdot O(\log n) = O(n \log n)$. Upon the algorithm's execution, array *tail* contains the second coordinates of the points in a longest $\ll$-chain.

Given a string a, the LIS problem is a special case of the CAP problem on set $X = \{\langle \hat{\imath}; a\langle\hat{\imath}\rangle\rangle, \hat{\imath} \in \langle 0{:}\, n\rangle\}$, and can therefore be solved in the same asymptotic time. Upon the algorithm's execution, array *tail* contains the longest increasing subsequence of string a. ■

8.2 Semi-local LCSP

The semi-local LCSP problem is equivalent to a local version of the LIS problem.

Definition 8.6 *Given a string a, the* local LIS problem *asks for the LIS score in every substring of a.* □

Similarly to Definition 8.2, Definition 8.6 does not require the input string a to be a permutation string. The equivalence between the LIS and LCSP problems described in Section 8.1 extends to the local LIS and the semi-local LCSP problem.

The semi-local LCSP problem can be solved by adapting Algorithm 5.33 (semi-local LCS by recursive combing) to permutation strings.

Algorithm 8.7 (Semi-local LCSP by sparse recursive combing)

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

Output: a semi-local LCS oracle for a against b.

Description. Recursion on the input strings. The LCS kernel $\mathsf{P}_{a,b}$ is obtained by partitioning string a into two substrings, calling the algorithm recursively to obtain LCS kernels for each substring of a against the relevant subsequence of the other string, and then composing the two resulting LCS kernels.

Recursion base: $n = 1$. The LCS grid $\mathsf{G}_{a,b}$ consists of a single cell. Its LCS kernel is the 2×2 elementary kernel $\mathsf{P}_{a,b} \in \{\mathsf{I}, \mathsf{I}^{\mathsf{R}}\}$.

Recursive step: $n > 1$. Assume without loss of generality that n is even. Let $a = a'a''$, where strings a', a'' are of length $\frac{n}{2}$. Each of the substrings a', a'' is a permutation string over an alphabet of size $\frac{n}{2}$.

Let $I' = \{\hat{\imath} \mid b\langle\hat{\imath}\rangle \text{ occurs in } a'\}$, $I'' = \{\hat{\imath} \mid b\langle\hat{\imath}\rangle \text{ occurs in } a''\}$. We have $\langle 0{:}\, n\rangle = I' \sqcup I''$, where $\sqcup$ denotes the disjoint union of sets. Note that for all $\hat{\imath} \in I'$ (respectively $\hat{\imath} \in I''$), we have $\mathsf{P}_{a',b}\langle\hat{\imath};\hat{\imath}\rangle = 1$ (respectively, $\mathsf{P}_{a'',b}\langle\hat{\imath};\hat{\imath}\rangle = 1$). Therefore, the non-contiguous submatrix of $\mathsf{P}_{a',b}$ (respectively, $\mathsf{P}_{a'',b}$) formed by the intersection of rows and columns indexed by I' (respectively, I'') is equal to the $\frac{n}{2} \times \frac{n}{2}$ identity matrix I.

Let $b' = b|_{I'}$. We call the algorithm recursively on strings a', b', each of length $\frac{n}{2}$, obtaining the semi-local LCS kernel $\mathsf{P}_{a',b'}$. Then, the LCS kernel $\mathsf{P}_{a',b}$ is obtained by inserting a set of $\frac{n}{2}$ rows and $\frac{n}{2}$ columns into the LCS kernel $\mathsf{P}_{a',b'}$ at the indices in I', which corresponds to reinserting the $\frac{n}{2}$ deleted characters of string b into string b'. The newly inserted rows and columns are filled with (implicit) zeros and ones, so that they form a $\frac{n}{2} \times \frac{n}{2}$ non-contiguous submatrix equal to the identity matrix I.

Symmetrically, let $b'' = b|_{I''}$. The LCS kernel $\mathsf{P}_{a'',b}$ is obtained by calling the algorithm recursively on strings a'', b'', and then inserting a set of $\frac{n}{2}$ rows and $\frac{n}{2}$ columns into the resulting LCS kernel at the indices in I'', so that they form a $\frac{n}{2} \times \frac{n}{2}$ non-contiguous identity submatrix.

Given the LCS kernels $\mathsf{P}_{a',b}$, $\mathsf{P}_{a'',b}$, the output LCS kernel $\mathsf{P}_{a,b}$ is obtained by the algorithm of Theorem 5.30, which calls the algorithm of Theorem 4.16 (the Steady Ant algorithm) as a subroutine. Similarly to Algorithm 5.33, we have two nested recursions: the outer recursion of the current algorithm, and the inner recursion of Theorem 4.16.

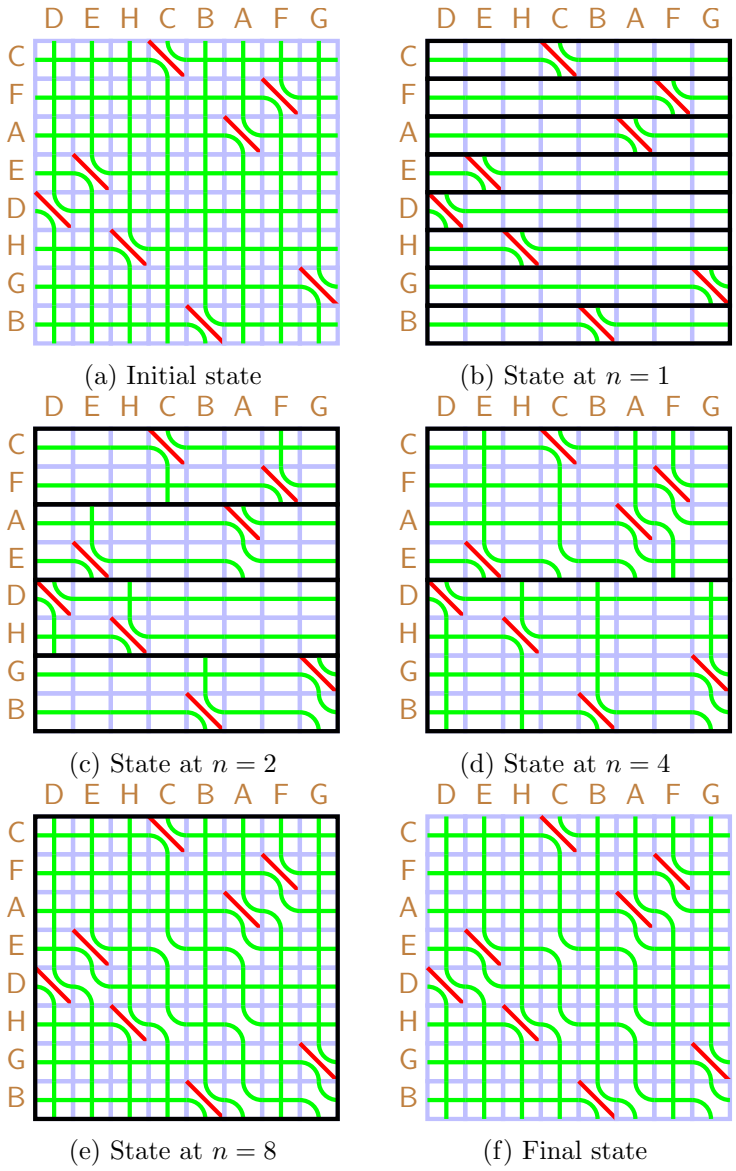
(End of recursive step)

Example 8.8 Figure 8.1 shows an execution of Algorithm 8.7 in terms of an embedded sticky braid construction for the LCS kernel $\mathsf{P}_{a,b}$ on permutation strings $a =$ "CFAEDHGB", $b =$ "DEHCBAFG", using the same conventions as Figure 5.7. ■

Theorem 8.9 *Assuming the input strings a, b are both permutation strings, the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O(n \log^2 n)$ by Algorithm 8.7.* □

Proof In Algorithm 8.7, 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 Theorem 5.30, 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 $\log n \cdot O(n \log n) = O(n \log^2 n)$. ■

By keeping the algorithm's intermediate data, we obtain a data structure that allows efficient traceback of any semi-local LCSP query, in time proportional to the size of the output (i.e. the length of the output subsequence).



(a) Initial state (b) State at $n = 1$

(c) State at $n = 2$ (d) State at $n = 4$

(e) State at $n = 8$ (f) Final state

Figure 8.1: Execution of Algorithm 8.7 (semi-local LCS by sparse recursive combing)

8.3 Window and cyclic LCSP

Window LCSP. A w-window and the window LCS problem have been defined in Definition 7.4. Recall that, given a string and a fixed length $w \geq 0$, a w-window is any substring of length w.

Definition 8.10 *The* window LCSP *problem is the window LCS problem specialised to permutation strings. Given a permutation string a and a window length w, the* window LIS problem *asks for the LIS score in every w-window of a.* □

The window LCSP problem and the window LIS problem in Definition 8.10 are clearly equivalent.

The window LIS problem has been studied by Albert et al. [12] and by Chen et al. [59]. In particular, work [59] gives an algorithm that reports all window LIS (as opposed to just their lengths) in time $O(output)$. In the same work, the algorithm is also generalised for reporting all LIS in an arbitrary subset of n substrings of the input permutation string, possibly of different sizes. Deorowicz [78] considered the problem of finding, for a given w, the maximum LIS score across all w-windows of the input string. The resulting algorithm runs in time $O\big(n \log\log n + \min(n\lambda, n\lceil \frac{\lambda^3}{w} \rceil \cdot \log\lceil \frac{w}{\lambda^2}+1 \rceil)\big)$, where λ is the output maximum LIS score. In all the above versions of the problem, the LIS score in each window can be as high as $\Theta(w)$, and therefore the algorithms' running time can be as high as $\Theta(nw)$.

The window LIS problem on a permutation string a of size n can be solved by the LCS kernel method as follows.

Theorem 8.11 *The window LIS problem can be solved in time $O(n \log^2 w)$.* □

Proof The algorithm runs in two phases.

Phase 1. Consider a subset of $2w$-windows in a, overlapping over prefixes and suffixes of length w: $a\langle 0\colon 2w\rangle$, $a\langle w\colon 3w\rangle$, $a\langle 2w\colon 4w\rangle$, ... We run Algorithm 8.7 (semi-local LCSP by sparse recursive combing) on each of these $2w$-windows against the identity permutation string id, obtaining the semi-local LCS kernels $\mathsf{P}_{a\langle kw:(k+2)w\rangle,\mathsf{id}}$.

Phase 2. We perform $2w-1$ string-substring LCS score queries for every w-window of a' against id. This can be done efficiently by Theorem 4.8 as a diagonal batch query. Every w-window of a is a substring in some string a' among the considered subset of overlapping $2w$-windows, hence we have obtained the full solution to the window LIS problem.

Running time analysis. The overall running time is dominated by Phase 1, which runs in time $O\big(\frac{2n}{w} \cdot w \log^2 w\big) = O(n \log^2 w)$ as claimed. ■

Cyclic LCSP. The cyclic LCS problem has been defined in Definition 7.6.

Definition 8.12 *The* cyclic LCSP *problem is the cyclic LCS problem specialised to permutation strings. Given a permutation string a, the* cyclic LIS problem *asks for the maximum LIS score across all cyclic shifts of a.* □

The cyclic LCSP problem and the cyclic LIS problem in Definition 8.12 are clearly equivalent.

The cyclic LIS problem has been considered by Albert et al. [11], who gave a Monte Carlo randomised algorithm, running in time $O(n^{1.5} \log n)$ with small error probability. Deorowicz [77] observed that [11] also provides a deterministic algorithm for cyclic LIS, running in time $O(n\lambda \log n)$, and gave an improved algorithm running in time $O\big(\min(n\lambda, n \log n + \lambda^3 \log n)\big)$.

Similarly to the cyclic LCS algorithm of Theorem 7.7, an algorithm for the cyclic LIS problem can be obtained by the LCS kernel method based on Theorem 8.11 as follows.

Theorem 8.13 *The cyclic LIS problem can be solved in time $O(n \log^2 n)$.*□

Proof The algorithm runs in two phases.

Phase 1. We solve the window LIS problem by Theorem 8.11 on string aa (a concatenation of string a with itself), which is of length $2n$, with window size $w = n$.

Phase 2. We take the maximum LIS score across all the windows.

Running time analysis. The running time is dominated by Phase 1, which runs in time $O(n \log^2 n)$. ∎

The resulting algorithm for the cyclic LIS problem improves on previous algorithms both in running time, and by being deterministic. In particular, our algorithm is faster than the algorithm of [77], unless $l = o\big((n \log n)^{1/3}\big)$.

8.4 Pattern-avoiding subsequence

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. $a\langle\hat{\imath}\rangle < a\langle\hat{\jmath}\rangle$ iff $b\langle\hat{\imath}\rangle < b\langle\hat{\jmath}\rangle$ for all $\hat{\imath}$, $\hat{\jmath}$. 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 8.14 The LIS problem can be interpreted as the longest X-subsequence problem, where $X = \{\text{“1”}, \text{“12”}, \text{“123”}, \ldots, \text{“123}\ldots\text{n”}\}$. ∎

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 8.15 The LIS problem on a permutation string can be interpreted as the longest $\{\text{“21”}\}$-avoiding subsequence problem. ∎

For a detailed introduction into these problems and their connections, see the work by Albert et al. [10] 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. [10] 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 [10] run in polynomial time, ranging from $O(n \log n)$ to $O(n^5)$. Two particular antipattern sets considered in [10] are (in that work's original notation):

$$C_3 = \{\text{“132”}, \text{“213”}, \text{“321”}\}$$
$$C_4 = \{\text{“132”}, \text{“213”}, \text{“312”}\}$$

For both these antipattern sets, algorithms given in [10] run in time $O(n^2 \log n)$.

We now show how the running time for the longest C_3- and C_4-avoiding subsequence problems can be improved, using the semi-local LCSP problem. hi

Longest C_3-avoiding subsequence. Permutation strings that are C_3-avoiding are all cyclic rotations of an increasing permutation string. The maximum length of such a subsequence in the target string can be found by the algorithm of Theorem 8.13, running in time $O(n \log^2 n)$.

Longest C_4-avoiding subsequence. Permutation strings that are C_4-avoiding are all obtained from an increasing permutation string by reversing some suffix. The maximum length of such a 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 apply Theorem 8.5 to string t, obtaining $n+1$ prefix LCSP scores

$$h[0] = 0 \qquad h[i] = \mathsf{score}_{LCS}\big(t\langle 0{:}\, i\rangle, \mathsf{id}\langle 0{:}\, t\langle i^-\rangle^+\rangle\big)$$

for all $i \in [1{:}\, n]$. Independently, we apply Theorem 8.9 on string t against the reverse identity permutation string $\overline{\mathsf{id}}$ of length n, and then apply Theorem 4.6, obtaining $n+1$ suffix-prefix LCSP scores

$$h'[0] = \mathsf{score}_{LCS}(t, \overline{\mathsf{id}}) \qquad h'[i] = \mathsf{score}_{LCS}\big(t\langle i{:}\, n\rangle, \overline{\mathsf{id}}\langle 0{:}\, n - t\langle i^-\rangle^+\rangle\big)$$

for all $i \in [1:n]$. Finally, we obtain the solution to the longest C_4-avoiding subsequence problem as

$$\max_{i \in [0:n]} \big(h[i] + h'[i]\big)$$

The overall running time is dominated by the algorithm of Theorem 8.9, which runs in time $O(n \log^2 n)$.

8.5 Piecewise monotone subsequence

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 (increasing or decreasing) pieces.

Definition 8.16 *Let a be a permutation string. The* longest k-increasing subsequence *problem asks for the length of a longest subsequence in a that is a concatenation of at most k increasing sequences. The* longest $(k-1)$-modal subsequence *problem asks for the length of a longest subsequence in a that is a concatenation of at most k sequences, alternating between increasing and decreasing.* □

For the longest $(k-1)$-modal subsequence problem, we assume without loss of generality that k is even.

Both problems can be solved as a special case of the LCSP problem. The length of a longest k-increasing subsequence can be obtained as $\mathsf{score}_{LCS}(\mathsf{id}^k, a)$, i.e. by comparing the concatenation of k copies of the identity permutation id against string a. Likewise, the length of a longest $(k-1)$-modal subsequence can be obtained as $\mathsf{score}_{LCS}\big((\mathsf{id}\,\overline{\mathsf{id}})^{k/2}, a\big)$, i.e. by comparing the concatenation of k alternating copies of id and its reverse $\overline{\mathsf{id}}$ against string a. In both cases, the resulting LCS grid is of size $kn \times n$, and contains kn match cells. Using standard sparse LCS algorithms [116, 18], such an instance of the LCS problem can be solved in time $O(nk \log n)$. Demange et al. [74] gave an algorithm for the longest $(k-1)$-modal subsequence problem, also running in time $O(nk \log n)$.

We now show algorithms for the longest k-increasing subsequence and the longest $(k-1)$-modal subsequence problems, improving on the above algorithms in running time for sufficiently large k.

Theorem 8.17 *The longest k-increasing and the longest $k-1$-modal subsequence problems can both be solved in time $O(n \log^2 n)$.* □

Proof To solve the longest k-increasing subsequence problem, we call the algorithm of Theorem 8.9 (semi-local LCSP) on strings id, a, obtaining the

semi-local LCS kernel $\mathsf{P}_{\mathsf{id},a}$, from which we extract the string-substring subkernel $\mathsf{P}^{\nearrow}_{\mathsf{id},a}$. Assume that k is a power of 2. We run the algorithm of Theorem 5.30 $\log k$ times in a repeated squaring procedure, obtaining the string-substring LCS kernels

$$\mathsf{P}^{\nearrow}_{\mathsf{id},a} \boxdot \mathsf{P}^{\nearrow}_{\mathsf{id},a} = \mathsf{P}^{\nearrow}_{\mathsf{id}^2,a}$$
$$\mathsf{P}^{\nearrow}_{\mathsf{id}^2,a} \boxdot \mathsf{P}^{\nearrow}_{\mathsf{id}^2,a} = \mathsf{P}^{\nearrow}_{\mathsf{id}^4,a}$$
$$\dots$$
$$\mathsf{P}^{\nearrow}_{\mathsf{id}^{k/2},a} \boxdot \mathsf{P}^{\nearrow}_{\mathsf{id}^{k/2},a} = \mathsf{P}^{\nearrow}_{\mathsf{id}^{k},a}$$

For a general k (not necessarily a power of 2), we can still obtain matrix $\mathsf{P}^{\nearrow}_{\mathsf{id}^k,a}$ in at most $2\log k$ applications of Theorem 5.30, involving repeated squaring and multiplication by $\mathsf{P}^{\nearrow}_{\mathsf{id},a}$. We then obtain $\mathsf{score}_{LCS}(\mathsf{id}^k, a) = \mathsf{H}^{\nearrow}_{\mathsf{id}^k,a}[0;n]$ by Theorem 5.17.

To solve the longest $(k-1)$-modal subsequence problem, we call the algorithm of Theorem 8.9 (semi-local LCSP) twice: on strings id, a, and on strings $\overline{\mathsf{id}}$, a. As as a result, we obtain the semi-local LCS kernels $\mathsf{P}_{\mathsf{id},a}$ and $\mathsf{P}_{\overline{\mathsf{id}},a}$, respectively. We then obtain the LCS kernel $\mathsf{P}_{\mathsf{id}\,\overline{\mathsf{id}},a}$ by Theorem 5.30. From this kernel, we extract the string-substring subkernel $\mathsf{P}^{\nearrow}_{\mathsf{id}\,\overline{\mathsf{id}},a}$. Analogously to the above algorithm for the longest k-increasing subsequence problem, we then obtain the string-substring LCS kernel $\mathsf{P}^{\nearrow}_{(\mathsf{id}\,\overline{\mathsf{id}})^{k/2},a}$ in at most $2(\log k - 1)$ applications of Theorem 5.30, involving repeated squaring and multiplication by $\mathsf{P}^{\nearrow}_{\mathsf{id}\,\overline{\mathsf{id}},a}$. We then obtain $\mathsf{score}_{LCS}(\mathsf{id}\,\overline{\mathsf{id}}, a) = \mathsf{H}^{\nearrow}_{\mathsf{id}\,\overline{\mathsf{id}},a}[0,n]$ by Theorem 5.17.

Both described algorithms run in time $O(n\log^2 n) + \log k \cdot O(n \log n) = O(n\log^2 n)$. ∎

The algorithm of Theorem 8.17 is faster than both the sparse LCS approach and the algorithm of [74], for all $k = \omega(\log n)$.

8.6 Affinity-sensitive semi-local LCS

In this chapter, we have considered semi-local comparison of permutation strings, and some of its applications. We now extend these results to more general sparse comparison problems.

Permutation string vs general string. Let a be a permutation string of length m, and let b be a general string (with possibly repeating characters) of length n. Without loss of generality, we assume $m \le n$ (otherwise, string a will contain characters not occurring in string b, which can be deleted

without affecting any LCS scores). The LCS grid $\mathsf{G}_{a,b}$ for such strings contains at most n match cells, and the semi-local LCS problem is equivalent to the local LIS problem on a string of length n over an alphabet of size m.

Lemma 8.18 *Assuming the input strings a is a permutation string, while b is a general string, the semi-local LCS oracle of Theorem 5.21 can be obtained in time $O(n \log^2 m)$.* □

Proof We extend Algorithm 8.7 by allowing b to be a general string. The recursion tree is a balanced binary tree of height $\log m$. At the top level of recursion, we have the partitioning $a = a'a''$, where a', a'' are both of length $\frac{m}{2}$. Strings b', b'' are defined as in Algorithm 8.7; in contrast with that algorithm, they may no longer be of equal length, but their total combined length is still n. In every subsequent recursion level, we have a partitioning of a into substrings of equal length; the total length of these substrings across a recursion level is m. Every substring in the partitioning of a is compared against a substring of b. These substrings of b are of variable lengths; the total length of these substrings across a recursion level is n. In every recursion level, the running time is dominated by the combined running time of the algorithm of Theorem 5.30 on substrings of a against the substrings of b, which is $O(n \log m)$. There are $\log m$ recursion levels, therefore the overall running time is $\log m \cdot O(n \log m) = O(n \log^2 m)$. ■

Affinity-sensitive semi-local LCS. A highly relevant parameter in string comparison is the total number of common character matches between the input strings.

Definition 8.19 *The* affinity *of strings a, b is the number $\rho \le mn$ of match cells in the LCS grid $\mathsf{G}_{a,b}$.* □

Without loss of generality, we assume $m, n \le \rho$.

Affinity-sensitive string comparison algorithms have been studied for a long time. Hunt and Szymanski [116] gave an algorithm for affinity-sensitive LCS, running in time $O(\rho \log n)$. Apostolico and Guerra [18] improved this to $O\big(m \log n + \rho \log \frac{mn}{r}\big)$. Affinity-sensitive semi-local LCS has been considered by Krusche and Tiskin [140], [138, Section 5.3], who gave an algorithm running in time $n\rho^{1/2}$, assuming $m = n$. Matarazzo et al. [157] improved this to $O\big(\rho \log^3 \frac{n^2}{\rho}\big)$ under the same assumption. We now give a further improvement via an extension and a careful analysis of our previous techniques.

Theorem 8.20 *Let input strings a, b have affinity ρ. The semi-local LCS oracle of Theorem 5.21 can be obtained in time $O\big(\rho \log^2 \frac{mn}{\rho}\big)$.* □

Proof We extend Algorithm 8.7 (semi-local LCSP by sparse recursive combing) by allowing both a and b to be general strings. The recursion tree is a balanced binary tree of height $\log m$.

Consider the top part of the recursion tree, consisting of $\log \frac{\rho}{n}$ recursion levels. In every level, we have a partitioning of a into substrings of equal length, which is at most m. Each substring in the partitioning of a is compared against a substring of b of length at most n. In every recursion level, the running time is dominated by the combined running time of the algorithm of Theorem 5.30 on substrings of a against the substrings of b, which is $k \cdot O(n \log \frac{m}{k})$, where $k \leq \frac{\rho}{n}$ is the number of subproblems in the given recursion level. The running time of the top part of the tree is dominated by level $\log \frac{\rho}{n}$, where we have $k = \frac{\rho}{n}$ subproblems, running in combined time

$$\tfrac{\rho}{n} \cdot O\big(n \log \tfrac{m}{\rho/n}\big) = O\big(\rho \log \tfrac{mn}{\rho}\big)$$

Now consider the remaining bottom $\log m - \log \frac{\rho}{n} = \log \frac{mn}{\rho}$ recursion levels. In every level, we have a partitioning of a into substrings of equal length, which is at most m. Each substring in the partitioning of a is compared against a substring of b of length at most n. These substrings of b are of variable lengths; their total combined length is at most ρ. In every recursion level, the running time is dominated by the combined running time of the algorithm of Theorem 5.30 on substrings of a against the substrings of b, which is $O(\rho \log \frac{m}{k}) = O(\rho \log \frac{mn}{\rho})$, where $k \geq \frac{\rho}{n}$ is the number of subproblems in the given recursion level. Since there are $\log \frac{mn}{\rho}$ levels, the overall running time of the bottom part of the recursion tree is

$$\log \tfrac{mn}{\rho} \cdot O\big(\rho \log \tfrac{mn}{\rho}\big) = O\big(\rho \log^2 \tfrac{mn}{\rho}\big)$$

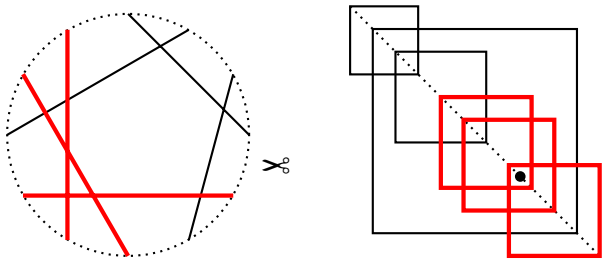
The total running time of our affinity-sensitive semi-local LCS algorithm is obtained as the sum of the running times of the top and bottom parts of the recursion tree:

$$O\big(\rho \log \tfrac{mn}{\rho}\big) + O\big(\rho \log^2 \tfrac{mn}{r}\big) = O\big(\rho \log^2 \tfrac{mn}{\rho}\big) \qquad \blacksquare$$

8.7 Maximum clique in a circle graph

Circle graphs are an important combinatorial class of graphs; see e.g. [86, 104].

Definition 8.21 *A* circle graph *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 their corresponding chords intersect. 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.* □



(a) The chord model (b) An interval model

Figure 8.2: A circle graph and its maximum clique

A 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 their corresponding intervals intersect but do not contain one another. Circle graphs have many applications, see e.g. [216] and references therein.

Example 8.22 Figure 8.2 shows an instance of the maximum clique problem on a six-node circle graph. Figure 8.2a 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. Figure 8.2b 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. ■

We consider the maximum clique problem on a circle graph. It has long been known that the maximum clique problem in a circle graph on n nodes is solvable in polynomial time [95]. A number of algorithms have been proposed for this problem [183, 114, 156, **?**]; the problem has also been studied in the context of line arrangements in the hyperbolic plane [128, 81]. 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 [201, 204], 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 implicit distance multiplication procedure of Theorem 4.16 (the Steady Ant algorithm).

Our algorithm takes as input the interval model of a circle graph G on n nodes. Without loss of generality, we may assume that all interval endpoints are pairwise distinct half-integers in $\langle 0{:}\,2n\rangle$. The interval model is represented by a permutation string $a\langle 0{:}\,2n\rangle$, where for each interval endpoint $\hat{\imath} \in \langle 0{:}\,2n\rangle$, $a\langle\hat{\imath}\rangle$ gives that interval's opposite endpoint. Note that for all $\hat{\imath} < \hat{\jmath}$, an interval with left endpoint $\hat{\imath}$ does not contain an interval with left endpoint $\hat{\jmath}$, if and only if $a\langle\hat{\imath}\rangle < a\langle\hat{\jmath}\rangle$. Various alternative representations

of interval models (e.g. the ones used in [183, **?**]) 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 setting, we only need to consider integer indices as intersection points.

Consider a clique in G. Let $k \in [1\colon 2n-1]$ 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. Consider the identity permutation string $\mathsf{id}\langle 0\colon 2n\rangle$. From the observations above, it follows that the clique corresponds to a common subsequence of a prefix $a\langle 0\colon k\rangle$ and a suffix $\mathsf{id}\langle 2n-k\colon 2n\rangle$. Therefore, the maximum clique problem can be solved as an instance of the semi-local LCS problem.

Algorithm 8.23 (Max-clique in a circle graph)

Input: permutation string a of size $2n$, representing the interval model of a circle graph G.

Output: maximum-size clique of G.

Description. The algorithm runs in three phases.

Phase 1. We obtain the semi-local LCSP oracle for strings a, $\mathsf{id}\langle 0\colon 2n\rangle$ by Theorem 8.9.

Phase 2. The size of the maximum clique can now be obtained as

$$\max_{k\in[0:2n]} \mathsf{score}_{LCS}\big(a\langle 0\colon k\rangle, \mathsf{id}\langle 2n-k\colon 2n\rangle\big)$$

For each k, the prefix-suffix LCSP score is queried from the oracle by Theorem 4.6. The value k^* for which the maximum score is attained gives a common intersection point of the clique intervals.

Phase 3. Let $I = \{\hat{\imath} \mid a\langle i\rangle \in \langle 2n-k^*\colon 2n\rangle\}$, $a' = a|_I$. The intervals defining the maximum clique can now be obtained by applying Theorem 8.5 to string a', and then tracing back the elements of the resulting LIS. ■

Theorem 8.24 *The maximum clique problem in a circle graph can be solved in time $O(n \log^2 n)$ by Algorithm 8.23.* □

Proof We consider each phase of Algorithm 8.23.

Phase 1. By Theorem 8.9, the running time is $O(n \log^2 n)$.

Phase 2. By Theorem 4.6, the overall running time of all the prefix-suffix queries is $O(n \log^2 n)$, if the queries are performed independently. Furthermore, by Theorem 4.8 these queries can be combined into a diagonal batch query, reducing the running time to $O(n)$.

Phase 3. By Theorem 8.5, the running time is $O(n \log n)$.

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

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

- the size l of the maximum clique;
- the *cutwidth* d of the interval model, i.e. the maximum number of intervals containing a given point, taken across all points in the line (sometimes also called *density*; see e.g. [**?**, 123]);
- 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 permutation string a representing an interval model, it is straightforward to obtain its cutwidth in time $O(n)$ by a linear scan of the string, keeping track of the intervals' nesting level.

Apostolico et al. [**?**] gave algorithms for the parameterised maximum clique problem in a circle graph, running in time $O(n \log n + e)$ and $O\big(n \log n + nl \log(n/l)\big)$. They also describe an algorithm for the maximum independent set problem, parameterised by the interval model's cutwidth.

We describe an extended version of Algorithm 8.23 (Maximum clique problem in a circle graph), which is parameterised by the cutwidth of the input interval model, and improves on the algorithms of [**?**] for most values of the parameters.

Algorithm 8.25 (Max-clique in a circle graph, cutwidth-sensitive)

Input: string a of size $2n$, representing the interval model of a circle graph G.

Output: maximum-size clique of G.

Parameter: the interval model's cutwidth d, $d \leq n$.

Description. Since fixed cutwidth guarantees that the intervals are "spread out" along the interval $\langle 0 \colon 2n \rangle$, we spread the computation accordingly. We partition the interval model, originally defined on the interval $\langle 0 \colon 2n \rangle$, into $\frac{2n}{d}$ smaller, partially overlapping submodels, each consisting of the intervals intersecting one of a disjoint set of intervals $\langle \hat{r}^- d \colon \hat{r}^+ d \rangle$, $\hat{r} \in \big\langle 0 \colon \frac{2n}{d} \big\rangle$. A

semi-local LCSP problem corresponding to each submodel is solved independently, and the best of these solutions is then selected as the solution for the main problem.

Phase 1. For each $\hat{r} \in \langle 0\colon \frac{2n}{d} \rangle$ independently, we obtain the semi-local LCSP oracle for strings $a_{\hat{r}} = a\langle 0\colon \hat{r}^+ d\rangle$, $\mathsf{id}_{\hat{r}} = \mathsf{id}\langle \hat{r}^- d\colon 2n\rangle$ by Theorem 8.9.

Phase 2. The size of the maximum clique can now be obtained as

$$\max_{k\in[0:2n]} \mathsf{score}_{LCS}\big(a\langle 0\colon k\rangle, \mathsf{id}\langle 2n-k\colon 2n\rangle\big) =$$
$$\max_{k\in[0:2n]} \mathsf{score}_{LCS}\big(a_{\lfloor \frac{k}{d} \rceil}\langle 0\colon k\rangle, \mathsf{id}_{\lfloor \frac{k}{d} \rceil}\langle 2n-k\colon 2n\rangle\big)$$

where $\lfloor x \rceil$ denotes the nearest half-integer to x. For each k, the prefix-suffix LCSP score is queried from the oracle for strings $a_{\lfloor \frac{k}{d} \rceil}, \mathsf{id}_{\lfloor \frac{k}{d} \rceil}$ by Theorem 4.6. The value k^* for which the maximum score is attained gives a common intersection point of the clique intervals.

Phase 3. As in Algorithm 8.23. ∎

Theorem 8.26 *The maximum clique problem in a circle graph of cutwidth d can be solved in time $O(n \log^2 d)$.* □

Proof We consider each phase of Algorithm 8.25.

Phase 1. Let $\hat{r} \in \langle 0\colon \frac{2n}{d} \rangle$. Consider string decomposition $a_{\hat{r}} = a'_{\hat{r}} a''_{\hat{r}}$, where

$$a'_{\hat{r}} = a\langle 0\colon \hat{r}^- d\rangle \qquad a''_{\hat{r}} = a\langle \hat{r}^- d\colon \hat{r}^+ d\rangle$$

The LCS grid $G = \mathsf{G}_{a_{\hat{r}},\mathsf{id}_{\hat{r}}}$ is the composite of LCS grids $G' = \mathsf{G}_{a'_{\hat{r}},\mathsf{id}_{\hat{r}}}$ and $G'' = \mathsf{G}_{a''_{\hat{r}},\mathsf{id}_{\hat{r}}}$. Grid G' contains at most d match cells, since every match in this grid corresponds to an interval in the interval model of G, containing point $\hat{r}^- d$. Therefore, there can be at most d matches by the definition of cutwidth. Grid G'' also contains at most d match cells, since string $a''_{\hat{r}}$ has length d. Hence, grid G contains at most $d + d = 2d$ match cells. By Theorem 8.9, the running time of each iteration is $O(d \log^2 d)$, and the overall running time of this phase is $O(n/d \cdot d \log^2 d) = O(n \log^2 d)$.

Phase 2. By Theorem 4.6, the overall running time of all the prefix-suffix queries is $O(n \log^2 d)$, if the queries are performed independently. Furthermore, by Theorem 4.8 these queries can be combined into a diagonal batch query, reducing the running time to $O\big(\frac{n}{d} \cdot d\big) = O(n)$.

Phase 3. String a' contains at most d characters, since every such character corresponds to an interval containing point k^*. By Theorem 8.5, the running time is $O(d \log d)$.

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

Algorithm 8.25 is faster than the $O(n \log n + e)$ algorithm of [**?**], unless $e = o(n \log^2 d) = O(n \log^2 n)$. It is also faster than the $O\big(n \log n + nl \log(n/l)\big)$ algorithm of [**?**], unless $l = o\big(\frac{\log^2 d}{\log n}\big) = O(\log n)$.

Max-clique with shared endpoints. We now relax the assumption that all the chord (equivalently, interval model) endpoints must be distinct. Instead, we allow an endpoint to be shared by several chords (intervals). In the definition of a circle graph, the chords are taken to be open intervals: two nodes are adjacent, whenever the interiors of their corresponding chords intersect. We denote by k the total number of distinct chord (interval) endpoints, and use this value as the problem's parameter, keeping n, $\frac{k}{2} \leq n \leq \frac{k(k-1)}{2}$, for the number of intervals (i.e. nodes of the circle graph).

A circle graph with shared endpoints is still isomorphic to the overlap graph of its interval model, where the intervals are taken to be open intervals.

Ward et al. [216] considered the maximum-weight clique problem in a vertex-weighted circle graph with shared interval endpoints. They gave an algorithm running in time $O(k^3)$, and conjectured that in the unweighted case, the size of the maximum clique can be obtained in time $O(k^2 \log k)$. We show that their conjecture holds and can even be exceeded in the unweighted case.

Theorem 8.27 *The maximum clique problem in a circle graph with k distinct chord (interval) endpoints can be solved in time $O\big(\frac{k^2(\log\log k)^2}{(\log k)^2}\big)$ or $O\big(n \log^2 \frac{k^2}{n}\big)$.* □

Proof Similarly to the algorithm of Theorem 8.24, the computation is dominated by solving the prefix-suffix LCS problem on a string of length k and self-affinity n. The claimed running time can be obtained by plugging into the algorithm of Theorem 8.24 (maximum clique in a circle graph) either the algorithm of Theorem 5.42 (semi-local LCS with micro-block speedup), or the algorithm of Theorem 8.20 (affinity-sensitive semi-local LCS). ■

8.8 Linear graph comparison

The concept of a circle graph is generalised by that of a *linear graph*, introduced by Davydov and Batzoglou [72]. A linear graph is defined by an interval model, similar to that of a circle graph introduced in Definition 8.21. A *pattern* in a linear graph is defined as an ordered subset of intervals, all of which satisfy pairwise a prescribed subset of relations: disjointness ($<$), containment ($\sqsubset$) and/or overlapping without containment ($\between$).

Fertin et al. [89] considered the *maximum common S-structured pattern (S-MCSP)* problem. The problem asks for the maximum-size common pattern in a set of r linear graphs, each defined by at most m intervals, where the structure of the common pattern is constrained by a prescribed subset of relations $S \subseteq \{<, \sqsubset, \between\}$. In particular:

- The $\{\between\}$-MCSP problem with $r = 1$ asks for the maximum-size subset of pairwise overlapping intervals; this is equivalent to asking for the

maximum clique of a circle graph. For general r, the problem is equivalent to asking for the minimum-sized clique among maximum cliques of the r input circle graphs.

- The $\{<, \sqsubset\}$-MCSP problem with $r = 1$ asks for the maximum-size subset of pairwise non-overlapping intervals; this is equivalent to asking for the maximum independent set of a circle graph. For general r, the problem is equivalent to asking for the maximum-size commonly structured subset among maximum independent sets of the r input circle graphs.

- The $\{<, \sqsubset, \between\}$-MCSP problem asks for the maximum commonly-structured subset of intervals without any a priori constraints on its structure.

Extending and generalising a number of previous results, paper [89] considers the S-MCSP problem, where S runs over all seven nonempty subsets of $\{<, \sqsubset, \between\}$. For some of these seven variants, the algorithms use as a subroutine the algorithm of [201, 204] for the maximum clique problem in a circle graph. By plugging in the more efficient Algorithm 8.23, we can obtain 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 $\{\between\}$-MCSP problem is solved in [89] by finding the maximum clique independently for r circle graphs, each corresponding to one of the input linear graphs, in overall time $O(rm^{1.5})$. By plugging in Algorithm 8.23, the running time is improved to $O(rm \log^2 m)$.

The $\{<, \between\}$-MCSP problem is shown in [89] to be NP-hard, and to admit a polynomial-time $2h(k)$-approximation, where $h(k) = \sum_{1 \le i \le k} 1/i = \ln k + O(1)$; for the rest of this section, k denotes the size of the optimal solution to the problem. The approximation is obtained by $O(rm)$ 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 $O(m^{2.5} \log m)$, therefore the overall running time is $rm \cdot O(m^{2.5} \log m) = O(rm^{3.5} \log m)$. By a straightforward extension of Algorithm 8.23, the running time of the subroutine is improved to $O(m \log^2 m)$, therefore the overall running time of the approximation algorithm is improved to $rm \cdot O(m \log^2 m) = O(rm^2 \log^2 m)$.

The $\{\sqsubset, \between\}$-MCSP problem is also shown in [89] to be NP-hard, and to admit a polynomial-time $k^{1/2}$-approximation. The approximation is obtained by combining exact solutions for the $\{\sqsubset\}$-MCSP and $\{\between\}$-MCSP problems on the same inputs. The exact solution for the $\{\between\}$-MCSP is the bottleneck; by plugging in the improved algorithm for this problem described above, the running time of the approximation algorithm for the $\{\sqsubset, \between\}$-MCSP problem is improved from $O(rm^{1.5})$ to $O(rm \log^2 m)$.

Finally, the $\{<, \sqsubset, \between\}$-MCSP problem is shown in [89] to be NP-hard, and to admit several polynomial-time approximations. In particular, it

has an $O(k^{2/3})$-approximation algorithm running in time $O(rm^{1.5})$, and an $O\big((k \log k)^{1/2}\big)$-approximation algorithm running in time $O(rm^{3.5} \log m)$. By using the techniques described above, the running times of these approximation algorithms are improved respectively to $O(rm \log^2 m)$ and $O(rm^2 \log^2 m)$.

Chapter 9

Weighted string comparison

9.1 Scuffles and copulas

Generalisations of matrices to bivariate functions of reals, or measures on a rectangle, understood as "continuous matrices", are well-known in mathematics. This is done e.g. in the context of integral transforms (see e.g. [73]), where such generalised matrices are known as "integration kernels" matrix factorisations [**?**], where they are termed "cmatrices", and convergent sequences of graphs and permutations, where they are termed "graphons" and "permutons" (see e.g. [**?**]).

In this chapter, we present a continuous version of the LCS kernel method, that will help us to extend its application beyond rational alignment scores.

Scuffles and scuffle measures. We define continuous analogues of concepts introduced in Chapters 2 and 3, beginning with (sub)permutations.

Definition 9.1 *Let $[\![X]\!]$, $[\![Y]\!]$ be real intervals. Let $\mathcal{X}$, $\mathcal{Y}$ be finite sets of open disjoint subintervals in $[\![X]\!]$, $[\![Y]\!]$, respectively, both with an identical multiset of subinterval lengths. A* subscuffle $\sigma\colon [\![X]\!] \leftrightarrow [\![Y]\!]$ of order $|\mathcal{X}| = |\mathcal{Y}|$ *is a bijective mapping from* $\operatorname{dom}\sigma = \bigcup\mathcal{X}$ *to* $\operatorname{ran}\sigma = \bigcup\mathcal{Y}$*, that acts on every interval in $\mathcal{X}$ as a translation with reversal: we have*

$$\sigma|_{(\!|x_0;x_1|\!)}\colon x \mapsto x_0 + \sigma(x_0) - x$$

for every $(\!|x_0;x_1|\!) \in \mathcal{X}$. A subscuffle σ is called a scuffle*, if* $\operatorname{cl}(\operatorname{dom}\sigma) = [\![X]\!]$, $\operatorname{cl}(\operatorname{ran}\sigma) = [\![Y]\!]$*. We denote* $\operatorname{Dom}\sigma = \mathcal{X}$, $\operatorname{Ran}\sigma = \mathcal{Y}$*. For any subset $U \subseteq \operatorname{dom}\sigma$, we denote its image by $\sigma(U) \subseteq \operatorname{ran}\sigma$.* □

A (sub)scuffle σ of a given order can be represented as a (sub)scuffle of any higher order by subdividing the intervals in $\mathcal{X}$ at a finite number of points, and restricting σ to the interiors of the resulting subintervals.

A (sub)permutation can be treated as a discrete (sub)scuffle as follows.

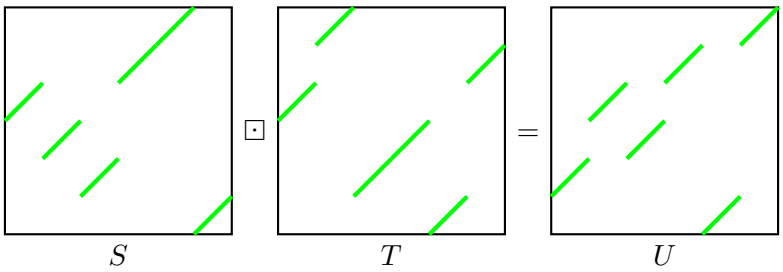


Figure 9.1: Sticky product $S \boxdot T = U$

Definition 9.2 *A (sub)permutation* $\pi\colon \langle I \rangle \leftrightarrow \langle J \rangle$ *corresponds to (sub)scuffle* $\tilde{\pi}\colon [\![I]\!] \leftrightarrow [\![J]\!]$ *as follows: for every* $\langle \hat{\imath}; \hat{\jmath} \rangle \in \pi$*, we have* $(\!|\hat{\imath}^-\colon \hat{\imath}^+; \hat{\jmath}^-\colon \hat{\jmath}^+|\!) \in \tilde{\pi}$*.*

We generalise Definition 4.2 ((sub)permutation matrix) as follows.

Definition 9.3 *Let* $\sigma\colon [\![X]\!] \leftrightarrow [\![Y]\!]$ *be a (sub)scuffle. The corresponding* (sub)scuffle measure $S[\![X; Y]\!]$ *is the singular continuous push-forward measure on* $[\![X; Y]\!]$*, induced by the Lebesgue measure on* $[\![X]\!]$ *and the natural embedding* $x \mapsto [\![x; \sigma(x)]\!]$ *of* $\operatorname{dom}\sigma$ *into* $[\![X; Y]\!]$*. Any pair* $[\![x; \sigma(x)]\!]$*,* $x \in \operatorname{dom}\sigma$ *will be called a* nonzero *of* S*.* □

We denote by $\operatorname{nz} S$ the set of nonzeros of S; note that $\operatorname{cl}(\operatorname{nz} S) = \operatorname{supp} S$.

Example 9.4 Figure 9.1 shows a triple of scuffle measures S, T (both of order 5), U (of order 6), such that $S \boxdot T = U$. Nonzeros are indicated by green intervals. ■

A notion of (sub)bistochastic measure can be introduced by a suitable generalisation of Definition 3.9 ((sub)bistochastic matrix), so that (sub)scuffle measures are (sub)bistochastic.

A (sub)permutation matrix can be treated as a discrete (sub)scuffle measure as follows.

Definition 9.5 *A (sub)permutation matrix* $P\langle I; J \rangle$ *corresponds to (sub)scuffle measure* $\tilde{P}[\![I; J]\!]$*, if the same relation holds for their defining (sub)permutation and (sub)scuffle. For every nonzero* $P\langle \hat{\imath}; \hat{\jmath} \rangle = 1$*, we have* $[\![\hat{\imath} + x; \hat{\jmath} - x]\!] \in \operatorname{nz} \tilde{P}$ *for all* $0^- < x < 0^+$*.* □

Example 9.6 Scuffle measures S, T in Figure 9.1 can both also be considered to be of order 6, by splitting the longer nonzero interval in the middle. In this way, each of measures S, T, U corresponds to a 6×6 permutation matrix. ■

Monge and unit-Monge copulas. We generalise Definition 3.8 (Monge matrix) as follows.

Definition 9.7 *A* copula[1] $A[\![X;Y]\!]$ *is a function* $A\colon [\![X;Y]\!] \to \mathbb{R}$. *Copula* A *is a* Monge copula *(also called a* lattice subadditive function*), if*

$$A[\![i;j]\!] + A[\![i';j']\!] \leq A[\![i;j']\!] + A[\![i';j]\!]$$

for all $i \leq i'$, $j \leq j'$. □

In general, a Monge copula may have no finite (nor even countable) representation. In this work, we will only be concerned with finitely representable Monge copulas, obtained as a continuous generalisation of unit-Monge matrices. We generalise Definition 3.1 (dominance-sum matrix) and Definition 4.4 ((sub)unit-Monge matrix) as follows.

Definition 9.8 *Let* $S[\![X;Y]\!]$ *be a measure. Its* dominance-measure copula $S^{\Sigma}[\![X;Y]\!]$ *is a continuous nonnegative copula, defined by*

$$S^{\Sigma}[\![x;y]\!] = \int_{[\![x:;:y]\!]} \mathrm{d}S$$

where $x \in X$, $y \in Y$. □

In particular, the dominance copula is well-defined on a (sub)scuffle measure.

Definition 9.9 *A* simple (sub)unit-Monge copula *of order* r *is the dominance-measure copula of some (sub)scuffle measure of the same order.* □

A notion of Σ-ν-(sub)bistochastic copula can be introduced by a suitable generalisation of **??**, so that (sub)unit-Monge copulas are Σ-ν-(sub)bistochastic.

9.2 Copula monoids

Copula distance multiplication. We generalise Definition 2.1 (tropical matrix multiplication) as follows.

Definition 9.10 *Let* $A[\![X;Y]\!]$, $B[\![Y;Z]\!]$, $C[\![X;Z]\!]$ *be copulas.* Tropical copula multiplication $A \odot B = C$ *is defined by*

$$C[\![x;z]\!] = \bigoplus_{y \in [\![Y]\!]} \big(A[\![x;y]\!] \odot B[\![y;z]\!]\big) = \min_{y \in [\![Y]\!]} \big(A[\![x;y]\!] + B[\![y;z]\!]\big)$$

for all $x \in [\![X]\!]$, $z \in [\![Z]\!]$. □

[1] To quote from Townsend and Trefethen [**?**], "we are well aware that it is a little odd to introduce a new term for what is, after all, nothing more than a bivariate function." Still, this terminology is necessary in order to mirror our terminology for matrices. It is somewhat easier to pronounce than "cmatrices" of [**?**], and different in that a copula is not required to be a continuous function.

In general, the tropical product $A \odot B$ may not exist, since the minimum in Definition 9.10 may not be attained for some x, z. However, the tropical product always exists when copulas A, B are continuous, by Weierstrass' extreme value theorem. Therefore, a monoid of continuous copulas with respect to tropical multiplication can be defined as a suitable generalisation of the tropical matrix monoid.

Closure properties of Monge, simple Σ-bistochastic and simple Σ-subbistochastic copulas with respect to tropical multiplication, can be stated and proved by a suitable generalisation of Theorems 3.12 and 3.15. Due to these properties, monoids of such copulas can be defined as suitable generalisations of the corresponding matrix monoids.

Measure sticky multiplication. We extend the definition of matrix sticky multiplication to measures as follows.

Definition 9.11 *Let $A[\![I;J]\!]$, $B[\![J;K]\!]$, $C[\![I;K]\!]$ be measures. The* measure sticky multiplication *$S \boxdot T = U$ is defined by $S^{\Sigma} \odot T^{\Sigma} = U^{\Sigma}$.* □

This definition is indeed consistent with sticky multiplication of permutation matrices.

Theorem 9.12 *Let P, Q, R be permutation matrices. We have $P \boxdot Q = R$, if and only if $\tilde{P} \boxdot \tilde{Q} = \tilde{R}$ for their corresponding scuffle measures.* □

Proof Straightforward by the definitions. ■

Sticky multiplication on general scuffle measures can be regarded as a limiting case of sticky multiplication on permutation matrices of increasing order, where the multiplicand matrices are taken to be suitable discretisations of the corresponding multiplicand measures, and the sticky product measure is obtained by Theorem 9.12. Therefore, sticky multiplication of arbitrary scuffle measures is well-defined by a compactness argument.

Definition 9.13 *(Sub)scuffles $\sigma\colon X \leftrightarrow Y$ and $\tau\colon Y \leftrightarrow Z$, as well as their corresponding (sub)scuffle measures, are called* compatible*, when* $\operatorname{ran}\sigma = \operatorname{dom}\tau$. □

Note that compatible (sub)scuffles have the same order, equal to the number of intervals in $\operatorname{dom}\sigma$, $\operatorname{ran}\sigma = \operatorname{dom}\tau$, and $\operatorname{ran}\tau$. In contrast with permutation matrices, sticky multiplication of scuffle measures may increase their order, even if the multiplicands are compatible.

Example 9.14 Scuffle measures U, V in Figure 9.1 are of order 5 and compatible; their sticky product $S \boxdot T = U$ is of order 6. ■

An upper bound on the growth of the scuffle measure order under sticky multiplication can be obtained as follows.

Theorem 9.15 *Let S, T, U be (sub)scuffle measures. Let $S \boxdot T = U$. If S, T are of order r and compatible, then U is of order at most $\frac{r(r-1)}{2}$.* □

PROOF Without loss of generality, we may assume that all r defining intervals in the scuffles for S, T are of different lengths $l_1 < l_2 < \ldots < l_r$. A product of compatible scuffle measures can be discretised by replacing a real interval of length l_k with a half-integer interval of length k. The discretised product has order $\frac{r(r-1)}{2}$, which provides the upper bound on the order of the original product. ■

In general, the sticky product of subscuffle matrices does not preserve their order; on the contrary, the order of the product may be substantially higher than the order of the multiplicands. Still, a continuous analogue of Theorem 4.16 holds, although the runtime of the algorithm has to bounded by a function of the output size, rather than input size.

Theorem 9.16 *Let S, T, U be (sub)scuffle measures. Let $S \boxdot T = U$. If S, T are of order r and compatible, then, given the defining interval endpoints of S, T, the defining interval endpoints of U can be computed in time $O(r^2)$.* □

PROOF The general structure of the algorithm is similar to that of Theorem 4.16. We may assume without loss of generality that r is a power of 2. Measure S is partitioned vertically, and measure T horizontally, into two pairs of compatible measures of order $r/2$. The two resulting subproblems are solved independently by recursion. We then perform discretisation as described in the proof of Theorem 9.15, and then perform the conquer phase of Theorem 4.16 on the discretised solutions to the subproblems. The running time is dominated by the top level of recursion, which runs in time $O(n^2)$. ■

Generalised Monge monoid Monge copulas closed

Generalised unit-Monge monoid unit-Monge copulas closed increases order on unit-Monge

smult algorithm

Improves Russo.

Relates to Comet?

9.3 Heaviest common subsequence (HCS)

Jacobson and Vo [**?**] introduced a natural generalisation of the LCS problem, where the score of a character match may depend on the characters' position.

Definition 9.17 *Let a, b be strings. Let $w\langle \hat{\imath}; \hat{\jmath} \rangle$ be a real-valued* weight function*, which may depend on a, b. The* (weighted) score *of a common*

subsequence of a, b is the sum of all values $w\langle\hat{\imath};\hat{\jmath}\rangle$, where character $a\langle\hat{\imath}\rangle$ is aligned with character $b\langle\hat{\jmath}\rangle$ in the subsequence. □

Definition 9.18 *Let a, b be strings. Assuming a fixed weight function, the* heaviest common subsequence (HCS) score*, denoted* $\mathsf{score}_{HCS}(a,b)$*, is the maximum weighted score across all possible common subsequences of a, b. Given strings a, b, the* HCS problem*, the* HCS grid $\mathrm{G}_{a,b}$*, the* prefix HCS problem*, the* semi-local HCS problem*, analogous problems on permutation strings (HCSP, prefix HCSP, semi-local HCSP), the* heaviest increasing subsequence (HIS) problem*, and the* local HIS problem *are defined analogously to Definitions 5.1, 5.4, 5.7, 5.11, 8.1, 8.2 and 8.6, replacing LCS scores with HCS scores.* □

The HCS and prefix HCS problems can be solved by an extension of the classical dynamic programming algorithm, running in time $O(n^2)$.

9.4 Semi-local HCS

HCS kernel: scuffle measure of order $O(n^2)$

Klein's algorithm $O(n^2 \log n)$

recursive combing with HCS kernel composition - extra log factor?

9.5 HCS on permutation strings (HCSP)

[[[HCSP.

[**?**]

Weighted canonical antichain partition? $O(n \log n)$

[[[

9.6 Semi-local HCSP (TODO)

Semi-local HCSP

HCS kernel: scuffle measure, order $2n$? by Klein's lemma

Semi-local HCSP Algorithm: as in LCSP, replacing mmult by fmmult, order bounded

$O(n \log^2 n)$

9.7 Window and cyclic HCSP (TODO)

Replacing semi-local LCSP by semi-local HCSP

$O(n \log^2 n)$]]]

9.8 Affinity-sensitive semi-local HCS (TODO)

9.9 Heaviest clique in a circle graph (TODO)

Replacing semi-local LCSP by semi-local HCSP
$O(n \log^2 n)$

Chapter 10

Compressed string comparison

10.1 Grammar-compressed (GC-) strings

Nowadays most of the data used in science and technology are compressed. From an algorithmic viewpoint, 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. [15] and by Rytter [184]; for a recent survey on the topic, see Lohrey [150]. Efficient algorithms for compressed strings can also be applied to achieve speedup over ordinary string processing algorithms for plain strings that are highly compressible.

The following generic compression model is well-studied, and covers many data compression formats used in practice.

Definition 10.1 *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:*

$$t_k = \alpha \qquad t_k = uv$$

where α is an alphabet character, and each of u, v is either an alphabet character, or symbol t_i for some i, $1 \leq i < k$. □

We identify every symbol t_k with the string it expands to; 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 10.2 The *Fibonacci string* "ABAABABAABAAB" of length 13 can be represented by the following SLP of length 6:

$$t_1 = \mathsf{A} \quad t_2 = t_1\mathsf{B} \quad t_3 = t_2t_1 \quad t_4 = t_3t_2 \quad t_5 = t_4t_3 \quad t_6 = t_5t_4$$

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. ■

As a special case, grammar compression includes the classical compression schemes LZ78 and LZW by Ziv, Lempel and Welch [223, 217]. 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 [222] and run-length compression, do not fit directly into the grammar compression model.

Kida et al. [132] generalised grammar compression to a richer model, called a *collage system*. Grammar compression is a equivalent to a subclass of collage systems called *regular*.

10.2 Semi-local LCS on GC-strings

Plain string against GC-string. Consider the LCS problem on a plain string a of length m and a GC-text string b of length n, generated by an SLP of length $\bar{n}$. 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 $\bar{n}$).

In the special case of LZ77 or LZW compression of the text, the algorithm of Crochemore et al. [67] solves the LCS problem in time $O(m\bar{n})$. Thus, surprisingly, LZ77 or LZW compression of one of the input strings only slows down the LCS computation by a constant factor relative to the classical dynamic programming LCS algorithm, or by a polylogarithmic factor relative to the best known LCS algorithms.

The general case of arbitrary GC-strings appears more difficult. A GC-string is a special case of a context-free language (one that consists of a single string). Therefore, the LCS problem on a plain string against a GC-string can be regarded as a special case of the edit distance problem between a string a and context-free language. For this more general problem, Myers [164] gave an algorithm running in time $O(m^3\bar{n} + m^2 \cdot \bar{n} \log \bar{n})$. In [198], we gave an algorithm for the extended substring-string LCS problem on a plain string against a GC-string, running in time $O(m^{1.5}\bar{n})$. Lifshits [148] asked whether the LCS problem in the same setting can be solved in time $O(m\bar{n})$.

We now consider the semi-local LCS problem on a plain string a against a GC-string b. First, we observe that the semi-local LCS kernel $\mathsf{P}_{a,b}$ contains $m+n$ nonzeros, therefore its size may be exponential in the input size, and therefore prohibitive to compute and store. However, the extended substring-string LCS kernel $\mathsf{P}^{\swarrow(k)}_{a,b}$ contains only m nonzeros for any k. Having obtained $\mathsf{P}^{\swarrow(k)}_{a,b}$ for $k = 0, n$, we can use the structure of the SLP generating string b to recover the missing implicit string-substring scores. The resulting algorithm is as follows.

Algorithm 10.3 (Semi-local LCS: plain string vs GC-string)

Input: plain string a; SLP of length $\bar{n}$, generating string b.

Output: a semi-local LCS oracle for a against b.

Description. We obtain extended substring-string LCS kernels $\mathsf{P}^{\swarrow(k)}_{a,b}$, $k = 0, n$ by recursion on the input SLP as follows. Let $b = b'b''$ be the statement generating b.

Recursion base: $n = \bar{n} = 1$. The semi-local LCS kernel $\mathsf{P}_{a,b}$ is obtained by either Algorithm 5.33 or Algorithm 5.36 on plain strings a, b.

Recursive step: $n \geq \bar{n} > 1$. We call the algorithm recursively on a against b', b'', obtaining extended substring-string LCS kernels $\mathsf{P}^{\swarrow(k)}_{a,b'}$, $k = 0, n'$, $\mathsf{P}^{\swarrow(k)}_{a,b''}$, $k = 0, n''$. Given these kernels, extended substring-string LCS kernels $\mathsf{P}^{\swarrow(k)}_{a,b}$, $k = 0, n', n$, are computed by Corollary 5.32. (End of recursive step.)

Throughout the computation, each of the kernels has m nonzeros. However, their indices may range up to $m + n$, which may be exponential in the input size. To prevent this from happening, in every recursion level we delete most of the zero rows and zero columns from the kernels, shrinking their index range. Kernels $\mathsf{P}^{\swarrow(k)}_{a,b}$, $k = 0, n', n$, taken together, have $3m$ nonzeros, therefore there are at most $3m$ indices corresponding to a nonzero row in at least one of the kernels (possibly fewer, if a nonzero row index is shared by more than one kernel), and similarly for nonzero columns. Therefore, we can delete a sufficient number of zero rows and zero columns from the kernels, remapping row and column indices to a contiguous range up to $3m$, while preserving the linear order of the indices and the applicability of Corollary 5.32.

The extended substring-string LCS kernels $\mathsf{P}^{\swarrow(k)}_{a,b}$, $k = 0, n$ represent implicitly the suffix-prefix, prefix-suffix and substring-string LCS scores for p against t by Theorem 5.17 (LCS matrix canonical decomposition).

It remains to show how string-substring LCS scores are represented. The cross-semi-local LCS kernel $\mathsf{P}^{\swarrow(n')}_{a,b}$ represents implicitly the string-cross-substring LCS scores for a against the decomposition $b = b'b''$ in every recursive step. Given a query substring of t, we follow recursively the index remapping, in order to find the unique SLP statement for which that query

substring is a cross-substring. The cross-semi-local LCS kernel $\mathsf{P}^{\swarrow(n')}_{a,b}$ for that statement represents implicitly the string-substring score in question.

Theorem 4.6 can be used for querying efficiently all the implicit scores.■

Theorem 10.4 *There exists an oracle for semi-local LCS on a plain string against a GC-string with*

- *size* $O(m\bar{n})$ *and query time* $O(s+\bar{n})$*;*
- *size* $O\big((m+n)\log(m+n)\bar{n}\big)$ *and query time* $O(\log^2 s+\bar{n})$

where s *is the length of the shorter string or substring in the query. Such an oracle can be obtained in time* $O(m\log m\cdot\bar{n})$*.* □

Proof In Algorithm 10.3, the running time of a recursive step is dominated by the application of Corollary 5.32, which runs in time $O(m\log m)$. There are $\bar{n}$ recursive steps in total, therefore the overall running time is $O(m\log m\cdot\bar{n})$.

The data structure of Theorem 4.6 can be built in time $O(m\log m)$ in every recursion level, therefore the total time required for building such data structures is also $O(m\log m\cdot\bar{n})$.

A suffix-prefix, prefix-suffix or string-substring LCS score query has to provide an index (or a pair of indices) into the uncompressed string t, which needs to be represented in space $O(\bar{n})$. The query time is dominated by the inverse remapping of indices, which can be performed recursively in time $O(\bar{n})$.

A substring-string LCS score query can be answered in time $O(\log^2 m)$ by Theorem 4.6. ■

?? implies a solution for the (global) LCS problem between a plain string and a GC-string, running in time $O(m\log m\cdot\bar{n})$; the global LCS score can be obtained from either of the extended substring-string matrices $\mathsf{P}^{\swarrow(0)}_{p,t}$, $\mathsf{P}^{\swarrow(n)}_{p,t}$ by Theorem 5.17 (LCS matrix canonical decomposition). Neither the computation of the cross-semi-local matrices, nor the data structure of Theorem 4.6 are required in this case. This running time should be contrasted with standard LCS algorithms on plain strings, running in time $O\big(\frac{mn(\log\log n)^2}{\log^2 n}+n\big)$ [155, 67].

GC-string against GC-string. The LCS problem on a pair of GC-strings has been considered by Lifshits and Lohrey [149, 150], and proven to be PP-hard (and therefore NP-hard).

Hermelin et al. [**?**] and Gawrychowski [97] refined the application of our techniques as follows. They consider the rational-weighted alignment problem (equivalently, the LCS or Levenshtein distance problems) 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 $r = m + n$. The algorithm of [**?**] runs in time $O\big(r \log(r/\bar{r}) \cdot \bar{r}\big)$, which is improved in [97] to $O\big(r \log^{1/2}(r/\bar{r}) \cdot \bar{r}\big)$. In both cases, the algorithm of Theorem 4.16 (the Steady Ant algorithm) is used as a subroutine.

10.3 Subsequence matching in GC-strings

Subsequence recognition. The SR problem on a plain pattern p against a GC-text t can be solved by a simple folklore algorithm as follows. First, we generalise the problem slightly, looking for the length of the longest prefix of p that is a subsequence of t.

Algorithm 10.5 (SR: plain pattern vs GC-text)

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

Output: length k of the longest prefix of p that is a subsequence of t: 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 p' denote the suffix of p obtained by removing that prefix, and let k'' be the length of the longest prefix of p' that is a subsequence of t''. We first call the algorithm recursively on p and t' to obtain k', and then on p' and t'' to obtain k''. We then return $k = k' + k''$. (End of recursive step) ■

Theorem 10.6 *The SR problem on a plain pattern against a GC-text can be solved in time $O(m\bar{n})$.* □

Proof Algorithm 10.5 runs in time $O(k\bar{n})$, which can be proved by structural induction on the SLP. The running time of the two recursive calls is respectively $O(k'\bar{n})$ and $O(k''\bar{n})$ by the inductive hypothesis. Therefore, the overall running time of the algorithm is $O(k'\bar{n}) + O(k''\bar{n}) + O(1) = O(k\bar{n}) = O(m\bar{n})$. ■

Subsequence matching. We now consider the SR problem on a plain pattern p against a GC-text t. The number of matching substrings in this setting may be exponential in the input size. Therefore, it is natural to parameterise the running time by the output size, which we denote by *output*. We refer to this version of the SM problem as the *reporting version*. We will also consider the *counting version*, which, instead of the locations of the matching substrings, only asks for their overall number. The running time

of our reporting SM algorithm will only differ from its counting version by the additive term $O(output)$.

The minimal-substring and window SM problems for a plain pattern against a GC-text have been considered by Cégielski et al. [52]. For both problems, they gave an algorithm running in time $O(m^2 \log m \cdot \bar{n} + output)$. In [198], we gave an algorithm reducing the running time for these problems to $O(m^{1.5}\bar{n} + output)$, and then in [207] to $O(m \log m \cdot \bar{n} + output)$.

An algorithm for minimal-substring SM in a GC-text can be obtained by the LCS kernel method as an extension of Algorithm 5.47.

Algorithm 10.7 (Minimal-substring SM: plain pattern vs GC-text)

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

Output: implicit locations of inclusion-minimal matching substrings in t.

Description. The algorithm runs in two phases.

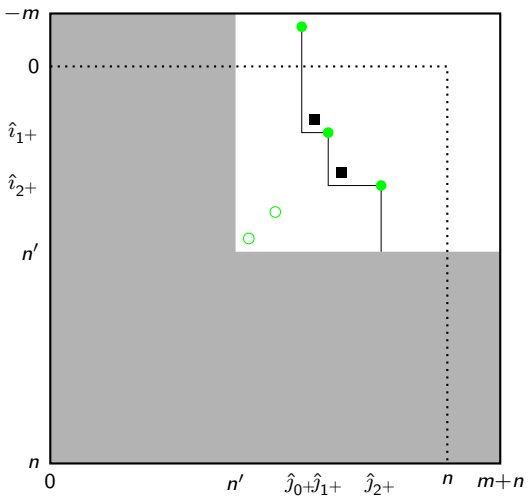
Phase 1. We execute Algorithm 10.3 (semi-local LCS for a plain pattern against a GC-text), obtaining for every SLP statement $t = t't''$ the extended substring-string LCS kernels $\mathsf{P}^{\swarrow(k)}_{p,t}$, $k = 0, n', n$. As in Algorithm 10.3, index remapping is performed in the background in order to prevent exponential growth of the indices.

Phase 2. For every SLP statement $t = t't''$, we execute Algorithm 5.47 (minimal-substring SM), replacing the semi-local LCS kernel $\mathsf{P}_{p,t}$ with the cross-semi-local LCS subkernel $\mathsf{P}^{\swarrow(n')}_{p,t}$. As in Algorithm 10.3, we obtain the $\ll$-chain $L \cap \langle -m : n'; n' : m + n \rangle$ of $\gtrsim$-maximal nonzeros in the subkernel $\mathsf{P}^{\swarrow(n')}_{p,t}$, ignoring the nonzeros of the LCS kernel $\mathsf{P}_{p,t}$ outside the extended string-substring range $\langle -m : n'; n' : m + n \rangle$, since such nonzeros have been processed at lower recursion levels. We then obtain the skeleton $\ll$-chain $M \cap [-m : n'; n' : m + n]$. All inclusion-minimal matching non-degenerate cross-substrings of t at the current recursion level can now be obtained as the elements of subchain $M \cap [-m : n'; n' : m + n] \cap [0 : n' - 1; n' + 1 : n] = M \cap [0 : n' - 1; n' + 1 : n]$. ■

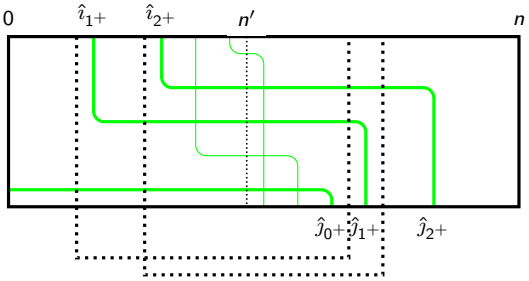
Theorem 10.8 *The counting version of both the minimum-substring and the window SM problems on a plain pattern against a GC-text can be solved in time $O(m \log m \cdot \bar{n})$. The reporting version of each problem can then be solved in additional time $O(output)$.* □

Proof In Phase 1 of Algorithm 10.7, the extended substring-string LCS kernels for all SLP statements are obtained in time $O(m \log m \cdot \bar{n})$ **??**.

In Phase 2, for every SLP statement $t = t't''$, Algorithm 10.7 returns the relative locations of all inclusion-minimal matching non-degenerate cross-substrings of t. Furthermore, every non-trivial substring of the uncompressed text corresponds to a non-degenerate cross-substring for some SLP



(a) Cross-semi-local LCS kernel $\mathsf{P}^{\swarrow(n')}_{p,t}$; chains L, $M \cap [-m : n'; n' : m+n]$



(b) Corresponding sticky braid

Figure 10.1: Execution of Algorithm 10.7 (minimal-substring SM in GC-text)

statement, under an appropriate transformation of indices. By another recursion on the structure of the SLP, it is now straightforward to obtain either the absolute locations, or the count of all the inclusion-minimal matching substrings, as well as all the matching substrings of length w.

The total running time is dominated by Phase 1, and is therefore as claimed. ∎

Example 10.9 Figure 10.1 illustrates the execution of Phase 2 of Algorithm 10.7 for a particular SLP statement, using the same data and conventions as Figure 5.10. ∎

Yamamoto et al. [220], using elementary techniques, gave an even faster algorithm than that of Theorem 10.8, running in time $O(m \cdot \bar{n} + \mathit{output})$.

10.4 Approximate matching in GC-strings

We now consider the AMatch problem and its special cases (EDMatch, Levenshtein EDMatch) on a GC-text. Similarly to the SM problem, the output size of the AMatch problem in a GC-text may be exponential in its input size. Therefore, in keeping with the setup of Section 10.3, we consider the problem's counting version in addition to its reporting version, and parameterise the running time by *output*.

The threshold Levenshtein problem on a compressed text has been studied by Kärkkäinen et al. [127]. For a plain pattern of length m, a GC-text of length $\bar{n}$, and an edit distance threshold $k > 0$, the (suitably generalised) algorithm of [127] solves the threshold Levenshtein problem in time $O(m\bar{n}k^2 + \mathit{output})$. In the special case of LZ78 or LZW compression, the running time is reduced to $O(m\bar{n}k + \mathit{output})$. Bille et al. [36] gave an efficient general scheme for adapting an arbitrary AMatch algorithm to work on a GC-text. The running time of the resulting algorithms is parameterised by the plain text length n; note that $\log n \leq \bar{n}$. In particular, when the algorithms by Landau and Vishkin [145] and by Cole and Hariharan [60] are each plugged into this scheme, the resulting algorithm solves the threshold Levenshtein respectively in time

$$O\big(\bar{n}mk + \bar{n}\log n + \mathit{output}\big)$$
$$O\big(\bar{n}(m + k^4) + \bar{n}\log n + \mathit{output}\big)$$

In the special case of LZ78 or LZW compression, Bille et al. [34] show that it is possible to remove the term $\bar{n}\log n$, reducing the running time to $O(\bar{n}mk + \mathit{output})$ for the algorithm based on [145], and to $O\big(\bar{n}(m + k^4) + \mathit{output}\big)$ for the one based on [60].

Assuming a rational scoring scheme with constant denominator, an algorithm for threshold AMatch in a GC-text can be obtained by the LCS kernel method as follows.

Theorem 10.10 *Assuming a rational scoring scheme with denominator $\nu = O(1)$, the counting version of both the minimum-substring and the window threshold AMatch problems for a plain pattern against a GC-text can be solved in time $O(m\log m \cdot \bar{n})$. The reporting version of each problem can then be solved in additional time $O(\mathit{output})$.* □

Proof Let (w_+, w_0, w_-) be the scoring scheme. Without loss of generality, assume $w_+ - 2w_- = 1$ (otherwise, the scoring scheme can be scaled to satisfy this condition without affecting the correctness of the algorithm).

Consider the corresponding regular scoring scheme $(1, w_0^* = w_0 - 2w_-, 0)$. We follow the structure of Algorithm 10.7 (minimal-substring SM in GC-text) replacing the extended substring-string LCS kernels with alignment kernels and composing them by Corollary 6.21. In Phase 1 of the algorithm,

for each SLP statement we obtain the extended substring-string alignment kernels $\mathrm{S}^{*\nearrow(k)}_{p,t}$, $k = 0, n', n$. As in Algorithm 10.3 and Theorem 10.8, index remapping is performed in the background in order to prevent exponential growth of the indices.

In Phase 2, for every SLP statement we perform a computation analogous to the algorithm of Theorem 6.29, replacing the semi-local alignment kernel $\mathrm{S}_{p,t}$ with its cross-semi-local submatrix $\mathrm{S}^{\nearrow(n')}_{p,t}$. We then reverse score regularisation and perform row maxima searching, obtaining for every SLP statement the relative locations of its inclusion-minimal matching non-degenerate cross-substrings. Similarly to Algorithm 10.3 and Theorem 10.8, it is now straightforward to obtain either the absolute locations, or the count of all the inclusion-minimal matching substrings, as well as all the matching substrings of length w.

The total running time is dominated by Phase 1, and is therefore as claimed. ∎

Theorem 10.10 improves on the algorithm of [127] for $k = \omega\big((\log m)^{1/2}\big)$ in the case of general GC-compression, and $k = \omega(\log m)$ in the case of LZ78 or LZW compression. Theorem 10.10 also improves on the algorithms of [34, 36] for $k = \omega\big((m \log m)^{1/4}\big)$, in the case of both general GC-compression and LZ78 or LZW compression.

Chapter 11

Beyond semi-locality

11.1 Window-substring comparison

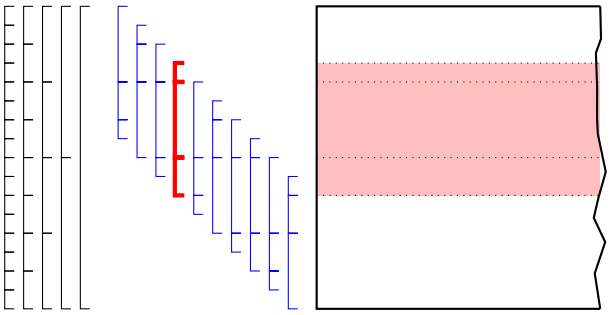
Dot plots. Recall that, given a string and a fixed length $w \geq 0$, a w-window is any substring of length w. Given input strings a, b, a *dot plot* (also known as a *diagonal plot* or a *dot matrix*) gives a comparison score for every w-window of string a against every w-window of string b.

Dot plots were introduced by Gibbs and McIntyre [102] and by Maizel and Lenk [152] (see also Crochemore et al. [64]). As a form of local string alignment, they provide a convenient method for analysing and visualising string similarity in computational molecular biology. Classical dot plots compare pairs of windows in terms of their Hamming score, i.e. the count of matching character pairs with no gaps allowed. Such a *Hamming-scored* dot plot can be computed in time $O(mn)$ by the algorithm of [152, 161], which has been implemented in several software packages (see e.g. [196, 177, 1]). A faster suffix-tree based algorithm has been proposed and implemented by Krumsiek et al. [137]. Enhancements of the dot plot approach have been proposed by Huang and Zhang [115] and by Putonti et al. [175].

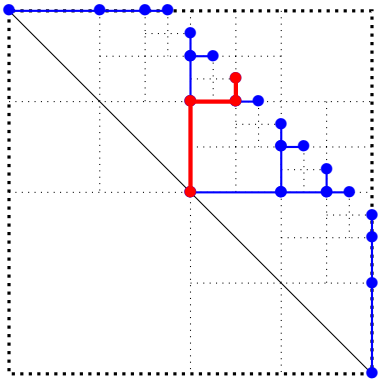
Hamming-scored dot plots have been used in the past for obtaining biologically significant window-window alignments. However, the Hamming scoring scheme used for each window pair only considers matches and mismatches, and does not account for gaps in the alignment. For this reason, it is less sensitive than even the basic LCS-scored alignment, and especially than an alignment with a well-chosen scoring scheme.

Window-substring LCS. We define an alternative to Hamming-scored dot plots, which is based on the LCS scoring scheme.

Definition 11.1 *Given strings a, b, and a window length w, the* window-substring *(respectively,* window-window*)* LCS problem *asks for the LCS score of every w-window in string a against every substring (respectively, every w-window) in string b.* □



(a) Canonical substrings and w-windows



(b) Window decomposition forest

Figure 11.1: An execution of Algorithm 11.2 (window-substring LCS)

A naive approach to the window-substring LCS problem would be to treat it as $O(m)$ independent instances of the string-substring LCS problem, one for each window of string a against the whole string b. Using Algorithm 5.36 (semi-local LCS by iterative combing) as a subroutine, the resulting algorithm would run in overall time $O(m) \cdot O(nw) = O(mnw)$. If window length w is sufficiently large, then the running time could be reduced slightly by the micro-block speedup of Algorithm 5.41.

Using the LCS kernel approach, we can obtain an algorithm for window-substring LCS that improves substantially on the described naive approach.

Algorithm 11.2 (Window-substring LCS)

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

Output: implicit string-substring LCS scores for every w-window of a against whole b.

Description. The window-substring LCS kernel $\mathsf{P}^{\nearrow}_{a\langle k:l\rangle,b}$ for every $k, l \in [0:m]$, $l - k = w$, is obtained as follows.

Let s be an arbitrary power of 2, $1 \leq s \leq m$. We define an (s,t)-*substring* in a to be a substring $a\langle k\colon l\rangle$, $l-k = st$, such that k and l are both multiples of s. A $(1,t)$-substring is the same as a t-window. A *canonical s-window* is an $(s,1)$-substring for some s. Every substring of a of length t can be decomposed into a concatenation of at most $\log t$ canonical s-windows with various values of s.

In the following, by processing a substring $a\langle k\colon l\rangle$, we mean computing the string-substring LCS kernel $\mathsf{P}^{\nearrow}_{a\langle k:l\rangle,b}$. Without loss of generality, we assume that m is a power of 2. The algorithm proceeds in two phases as follows.

1. Processing recursively all $(s,1)$-substrings for a fixed s, $1 \leq s \leq m$, with $s = 1$ as recursion base. The task of processing all canonical windows is performed by calling this procedure with $s = m$.

2. Processing recursively all (s,t)-substrings for fixed s, t, $1 \leq st \leq m$, with $t = 1$ as recursion base. The task of processing all w-windows is performed by calling this procedure with $s = 1$, $t = w$. The procedure is based on a zeroless binary representation of w, which is similar to a binary representation, but with digit set $\{0,1\}$ replaced by $\{1,2\}$ (see e.g. [169, p.124]).

We now describe each phase in more detail.

Phase 1 (Processing $(s,1)$-substrings, $1 \leq s \leq m$). Such substrings are processed recursively as follows.

Recursion base: $s = 1$. For every $(1,1)$-substring $a\langle k\colon l\rangle$ of length $l-k = 1$, matrix $\mathsf{P}^{\nearrow}_{a\langle k:l\rangle,b}$ is computed by Algorithm 5.36 (semi-local LCS by iterative combing) on string $a\langle k\colon l\rangle$ against string b.

Recursive step: $s > 1$. We call the procedure recursively to process all $(\frac{s}{2},1)$-substrings. Let $a\langle k\colon l\rangle$, $l-k = s$, be an $(s,1)$-substring. We have

$$a\langle k\colon l\rangle = a\big\langle k\colon \tfrac{k+l}{2}\big\rangle\, a\big\langle \tfrac{k+l}{2}\colon l\big\rangle$$

where the two substrings in the right-hand side are both $(\frac{s}{2},1)$-substrings that have been processed by the recursive call. We can now process string $a\langle k\colon l\rangle$ by Theorem 5.30.

(End of recursive step)

Phase 2 (Processing (s,t)-substrings, $1 \leq st \leq m$). Such substrings are processed recursively as follows.

Recursion base: $t = 1$. For any s, all $(s,1)$-substrings have already been processed in the first phase.

Recursive step: $t > 1$. Let $a\langle k\colon l\rangle$, $l-k = st$, be an (s,t)-substring. Consider its decompositions into a concatenation of substrings

$$a\langle k\colon l\rangle = a\langle k\colon l-s\rangle\, a\langle l-s\colon l\rangle = a\langle k\colon k+s\rangle\, a\langle k+s\colon l\rangle \tag{11.1}$$

■

We have two cases.

Case 1: t is even. We call the procedure recursively to process all $(s, t-1)$-substrings. In (11.1), substrings $a\langle k\colon l-s\rangle$ and $a\langle k+s\colon l\rangle$ are both $(s,t-1)$-substrings that have been processed in the recursive call.

Case 2: t is odd. We call the procedure recursively to process all $(2s, \frac{t-1}{2})$-substrings (in other words, every other $(s,t-1)$-substring). Now consider an (s,t)-substring $a\langle k\colon l\rangle$. In (11.1), substrings $a\langle k\colon l-s\rangle$ and $a\langle k+s\colon l\rangle$ are both $(s,t-1)$-substrings. Furthermore, if $\frac{k}{s}$ is even, then $\frac{l}{s} = \frac{l-k}{s} + \frac{k}{s} = t + \frac{k}{s}$ is odd, and substring $a\langle k\colon l-s\rangle$ is a $(2s, \frac{t-1}{2})$-substring that has been processed in the recursive call. Likewise, if $\frac{k}{s}$ is odd, then $\frac{l}{s}$ is even, and substring $a\langle k+s\colon l\rangle$ is a $(2s, \frac{t-1}{2})$-substring that has been processed by the recursive call.

In both above cases, substrings $a\langle l-s\colon l\rangle$ and $a\langle k\colon k+s\rangle$ are both canonical $(s,1)$-substrings that have been processed in the first phase. We can now process string $a\langle k\colon l\rangle$ by Theorem 5.30, choosing the appropriate decomposition in (11.1).

(End of recursive step)

Theorem 11.3 *The window-substring LCS problem can be solved implicitly in time $O(mn)$.* □

Proof We consider each phase of Algorithm 11.2.

First phase. Starting at the top level of the recursion tree, the number of sticky matrix multiplications doubles in every subsequent level. The running time for each matrix multiplication is $O(n \log s)$, where the parameter s halves in every subsequent level. Therefore, the running time of the whole phase is dominated by the bottom level of the recursion tree, where we have $O(m)$ matrix multiplications, each running in time $O(n)$. The running time of the whole phase is $O(m) \cdot O(n) = O(mn)$.

Second phase. Starting at the top level of the recursion tree, the number of sticky matrix multiplications halves in every level for which the parameter t is odd, which is at least a half of all levels. In every level for which t is even, the number of matrix multiplications does not change. The running time for each matrix multiplication is $O(n \log s)$ by Theorem 5.30. Here, the parameter s doubles in every level where the parameter t is odd, and does not change in every level where the parameter t is even. Therefore,

the running time of the whole phase is dominated by the top level of the recursion tree, where we have $O(m)$ matrix multiplications, each running in time $O(n)$. The running time of the whole phase 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 $O(mn)$.

The solution of the window-substring LCS problem can be represented implicitly by the string-substring LCS kernel for each w-substring of a against b. Random access to explicit window-substring LCS scores can then be performed by Theorem 4.6 in time $O(\log^2 n)$. An explicit solution to the window-window LCS problem (i.e. an alignment plot) can be obtained by Theorem 4.8 in additional time $O(mn)$, by performing a diagonal batch query directly on each of the string-substring LCS kernels. ∎

Note that the asymptotic running time of Algorithm 11.2 is independent of the window length w, and matches the running time of Hamming-scored dot plot algorithms.

Example 11.4 Figure 11.1 shows shows the computation of window-substring LCS by Algorithm 11.2 and Theorem 11.3 on string a of length 16 with window size 7, against string b of arbitrary length. Figure 11.1a shows the canonical substrings ($(s, 1)$-substrings) of a for $s = 1, 2, 4, 8, 16$ in black, and 7-windows ($(1, 7)$-substrings) in blue. For each 7-window, the figure shows its decomposition into canonical substrings. For one of the 7-windows, highlighted in thick red, the corresponding area is outlined in the alignment dag. Figure 11.1b represents substrings of a by points in the plane, and the decompositions into canonical substrings by a forest of trees. Each 7-window $a\langle k : l\rangle$ corresponds to a leaf of a decomposition tree. The canonical substrings in a decomposition of the 7-window correspond to the edges on the path from that leaf to the root of the tree; for the 7-window shown in red, its corresponding path is also highlighted in red. ∎

Fixed total length LCS. Definition 11.1 and Algorithm 11.2 can be extended to other types of *length-constrained comparison*. In particular, we can consider a related symmetric problem, where the total length of strings a, b is fixed.

Definition 11.5 *Given strings a, b, and a length parameter w, the* fixed total length LCS problem *asks for the LCS score of every substring in string a against every substring in string b, such that the sum of the two substrings' lengths is w.* □

Note that the window-window LCS problem with window length w (Definition 11.1) is a special case of both the window-substring LCS problem with

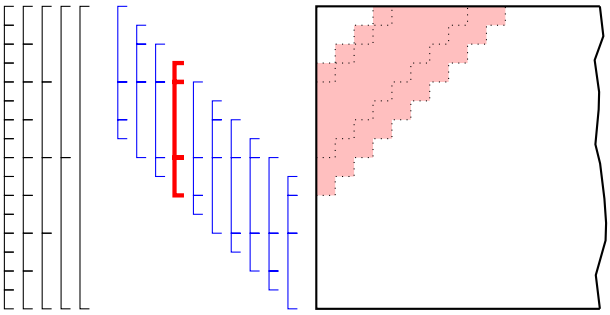


Figure 11.2: Fixed total length LCS

window length w, and the fixed total length LCS problem with total length parameter $2w$.

The fixed total length LCS problem can be solved by a technique similar to Algorithm 11.2 (window-substring LCS) and Theorem 11.3. Observe that a window-substring LCS kernel corresponds to a horizontal rectangular strip of width w in the LCS grid, defined by the window in string a and the whole string b. To obtain an algorithm for the fixed total length LCS problem, we replace these with an alternative type of kernel, corresponding to an anti-diagonal strip of rectilinear width w. Otherwise, the algorithm's structure is identical to Algorithm 11.2, and the running time remains $O(mn)$, as in Theorem 11.3.

Example 11.6 Figure 11.2 shows the computation of a fixed total length LCS on string a of length 16 with length parameter 7, against string b of arbitrary length. Conventions are the same as in Figure 11.1. ■

Window-substring and fixed total length alignment. Definitions 11.1 and 11.5 can be extended to alignment under an arbitrary scoring scheme.

Definition 11.7 *Assuming a fixed scoring scheme, the* window-substring, window-window, *and* fixed total length alignment problems *are defined analogously to Definitions 11.1 and 11.5 (window-window, window-substring, and fixed total length LCS problems), replacing the LCS score by the alignment score.* □

By analogy with Hamming-scored dot plots, given a fixed window length, we call the explicit matrix of all window-window alignment scores an *alignment-scored dot plot*, or simply an *alignment plot*.

Computation of an alignment plot is a key subroutine in the approximate repeat searching method by Federico et al. [87]. A concept similar to alignment plots is also introduced by Rasmussen et al. [176], who propose an algorithm for local alignment in fixed-length windows. The difference of our approach from that of [176] is that the latter is designed to be very efficient

when the similarity threshold is high, but becomes prohibitively expensive when looking for relatively low-similarity matches (e.g. 70% similarity).

Theorem 11.8 *Assuming a rational scoring scheme with denominator $\nu = O(1)$, the window-substring, window-window, and fixed total length alignment problems can all be solved implicitly in time $O(mn)$.* □

Proof The solution can be obtained as in Algorithm 11.2 (window-substring LCS) and the subsequent discussion of fixed total length LCS, replacing the LCS grid $\mathsf{G}_{a,b}$ with the alignment grid $\mathrm{G}_{a,b}$, and LCS kernels with alignment kernels. Matrix composition by Theorem 5.30 is replaced with Theorem 6.20. The claimed running time is obtained as in Theorem 11.3. ■

The approach of Theorem 11.8 would work equally efficiently to obtain an alignment plot for any similarity threshold. Using these ideas, our alignment plot method has been implemented and applied to the detection of evolutionary conserved regions in plant and animal DNA by Picot et al. [172], Baxter et al. [27], Davies et al. [71].

11.2 Fragment-substring comparison

Fragment-substring LCS. We now generalise fixed-length substring comparison as follows. Suppose that an arbitrary set of substrings of various lengths in string a are prescribed as *fragments*. We assume that all the fragments are non-empty, and denote their number by r, $m \leq r \leq \binom{m+1}{2}$.

Definition 11.9 *Given strings a, b, and a set of fragments in a, the* fragment-substring LCS problem *asks for the LCS score of every fragment in a against every substring in b.* □

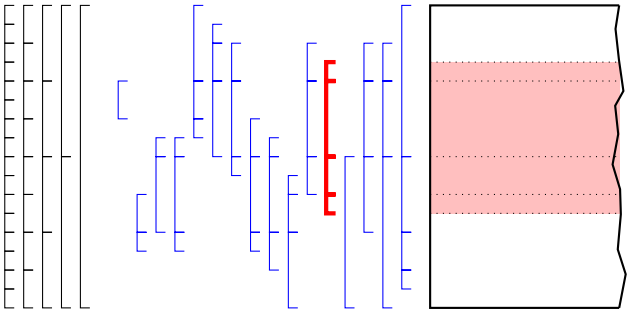
The window-substring LCS problem is a special case of the fragment-substring LCS problem, where the fragments are the w-windows in a.

A naive approach to the fragment-substring LCS problem would be to treat it as r independent instances of the string-substring LCS problem, one for each window of string a against the whole string b. Using Algorithm 5.36 (semi-local LCS by iterative combing) as a subroutine, the resulting algorithm would run in overall time $O(r) \cdot O(mn) = O(rmn)$. The running time could be reduced slightly by the micro-block speedup of Algorithm 5.41.

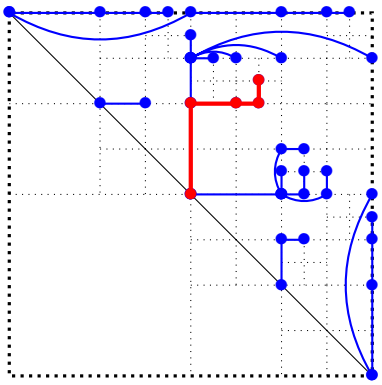
Using the LCS kernel approach, we can obtain an algorithm for fragment-substring LCS that improves substantially on the described naive approach.

Algorithm 11.10 (Fragment-substring LCS)

Input: strings a, b of length m, n, respectively; a set of r supporting intervals for the fragments in a.



(a) Canonical substrings and fragments



(b) Fragment decomposition forest

Figure 11.3: An execution of Algorithm 11.10 (fragment-substring LCS)

Output: implicit string-substring LCS scores for every fragment of a against whole b.

Description. The string-substring LCS kernel $\mathsf{P}^{\nearrow}_{a',b}$ for every fragment a' is obtained as follows.

The algorithm's structure is similar to that of Algorithm 11.2, and relies on the same definition of (s,t)-substring. As in Algorithm 11.2, without loss of generality, we assume that m is a power of 2. The algorithm proceeds in two phases as follows.

1. As in Algorithm 11.2.

2. We give a recursive procedure for processing a set of fragments, assuming that every fragment in the set is an (s,t)-substring for a fixed s and an arbitrary t, $1 \le st \le m$, with $t = 1$ as recursion base. The original set of fragments is processed by calling this procedure with $s = 1$. ■

We now describe each phase in more detail.

First phase. As in Algorithm 11.2.

Second phase. We give a recursive procedure for processing a set of fragments, each assumed to be an (s,t)-substring for a fixed s and an arbitrary t, $1 \le st \le m$.

Recursion base: $t = 1$. Every fragment is an $(s,1)$-substring for some s, therefore it has already been processed in the first phase.

Recursive step: $t > 1$. Let $a\langle k{:}\, l\rangle$, $l - k = st$ be a fragment. Consider its decompositions (11.1) into a concatenation of substrings. For each fragment, we define its *parent* and call the procedure on the set of all parents recursively as follows. We have two cases.

Case: for some fragments, value t is even. Given a fragment $a\langle l{:}\, s\rangle$ with an even value of t, we choose arbitrarily either of substrings $a\langle k{:}\, l - s\rangle$ and $a\langle k + s{:}\, l\rangle$ as the fragment's parent. A fragment with an odd value of t is considered to be its own parent. We call the procedure recursively to process the set of all fragments' parents (note that there may be fewer parents than original fragments, since a parent may be shared by several different fragments). In (11.1), substrings $a\langle k{:}\, l-s\rangle$ and $a\langle k+s{:}\, l\rangle$ are both $(s,t-1)$-substrings, at least one of which has been processed in the recursive call as the parent of $a\langle l{:}\, s\rangle$.

Case: for all fragments, value t is odd. Given a fragment $a\langle l{:}\, s\rangle$ with an even value of t, we choose substring $a\langle k{:}\, l - s\rangle$ as the fragment's parent if $\frac{k}{s}$ is even, and substring $a\langle k + s{:}\, l\rangle$ if $\frac{k}{s}$ is odd. We call the procedure recursively to process the set of all fragments' parents. In (11.1), the substrings $a\langle k{:}\, l - s\rangle$ and $a\langle k + s{:}\, l\rangle$ are both $(s, t - 1)$-substrings, and one of them is a $(2s, \frac{t-1}{2})$-substring that has been processed by the recursive call as the parent of $a\langle l{:}\, s\rangle$.

In both above cases, substrings $a\langle l-s{:}\, l\rangle$ and $a\langle k{:}\, k+s\rangle$ are both canonical $(s,1)$-substrings that have been processed in the first phase. We can now process fragment $a\langle k{:}\, l\rangle$ by Theorem 5.30, choosing the appropriate decomposition in (11.1).

(End of recursive step)

Theorem 11.11 *The fragment-substring LCS problem can be solved implicitly in time* $O(rn \log^2 m)$. □

Proof We consider each phase of Algorithm 11.10.

First phase. As in Algorithm 11.2, the total running time of this phase is $O(mn)$.

Second phase. The recursion has $\log m$ levels. In every level of the recursion, the number of matrix multiplications is at most $O(r)$. The running time for each matrix multiplication is at most $O(n \log m)$ by Theorem 5.30. Therefore, the running time of the whole phase is $O(r \log m \cdot n \log m) = O(rn \log^2 m)$.

Total. The overall running time is dominated by the second phase, and is therefore $O(rn \log^2 m)$.

Random access to explicit fragment-substring LCS scores can be performed by Theorem 4.6 in time $O(\log^2 n)$. ∎

Example 11.12 Figure 11.3 shows an execution of Algorithm 11.10 on string a of length 16, with 16 fragments of various sizes, against string b of arbitrary length. Conventions are the same as in Figure 11.1. ∎

Fragment-substring alignment. Definition 11.9 can be extended to alignment under an arbitrary scoring scheme.

Definition 11.13 *Assuming a fixed scoring scheme, the* fragment-substring alignment problem *is defined analogously to Definition 11.9 (fragment-substring LCS problem), replacing the LCS score by the alignment score.* □

Theorem 11.14 *Assuming a rational scoring scheme with denominator $\nu = O(1)$, the fragment-substring alignment problem can be solved implicitly in time $O(rn \log^2 m)$.* □

Proof The solution can be obtained as in Algorithm 11.10 (fragment-substring LCS), replacing the LCS grid $\mathsf{G}_{a,b}$ with the alignment grid $\mathrm{G}_{a,b}$, and LCS kernels with alignment kernels. Matrix composition by Theorem 5.30 is replaced with Theorem 6.20. The claimed running time is obtained as in Theorem 11.11. ∎

11.3 Fully-local comparison (TODO)

Local LCS oracle
- naive comp, memory $O(n^4)$, query $O(1)$
- Schmidt comp, memory $O(n^3)$, query $O(\log^2 n)$
- Sakai [189] comp, memory $O(n^2 polylog(n))$, query $O(n^{1/2} polylog(n))$
- Charalampopoulos+ oracle

11.4 Spliced comparison

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 such approach is *spliced comparison* by Gelfand et al. [100] (see also [**?**, 168]), which scores different candidate exon chains within a DNA sequence by comparing them to a known reference gene sequence. In keeping with the terminology and notation of Section 11.2, we call the candidate exons *fragments*, and denote their number by

r, $m \leq r \leq \binom{m+1}{2}$. We say that a subset of fragments form a *disjoint precedence chain*, if the endpoints of their supporting intervals form a chain in the disjoint precedence order. We identify a disjoint precedence chain with the string obtained by concatenating all its fragments in the precedence order.

Spliced LCS. We define spliced comparison by LCS scores as follows.

Definition 11.15 *The* spliced LCS score *for strings a, b, and a set of fragments in a, is the maximum LCS score of a disjoint precedence chain in a against whole string b. Given strings a, b, and a set of fragments in a, the* spliced LCS problem *asks for the spliced LCS score of a against b.* □

The biological interpretation of Definition 11.15 is as follows. The two DNA sequences correspond to strings a, b. Fragments in string a correspond to candidate exons, and their disjoint precedence to a gene assembly. The algorithm for spliced LCS based on [100] runs in time $O(m^2n)$.

In general, the number of candidate exons r may be as high as $\binom{m+1}{2} = O(m^2)$. Nevertheless, it is realistic to assume that, prior to assembly, the set of candidate exons undergoes some filtering, after which only a small fraction of candidate exons remains. An algorithm for spliced LCS by Kent et al. [131] assumes $r = O(m)$, and runs in time $O(m^{1.5}n)$. By an application of Theorem 11.11, the running time can be reduced to $O(mn \log^2 m)$. An improved algorithm by Sakai [187] runs in time $O(mn \log n)$.

For higher values of r, all the above algorithms provide a smooth transition in running time to the dense case $k = \binom{m+1}{2}$. In that case, the algorithms' running time $O(m^2n)$ is asymptotically equal to the algorithm based on [100].

We now describe an algorithm for spliced LCS, based on [187]. The algorithm uses a generalisation of the standard network alignment method, equivalent to the one used by [131].

Algorithm 11.16 (Spliced LCS)

Input: strings a, b of length m, n, respectively; a set of r supporting intervals for the fragments in a.

Output: disjoint precedence chain in a with the highest LCS score against b.

Description. The algorithm runs in two phases.

First phase. As in Algorithm 11.2.

Second phase. The highest-scoring disjoint precedence chain is obtained by dynamic programming as follows. Let $u_l[j] = j - h_l[j]$, $l \in [0\colon m]$, $j \in [1\colon n]$, where $h_l[j]$ is the highest LCS score across all disjoint precedence chains in prefix string $a\langle 0\colon l\rangle$, taken against prefix string $b\langle 0\colon j\rangle$. We assign $u_0[j] = j - h_0[j] = j$ for all j, where h_0 is the zero vector. We then obtain the

vectors u_l in the order of increasing l. Let $a_0 = a\langle k_0 : l\rangle$, $a_1 = a\langle k_1 : l\rangle$, ..., $a_t = \langle k_t : l\rangle$ be all the fragments of a terminating at index l. We have

$$u_l = u_{l-1} \oplus \left(u_{k_0} \boxdot \mathsf{P}^{\nearrow}_{a_0,b}\right) \oplus \left(u_{k_1} \boxdot \mathsf{P}^{\nearrow}_{a_1,b}\right) \oplus \ldots \oplus \left(u_{k_t} \boxdot \mathsf{P}^{\nearrow}_{a_t,b}\right) \quad (11.2)$$

for all $j \in [1 : n]$. Using the decomposition of each of a_0, a_1, ..., a_t into up to $\log m$ canonical substrings, each vector-matrix sticky product in (11.2) is computed by up to $\log m$ instances of vector-matrix sticky product with an LCS kernel for a canonical substring against b. The solution score is given by the value $h_m(n) = n - u_m(n)$. The solution disjoint precedence chain can be obtained by tracing the dynamic programming sequence backwards from vector h_m to vector h_0. ■

Theorem 11.17 *The spliced LCS problem can be solved in time $O(rn \log m)$.*□

Proof We consider each phase of Algorithm 11.16.

First phase. As in Algorithm 11.2, the total running time of this phase is $O(mn)$.

Second phase. For each of r fragments 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 4.15. Therefore, the total running time of this phase is $O(rn \log m)$.

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

Spliced alignment. Definition 11.15 can be extended to alignment under an arbitrary scoring scheme.

Definition 11.18 *Assuming a fixed scoring scheme, the* spliced alignment problem *is defined analogously to Definition 11.15 (the spliced LCS problem), replacing the LCS score by the alignment score.* □

Theorem 11.19 *Assuming a rational scoring scheme with denominator $\nu = O(1)$, the spliced alignment problem can be solved in time $O(rn \log m)$.*□

Proof The solution can be obtained as in Algorithm 11.16 (spliced LCS), replacing the LCS grid $\mathsf{G}_{a,b}$ with the alignment grid $\mathrm{G}_{a,b}$, and LCS kernels with alignment kernels. Matrix composition by Theorem 5.30 is replaced with Theorem 6.20. The claimed running time is obtained as in Theorem 11.17. ■

Similarly to Theorem 11.11, a sharper analysis for values of r close to $\Theta(m^2)$ leads to a smooth transition to the running time $O(m^2 n)$ in the dense case $r = \Theta(m^2)$, which is asymptotically equal to the running time of the dense spliced alignment algorithm of [100].

11.5 Smith–Waterman alignment

We continue to explore different versions of local string comparison. In the previous sections, we have considered comparison problems on a prescribed subset of substrings in string a, against all substrings in string b. The comparison score in these problems is taken separately for every pair of counterpart substrings.

We now consider a different type of local comparison, seeking to maximise the score taken across a range of pairs of counterpart substrings. Specifically, let us consider a fixed prefix in each input string, and aim to maximise the score taken across different suffixes of the counterpart prefixes. Note that the set of suffixes of a given prefix is nested. Under the LCS scoring scheme, maximising scores across such nested substrings is trivial: a substring in either of the input strings will always be preferred to any of its proper sub-substrings in terms of the LCS score. However, under more general scoring schemes such alignment becomes non-trivial, and has substantial applications.

Classical Smith–Waterman alignment. A classical alignment problem of this type was introduced by Smith and Waterman [195].

Definition 11.20 *Given strings a, b, and assuming a fixed scoring scheme, the* Smith–Waterman (SW) alignment problem *is defined as follows. For every prefix $a\langle 0\colon l\rangle$ and every prefix $b\langle 0\colon j\rangle$, the problem asks for the maximum alignment score over all possible choices of a suffix $a\langle k\colon l\rangle$ and a suffix $b\langle i\colon j\rangle$ from these prefixes:*

$$h[l;j] = \max_{k\in[0:l],i\in[0:j]} \mathsf{score}\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)$$

where $l \in [0\colon m]$, $j \in [0\colon n]$. □

A solution to the SW-alignment problem allows one to obtain, by another layer of maximisation, the highest alignment score across all substring pairs in a, b.

An efficient dynamic programming algorithm for SW-alignment was given by Smith and Waterman [195] and by Gotoh [107].

Algorithm 11.21 (SW-alignment)

Parameters: scoring scheme $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$.

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

Output: SW-alignment scores for a against b.

Description. We assign

$$h[l;0] \leftarrow 0 \qquad h[0;j] \leftarrow 0$$

for all $l \in [0\colon m]$, $j \in [0\colon n]$. We then iterate through the cells of $\mathrm{G}_{a,b}$ for all $\hat{l} \in \langle 0\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$ in lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the cells. In each iteration, we perform a score assignment as follows:

$$h[\hat{l}^+;\hat{\jmath}^+] \leftarrow \max \begin{cases} h[\hat{l}^-;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ h[\hat{l}^-;\hat{\jmath}^+] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^+] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ h[\hat{l}^+;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^+;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ 0 \end{cases} \tag{11.3}$$

Thus, an iteration is similar to that of Algorithm 6.5, except that any negative score is reset to 0 in the final line of (11.3), corresponding to an alignment of a pair of empty substrings. ∎

Theorem 11.22 *The SW-alignment problem can be solved in time $O(mn)$.*□

Proof In Algorithm 11.21, each iteration runs in constant time. The overall running time is $mn \cdot O(1) = O(mn)$. ∎

Bounded-length Smith–Waterman alignment. A commonly observed drawback in biological applications of SW-alignment is that it seeks to maximise the scores across a range of counterpart substring pairs, regardless of their lengths. However, if the substrings are too short, then their alignment may be less significant biologically than a slightly lower-scoring alignment of much longer substrings. To address this issue, various alternative definitions of local string comparison have been proposed, taking substring lengths into account; see [21] for a survey. Algorithms for such length-sensitive alignment problems are typically more computationally expensive than Algorithm 11.21 (SW-alignment); for this reason, approximation versions of such problems have also been considered [21].

The most direct approach to defining length-sensitive local alignment requires that the aligned substrings satisfy a specific lower bound on their lengths, thus filtering out substrings that are too short.

Definition 11.23 *Given strings a, b, a length threshold $w \geq 0$, and assuming a fixed scoring scheme, the* one-sided bounded length SW-alignment *(*B1LSW-alignment*),* two-sided bounded length SW-alignment *(*B2LSW-alignment*), and* bounded total length SW-alignment*) (*BTLSW-alignment*)* problems *are defined as follows. For every prefix $a\langle 0\colon l\rangle$ and every prefix $b\langle 0\colon j\rangle$, each problem asks for the maximum alignment score over all possible choices of a suffix $a\langle k\colon l\rangle$ and a suffix $b\langle i\colon j\rangle$ from these prefixes, where $l-k \geq w$ (respectively, $l-k+j-i \geq w$):*

$$h_{1\mathsf{len}\geq w}[l;j] = \max_{\substack{k\in[0:l],i\in[0:j]\\ l-k\geq w}} \mathsf{score}\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)$$

$$h_{\mathsf{2len}\geq w}[l;j] = \max_{\substack{k\in[0:l],i\in[0:j]\\ l-k\geq w;\, j-i\geq w}} \mathsf{score}\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)$$

$$h_{\mathsf{tlen}\geq w}[l;j] = \max_{\substack{k\in[0:l],i\in[0:j]\\ l-k+j-i\geq w}} \mathsf{score}\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)$$

where $l \in [0\colon m]$, $j \in [0\colon n]$; also, $l \geq w$ for B1LSW-alignment; $l, j \geq w$ for B2LSW-alignment; $l + j \geq w$ for BTLSW-alignment. □

A solution to the B(1,2,T)LSW-alignment problem allows one to obtain, by another layer of maximisation, the highest alignment score across all substring pairs in a, b, satisfying the respective substring length constraint.

The BTLSW-alignment problem was introduced by Arslan and Eğecioğlu [20] (see also [21]) under the name "LAt" (local alignment with length threshold). They describe a dynamic programming algorithm solving the problem in time $O(mn^2)$. They also give an approximation algorithm for a relaxed problem running in time $O(rmn)$, where r is a parameter controlling the accuracy of approximation.

We now show that, assuming a rational scoring scheme, the B(1,2,T)LSW-alignment problem can be solved exactly by an efficient algorithm. In fact, our exact algorithm is asymptotically faster not only than the exact algorithm of [20], but also than their approximation algorithm. We first describe an algorithm for B1LSW-alignment, and then adjust it for B2LSW- and BTLSW-alignment.

Algorithm 11.24 (B1LSW-alignment)

Parameters: a rational scoring scheme $(\mathrm{w}_+, \mathrm{w}_0, \mathrm{w}_-)$, where $\mathrm{w}_- < 0$.

Input: strings a, b of length m, n, respectively; length threshold $w \geq 0$.

Output: B1LSW-alignment scores for a against b.

Description. The algorithm runs in three phases.

1. Solving the window-substring alignment problem for every w-window in a against every substring in b;

2. Solving the complete AM problem for every w-window in a against b. For every w-window $a\langle k\colon l\rangle$ in a and every prefix $b\langle 0\colon j\rangle$ of b, we find the maximum alignment score of $a\langle k\colon l\rangle$ against all possible choices of a suffix $b\langle i\colon j\rangle$ from this prefix.

3. Extending the complete AM scores to B1LSW-alignment scores by a dynamic programming procedure that generalises Algorithm 11.21 (SW-alignment).

We now describe each of these phases in more detail.

Phase 1 (Solving the window-substring alignment problem). We obtain the window-substring alignment kernel $\mathrm{S}^{\nearrow}_{a\langle l-w:l\rangle,b}$ for every $l \in [0\colon m]$ by Theorem 11.8.

Phase 2 (Solving the complete AM problem for every w-window in a against b). For every w-window $a\langle l-w\colon l\rangle$ in a and every prefix $b\langle 0\colon j\rangle$ of b, we find the maximum alignment score of $a\langle l-w\colon l\rangle$ against all possible choices of a suffix $b\langle i\colon j\rangle$ from this prefix. These scores correspond to column maxima of the alignment matrices $\mathrm{H}^{\nearrow}_{a\langle l-w:l\rangle,b}$:

$$h_{\mathsf{1len}=w}[l;j] = \max_{i\in[0:j]} \mathrm{H}^{\nearrow}_{a\langle l-w:l\rangle,b}[i;j]$$

Every such matrix is represented implicitly by the window-substring alignment kernel $\mathrm{S}^{\nearrow}_{a\langle l-w:l\rangle,b}$, obtained in the first phase, hence its column maxima can be obtained by Theorem 6.29.

Phase 3 (Extending the complete AM scores to B1LSW-alignment scores). We assign

$$h_{\mathsf{1len}\geq w}[l;0] \leftarrow 0 \qquad h_{\mathsf{1len}\geq w}[w;j] \leftarrow h_{\mathsf{1len}=w}[w;j]$$

for all $l \in [w\colon m]$, $j \in [0\colon n]$. We then iterate through the cells of $\mathrm{G}_{a,b}$ for all $\hat{l} \in \langle w\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$ in lexicographic order, or in any other total order compatible with the $\ll$-dominance partial order of the cells. In each iteration, we perform a score assignment similar to that of Algorithm 11.21, except that the reset score value now corresponds to an alignment of a w-window in a against a substring of b, obtained in the second phase:

$$h_{\mathsf{1len}\geq w}[\hat{l}^+;\hat{\jmath}^+] \leftarrow \max \begin{cases} h[\hat{l}^-;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ h[\hat{l}^-;\hat{\jmath}^+] + \mathsf{score}\big([\hat{l}^-;\hat{\jmath}^+] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ h[\hat{l}^+;\hat{\jmath}^-] + \mathsf{score}\big([\hat{l}^+;\hat{\jmath}^-] \to [\hat{l}^+;\hat{\jmath}^+]\big) \\ h_{\mathsf{1len}=w}[\hat{l}^+;\hat{\jmath}^+] \end{cases} \tag{11.4}$$

■

Theorem 11.25 *Assuming a rational scoring scheme with denominator $\nu = O(1)$, the B(1,2,T)LSW-alignment problems can each be solved in time $O(mn)$.* □

Proof In Algorithm 11.24, the running time of each of the three phases, and therefore the overall running time, is $O(mn)$.

The B2LSW-alignment problem can be solved by an algorithm similar to Algorithm 11.24, where in Phase 2, instead of the complete AM problem, we solve the window-window alignment problem. We define

$$h_{\mathsf{2len}=w}[l;j] = \mathrm{H}^{\nearrow}_{a\langle l-w:l\rangle,b}[i;i-w]$$

In Phase 3, we assign

$$h_{\mathsf{2len}\geq w}[l;w] \leftarrow h_{\mathsf{2len}=w}[l;w] \qquad h_{\mathsf{2len}\geq w}[w;j] \leftarrow h_{\mathsf{2len}=w}[w;j]$$

for all $l \in [w\colon m]$, $j \in [w\colon n]$. We then iterate through the cells of $\mathrm{G}_{a,b}$ for all $\hat{l} \in \langle w\colon m\rangle$, $\hat{\jmath} \in \langle w\colon n\rangle$, as before; however, we use $h_{\mathsf{2len}\geq w}[\hat{l}^+;\hat{\jmath}^+]$ in the left-hand side of the assignment in (11.4), and $h_{\mathsf{2len}=w}[\hat{l}^+;\hat{\jmath}^+]$ as the reset value in the bottom line of the right-hand side.

The BTLSW-alignment problem can also be solved by an algorithm similar to Algorithm 11.24, where instead of the window-substring alignment problem, in Phase 1 we solve the fixed total length alignment problem, and in Phase 2 we define $h_{\mathsf{tlen}=w}[l;j]$ as column maxima in the resulting implicit alignment matrices. In Phase 3, we assign

$$h_{\mathsf{tlen}\geq w}[l;0] \leftarrow 0 \qquad h_{\mathsf{tlen}\geq w}[0;j] \leftarrow 0$$

for all $l \in [w\colon m]$, $j \in [w\colon n]$, and also

$$h_{\mathsf{tlen}\geq w}[l;j] \leftarrow 0$$

for all $l \in [0\colon w]$, $j \in [0\colon w]$, $l + j = w$. We then iterate through the cells of $\mathrm{G}_{a,b}$ for all $\hat{l} \in \langle 0\colon m\rangle$, $\hat{\jmath} \in \langle 0\colon n\rangle$, $l + j = w$, as before; however, we use $h_{\mathsf{tlen}\geq w}[\hat{l}^+;\hat{\jmath}^+]$ in the left-hand side of the assignment in (11.4), and $h_{\mathsf{tlen}=w}[\hat{l}^+;\hat{\jmath}^+]$ as the reset value in the bottom line of the right-hand side.

For each of the three problems, the running time remains $O(mn)$. ∎

Barsky et al. [25] introduced the problem of finding *all error-bounded approximate matches*: given strings a, b, a length threshold $w \geq 0$, and an edit distance threshold k, the problem asks for all pairs of inclusion-maximal substrings in a, b, respectively, where the length of either substring is at least w, and the Levenshtein distance between the substrings is at most k. They gave an algorithm running in time $O(mnk^3)$. By applying the above solution for the B2LSW-alignment problem with the Levenshtein scoring scheme, and then selecting the required set of substring pairs, we obtain a faster algorithm for the all error-bounded approximate matches problem, running in time $O(mn)$.

Normalised Smith–Waterman alignment. Arslan et al. [22] (see also [21]) introduced another, more sophisticated approach to length-sensitive local alignment, by incorporating the substrings' lengths into the scoring function.

Definition 11.26 *Let a, b be strings. Assuming a fixed scoring scheme, their* normalised alignment score *is defined as their alignment score divided by their total length:*

$$\mathsf{nscore}(a,b) = \frac{\mathsf{score}(a,b)}{m+n}$$ □

Analogously to Definition 11.23 (the BTLSW-alignment problem), the normalised BTLSW-alignment problem was introduced by Arslan and Eğecioğlu [20] (see also [21]) under the name *normalised local alignment with length threshold (NLAt)*. They give an approximation algorithm for a relaxed problem running in time $O(rmn \log n)$, assuming a rational scoring scheme, where r is a parameter controlling the accuracy of approximation. The algorithm uses their approximation algorithm for ordinary (unnormalised) BTLSW-alignment as a subroutine; the only dependence on parameter r is within that subroutine. By replacing this subroutine with the algorithm of Theorem 11.25, we obtain an exact algorithm for normalised BTLSW-alignment running in time $O(mn \log n)$, assuming a rational scoring scheme.

Definition 11.27 *Given strings a, b, a length threshold $w \geq 0$, a score threshold h, and assuming a fixed scoring scheme, the* threshold normalised BTLSW-alignment problem *is defined as follows. For every prefix $a\langle 0\colon l\rangle$ and every prefix $b\langle 0\colon j\rangle$, we consider the maximum normalised alignment score over all possible choices of a suffix $a\langle k\colon l\rangle$ and a suffix $b\langle i\colon j\rangle$ from these prefixes, where $l - k + j - i \geq w$:*

$$\begin{aligned}\hbar_{\mathsf{tlen}\geq w}[l; j] &= \max_{\substack{k\in[0:l], i\in[0:j]\\ l-k+j-i\geq w}} \mathsf{nscore}\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)\\ &= \max_{\substack{k\in[0:l], i\in[0:j]\\ l-k+j-i\geq w}} \frac{\mathsf{score}\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)}{l-k+j-i}\end{aligned}$$

where $l \in [0\colon m]$, $j \in [0\colon n]$, $l + j \geq w$. The problem asks for those indices l, j, for which $\hbar_{\mathsf{tlen}\geq w}[l; j] \geq h$. □

The threshold normalised BTLSW-alignment problem was introduced by Arslan and Eğecioğlu [20] (see also [21]) under the name "Qt". They describe a solution for this problem based on an application of BTLSW-alignment, with the same running time both in the exact and the approximate versions. By plugging in the algorithm of Theorem 11.25, we obtain an efficient exact algorithm for the threshold normalised BTLSW-alignment problem.

Theorem 11.28 *Assuming a rational scoring scheme with denominator $\nu = O(1)$, the threshold normalised BTLSW-alignment problem can be solved in time $O(mn)$.* □

Proof Let u, v be strings of length s, t respectively. We have $\mathsf{nscore}(u, v) = \frac{\mathsf{score}(u,v)}{s+t} \geq h$, iff $\mathsf{score}(u, v) - h \cdot (s+t) \geq 0$. Consider the standard BTLSW-alignment problem with the modified score function

$$\begin{aligned}&\mathsf{score}'(u, v) = \mathsf{score}(u, v) - h \cdot (s + t) =\\ &(\mathrm{w}_+ - 2\mathrm{w}_-) \cdot k_+ + (\mathrm{w}_0 - 2\mathrm{w}_-) \cdot k_0 + \mathrm{w}_- \cdot (s + t) - h \cdot (s + t) =\end{aligned}$$

$$(\mathrm{w}'_+ - 2\mathrm{w}'_-) \cdot k_+ + (\mathrm{w}'_0 - 2\mathrm{w}'_-) \cdot k_0 + \mathrm{w}'_- \cdot (s + t)$$

where k_+, k_0 is the number of matching (respectively, mismatching) character pairs in the highest-scoring alignment of u, v, and

$$\mathrm{w}'_+ = \mathrm{w}_+ - 2h \qquad \mathrm{w}'_0 = \mathrm{w}_0 - 2h \qquad \mathrm{w}'_- = \mathrm{w}_- - h$$

We solve this BTLSW-alignment problem in time $O(mn)$, using the algorithm of Theorem 11.25. Denote the solution by

$$h'_{\mathsf{tlen}\geq w}[l; j] = \max_{k\in[0:l],i\in[0:j]} \mathsf{score}'\big(a\langle k\colon l\rangle, b\langle i\colon j\rangle\big)$$

where $l \in [0\colon m]$, $j \in [0\colon n]$. We now have the solution for the original threshold normalised BTLSW-alignment problem: for all l, j, we have $\hbar_{\mathsf{tlen}\geq w}[l; j] \geq h$, if and only if $h'_{\mathsf{tlen}\geq w}[l; j] \geq 0$. ∎

Chapter 12

String comparison by transposition networks

12.1 Transposition networks (TN)

Comparison networks were first considered as a computation model by Batcher [26] (see also [62, 9]).

Definition 12.1 *A* circuit *represents a computation as a dag (directed acyclic graph). The internal nodes of a circuit are labeled by elementary operations on values, which are passed along the edges; source and sink nodes represent the inputs and outputs, respectively. A* comparator node *(or simply* comparator*) is a node of indegree and outdegree 2, which sorts its two operands in increasing order. In other words, a comparator node compares the operands on the incoming edges, and returns each of the minimum and the maximum operand on a prescribed outgoing edge. A* comparison network *is a circuit where all internal nodes are comparator nodes.* □

The most well-studied types of comparison networks are *sorting networks*, that sort every possible permutation of inputs, and *merging network*, that merge two disjoint pre-sorted subsets of inputs. In particular, Batcher [26] gave classical merging networks with $O(n \log n)$ comparators, and sorting networks with $O(n \log^2 n)$ comparators. Ajtai et al. [7, 8] gave an asymptotically optimal sorting network with $O(n \log n)$ comparators; their construction was subsequently simplified by Paterson [170] and by Seiferas [191].

Comparison networks are usually visualised by *wire diagrams* (also known as *Knuth diagrams*), where the values propagate across the network along a set of parallel *wires*. Every comparator is represented by a directed line segment, drawn orthogonally between two (not necessarily adjacent) wires. The order in which a comparator returns the minimum and the maximum output is consistent across all the comparators in the network. The most

common convention on wire diagrams (adopted e.g. by Knuth [135]) is to draw the wires horizontally, directed from left to right; sometimes (e.g. in [170]), they are drawn vertically, directed from top to bottom. In our setting, it will be convenient to draw the wires diagonally, directed from top-left to bottom-right. Comparator segments will be directed so that the minimum output is returned on the bottom-left, and the maximum on the top-right.

We will be dealing exclusively with the following restricted type of comparison network.

Definition 12.2 *A comparison network is called a* transposition network*, if it can be represented by a wire diagram, where all the comparisons are between adjacent wires.* □

12.2 Semi-local LCS with TN

From **??** and Chapter 5, we already know that the structure of semi-local string comparison is isomorphic to the unit-Monge and the sticky braid monoids. In this chapter, we show that this structure can be expressed in yet another alternative form, based on the transposition networks.

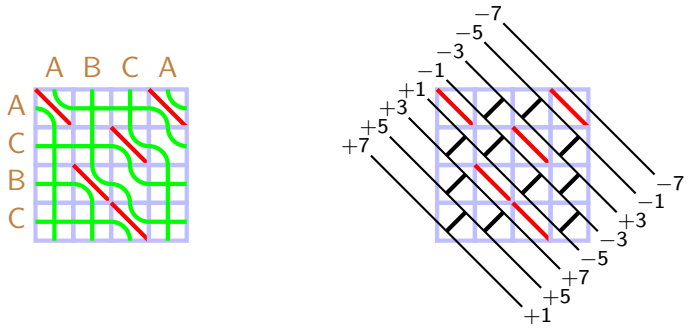
Every triangulated grid (as in Definition 5.3) with zero-one edge scores can be associated with a transposition network as follows.

Definition 12.3 *Let G be a triangulated grid with horizontal and vertical edges having score* 0*, and diagonal edges having score either* 0 *or* 1*. Its corresponding transposition network* $\mathsf{N}(G)$ *has a diagram of* $m+n$ *parallel wires, laid over grid G diagonally, so that every horizontal and every vertical edge in G is crossed by exactly one wire. Hence, every cell is crossed diagonally by exactly two adjacent wires: one from the left edge to the bottom edge, the other from the top edge to the right edge. The cell contains a comparator joining these two wires, if and only if the cell's diagonal edge has score* 0*.* □

Example 12.4 Figure 12.1 illustrates Definition 12.3 for strings $a =$ "ACBC", $b =$ "ABCA". Subfigure Figure 12.1a shows the LCS grid $\mathsf{G}_{a,b}$ and the LCS kernel $\mathsf{G}_{a,b}$, represented by an embedded reduced sticky braid. Subfigure Figure 12.1b shows in black the transposition network $\mathsf{N}(\mathsf{G}_{a,b})$. By Definition 12.3, a cell contains a comparator, if and only if it does not contain a red diagonal edge. ■

Let us denote the input and output arrays of a transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ by $x\langle -m\!:n\rangle$ and $y\langle 0\!:m+n\rangle$, respectively. Assuming all input values of the network are distinct, each value traces a well-defined path through the network. We write $\pi\langle\hat{\imath}\rangle = \hat{\jmath}$, if the input $x\langle\hat{\imath}\rangle$ ends up as the output $y\langle\hat{\jmath}\rangle$.

Although the transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ is in general neither a merging nor a sorting network, it is still useful to consider its operation on a



(a) LCS grid $\mathsf{G}_{a,b}$ with an embedded sticky braid

(b) Corresponding transposition network

Figure 12.1: Semi-local LCS by sticky braid combing with a transposition network

certain type of input. Let the input array x be anti-sorted (i.e. sorted in decreasing order). It turns out that, given such an input, the operation of the network $\mathsf{N}(\mathsf{G}_{a,b})$ emulates sticky braid combing, and the resulting bijection π coincides with the bijection defined by the semi-local LCS kernel $\mathsf{P}_{a,b}$.

Theorem 12.5 *Consider an LCS grid $\mathsf{G}_{a,b}$ and the corresponding transposition network $\mathsf{N}(\mathsf{G}_{a,b})$. For all $\hat{\imath} \in \langle -m : n \rangle$, $\hat{\jmath} \in \langle 0 : m+n \rangle$, we have $\pi\langle\hat{\imath}\rangle = \hat{\jmath}$, if and only if $\mathsf{P}_{a,b}\langle\hat{\imath}, \hat{\jmath}\rangle = 1$.* □

Proof Consider any pair of input values in $\mathsf{N}(\mathsf{G}_{a,b})$. Initially, these two values are anti-sorted. Assume that the two values in question meet at some comparator. If by that point their paths have not yet crossed, then they arrive at the comparator anti-sorted, and leave it sorted, so the two paths cross for the first time. Otherwise, the two paths have previously crossed once, therefore the values arrive and leave the comparator sorted, and their paths never cross again.

Since the comparators are located in mismatch cells of $\mathsf{G}_{a,b}$, the described operation of each comparator is equivalent to sticky braid combing. Therefore, the paths of the values correspond to the strands in the resulting reduced sticky braid, and the output of $\mathsf{N}(\mathsf{G}_{a,b})$ is described in terms of the LCS kernel $\mathsf{P}_{a,b}$ as claimed. ■

Example 12.6 Figure 12.1 illustrates the connection between sticky braids and transposition networks, as described by Theorem 12.5. Subfigure 12.1a shows the reduced sticky braid obtained by Algorithm 5.36 (semi-local LCS by iterative combing), embedded in the LCS grid $\mathsf{G}_{a,b}$. Subfigure 12.1b shows an anti-sorted input array of distinct values, and the corresponding output array. Each value in Subfigure 12.1b traces a path through the network; the paths are not shown explicitly, but can be reconstructed by executing the

network "by hand". Every path turns out to be identical to the layout of the corresponding strand in Subfigure 12.1a. ■

Extending Theorem 12.5, it is natural to consider the situation where the input values to a transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ are anti-sorted, but not necessarily distinct. An extreme case of that is an anti-sorted *binary input*: a sequence of ones, followed by a sequence of zeros. While in this case, the information on semi-local LCS scores is lost, it turns out that the output still contains sufficient information to obtain the ordinary (global) LCS score.

Theorem 12.7 *Let the transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ operate on an anti-sorted input array $x\langle -m\colon n\rangle$, consisting of m 1-values and n 0-values:*

$$x\langle \hat{\imath}\rangle = \begin{cases} 1 & \text{if } \hat{\imath} \in \langle -m\colon 0\rangle \\ 0 & \text{if } \hat{\imath} \in \langle 0\colon n\rangle \end{cases}$$

Let $y\langle 0\colon m+n\rangle$ be the output array of $\mathsf{N}(\mathsf{G}_{a,b})$. Then we have

$$lcs(a,b) = \textstyle\sum_{\hat{\jmath}\in\langle 0:n\rangle} y\langle\hat{\jmath}\rangle = m - \sum_{\hat{\jmath}\in\langle n:m+n\rangle} y\langle\hat{\jmath}\rangle \qquad \square$$

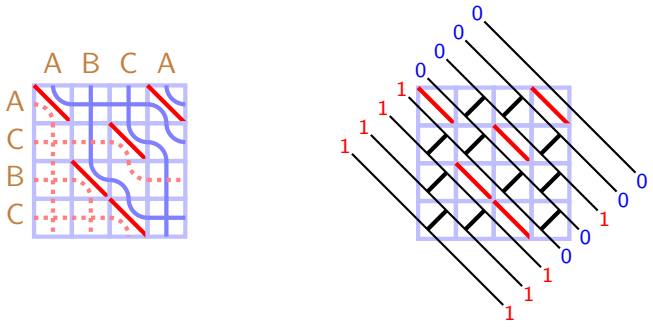
Proof We have

$$\begin{aligned}
&lcs(a,b) = n - \mathsf{P}^{\Sigma}_{a,b}[0,n] = && \text{(Theorem 5.17; definition of } \Sigma)\\
&n - \textstyle\sum_{\langle\hat{\imath},\hat{\jmath}\rangle\in\langle 0:n|0:n\rangle} \mathsf{P}_{a,b}\langle\hat{\imath},\hat{\jmath}\rangle = && (\mathsf{P}_{a,b} \text{ is a permutation matrix})\\
&n - \big(n - \textstyle\sum_{\langle\hat{\imath},\hat{\jmath}\rangle\in\langle -m:0|0:n\rangle} \mathsf{P}_{a,b}\langle\hat{\imath},\hat{\jmath}\rangle\big) = && \text{(cancellation)}\\
&\textstyle\sum_{\langle\hat{\imath},\hat{\jmath}\rangle\in\langle -m:0|0:n\rangle} \mathsf{P}_{a,b}\langle\hat{\imath},\hat{\jmath}\rangle = && (\mathsf{P}_{a,b} \text{ is a permutation matrix})\\
&\big|\big\{\langle\hat{\imath},\hat{\jmath}\rangle \in \langle -m\colon 0 \mid 0\colon n\rangle\colon \mathsf{P}_{a,b}\langle\hat{\imath},\hat{\jmath}\rangle = 1\big\}\big| = && \text{(Theorem 12.5)}\\
&\big|\big\{\langle\hat{\imath},\hat{\jmath}\rangle \in \langle -m\colon 0 \mid 0\colon n\rangle\colon \pi\langle\hat{\imath}\rangle = \hat{\jmath}\big\}\big| = && \text{(definition of } x)\\
&\big|\big\{\hat{\jmath} \in \langle 0\colon n\rangle\colon x\langle\pi^{-1}\langle\hat{\jmath}\rangle\rangle = 1\big\}\big| = && \text{(definition of } \pi)\\
&\big|\big\{\hat{\jmath} \in \langle 0\colon n\rangle\colon y\langle\hat{\jmath}\rangle = 1\big\}\big| = && (y \text{ binary})\\
&\textstyle\sum_{\hat{\jmath}\in\langle 0:n\rangle} y\langle\hat{\jmath}\rangle = && (\textstyle\sum_{\hat{\jmath}\in\langle 0:m+n\rangle} y(\hat{\jmath}) = \sum_{\hat{\imath}\in\langle -m:n\rangle} x(\hat{\imath}) = m)\\
&m - \textstyle\sum_{\hat{\jmath}\in\langle n:m+n\rangle} y\langle\hat{\jmath}\rangle && \blacksquare
\end{aligned}$$

Example 12.8 Figure 12.2 illustrates Theorem 12.7 on the same pair of strings as Figure 12.1. Subfigure 12.2a shows the sticky braid of Subfigure 12.1a, embedded in the LCS grid $\mathsf{G}_{a,b}$. The strands originating at the top (respectively, the left-hand side) of the grid are shown by solid (respectively, dotted) lines. Subfigure Figure 12.2b shows the transposition network of Subfigure 12.1b, embedded in the LCS grid $\mathsf{G}_{a,b}$. The network is given an anti-sorted binary input. The path of each 0-value (respectively, 1-value) corresponds to a solid blue (respectively, dotted red) strand. We have

$$lcs(a,b) = \textstyle\sum_{\hat{\jmath}\in\langle 0:4\rangle} y\langle\hat{\jmath}\rangle = 3$$
$$lcs(a,b) = 4 - \textstyle\sum_{\hat{\jmath}\in\langle 4:8\rangle} y\langle\hat{\jmath}\rangle = 4 - 1 = 3$$

as claimed by Theorem 12.7. ■



(a) LCS grid with a partitioned sticky braid

(b) Corresponding transposition network with binary input

Figure 12.2: LCS by binary iterative combing with a transposition network

12.3 (Dis)similarity-sensitive LCS

An algorithm's complexity is most commonly defined to be a function of a single argument: the input size. However, in the pursuit of efficiency, algorithms may also be designed to be sensitive to various other parameters of the input. In the context of string comparison, the two most relevant parameters are:

- the input strings' alignment score; we consider primarily the LCS score $\lambda = lcs(a, b)$;
- the input strings' edit distance; we consider primarily the LCS distance $\kappa = dist_{LCS}(a, b) = m + n - 2\lambda$.

In this section, we study algorithms for the LCS problem that are sensitive to these parameters. We consider *dissimilarity-sensitive* and *similarity-sensitive* LCS algorithms. The running times of such algorithms are parameterised respectively by λ and κ, so that advantage can be taken of the low value of a parameter. More generally, one can also use weighted alignment scores or weighted edit distances (e.g. the Levenshtein distance) as parameters.

As we aim for algorithms that, for a low value of the parameter, run substantially faster than $\Theta(mn)$, we cannot afford to perform all the mn pairwise comparisons of characters from each string. Assuming an ordered alphabet, the "missing" comparisons can be obtained by transitivity, and algorithms with running time $o(mn)$ become possible.

For simplicity, we ignore the trivial cases $\lambda = 0$ (the two input strings have no characters in common) and $\kappa = 0$ (the two input strings are identical). As usual, we denote the alphabet size by σ. Without loss of generality, we assume $m \leq n$.

Dissimilarity-sensitive comparison. In this type of comparison, the overall number of matching character pairs will be low. To locate these matching pairs effectively, the input strings a and b are preprocessed into a data structure that allows efficient queries defined by the *string identification problem* [5, 113, 180]. The preprocessing builds a binary search tree on each input string, and returns a partitioning of both a and b into character equality classes. This preprocesing procedure runs in time $m \log \sigma$ (respectively, $n \log \sigma$).

After the preprocessing, the LCS problem can be solved by one of the algorithms due to Hirschberg [112], Hsu and Du [113] (see also Apostolico [16]), Apostolico and Guerra [18]. All these algorithms run in time $O(n\lambda)$. Apostolico, Browne and Guerra [17] proposed another algorithm that requires no preprocessing, and runs in time $O(n\lambda \log \sigma)$ and linear space.

Similarity-sensitive comparison. In this type of comparison, the overall number of matching character pairs may be as high as mn. However, a similarity-sensitive LCS algorithm may not need to look at all these matches; speaking informally, a good algorithm "will know where to look for relevant matches". In fact, it is sufficient to consider $O(n \cdot \kappa)$ character matches overall. In contrast to dissimilarity-sensitive comparison, there is no need for preprocessing the input strings.

Efficient similarity-sensitive LCS algorithms have been given by Ukkonen [211], Myers [163], Wu et al. [219]. All these algorithms run in time $O(n \cdot \kappa)$. Apostolico, Browne and Guerra [17] proposed another algorithm that runs in time $O(n \cdot \kappa)$ and linear space.

Flexible comparison. This type of comparison is sensitive to both dissimilarity and similarity. Flexible comparison can be achieved by preprocessing the input string as for dissimilarity-sensitive comparison, and then running both comparison types alongside each other. However, dedicated flexible comparison algorithms have also been proposed. In particular, the flexible LCS algorithm by Hirschberg [112] runs in time $O(\lambda\kappa \log n)$, and the one by Rick [180] in time $O(\lambda\kappa)$ and linear space.

We now describe a flexible LCS algorithm based on the comparison network method, matching the above algorithms in running time. We call it the *waterfall algorithm*.

Algorithm 12.9 (Flexible LCS: The waterfall algorithm)

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

Output: the LCS score $lcs(a, b)$.

Description. We preprocess input strings to build a data structure for efficient querying of the matches, and also *match successor queries*: given $\hat{l}$, $\hat{\imath}$, find lowest $\hat{\jmath} \geq \hat{\imath}$ such that $a(\hat{l}) = b(\hat{\jmath})$.

The algorithm is based on transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ with binary input, as described by Theorem 12.7. Such a network can be evaluated efficiently as follows. The grid $\mathsf{G}_{a,b}$ is processed row-by-row. Instead of performing binary value comparisons within individual mismatch cells of a given row of cells, we partition the row into contiguous horizontal blocks, and combine the comparisons within each block into a single constant-time operation. A block may contain both match and mismatch cells; as we move vertically from one grid row to the next, the blocks may split or merge. We keep track of each block's endpoints in a parameter-sensitive way, achieving an overall speedup whenever either of the parameters λ, κ is low.

Let us now fill in the details. Consider a row of nodes in the LCS grid $\mathsf{G}_{a,b}$, corresponding to a fixed index $l \in [0\colon m]$. The nodes in this row are connected by n horizontal edges, each of which is crossed by a single wire of the transposition network $\mathsf{N}(\mathsf{G}_{a,b})$. Hence, when running the network on a binary input as in Theorem 12.7, each row of nodes corresponds to a sequence of n binary values. We regard this sequence as partitioned into contiguous runs of 0-values, called *blocks*, alternating with contiguous runs of 1-values, called *gaps*. We only need to deal explicitly with the blocks, leaving the processing of gaps implicit (hence the block/gap terminology). If the input strings happen to be highly dissimilar or highly similar, i.e. the respective parameter λ or κ is low, then every grid row will have only a small number of blocks (and gaps).

Consider the operation of the transposition network row by row, moving downwards across the LCS grid. In the top row of nodes $l = 0$, we have a single block of 0-values coming as input to the network, with index set spanning the full row $\langle 0\colon n\rangle$.

Now, given a fixed row of cells $\hat{l} \in \langle 0\colon m\rangle$, consider a transition from its top boundary $\hat{l}^-$ to its bottom boundary $\hat{l}^+$. Given the endpoints of all blocks in row of nodes $\hat{l}^-$, we need to find the endpoints for the blocks in row of nodes $\hat{l}^+$. This transformation of endpoints can be achieved in two phases: *block splitting* and *block merging*.

Block splitting. Consider a block in row of nodes $\hat{l}^-$, spanning the index set $\langle i_0\colon i_1\rangle$. Let $\hat{\imath} \in \langle i_0\colon i_1\rangle$ be the index of the leftmost match cell (if one exists) immediately below the given block: $\hat{\imath} = \min\{\hat{k} \in \langle i_0\colon i_1\rangle \mid a(\hat{l}) = b(\hat{k})\}$. If index $\hat{\imath}$ exists, then block $\langle i_0\colon i_1\rangle$ is split at this index into a pair of subblocks:

- the left subblock $\langle i_0\colon \hat{\imath}^-\rangle$, which is empty if $i = i_0^+$, and consists of the whole block $\langle i_0\colon i_1\rangle$ if index $\hat{\imath}$ does not exist; this subblock is kept unchanged in row of nodes $\hat{l}^+$;
- the right subblock $\langle \hat{\imath}^-\colon i_1\rangle$; this subblock is shifted by one unit to the right, forming a new block $\langle \hat{\imath}^+\colon i_1{+}1\rangle$ in row of nodes $\hat{l}^+$ (unless $i_1 = n$, in which case the rightmost 0-value is considered to be "shifted out" off the right boundary of the grid, and the resulting new block is $\langle \hat{\imath}^+\colon i_1\rangle$).

A unit-length gap $\langle \hat{\imath}^- : \hat{\imath}^+ \rangle$ is created between the two subblocks in row $\hat{l}^+$. The gap at the right of the right subblock (which exists in row $\hat{l}^-$, unless $i_1 = n$) is reduced by one unit in row $\hat{l}^+$.

Block merging. This occurs whenever a gap between two blocks has been closed (reduced to length 0) as a result of shifting a subblock in the previous splitting phase. Suppose that subblock shifting has resulted in a pair of touching blocks $\langle i_0 : i_1 \rangle$ and $\langle i_1 : i_2 \rangle$ in row $\hat{l}^+$. Then, we merge them into a single block $\langle i_0 : i_2 \rangle$. Note that the left touching block $\langle i_0 : i_1 \rangle$ must have been split off the right-hand side of a larger block in the preceding splitting phase. Therefore, blocks can only merge in pairs: it is impossible for three or more blocks to merge together in the same merging phase. Upon the completion of all block merging, we have the correct block endpoints in row of nodes $\hat{l}^+$.

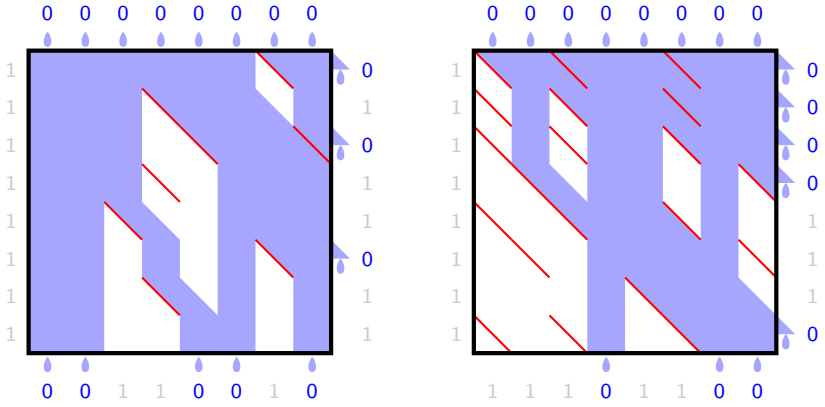
After all the m rows of cells in the LCS grid have been processed, the values $y\langle \hat{\jmath} \rangle$, $\hat{\jmath} \in \langle 0 : m + n \rangle$ are obtained as follows:

- for $\hat{\jmath} \in \langle 0 : n \rangle$, the values $y\langle \hat{\jmath} \rangle$ correspond to the sequence of blocks and gaps resulting from the final iteration of the split/merge procedure;
- for $\hat{\jmath} \in \langle n : m + n \rangle$, the values $y\langle \hat{\jmath} \rangle$ correspond, in reverse order, to the sequence of 0-values (and, implicitly, 1-values) "shifted out" off the right boundary of the grid in each of the m iterations of the split/merge procedure.

The LCS score of the input strings a, b can now be obtained by Theorem 12.7.

We now need to show that the described algorithm emulates correctly the operation of the transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ on binary input, as defined by Theorem 12.7. Consider the operation of the algorithm in row of cells $\hat{l}$. The operation within an individual cell depends on whether the top edge of the cell belongs to a left subblock (which can be the whole block in case it does not get split), a right subblock, or a gap. We also need to consider separately the leftmost cell under a right subblock (the "subblock splitting cell"), and the leftmost cell under a gap following a right subblock (the "gap filling cell"). We thus have the following cases for a cell's operation (where L = "left", R = "right", T = "top", B = "bottom").

Cell's top edge	Input		Output		Match?
	L	T	B	R	
In L subblock	1	0	0	1	mismatch
In R subblock, leftmost	1	0	1	0	match
In R subblock, other	0	0	0	0	either
In gap after R subblock, leftmost	0	1	0	1	either
In gap, other	1	1	1	1	either



(a) Low-similarity (b) High-similarity

Figure 12.3: The waterfall algorithm

It is now straightforward to check that the input-output relationship defined by the above table is consistent with the operation of all the individual comparators in the network $\mathsf{N}(\mathsf{G}_{a,b})$, and that the algorithm's overall running time is $O(\lambda \cdot \kappa)$. ■

The name "waterfall algorithm" for Algorithm 12.9 is justified by the following interpretation. Let us think of the 0-values as unit-width jets of non-compressible water, falling through the LCS grid under gravity. The blocks of adjacent 0-values correspond to wider jets of different widths. These jets may split or merge while flowing through the grid. Initially, there is just a single wide jet of width n, falling vertically down through the top boundary of the grid. The diagonal match edges are sloping roofs that form barriers for the water: whenever a jet hits a roof in its path, it is displaced by one unit to the right, following the roof's slope. This displacement may result in a wide jet splitting at the tip of the roof. A jet displaced by a roof may also fill a unit gap to its right, merging with the jet falling vertically just beyond that gap. The amount of water that emerges from the grid at its right-hand side (respectively, at its bottom) equals the LCS score λ (respectively, the LCS distance κ) of the input strings.

Example 12.10 Figure 12.3 illustrates two separate executions of Algorithm 12.9 (the waterfall algorithm):

- Figure 12.3a shows the low-similarity case, with (implicit) input strings $a =$ "ABCBDABE", $b =$ "FFDBCFAC";
- Figure 12.3b shows the high-similarity case, with (implicit) input strings $a =$ "AAABABCA", $b =$ "ABADCADB".

Following our usual convention, the diagonal edges in match cells are shown in red. The rest of the LCS grid, as well as the input strings, are kept

implicit. The anti-sorted input to the transposition network is represented by the sequences of red 1-values and blue 0-values along the left (respectively, the top) boundary of the LCS grid.

The iterative procedure of block splitting and merging is shown by filling in the interior of the LCS grid with blue and white pattern as follows. Every horizontal edge in the LCS grid corresponds to a blue (respectively, white) streak in the pattern, whenever that edge is crossed by an (implicit) wire carrying a 0- (respectively, 1-) value. Therefore, for each row index l in the grid, the corresponding sequence of blocks and gaps is represented by a sequence of continuous blue and white streaks in a horizontal line at level l. The transition of block sequences between every pair of successive rows is shown by connecting the corresponding pairs of blue streaks with a blue strip. For the vertical (respectively, diagonal) transition of a single 0-value, the connecting strip has the shape of a unit square (respectively, unit-width parallelogram).

The output of the transposition network is represented by the mixed sequence of red 1-values and blue 0-values along the bottom and right boundaries of the LCS grid. Each output 0-value is also shown by a blue tab. By Theorem 12.7, we have

$$lcs(a,b) = \sum_{\hat{\jmath}\in\langle 0:8\rangle} y\langle\hat{\jmath}\rangle = 3 = 8 - \sum_{\hat{\jmath}\in\langle 4:8\rangle} y\langle\hat{\jmath}\rangle = 8-5 = 3$$
$$lcs(a,b) = \sum_{\hat{\jmath}\in\langle 0:8\rangle} y\langle\hat{\jmath}\rangle = 5 = 8 - \sum_{\hat{\jmath}\in\langle 4:8\rangle} y\langle\hat{\jmath}\rangle = 8-3 = 5$$

in Figures 12.3a and 12.3b, respectively.

Just as the splitting/merging procedure in Algorithm 12.9 is not symmetric with respect to blocks and gaps, so the pattern in Figure 12.3 is not symmetric either with respect to horizontal and vertical directions, or with respect to 0- and 1-values. For this reason, while the blocks are represented by the usual blue colour, the gaps are left uncoloured (i.e. are represented by the background white). The red colour, which we would normally use to represent 1-values, is not present in the pattern. ∎

Threshold LCS. Parameterised LCS algorithms are closely related to *threshold LCS* algorithms. Here, a threshold value for the parameter λ or κ is given, and the algorithm is required to output the LCS score of the input strings, as long as this value is below the threshold, or to report "excess", if the LCS score (respectively, LCS distance) exceeds the threshold; in the latter case, there is no requirement for the score to be output explicitly. In general, a threshold algorithm can be obtained from a parameterised algorithm by setting an appropriate time-out threshold as a function of the parameter threshold, and reporting "excess", if the algorithm's running time exceeds the time-out threshold. Alternatively, a threshold LCS algorithm can be obtained from the waterfall algorithm (Algorithm 12.9) by reporting

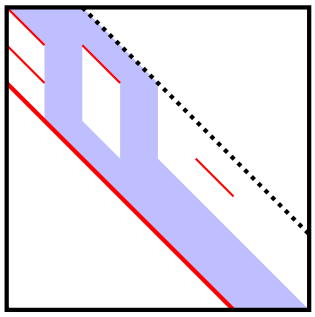


Figure 12.4: High-similarity banded waterfall algorithm

"excess", if the current number of blocks exceeds the threshold value for λ (respectively, κ).

Another efficient approach to threshold similarity-sensitive string comparison is by a technique known as *banded alignment* (see e.g. [121] and references therein). Let κ denote a threshold on $dist_{LCS}(a, b)$, i.e. the LCS distance between the input strings a, b. Assume for simplicity that the value κ is odd, and that $m = n$. Under these assumptions, a highest-scoring path in the LCS grid $\mathsf{G}_{a,b}$ must lie strictly within a symmetric diagonal band of width $\kappa + 1$, unless $dist_{LCS}(a, b)$ exceeds κ. Hence, the waterfall algorithm (Algorithm 12.9) can be modified as follows. First, we ignore all character matches outside the band, as well as on the band's lower-left and upper-right boundaries. We also ignore all the (explicit) input 0-values and all the (implicit) input 1-values outside the band; therefore, we only have $\frac{k+1}{2}$ (explicit) input 0-values at the top of the band, and $\frac{k+1}{2}$ (implicit) input 1-values at the left-hand side of the band. Finally, we create artificial *separator matches* along the bottom-left boundary of the band. The LCS score $lcs(a, b)$ can be obtained by counting the output 0-values at the bottom of the band (or, symmetrically, the output 1-values at the right-hand side of the band). The algorithm reports "exceed", if all the input 0-values are output at the bottom of the band. The algorithm runs in time $O\big(\frac{m\kappa}{v}\big)$.

Figure 12.4 shows a run of the resulting banded high-similarity waterfall algorithm, for $\kappa = 3$. The visual conventions are similar to those in Figures 12.3a and 12.3b. Note that the band is of width $\kappa + 1 = 4$, and that there are $\frac{\kappa+1}{2} = 2$ input 0-values at the band's top. Both these values end up as output 0-values at the band's bottom, hence the algorithm returns "exceed" in the given run.

Threshold multi-string LCS. Still further optimisation is possible in the case of *multi-string* comparison. This type of comparison has been considered e.g. by by Hyyrö et al. [118]. Here, we asked to compute the LCS score for string a against each of the r strings $b_0, \ldots, b_{r-1}$, all of length n. We assume that we are given a single threshold κ on all $dist_{LCS}(a, b_s)$,

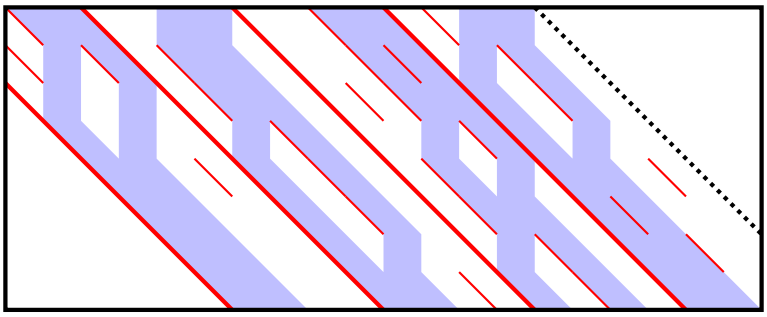


Figure 12.5: High-similarity multi-string banded waterfall algorithm

$0 \leq s < r$. As before, we assume for simplicity that the value κ is odd, and that $m = n$. The problem can be solved by r independent runs of the high-similarity bit-parallel waterfall algorithm. However, it is possible to combine these runs into a single bit-parallel computation, where in each step, we evaluate a single row from every one of the r bands. The bands are packed together in a single super-band of width $r(\kappa + 1)$; individual bands within the super-band are separated by diagonals of separator matches.

Figure 12.5 shows a run of the resulting banded high-similarity multi-string waterfall algorithm, for $\kappa = 3$ and $r = 4$. The leftmost band (band 0) is identical to the band in Figure 12.4. The algorithm returns "exceed" for bands 0, 1, 3. For band 2, we have a single 0-value output at the bottom of the band, hence $lcs(a, b_2) = n - 1$ by Theorem 12.7.

12.4 LCS of random strings (TODO)

Chvatal-Sankoff constants

Random grid-diagonal graph.

Bernoulli model:

Discrete-time TASEP for LCS. Tracy-Widom distribution.

Discrete-time multi-type TASEP for semi-local LCS.

Genuine random strings.

Frog dynamics. Computational experiments.

Chapter 13

Parallel string comparison

13.1 LCS with bit parallelism

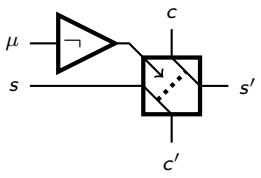
The most efficient practical method for computing the (global) LCS score for a pair of strings is by *bit-parallel* algorithms. These algorithms take advantage of bitwise Boolean operations on bit vectors available in modern processors, often in combination with arithmetic operations on the same vectors as integers. We denote by v the *word* (standard bit vector) length; for instance, $v = 64$ when implementing algorithms on a 64-bit processor architecture.

Early bit-parallel string comparison algorithms were given by Allison and Dix [13] and by Myers [165]. Crochemore at al. [65] proposed an efficient bit-parallel LCS algorithm, running in time $O(\frac{mn}{v})$. For every v cells of the LCS grid $\mathsf{G}_{a,b}$, the algorithm only performs five elementary operations (one arithmetic and four Boolean). Hyyrö [117] improved this to four operations (two arithmetic and two Boolean).

Both algorithms [65, 117] can be viewed as an implementation of a binary transposition network of Definition 12.3 by standard bit-parallel processor instructions. Figure 13.1 shows the main idea of such an implementation for the algorithm of [65]. Consider a cell in a binary transposition network, and let us denote its input bits by s, c, and its output bits by s', c', as shown in Figure 13.1a. Let us denote by μ an extra input bit, which takes value 1 if and only if the current grid cell is a match cell. The operation of a network cell is fully described by the truth table in Figure 13.1b.

Now consider a Boolean circuit shown in Figure 13.1c. The circuit consists of an $\wedge$-gate and a *full adder* element, which adds its three input bits arithmetically, and returns the sum as two separate output bits. Let us again denote the input bits by s, c, μ, and the output bits by s', c'. Then, the circuit computes a Boolean-arithmetic expression

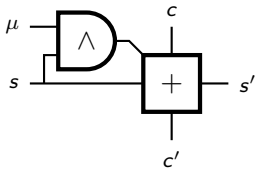
$$2c' + s' \leftarrow s + (s \wedge \mu) + c$$



(a) Binary transposition network cell

s	0	1	0	1	0	1	0	1
c	0	0	1	1	0	0	1	1
μ	0	0	0	0	1	1	1	1
s'	0	1	1	**1**	0	0	1	1
c'	0	0	0	1	0	1	0	1

(b) Corresponding truth table



(c) Boolean circuit for $2c' + s' \leftarrow s + (s \wedge \mu) + c$

s	0	1	0	1	0	1	0	1
c	0	0	1	1	0	0	1	1
μ	0	0	0	0	1	1	1	1
s'	0	1	1	**0**	0	0	1	1
c'	0	0	0	1	0	1	0	1

(d) Corresponding truth table

Figure 13.1: Binary transposition network via a Boolean circuit

The operation of such a circuit is fully described by the truth table in Figure 13.1d.

Note that the truth tables in Figures 13.1b and 13.1d differ in just the one highlighted bit. This difference can be corrected by two extra Boolean operations, resulting in a Boolean-arithmetic expression that fully implements the operation of a single transposition network cell:

$$2c' + s' \leftarrow \big(s + (s \wedge \mu) + c\big) \vee (s \wedge \neg\mu) \tag{13.1}$$

Now consider a row of n cells in the LCS grid $\mathsf{G}_{a,b}$, assuming for the moment $n \leq v$. Let S denote a word variable that will hold the input s, and then the output s', for each cell, least significant bit first. Likewise, let M denote a word constant that holds the match parameter μ for each cell, least significant bit first. (The input c and output c' will not be represented explicitly, but will instead correspond to a propagating carry bit in word integer addition, from the least significant bit all the way to the most significant bit.) The operation of the transposition network $\mathsf{N}(\mathsf{G}_{a,b})$ in the given row of cells corresponds to evaluating an expression

$$S \leftarrow \big(S + (S \wedge M)\big) \vee (S \wedge \neg M) \tag{13.2}$$

which is obtained from (13.1) by identifying the output c' of each cell with the input c of the next cell in the row. Here, the Boolean operations are bitwise, and the addition is on integers represented by the words. Note that for an exact correspondence with (13.2), the roles of 0-values and 1-values in the waterfall algorithm (Algorithm 12.9) must be exchanged (or, alternatively, the cells must be composed into words by columns, rather than by rows).

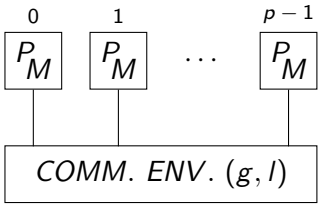


Figure 13.2: The BSP computer

If $n > v$, then each row of the LCS grid is partitioned into $\lceil \frac{n}{v} \rceil$ words of length v. In this case, expression (13.2) needs to be modified to allow carry propagation from each word to the next word in its row.

Bit parallelism can be combined efficiently with banded algorithms for threshold LCS, described in Section 12.3. This is particularly easy when the bandwidth $m + 1 \leq v - 1$. In such a case, every row of the band fits into a single word. The bit-parallel five- (respectively, four-) instruction sequence of either of [65, 117] can be used; the only modification required is an extra shift instruction in each row, to account for the band right-shifting by 1 when moving to the next row.

Finally, we note that all the described bit-parallel LCS algorithms can be generalised to alignment under a rational scoring scheme, using the approach described in Chapter 6. For example, we can obtain bit-parallel algorithms for Levenshtein-scored alignment (or, symmetrically, Levenshtein edit distance) by blowing up the LCS grid by a factor of 2 in each dimension.

13.2 LCS with subword parallelism (TODO)

Subword-parallel, including MP-RAM.

13.3 Bulk-synchronous parallelism (BSP)

The BSP model The model of *bulk-synchronous parallel (BSP) computation* was introduced by Valiant [212] as a "bridging model" for general-purpose parallel computing. The BSP model can be regarded as an abstraction of both parallel hardware and software, and supports an approach to parallel computation that is both architecture-independent and scalable. The main principles of BSP are the treatment of a communication medium as an abstract fully connected network, and the decoupling of all interaction between processors into point-to-point asynchronous data communication and barrier synchronisation. Such a decoupling allows an explicit and independent cost analysis of local computation, communication and synchronisation, all of which are viewed as limited resources. A *BSP computer* (Figure 13.2) contains

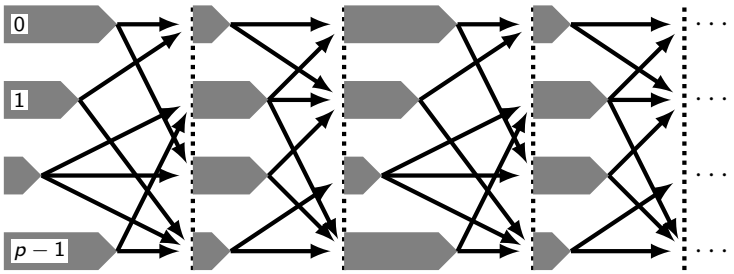


Figure 13.3: BSP computation

- p *processors*; each processor has a local memory and is capable of performing an elementary operation or a local memory access every time unit;

- a *communication environment*, capable of accepting a word of data from every processor, and delivering a word of data to every processor, every g time units;

- a *barrier synchronisation* mechanism, capable of synchronising all the processors simultaneously every l time units.

The processors may follow different threads of computation, and have no means of synchronising with one another apart from the global barriers.

A BSP computation is a sequence of *supersteps* (see Figure 13.3). The processors are synchronised between supersteps; the computation within a superstep is completely asynchronous. Consider a superstep s in which every processor performs a maximum of $comp_s$ local operations, sends a maximum of $comm_s^{out}$ data values, and receives a maximum of $comm_s^{in}$ data values. The value $comp$ is the *local computation cost*, and $comm_s = comm_s^{out} + comm_s^{in}$ is the *communication cost* of the superstep. The total superstep cost is defined as $comp_s + comm_s \cdot g + l$, where the *communication gap* g and the *latency* l are parameters of the communication environment. For a computation comprising *sync* supersteps with local computation costs $comp_s$ and communication costs $comm_s$, $1 \leq s \leq S$, the total cost is $comp + comm \cdot g + sync \cdot l$, where

- $comp = \sum_{1 \leq s \leq S} comp_s$ is the total *local computation cost*;

- $comm = \sum_{1 \leq s \leq S} comm_s$ is the total *communication cost*;

- *sync* is the *synchronisation cost*.

The values of *comp*, *comm* and *sync* typically depend on the number of processors p and on the problem size.

BSP algorithms As a simple parallel computation model, BSP lends itself to the design of efficient, well-structured algorithms. The aim of BSP algorithm design is to minimise the resource consumption of the BSP computer: the local computation cost, the communication cost, the synchronisation cost, and also sometimes the memory cost. Since the aim of parallel computation is to obtain speedup over sequential execution, it is natural to require that a BSP algorithm should be *work-optimal* relative to a particular "reasonable" sequential algorithm, i.e. the local computation cost *comp* should be proportional to the running time of the sequential algorithm, divided by p. It is also natural to allow a reasonable amount of *slackness*: an algorithm only needs to be efficient for $n \gg p$, where n is the problem size. The asymptotic dependence of the minimum value of n on the number of processors p must be clearly specified by the algorithm.

Many important problems have work-optimal BSP algorithms that run in the number of supersteps *sync* that is independent of the problem size n (but may depend on the number of processors p). We shall call such algorithms *coarse-grained*. For instance, the problem of matrix multiplication has a BSP algorithm running in $comp = O\big(\frac{n^3}{p}\big)$, $comm = O\big(\frac{n^2}{p^{2/3}}\big)$, $sync = O(1)$ [159], and the problem of matrix LU decomposition without pivoting has a BSP algorithm running in $comp = O\big(\frac{n^3}{p}\big)$, $comm = O\big(\frac{n^2}{p^\alpha}\big)$, $sync = O(p^\alpha)$ for all α, $\frac{1}{2} \leq \alpha \leq \frac{2}{3}$ [202]. These algorithms are work-optimal with respect to the standard $O(n^3)$ sequential matrix multiplication and Gaussian elimination, respectively; they also generalise to subcubic Strassen-like multiplication schemes [**?**].

Work-optimal algorithms with *sync* not depending on n, and depending on p only moderately (e.g. sublinearly as in the examples above) are of particular interest; even more so, if the dependence of *sync* on p is polylogarithmic or even constant. We shall call a coarse-grained BSP algorithm *quasi-flat*, if $sync = O\big((\log p)^{O(1)}\big)$, and *flat*, if $sync = O(1)$. In the examples above, the BSP algorithm for matrix multiplication is flat, while the BSP algorithm for matrix LU decomposition is neither flat not quasi-flat. Somewhat surprisingly, the sorting problem has a flat work-optimal BSP algorithm [193, **?**], and the selection problem has a quasi-flat work-optimal BSP algorithm that is nearly flat [206]. The all-pairs shortest paths problem has a coarse-grained work-optimal BSP algorithm based on min-plus Gaussian elimination, running at the same asymptotic costs as matrix LU decomposition without pivoting; although this algorithm is not quasi-flat, the problem turns out to also have a better, quasi-flat algorithm [199].

For all BSP algorithms considered in this chapter it will be assumed that the input and the output are stored in an unlimited external memory, included in the communication environment; reading from and writing to this external memory by a processor is charged as (one-sided) communication. An algorithm running in communication $O\big(\frac{input}{p^c}\big)$, where $0 < c \leq 1$, and

input is the input size, will be called *communication-scalable*.

It will be assumed that all problem instances have sufficient slackness, which will be specified for every algorithm. In the description of BSP algorithms, we will ignore various insignificant irregularities arising from imperfect matching between integer parameters. This will allow us to avoid overloading our notation with floor and ceiling functions, assumptions on divisibility etc., which have to be made implicitly wherever necessary.

BSP divide-and-conquer A common approach to algorithm design, especially useful for obtaining quasi-flat BSP algorithms, is recursive partitioning of a problem into independent subproblems in a divide-and-conquer pattern. At a certain level in the recursion tree, p independent subproblems are generated; we shall call it the *ground level*. The recursion levels from the top level down to the ground level are *above ground*, and from the ground level down to the bottom level are *below ground*. The recursion tree is executed breadth-first, so all the subproblems at a given level are solved independently in parallel; however, this is done differently for above-ground levels and for the ground level. Every above-ground level has fewer than p subproblems, each of which is distributed over several (typically, all) processors and solved in parallel. The ground level has p *ground subproblems*, each of which is allocated to a separate processor and solved sequentially by its owner processor executing the subproblem's below-ground recursion subtree. Thus, a divide-and-conquer BSP algorithm runs in three consecutive stages:

- the *descending divide stage*, executing the divide phase in each node of the recursion tree from the top level down to the ground level; this stage forms p ground subproblems;
- the *ground stage*, executing the divide phase in each node of the recursion tree from the ground down to the bottom level, and then the conquer phase in each node from the bottom back up to the ground level; this stage solves each of the p ground subproblems sequentially, allocating each to a different processor;
- the *ascending conquer stage*, executing the conquer phase in each node of the recursion tree from the ground level up to the top level; this stage assembles the solution to the main problem from solutions to ground subproblems.

The operation of the descending divide and the ascending conquer stages is typically distributed across the processors. The transition from the descending divide stage to the ground stage, and that from the ground stage to the ascending conquer stage, typically involve communication and a synchronisation barrier in order to redistribute the data. The transition between

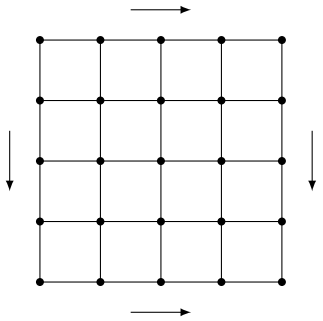


Figure 13.4: The ordered 2D grid dag

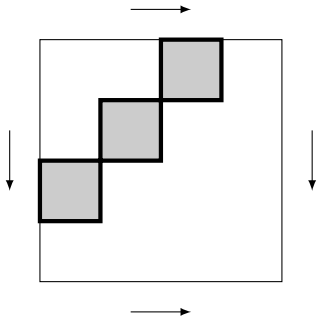


Figure 13.5: A snapshot of the ordered 2D grid computation

recursion levels within the descending divide and the ascending conquer stages may or may not involve communication; avoiding communication in such transition will be called *flattening*, and is typically necessary for obtaining flat or nearly flat BSP algorithms.

Ordered grid computation Many computational problems can be described as computing a directed acyclic graph (dag), which characterises the problem's data dependencies.

The *ordered 2D grid* dag of size n (Figure 13.4) consists of n^2 nodes arranged in an $n \times n$ grid, with edges directed top-to-bottom and left-to-right from each node to its neighbour nodes in the grid. The computation takes $2n$ inputs to the nodes on the left and top borders, and returns $2n$ outputs from the nodes on the right and bottom borders.

McColl [160] gave a BSP algorithm for the 2D ordered grid dag.

Theorem 13.1 *A 2D ordered grid dag can be computed in* $comp = O\big(\frac{n^2}{p}\big)$, $comm = O(n)$, $sync = O(p)$, *assuming* $n \geq p$. □

Proof The nodes are partitioned into a $p \times p$ grid of blocks. In every superstep, we evaluate an anti-diagonal *frontier* of at most p blocks, allocating each block to a different processor (Figure 13.5). The BSP cost and slackness analysis is straightforward. ■

The BSP costs of Theorem 13.1 have been proved optimal in [197]. Both Theorem 13.1 and the corresponding lower bounds have been vastly generalised by Ballard et al. [24].

Semi-local LCS Parallel algorithms for string comparison have been studied for a long time. Both Algorithm 5.8 (the classical dynamic programming LCS algorithm) and Algorithm 5.36 (semi-local LCS by incremental combing) have a structure that maps perfectly onto the 2D grid dag. Therefore, Theorem 13.1 applies directly to computing LCS and semi-local LCS. From now on, we assume for simplicity that $m = n$.

Theorem 13.2 *The semi-local LCS problem can be solved implicitly in* $comp = O\big(\frac{n^2}{p}\big)$, $comm = O(n)$, $sync = O(p)$, *assuming* $n \geq p$. □

Proof By Theorems 5.38 and 13.1. ■

Theorem 13.2 is very simple and seems difficult to improve. However, as we show in subsequent sections, it can still be substantially improved using the LCS kernel method.

13.4 Unit-Monge monoid with BSP

A crucial element in achieving an efficient parallelisation of the LCS kernel method is to obtain a parallel version of Theorem 4.16 (the Steady Ant algorithm). Despite the algorithm's intricate structure, it turns out that it can still be parallelised efficiently.

We first describe a quasi-flat work-optimal BSP algorithm for matrix sticky multiplication.

Theorem 13.3 (Quasi-flat Steady Ant) *Let* P, Q, R *be* $n \times n$ *(sub)permutation matrices, such that* $P \boxdot Q = R$. *Given the nonzeros of* P, Q, *the nonzeros of* R *can be obtained in* $comp = O\big(\frac{n \log n}{p}\big)$, $comm = O\big(\frac{n \log p}{p}\big)$, $sync = O(\log p)$, *assuming* $n \geq p^3$. □

Proof Without loss of generality, consider permutation matrices $P\langle 0{:}\,n; 0{:}\,n\rangle$, $Q\langle 0{:}\,n; 0{:}\,n\rangle$, $R\langle 0{:}\,n; 0{:}\,n\rangle$; the generalisation to subpermutation matrices is as in Theorem 3.15. The algorithm follows the divide-and-conquer structure of the sequential algorithm of Theorem 4.16 (the Steady Ant algorithm). Recursion level $\log p$ is designated as the ground level. Let $m = \frac{n}{p}$. The problem is solved in three consecutive stages.

Stage 1: Descending divide. We bundle together the divide phases of all the above-ground recursion levels, flattening them into a direct p-way partitioning of matrices P, Q into rectangular blocks

$$P = \begin{bmatrix} P_{0^+} & \dots & P_{p^-} \end{bmatrix} \qquad Q = \begin{bmatrix} Q_{0^+} \\ \vdots \\ Q_{p^-} \end{bmatrix}$$

where $P_{\hat{s}} = P\langle :; m\hat{s}^- : m\hat{s}^+\rangle$, $Q_{\hat{s}} = Q\langle m\hat{s}^- : m\hat{s}^+; :\rangle$, $\hat{s} \in \langle 0 : p\rangle$, are subpermutation matrices with m nonzeros each. We now have p ground subproblems

$$P_{0^+} \boxdot Q_{0^+} = R_{0^+} \qquad \ldots \qquad P_{p^-} \boxdot Q_{p^-} = R_{p^-}$$

where the product matrices $R_{\hat{s}}\langle 0 : n; 0 : n\rangle$, $\hat{s} \in \langle 0 : p\rangle$, are subpermutation matrices with m nonzeros each. Note that the multiplicands' nonzeros in all the subproblems have mutually disjoint index ranges in both coordinates. Therefore, matrix $\sum_{\hat{s}\in\langle 0:p\rangle} R_{\hat{s}}$, which represents the union of product nonzeros across all the subproblems, is a permutation matrix.

As in the algorithm of Theorem 4.16, we can now delete all zero rows and zero columns from matrices $P_{\hat{s}}$, $Q_{\hat{s}}$, $R_{\hat{s}}$, obtaining from each, after appropriate index remapping, a condensed $m \times m$ permutation matrix. We assign each of the resulting p ground subproblems to a separate processor. The whole stage runs in $comp, comm = O\big(\frac{n}{p}\big)$, $sync = O(1)$.

Stage 2: Ground. Every ground subproblem is solved sequentially by the algorithm of Theorem 4.16. The whole stage runs in $comp = O\big(\frac{n \log n}{p}\big)$, $comm = O\big(\frac{n}{p}\big)$, $sync = O(1)$.

Stage 3: Ascending conquer. In this stage, we execute the conquer phase in each of the $p - 1$ above-ground nodes of the recursion tree. Every such execution corresponds to the parallel solution of a sticky matrix multiplication subproblem, given the solutions of its two children subproblems as multiplicands. We keep the notation of Theorem 4.16 for the subproblems. Communication will be required in every node for transferring multiplicand nonzeros to the processors owning the corresponding parts of the product.

The computation in this stage is structured as follows. For the top-level main problem $P \boxdot Q = R$, its product matrix $R\langle 0 : n; 0 : n\rangle$ is partitioned into a $p \times p$ regular grid of blocks, each of size $m \times m$. The corners of the blocks correspond to integer grid points $[ms, mt]$, $s, t \in [0 : p]$. For a subproblem lower in the recursion tree, its product matrix will be partitioned into blocks accordingly, relative to its condensed index ranges.

As in the algorithm of Theorem 4.16, for every subproblem we will need to determine the meeting points of the left and the right border paths. A block of the product matrix that is visited by both these paths for a specific subproblem will be called an *essential block* for that subproblem. A border path enters an essential block at a single half-integer *entry point* on either its left or its bottom boundary, and leaves it at a single half-integer *exit point* on either its top or its right boundary. For every subproblem, we will perform the following substages:

3.1. identifying the subproblem's essential blocks and assign every such block to a processor, so that the blocks are distributed evenly across the processors

3.2. processing every essential block in parallel, obtaining the left and the right border paths' meeting points, and thereby the subproblem's fresh nonzeros.

We now describe each of these substages in more detail.

Substage 3.1: Identifying essential blocks. Consider the top-level main problem $P \boxdot Q = R$. Due to border path monotonicity, there are exactly $2p-1$ blocks visited by each of the border paths, therefore there are at most $2p-1$ essential blocks.

The blocks visited by each border path are identified by a sampling process. We *sample* dominance sums $R^{\Sigma}[0\colon n; 0\colon n]$ of the product matrix at the grid points $[mt, mu]$, $t, u \in [0\colon p]$ as follows. First, we sample dominance sums $R^{\Sigma}_{lo}[0\colon n; 0\colon n]$, $R^{\Sigma}_{hi}[0\colon n; 0\colon n]$ of the uncondensed children product matrices obtained in the previous recursion level, by evaluating them at the same set of grid points. (Note that although matrices R_{lo}, R_{hi} were already sampled at the previous recursion level as product matrices, it is still necessary to re-sample them here, since the sampling at the previous level was performed on the condensed form of these matrices, and therefore at a different set of grid points). It is straightforward to obtain the samples of R^{Σ}_{lo}, R^{Σ}_{hi} efficiently in parallel by a 2D prefix sums computation. We then obtain samples $R^{\Sigma}[ms, mt]$, $\Delta[ms, mt]$ by (4.3)–(4.6). At the top recursion level, we obtain $O(p^2)$ samples in $comp = O\big(\frac{n+p^2}{p}\big) = O\big(\frac{n}{p}\big)$, $comm = O\big(\frac{p^2}{p}\big) = O(p)$, $sync = O(1)$.

Using the samples of Δ, we now identify essential blocks of R. Consider a block $R\langle m\hat{t}^{-}\colon m\hat{t}^{+}; m\hat{u}^{-}\colon m\hat{u}^{+}\rangle$, for some $\hat{t}, \hat{u} \in \langle 0\colon p\rangle$. This block is visited by the left border path, if and only if

$$\Delta[m\hat{t}^{-}, m\hat{u}^{-}] < 0 \qquad \Delta[m\hat{t}^{+}, m\hat{u}^{+}] \geq 0$$

and by the right border path, if and only if

$$\Delta[m\hat{t}^{-}, m\hat{u}^{-}] \leq 0 \qquad \Delta[m\hat{t}^{+}, m\hat{u}^{+}] > 0$$

Therefore, a block should be marked as essential, if it is visited by both border paths:

$$\Delta[m\hat{t}^{-}, m\hat{u}^{-}] < 0 \qquad \Delta[m\hat{t}^{+}, m\hat{u}^{+}] > 0$$

A subproblem at a lower level of recursion will still be partitioned into a $p \times p$ grid of blocks; however, this grid will be different for every subproblem due to matrix condensation. Every subproblem will require p^2 samples, with block corners as sample points. In a given level, there are at most p subproblems, having n nonzeros in total, therefore $p \cdot O(p^2) = p^3$ samples will be required across the whole level. These samples can be obtained in $comp = O\big(\frac{n+p^3}{p}\big) = O\big(\frac{n}{p}\big)$, $comm = O\big(\frac{p^3}{p}\big) = O(p^2)$, $sync = O(1)$.

Since there are $\log p$ above-ground recursion levels, the sampling across all the levels can be completed in $comp = \log p \cdot O\big(\frac{n}{p}\big) = O\big(\frac{n \log p}{p}\big)$, $comm = \log p \cdot O(p^2) = O(p^2 \log p)$, $sync = O(1)$. For every subproblem, the essential blocks are determined similarly to the above.

Substage 3.2: Processing essential blocks. For every essential block, we will execute the following subsubstages:

3.2.1. communicate to the block's owner processor the necessary data from children subproblems;

3.2.2. evaluate children subproblem dominance sums at block boundaries and locate the entry points of the left and the right border path into the block;

3.2.3. traverse the sections of the subproblem's left and right border paths within the block and obtain the subproblem's fresh nonzeros within the block.

We now describe each of these subsubstages in more detail. While they all need to be performed for every subproblem, for simplicity of notation we first describe in detail the execution of each subsubstage on the top-level main problem $P \boxdot Q = R$, and then outline it for subproblems at lower levels.

Subsubstage 3.2.1: Communicating children subproblem data. Consider the top-level main problem $P \boxdot Q = R$. Out of p^2 blocks of R, at most $2p - 1$ blocks are essential; we assign these essential blocks to processors, so that at most two blocks are assigned to the same processor. For every essential block, its owner processor will need to receive the data necessary for traversing the two border paths within the block. Consider an essential block

$$R\langle m\hat{t}^- : m\hat{t}^+ ; m\hat{u}^- : m\hat{u}^+ \rangle = R\langle i_0 - m : i_0 ; m\hat{k}_0 : k_0 + m \rangle$$

for some $\hat{t}, \hat{u} \in \langle 0 : p \rangle$, where $i_0 = m\hat{t}^+$, $k_0 = m\hat{u}^-$. This block's owner processor receives

- the samples $R^{\Sigma}_{lo}[0, m\hat{u}^-]$, $R^{\Sigma}_{lo}[m\hat{t}^+, m\hat{u}^-]$, $R^{\Sigma}_{lo}[m\hat{t}^+, n]$, and analogously for R_{hi}; these samples have been obtained in the course of sampling matrix R^{Σ} in Substage 1;
- nonzeros of submatrices $R_{lo}\langle m\hat{t}^- : m\hat{t}^+ ; : \rangle$, $R_{lo}\langle : ; m\hat{u}^- : m\hat{u}^+ \rangle$, and analogously for R_{hi}, i.e. the nonzeros of R_{lo}, R_{hi} within a horizontal strip and a vertical strip of width m containing the current block of R; since $R_{lo} + R_{hi}$ is a permutation matrix, there are at most m nonzeros per strip, hence $2m$ nonzeros overall;

There are six samples and at most $2m$ nonzeros communicated to the current essential block's owner processor, in $comp, comm = O(1) + O(m) = O\big(\frac{n}{p}\big)$.

Subsubstage 3.2.2: Evaluating dominance sums at block boundaries. Given the samples $R^{\Sigma}_{lo}[m\hat{t}^{+}, m\hat{u}^{-}]$, $R^{\Sigma}_{lo}[m\hat{t}^{+}, n]$ as initial values, and the nonzeros of $R_{lo}\langle m\hat{t}^{-} : m\hat{t}^{+}; : \rangle$, we evaluate the vectors $R^{\Sigma}_{lo}[m\hat{t}^{-} : m\hat{t}^{+}; m\hat{u}^{-}]$, $R^{\Sigma}_{lo}[m\hat{t}^{-} : m\hat{t}^{+}; n]$ (i.e. the values of R^{Σ}_{lo} at the current block's left boundary, and at the block's projection onto the matrix' right boundary) by incremental columnwise queries of Theorem 4.8. Analogously, we evaluate both R^{Σ}_{lo} and R^{Σ}_{hi} at the current block's left, top, right and bottom boundaries, and at the block's projection onto the matrix' right and top boundaries. The computation uses only data local to the current block's owner processor.

Given the elements of R^{Σ}_{lo}, R^{Σ}_{hi} at the current block's left, top, right and bottom boundaries, we then obtain the corresponding elements of $\Delta = R^{\blacksquare}_{hi} - R^{\blacksquare}_{lo}$. At the current block's left boundary, we have

$$\Delta[i, m\hat{u}^{-}] = R^{\blacksquare}_{hi}[i, m\hat{u}^{-}] - R^{\blacksquare}_{lo}[i, m\hat{u}^{-}] = \big(R^{\Sigma}_{hi}[0, m\hat{u}^{-}] - R^{\Sigma}_{hi}[i, m\hat{u}^{-}]\big) - \big(R^{\Sigma}_{lo}[i, n] - R^{\Sigma}_{lo}[i, m\hat{u}^{-}]\big)$$

where the value $R^{\Sigma}_{hi}[0, m\hat{u}^{-}]$ on the right-hand side is a sample, and the remaining three values are elements of the vectors we have just obtained. We evaluate Δ analogously at the current block's three remaining boundaries.

The entry point of the left border path into the current block is a half-integer point on the block's bottom-left boundary that separates negative and nonnegative elements of Δ. The entry point of the right border path is determined symmetrically, separating nonpositive and positive elements. Overall, the described processing of the block's boundaries is performed in $comp = O(m) = O\big(\frac{n}{p}\big)$, without any communication or synchronisation.

A subproblem at a lower recursion level will have its essential blocks' boundaries processed analogously, in $comp = O(m) = O\big(\frac{n}{p}\big)$ across all the subproblems in a given level. Since there are $\log p$ above-ground recursion levels, this subsubstage can be completed across all the levels in $comp = O\big(\frac{n \log p}{p}\big)$.

Subsubstage 3.2.3: Traversing border paths. Any element of matrix $R^{\blacksquare}_{hi}$ within the current block can now be obtained from local data as

$$R^{\blacksquare}_{hi}[i, k] = R^{\blacksquare}_{hi}[i, m\hat{u}^{-}] + R^{\blacksquare}_{hi}[m\hat{t}^{-}, k] - R^{\blacksquare}_{hi}[m\hat{t}^{-}, m\hat{u}^{-}] + \sum R_{hi}\langle m\hat{t}^{-} : i; m\hat{u}^{-} : k\rangle$$

Any element of matrix $R^{\blacksquare}_{lo}$ within the current block can be obtained analogously. We then follow the path traversal procedure of Theorem 4.16, making use of incremental queries of Theorem 4.8 to update the nonzero count $\sum R_{hi}\langle m\hat{t}^{-} : i; m\hat{u}^{-} : k\rangle$ in the above expression for $R^{\blacksquare}_{hi}[i, k]$, and the analogous nonzero count for $R^{\blacksquare}_{lo}[i, k]$. The updated elements of $R^{\blacksquare}_{hi}$, $R^{\blacksquare}_{lo}$ along the

path are then used to update the corresponding element of $\Delta = R^{\bowtie}_{hi} - R^{\bowtie}_{lo}$, the sign of which guides the path traversal procedure of Theorem 4.16.

Once the left and the right border paths have been traversed in the current block, the fresh nonzeros of matrix R within the block can be obtained as in Theorem 4.16. Overall, the described border path traversal procedure is performed in $comp = O(m) = O\big(\frac{n}{p}\big)$, without any communication or synchronisation.

Overall, the described subsubstages of processing an essential block for the main problem are performed in $comp, comm = O\big(\frac{n}{p}\big)$, $sync = O(1)$. A subproblem at a lower recursion level will be processed in similar subsubstages. In a given level of the recursion tree, there are at most p subproblems, having n nonzeros in total. Furthermore, the blocks of every subproblem are distributed evenly across the processors. Therefore, the subsubstages can be performed in $comp, comm = O(p^2) + O(m) = O\big(\frac{n}{p}\big)$ and $sync = O(1)$ across all the subproblems in a given level. Since there are $\log p$ above-ground recursion levels, the whole ascending conquer stage can be completed in $comp, comm = O\big(\frac{n \log p}{p}\big)$, $sync = O(\log p)$.

Total cost. Across the three stages of the algorithm, we have

$$\begin{aligned} comp &= O\big(\tfrac{n}{p}\big) + O\big(\tfrac{n \log n}{p}\big) + O\big(\tfrac{n \log p}{p}\big) = O\big(\tfrac{n \log n}{p}\big) \\ comm &= O\big(\tfrac{n}{p}\big) + O\big(\tfrac{n}{p}\big) + O\big(\tfrac{n \log p}{p}\big) = O\big(\tfrac{n \log p}{p}\big) \\ sync &= O(1) + O(1) + O(\log p) = O(\log p) \end{aligned}$$ ∎

With a little more effort, it is possible to flatten the ascending conquer stage of the algorithm of Theorem 13.3, obtaining a flat work-optimal BSP algorithm. This comes at the expense of increasing the slackness requirement to superpolynomial; this requirement will be relaxed in subsequent sections.

Theorem 13.4 (Flat Steady Ant) *Let P, Q, R be $n \times n$ (sub)permutation matrices, such that $P \boxdot Q = R$. Given the nonzeros of P, Q, the nonzeros of R can be obtained in $comp = O\big(\frac{n \log n}{p}\big)$, $comm = O\big(\frac{n \log p}{p}\big)$, $sync = O(1)$, assuming $n \geq 2^{(\log p)^2}$.* □

Proof As before, without loss of generality we only consider permutation matrices. The algorithm's structure is similar to that of Theorem 13.3.

Stages 1–2: Descending divide and ground. As in Theorem 13.3.

Stage 3: Ascending conquer. We perform the same substages as in Theorem 13.3. However, in order to reduce the number of synchronisation barriers, we no longer perform communication and synchronisation between different recursion levels, replacing them by recomputation of the data that are no longer available from previous levels. All the $\log p$ recursion levels are thus executed simultaneously. The resulting flattened version of the ascending conquer stage is as follows.

Substage 3.1: Identifying essential blocks. When sampling the dominance sums R^{Σ} of the product matrix, the children product matrices R_{lo}, R_{hi} are not yet available, since they are produced simultaneously at different levels of the recursion tree. Instead, we sample the dominance sums R^{Σ} at every grid point $[ms, mt]$, $s, t \in [0\colon p]$ by evaluating recursively the dominance sums of the uncondensed children product matrices R_{lo}, R_{hi}, grandchildren product matrices, etc., invoking (4.3)–(4.5) in each recursive step. The base of the recursion evaluates the dominance sums $R^{\Sigma}_{\hat{s}}[ms, mt]$, $\hat{s} \in \langle 0\colon p\rangle$, on each of the uncondensed product matrices $R_{\hat{s}}$ obtained in the ground stage. It is straightforward to obtain the samples of $R^{\Sigma}_{\hat{s}}$ efficiently in parallel by a 2D prefix sums computation. In total, while sampling dominance sums for the top-level product matrix, we obtain $O(p^2)$ samples for each of the $2p-1$ above-ground descendant product matrices. Overall, $O(p^2) \cdot O(p) = O(p^3)$ samples are obtained in $comp = O\big(\frac{n+p^3}{p}\big) = O\big(\frac{n}{p}\big)$, $comm = O\big(\frac{p^3}{p}\big) = O(p^2)$, $sync = O(1)$.

At lower levels of the recursion tree, sampling dominance sums of a product matrix will require taking fewer samples from its descendant product matrices. Consider a fixed recursion level. Different subproblems at that level have disjoint sets of descendant subproblems, so the total of $O(p^3)$ recursive samples will be taken while sampling dominance sums of product matrices across the whole level. As before, these recursive samples can be obtained in $comp = O\big(\frac{n+p^3}{p}\big) = O\big(\frac{n}{p}\big)$, $comm = O\big(\frac{p^3}{p}\big) = O(p^2)$, $sync = O(1)$. Since there are $\log p$ above-ground recursion levels, the recursive sampling across all levels can be completed in $comp = O\big(\frac{n \log p}{p}\big)$, $comm = O(p^2 \log p)$. The sampling for each subproblem is independent, therefore we still have $sync = O(1)$. For every subproblem, the blocks visited by each border path are determined as before, and a block is marked as essential for a particular subproblem, if it is visited by both border paths.

Using the samples of R^{Σ}, we sample matrix Δ and identify essential blocks of matrix R as in Theorem 13.3.

Substage 3.2: Processing essential blocks. As in the previous substage, the children subproblem data are not yet available, so we must recompute them recursively from ground level subproblem data.

Subsubstage 3.2.1: Communicating ground level subproblem data. As in Theorem 13.3, consider the top-level main problem $P \boxdot Q = R$, and an essential block

$$R\langle m\hat{t}^{-}\colon m\hat{t}^{+}; m\hat{u}^{-}\colon m\hat{u}^{+}\rangle = R\langle i_0 - m\colon i_0; m\hat{k}_0\colon k_0 + m\rangle$$

for some $\hat{t}, \hat{u} \in \langle 0\colon p\rangle$, where $i_0 = m\hat{t}^{+}$, $k_0 = m\hat{u}^{-}$. This block's owner processor receives

- the samples $R^{\Sigma}_{\hat{s}}[0, m\hat{u}^{-}]$, $R^{\Sigma}_{\hat{s}}[m\hat{t}^{+}, m\hat{u}^{-}]$, $R^{\Sigma}_{\hat{s}}[m\hat{t}^{+}, n]$, for all $\hat{s} \in \langle 0\colon p\rangle$; these samples have been obtained in the course of sampling matrix R^{Σ} in Substage 1;

- nonzeros of submatrices $R_{\hat{s}}\langle m\hat{t}^-\colon m\hat{t}^+;:\rangle$, $R_{\hat{s}}\langle:;m\hat{u}^-\colon m\hat{u}^+\rangle$, for all $\hat{s}\in\langle 0\colon p\rangle$; i.e. the nonzeros of $R_{\hat{s}}$ within a horizontal strip and a vertical strip of width m containing the current block of R; since $\sum_{\hat{s}} R_{\hat{s}}$ is a permutation matrix, there are at most m nonzeros per strip, hence $2m$ nonzeros overall.

There are $3p$ samples and at most $2m$ nonzeros communicated to the current essential block's owner processor in *comp*, $comm = O(p) + O(m) = O\big(\frac{n}{p}\big)$.

Subsubstages 3.2.2–3.2.3: Evaluating dominance sums at block boundaries and traversing border paths. As in Theorem 13.3, we evaluate the vectors $R^{\mathbf{\Sigma}}_{\hat{s}}[m\hat{t}^-\colon m\hat{t}^+; m\hat{u}^-]$, $R^{\mathbf{\Sigma}}_{\hat{s}}[m\hat{t}^-\colon m\hat{t}^+; n]$ (i.e. the values of $R^{\mathbf{\Sigma}}_{\hat{s}}$ at the current block's left boundary, and at the block's projection onto the matrix' right boundary) for all $\hat{s}\in\langle 0\colon p\rangle$ by incremental columnwise queries of Theorem 4.8. Analogously, we evaluate $R^{\mathbf{\Sigma}}_{\hat{s}}$ at the current block's left, top, right and bottom boundaries, and at the block's projection onto the matrix' right and top boundaries. The computation uses only data local to the current block's owner processor.

We proceed to evaluate the dominance sums of the product matrices at each level recursively by (4.3)–(4.5). At every recursion level, we check whether the current block happens to be essential (as determined by the recursive samples of dominance sums at the block's corners), and if so, we traverse the two border paths and determine the fresh nonzeros as in Theorem 13.3. This includes the top level, since at that level the current block is already known to be essential.

At each recursion level, the total number of nonzeros participating in this computation is at most $2m = O(n/p)$. Furthermore, the computation starts on an uncondensed block at the top recursion level, but can then proceed by condensing the matrices at every level, deleting zero rows and columns from the block, along with the corresponding elements of the boundary vectors. Since there are $\log p$ above-ground recursion levels, each recomputing up to $\log p$ lower above-ground recursion levels, these subsubstages can be completed across all the levels in $comp = O\big(\frac{n\log^2 p}{p}\big)$, $comm = O\big(\frac{n\log p}{p}\big)$, $sync = O(1)$.

Total cost. Across the three stages of the algorithm, we have

$$\begin{aligned} comp &= O\big(\tfrac{n}{p}\big) + O\big(\tfrac{n\log n}{p}\big) + O\big(\tfrac{n\log^2 p}{p}\big) = O\big(\tfrac{n\log n}{p}\big)\\ comm &= O\big(\tfrac{n}{p}\big) + O\big(\tfrac{n}{p}\big) + O\big(\tfrac{n\log p}{p}\big) = O\big(\tfrac{n\log p}{p}\big)\\ sync &= O(1) + O(1) + O(1) = O(1) \end{aligned}$$

■

We observe that the algorithm of Theorem 13.4 has a superpolynomial slackness requirement, which is not entirely satisfactory, but seems necessary in order to achieve work-optimality. Furthermore, this algorithm would typically be used as a subroutine called a higher-level algorithm with a

higher local computation cost (such as the semi-local LCS algorithm in the following section). In this case, work-optimality of the subroutine may not be required, since its local computation cost would be dominated by the cost of the calling algorithm, and therefore the slackness condition may be relaxed.

13.5 Semi-local LCS with BSP

We now describe two types of BSP algorithms for semi-local LCS. Both types are based on recursive partitioning of the problem in a quaternary tree. They share the (trivial) descending divide stage, obtaining p ground subproblems of size $\frac{n}{p^{1/2}}$, and the ground stage, which solves every ground subproblem sequentially, obtaining its LCS kernel with $\frac{2n}{p^{1/2}}$ nonzeros; across all the ground subproblems, there are $p \cdot \frac{2n}{p^{1/2}} = 2np^{1/2}$ *ground nonzeros*. The two algorithm types differ in the ascending conquer stage, which may be based on either

- the sequential LCS kernel composition by Theorem 4.16; this algorithm type, although simpler than the second, has not been considered previously; the resulting algorithms are flat or nearly flat, but are not communication-scalable;
- the flat BSP LCS kernel composition by Theorem 13.4; this algorithm type is a refinement of [139, 141]; due to an extra layer of synchronisation compared to the first type, the resulting algorithm is only quasi-flat, but is communication-scalable.

Flat and nearly flat semi-local LCS In this type of algorithm, we flatten certain subsets of the above-ground recursion levels. Flattening the whole descending divide stage into a single superstep is easy; however, flattening the ascending conquer stage requires some care. At an extreme, doing it for the whole stage by collecting all ground nonzeros in a single processor would achieve $sync = O(1)$, at the expense of substantially superlinear $comm = O(np^{1/2})$. We take a more careful approach, achieving either nearly flat synchronisation with linear communication, or flat synchronisation with only slightly superlinear communication.

Theorem 13.5 *The semi-local LCS problem can be solved implicitly in* $comp = O\big(\frac{n^2}{p}\big)$, $comm = O(n)$, $sync = O(\log\log p)$, *assuming* $n \geq p$. □

Proof The semi-local LCS problem is partitioned recursively in a quaternary tree. Recursion level $\frac{\log p}{2}$ is designated as the ground level. The problem is solved in three consecutive stages.

Stage 1: Descending divide. We bundle together the divide phases of all the above-ground recursion levels, flattening them into a direct $p^{1/2}$-way partitioning of the input strings. Strings a, b are each partitioned into $p^{1/2}$ regular substrings of length $\frac{n}{p^{1/2}}$. The LCS grid is partitioned accordingly into a grid of $p = p^{1/2} \times p^{1/2}$ regular square blocks, each corresponding to a ground subproblem. We assign each of the resulting p ground subproblems to a separate processor. The whole stage runs in $comp, comm = O\big(\frac{n}{p^{1/2}}\big)$, $sync = O(1)$.

Stage 2: Ground. Every ground subproblem is solved sequentially by Algorithm 5.33 or Algorithm 5.36. The whole stage runs in $comp = O\big(\frac{n^2}{p}\big)$, $comm = O\big(\frac{n}{p^{1/2}}\big)$, $sync = O(1)$.

Stage 3: Ascending conquer. In this stage, we execute the conquer phases of all the above-ground recursion levels. Each semi-local LCS subproblem is solved by composing its four childrens' LCS kernels, obtaining the parent's LCS kernel. The composition is performed sequentially by Theorem 4.16. Without flattening, communication would be required for transferring the children's LCS kernels to the processor owning the parent's LCS kernel. In particular, the recursion level just above ground would be executed in parallel by all p processors in $comp = O\big(\frac{n \log n}{p^{1/2}}\big)$, $comm = O\big(\frac{n}{p^{1/2}}\big)$. The top recursion level would be executed sequentially by a single processor in $comp = O(n \log n)$, $comm = O(n)$. Across the whole stage, $comp$ and $comm$ would be dominated by the top level, hence the stage would run in $comp = O(n \log n)$, $comm = O(n)$, $sync = O(\log p)$.

In order to reduce the synchronisation cost, we bundle together subsets of successive levels, flattening them as follows. Consider bundling the bottom third (or any other constant positive fraction strictly less than $\frac{1}{2}$) of the above-ground levels, from the ground level $\frac{\log p}{2}$ to the *bundle level* $\frac{2}{3} \cdot \frac{\log p}{2}$. The bundle level has $p^{2/3}$ independent *bundle subproblems*, each of which is of size $\frac{n}{p^{1/3}}$, and has $p/p^{2/3} = p^{1/3}$ descendant ground subproblems. We assign every bundle subproblem to a different processor (the remaining $p - p^{2/3}$ processors are left idle), collect all the $p^{1/3} \cdot \frac{2n}{p^{1/2}} = \frac{2n}{p^{1/6}}$ kernel nonzeros of its descendant ground subproblems in its owner processor, and execute the ascending conquer phase for every bundle subproblem sequentially in $comp = O\big(\frac{n \log n}{p^{1/6}}\big)$, $comm = O\big(\frac{n}{p^{1/6}}\big)$. We have now obtained an LCS kernel with $\frac{2n}{p^{1/3}}$ nonzeros for each of the $p^{2/3}$ bundle subproblems. There are $p^{2/3} \cdot \frac{2n}{p^{1/3}} = 2np^{1/3}$ kernel nonzeros in total across all the bundle subproblems.

We continue bundling recursion levels in a similar fashion in subsequent rounds. At the beginning of round k, there are $2np^{2^{k-2}/3^{k-1}}$ kernel nonzeros: thus, as described above, at the beginning of round 1 there are $2np^{1/2}$ nonzeros, and at the beginning of round 2 (which is the end of round 1) there are $2np^{1/3}$ nonzeros. In each round, we bundle the bottom third of

the remaining levels, obtaining $p^{2/3^k}$ independent bundle subproblems, each of size $\frac{n}{p^{2^{k-1}/3^k}}$. Every bundle subproblem is assigned to a different processor (the remaining $p - p^{2/3^k}$ processors are left idle). The bundle is executed in

$$comp = O\big(\tfrac{np^{2^{k-2}/3^{k-1}} \log n}{p^{2/3^k}}\big) = O\big(\tfrac{n \log n}{p^{2^{k-2}/3^k}}\big)$$
$$comm = \tfrac{np^{2^{k-2}/3^{k-1}}}{p^{2/3^k}} = O\big(\tfrac{n}{p^{2^{k-2}/3^k}}\big)$$

We have now obtained an LCS kernel with $\frac{2n}{p^{2^{k-1}/3^k}}$ nonzeros for each of the $p^{2/3^k}$ bundle subproblems. We have

$$p^{2/3^k} \cdot \tfrac{2n}{p^{2^{k-1}/3^k}} = 2np^{2^{k-1}/3^k}$$

kernel nonzeros in total across all the bundle subproblems.

Across the whole stage, local computation and communication are dominated by the top level, hence the stage runs in $comp = O(n \log n)$, $comm = O(n)$, $sync = O(\log p)$. Every bundling round removes a constant fraction (one third) of the remaining above-ground levels. Therefore, the total number of rounds is $O(\log \log p)$. Every round runs in $sync = O(1)$, therefore the whole stage runs in $sync = O(\log \log p)$.

Total cost. Across the three stages of the algorithm, we have

$$comp = O\big(\tfrac{n}{p^{1/2}}\big) + O\big(\tfrac{n^2}{p}\big) + O(n) = O\big(\tfrac{n^2}{p}\big)$$
$$comm = O\big(\tfrac{n}{p^{1/2}}\big) + O\big(\tfrac{n}{p^{1/2}}\big) + O(n) = O(n)$$
$$sync = O(1) + O(1) + O(\log \log p) = O(\log \log p)$$ ∎

Theorem 13.5 provides a nearly flat work-optimal algorithm for semi-local LCS. At the expense of increasing the communication from linear to slightly superlinear, it is possible to modify the algorithm of Theorem 13.5, making it genuinely flat.

Theorem 13.6 *The semi-local LCS problem can be solved implicitly in* $comp = O\big(\tfrac{n^2}{p}\big)$, $comm = O(np^\epsilon)$, $sync = O(\log(1/\epsilon))$ *for all* $\epsilon > 0$, *assuming* $n \geq p$.□

Proof The algorithm's structure is similar to that of Theorem 13.5.

Stages 1–2: Descending divide and ground. As in Theorem 13.5.

Stage 3: Ascending conquer. We perform the same bundling procedure as in Theorem 13.5, stopping after l bundling rounds (the value of l to be determined later). We bundle all the remaining recursion levels together. As shown in the proof of Theorem 13.5, there are $2np^{2^{l-1}/3^l}$ LCS kernel nonzeros in total at the beginning of the final bundle. Let $l = \log_{3/2}(1/\epsilon)$, then $2np^{2^{l-1}/3^l} < 2np^{1/(3/2)^l} = 2np^\epsilon$. The resulting BSP algorithm runs in $comp = O\big(\tfrac{n^2}{p}\big)$, $comm = O(np^\epsilon)$, $sync = O(\log(1/\epsilon))$. ∎

A flat work-optimal BSP algorithm for the semi-local LCS problem is obtained by Theorem 13.6, taking $\epsilon = O(1)$.

Quasi-flat communication-scalable semi-local LCS Krusche and Tiskin [139, 142] gave BSP algorithms for the semi-local LCS problem, running in $comp = O\big(\frac{n^2}{p}\big)$, $comm = O\big(\frac{n \log p}{p^{1/2}}\big)$, $sync = O(\log p)$ (respectively $comm = O\big(\frac{n}{p^{1/2}}\big)$, $sync = O(\log^2 p)$). Thus, they traded off communication and synchronisation, while not being able to minimise them simultaneously in the quasi-flat, communication-scalable regime. Using the flat BSP algorithm of Theorem 13.4 for LCS kernel composition, we are able to remove this part of the tradeoff, achieving simultaneously the lower communication of [142], and the lower synchronisation of [139].

Theorem 13.7 *The semi-local LCS problem can be solved implicitly in* $comp = O\big(\frac{n^2}{p}\big)$, $comm = O\big(\frac{n}{p^{1/2}}\big)$, $sync = O(\log p)$, *assuming* $n \geq p$. □

Proof The algorithm's structure is similar to that of Theorem 13.5. However, we do not perform any bundling of recursion levels. Instead, we invoke the algorithm of Theorem 13.4 for the conquer phase in every above-ground subproblem.

Stages 1–2: Descending divide and ground. As in Theorem 13.5.

Stage 3: Ascending conquer. As in Theorem 13.5, in this stage we evaluate the conquer phase in each of the $O(p)$ above-ground nodes of the recursion tree. Every such evaluation corresponds to the solution of a semi-local LCS subproblem, given the solutions of its four children subproblems. Such a solution is obtained by composing the children LCS kernels, obtaining their parent LCS kernel. In contrast with Theorem 13.5, we replace sequential composition by Theorem 4.16 with parallel one by Theorem 13.4. The recursion level just above ground will still be running in $comp = O\big(\frac{n \log n}{p^{1/2}}\big)$, $comm = O\big(\frac{n}{p^{1/2}}\big)$. The top recursion level will run in $comp = O\big(\frac{n \log n}{p}\big)$, $comm = O\big(\frac{n \log p}{p}\big)$. Across the whole stage, local computation and communication are dominated by the ground level, hence the stage will run in $comp = O\big(\frac{n \log n}{p^{1/2}}\big)$, $comm = O\big(\frac{n}{p^{1/2}}\big)$, $sync = O(\log p)$.

Total cost. Across the three stages of the algorithm, we have

$$comp = O\big(\tfrac{n}{p^{1/2}}\big) + O\big(\tfrac{n^2}{p}\big) + O\big(\tfrac{n \log n}{p^{1/2}}\big) = O\big(\tfrac{n^2}{p}\big)$$
$$comm = O\big(\tfrac{n}{p^{1/2}}\big) + O\big(\tfrac{n}{p^{1/2}}\big) + O\big(\tfrac{n}{p^{1/2}}\big) = O\big(\frac{n}{p^{1/2}}\big)$$
$$sync = O(1) + O(1) + O(\log p) = O(\log p)$$ ■

Note that, in contrast with Theorem 13.4, the algorithm of Theorem 13.7 has a very modest slackness requirement, and in particular does not require

superpolynomial (or even superlinear) slackness. This is because the local computation in this algorithm is dominated heavily by the ground stage. Therefore, although the algorithm of Theorem 13.4 is invoked by the ascending conquer stage of Theorem 13.7, that invocation has its local computation cost masked by the ground stage, and therefore does not have to be work-optimal.

13.6 Semi-local LCSP with BSP(TODO)

Chapter 14

Conclusions

We have presented 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.

A number of questions related to the semi-local string comparison framework remain open. For instance, is it possible to perform sticky multiplication of permutation matrices even faster? Is it possible to compute a given sticky power of a permutation matrix asymptotically faster than with the standard square-and-multiply method? Can the sticky braid framework be extended to sequence alignment with affine gap penalties? Positive answers to these questions would improve and extend various algorithms presented in this work.

In summary, the algebra of string comparison turns out to be a powerful technique, which unifies, and often improves on, a number of existing approaches to various substring- and subsequence-related problems. It is likely that further development of this approach will give it even more scope and power.

Bibliography

[1] CLC Protein Workbench: User Manual, 2006.

[2] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight Hardness Results for LCS and Other Sequence Similarity Measures. In *Proceedings of FOCS*, pages 59–78. IEEE, oct 2015.

[3] A. Aggarwal, D. Kravets, J. K. Park, and S. Sen. Parallel Searching in Generalized Monge Arrays. *Algorithmica*, 19(3):291–317, nov 1997.

[4] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1–4):195—-208, 1987.

[5] 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.

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

[7] M Ajtai, J Komlós, and E Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM STOC*, pages 1–9, 1983.

[8] M Ajtai, J Komlós, and E Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[9] S W Al-Haj Baddar and K E Batcher. *Designing Sorting Networks: A New Paradigm*. Springer, 2011.

[10] 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.

[11] 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.

[12] Michael H Albert, Alexander Golynski, Angèle M Hamel, Alejandro López-Ortiz, S.Srinivasa Rao, and Mohammad Ali Safari. Longest increasing subsequences in sliding windows. *Theoretical Computer Science*, 321:405—-414, 2004.

[13] L Allison and T I Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23(5):305–310, 1986.

[14] 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.

[15] Amihood Amir, Gary Benson, and Martin Farach. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. *Journal of Computer and System Sciences*, 52:299—-307, 1996.

[16] A Apostolico. Remark on the Hsu–Du new algorithm for the longest common subsequence problem. *Information Processing Letters*, 25:235—-236, 1987.

[17] A Apostolico, S Browne, and C Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92:3—-17, 1992.

[18] A Apostolico and C Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987.

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

[20] Abdullah N. Arslan and Ömer Eğecioğlu. Dynamic Programming Based Approximation Algorithms for Sequence Alignment with Constraints. *INFORMS Journal on Computing*, 16(4):441–458, 2004.

[21] Abdullah N. Arslan and Ömer Eğecioğlu. *Dynamic and fractional programming-based approximation algorithms for sequence alignment with constraints*. Chapman and Hall/CRC Computer and Information Science Series. Chapman and Hall/CRC, 2007.

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

[23] M J Atallah, S R Kosaraju, L L Larmore, G L Miller, and S.-H. Teng. Constructing trees in parallel. In *Proceedings of the 1st ACM SPAA*, pages 421–431, 1989.

[24] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[25] M Barsky, U Stege, A Thomo, and C Upton. A graph approach to the threshold all-against-all substring matching problem. *Journal of Experimental Algorithmics*, 12:1.10:1—-1.10:26, 2008.

[26] K E Batcher. Sorting networks and their applications. In *AFIPS Conference Proceedings*, volume 32, pages 307–314. Thompson Book Company, 1968.

[27] Laura Baxter, Aleksey Jironkin, Richard Hickman, Jay Moore, Christopher Barrington, Peter Krusche, Nigel P. Dyer, Vicky Buchanan-Wollaston, Alexander Tiskin, Jim Beynon, Katherine Denby, and Sascha Ott. Conserved Noncoding Sequences Highlight Shared Components of Regulatory Networks in Dicotyledonous Plants. *The Plant Cell*, 24(10):3949–3965, 2012.

[28] W W Bein and P K Pathak. A characterization of the Monge property and its connection to statistics. *Demonstratio Mathematica*, 29:451–457, 1996.

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

[30] J L Bentley. Multidimensional Divide-and-Conquer. *Communications of the ACM*, 23:214—-229, 1980.

[31] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings - 7th International Symposium on String Processing and Information Retrieval, SPIRE 2000*, pages 39–48, 2000.

[32] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76:7—-11, 2000.

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

[34] Philip Bille, Rolf Fagerberg, and Inge Li 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.

[35] Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008.

[36] Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random Access to Grammar-Compressed Strings and Trees. *SIAM Journal on Computing*, 44(3):513—-539, jan 2015.

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

[38] Henrik Blunck and Jan Vahrenhold. In-Place Algorithms for Computing (Layers of) Maxima. *Algorithmica*, 57:1—-21, may 2010.

[39] Miklós Bóna. *Combinatorics of Permutations*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, 2004.

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

[41] Karl Bringmann and Marvin Künnemann. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. In *Proceedings of FOCS*, pages 79–97. IEEE, oct 2015.

[42] Anders Skovsted Buch, Andrew Kresch, Mark Shimozono, Harry Tamvakis, and Alexander Yong. Stable Grothendieck polynomials and K-theoretic factor sequences. *Mathematische Annalen*, 340(2):359–382, feb 2008.

[43] Adam L. Buchsbaum and Michael T. Goodrich. Three-Dimensional Layers of Maxima. *Algorithmica*, 39:275—-286, aug 2004.

[44] H Bunke and U Bühler. Applications of approximate string matching to 2D shape recognition. *Pattern Recognition*, 26(12):1797–1812, 1993.

[45] V Y Burdyuk and V N Trofimov. Generalization of the results of Gilmore and Gomory on the solution of the traveling salesman problem. *Engineering Cybernetics*, 14:12–18, 1976.

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

[47] Rainer E. Burkard. Monge properties, discrete convexity and applications. *European Journal of Operational Research*, 176(1):1–14, 2007.

[48] P Butković. *Max-linear Systems: Theory and Algorithms*. Springer Monographs in Mathematics. Springer, 2010.

[49] Jeff Calder, Selim Esedoḡlu, and Alfred O. Hero. A Hamilton–Jacobi Equation for the Continuum Limit of Nondominated Sorting. *SIAM Journal on Mathematical Analysis*, 46:603—-638, jan 2014.

[50] Jeff Calder, Selim Esedoḡlu, and Alfred O. Hero. A PDE-based Approach to Nondominated Sorting. *SIAM Journal on Numerical Analysis*, 53:82—-104, jan 2015.

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

[52] Patrick Cégielski, Irène Guessarian, Yury Lifshits, and Yuri Matiyasevich. Window Subsequence Problems for Compressed Texts. In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 127—-136. 2006.

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

[54] Timothy M. Chan and Mihai Pătraşcu. Counting Inversions, Offline Orthogonal Range Counting, and Related Problems. In *Proceedings of SODA*, pages 161—-173, Philadelphia, PA, 2010.

[55] 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.

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

[57] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. Dynamic String Alignment. In *Proceedings of CPM*, volume 161 of *LIPIcs*, pages 9:1—-9:13, 2020.

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

[59] Erdong Chen, Linji Yang, and Hao Yuan. Longest increasing subsequences in windows based on canonical antichain partition. *Theoretical Computer Science*, 378(3):223—-236, jun 2007.

[60] R Cole and R Hariharan. Approximate String Matching: A Simpler Faster Algorithm. *SIAM Journal on Computing*, 31:1761—-1782, 2002.

[61] J.-P. Comet. Application of Max-Plus algebra to biological sequence comparisons. *Theoretical Computer Science*, 293:189—-217, 2003.

[62] T H Cormen, C E Leiserson, R L Rivest, and C Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw–Hill, second edition, 2001.

[63] G Cormode and S Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1), 2007.

[64] M Crochemore, C Hancart, and T Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[65] M Crochemore, C S Iliopoulos, Y J Pinzon, and J F Reid. A fast and practical bit-vector algorithm for the Longest Common Subsequence problem. *Information Processing Letters*, 80:279—-285, 2001.

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

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

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

[69] Maxime Crochemore and Ely Porat. Fast computation of a longest increasing subsequence and application. *Information and Computation*, 208:1054—-1059, 2010.

[70] 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.

[71] Nathaniel J. Davies, Peter Krusche, Eran Tauber, and Sascha Ott. Analysis of 5' gene regions reveals extraordinary conservation of novel non-coding sequences in a wide range of animals. *BMC Evolutionary Biology*, 15(1):227, 2015.

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

[73] Lokenath Debnath and Dambaru Bhatta. *Integral Transforms and Their Applications*. Chapman and Hall/CRC, 2014.

[74] 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.

[75] Bangming Deng, Jie Du, Brian Parshall, and Jianpan Wang. *Finite dimensional algebras and quantum groups*. Number 150 in Mathematical Surveys and Monographs. American Mathematical Society, 2008.

[76] T Denton, F Hivert, A Schilling, and N M Thiéry. On the representation theory of finite J-trivial monoids. *Séminaire Lotharingien de Combinatoire*, 64(B64d), 2011.

[77] S Deorowicz. An algorithm for solving the longest increasing circular subsequence problem. *Information Processing Letters*, 109:630—-634, 2009.

[78] S Deorowicz. A cover-merging-based algorithm for the longest increasing subsequence in a sliding window problem. *Computers and Informatics*, 31:1217—-1233, 2012.

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

[80] Brian Drake, Sean Gerrish, and Mark Skandera. Two New Criteria for Comparison in the Bruhat Order. *The Electronic Journal of Combinatorics*, 11(1), mar 2004.

[81] A. Dress, J.H. Koolen, and V. Moulton. On Line Arrangements in the Hyperbolic Plane. *European Journal of Combinatorics*, 23(5):549—-557, 2002.

[82] C Ehresmann. Sur la topologie de certains espaces homogènes. *Annals of Mathematics*, 35(2):396–443, 1934.

[83] David Eisenstat and Philip N. Klein. Linear-time algorithms for max flow and multiple-source shortest paths in unit-weight planar graphs. In *Proceedings of STOC*, pages 735—-744, 2013.

[84] P Erdös and G Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

[85] E S Esyp, I V Kazachkov, and V N Remeslennikov. Divisibility Theory and Complexity of Algorithms in Free Partially Commutative Groups. In *Groups, Languages, Algorithms*, volume 378 of *Contemporary Mathematics*, pages 319–348. American Mathematical Society, 2005.

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

[87] M Federico, P Peterlongo, N Pisanti, and M.-F. Sagot. RIME: Repeat identification. *Discrete Applied Mathematics*, 163(3):275–286, 2013.

[88] Stefan Felsner and Lorenz Wernisch. Maximum k-Chains in Planar Point Sets: Combinatorial Structure and Algorithms. *SIAM Journal on Computing*, 28:192—-209, jan 1998.

[89] 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.

[90] 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*, 45:11—-18, 1993.

[91] Sergey Fomin and Curtis Greene. Noncommutative Schur functions and their applications. *Discrete Mathematics*, 193(1-3):179–200, nov 1998.

[92] Michael L Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29—-35, 1975.

[93] H N Gabow and R E Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[94] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, sep 1991.

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

[96] P Gawrychowski. Private communication.

[97] Paweł Gawrychowski. Faster algorithm for computing the edit distance between SLP-compressed strings. In *Lecture Notes in Computer Science*, volume 7608 of *Lecture Notes in Computer Science*, pages 229–236, 2012.

[98] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Improved Submatrix Maximum Queries in Monge Matrices. In *Proceedings of ICALP*, volume 8572 of *Lecture Notes in Computer Science*, pages 525–537, 2014.

[99] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Submatrix Maximum Queries in Monge Matrices Are Equivalent to Predecessor Search. In *Proceedings of ICALP*, volume 9134 of *Lecture Notes in Computer Science*, pages 580–592, 2015.

[100] 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*, 93:9061—-9066, 1996.

[101] 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.

[102] A J Gibbs and G A McIntyre. The Diagram, a Method for Comparing Sequences. *European Journal of Biochemistry*, 16(1):1–11, 1970.

[103] A Gogol-Döring and K Reinert. *Biological sequence analysis using the SeqAn C++ library*. Chapman & Hall / CRC Mathematical and Computational Biology Series. CRC Press, 2010.

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

[105] M Gondran and M Minoux. *Graphs, Dioids and Semirings*, volume 47 of *Operations Research/Computer Science Interfaces Series*. Springer, 2008.

[106] M. A. Gorsky. Subword Complexes and Nil-Hecke Moves. *Model. Anal. Inform. Sist.*, 20(6):121—-128, 2013.

[107] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[108] D.Yu. Grigor'ev. Additive complexity in directed computations. *Theoretical Computer Science*, 19(1):39–67, jul 1982.

[109] D. Gusfield, K. Balasubramanian, and D. Naor. Parametric optimization of sequence alignment. *Algorithmica*, 12(4-5):312–326, 1994.

[110] Adam Hammett and Boris Pittel. How often are two permutations comparable? *Transactions of the American Mathematical Society*, 360(09):4541–4568, apr 2008.

[111] D S Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[112] D S Hirschberg. Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM*, 24(4):664–675, 1977.

[113] W J Hsu and M W Du. New algorithms for the LCS problem. *Journal of Computer and System Sciences*, 29:133–152, 1984.

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

[115] Yue Huang and Ling Zhang. Rapid and sensitive dot-matrix methods for genome analysis. *Bioinformatics*, 20(4):460–466, 2004.

[116] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350—-353, 1977.

[117] H Hyyrö. Bit-Parallel LCS-length Computation Revisited. In *Proceedings of AWOCA*, 2004.

[118] H Hyyrö, K Fredriksson, and G Navarro. Increased Bit-Parallelism for Approximate String Matching. In *Proceedings of WEA*, volume 3059 of *Lecture Notes in Computer Science*, pages 285—-298, 2004.

[119] 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.

[120] G F Italiano and R Raman. Topics in Data Structures. In M J Atallah and M Blanton, editors, *Algorithms and Theory of Computation Handbook*, volume 1, chapter 5, pages 5–29. Chapman and Hall/CRC, second edition, 2010.

[121] 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. Chapman and Hall/CRC, 2006.

[122] J JáJá, C Mortensen, and Q Shi. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In *Proceedings of ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558—-568, 2004.

[123] Tao Jiang, Guohui Lin, Bin Ma, and Kaizhong Zhang. The longest common subsequence problem for arc-annotated sequences. *Journal of Discrete Algorithms*, 2(2):257–270, jun 2004.

[124] C R Johnson and S Nasserasr. TP_2 = Bruhat. *Discrete Mathematics*, 310(10–11):1627–1628, 2010.

[125] N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms*. Computational Molecular Biology. The MIT Press, 2004.

[126] Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix Maximum Queries in Monge Matrices and Partial Monge Matrices, and Their Applications. *ACM Transactions on Algorithms*, 13(2):1—-42, 2017.

[127] Juha Kärkkäinen, Gonzalo Navarro, and Esko Ukkonen. Approximate string matching on Ziv–Lempel compressed text. *Journal of Discrete Algorithms*, 1(3-4):313–338, jun 2003.

[128] A Karzanov. Combinatorial methods to solve cut-determined multiflow problems. In *Combinatorial Methods for Flow Problems*, volume 3, pages 6–69. VNIISI, 1979.

[129] C Kassel and V Turaev. *Braid Groups*, volume 247 of *Graduate Texts in Mathematics*. Springer, 2008.

[130] Toby Kenney. Coxeter groups, Coxeter monoids and the Bruhat order. *Journal of Algebraic Combinatorics*, 39(3):719–731, may 2014.

[131] Carmel Kent, Gad M. Landau, and Michal Ziv-Ukelson. On the Complexity of Sparse Exon Assembly. *Journal of Computational Biology*, 13:1013—-1027, jun 2006.

[132] 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*, 298:253—-272, 2003.

[133] P N Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of SODA*, pages 146—-155, 2005.

[134] D E Knuth. Permutations, matrices, and generalized Young tableaux. *Pacific Journal of Mathematics*, 34(3):709–727, 1970.

[135] D E Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1998.

[136] 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.

[137] J Krumsiek, R Arnold, and T Rattei. Gepard: A rapid and sensitive tool for creating dotplots on genome scale. *Bioinformatics*, 23:1026—-1028, 2007.

[138] P. Krusche. *Parallel String Alignments: Algorithms and Applications*. PhD thesis, University of Warwick, 2011.

[139] P. Krusche and A. Tiskin. Efficient parallel string comparison. In *Proceedings of ParCo*, volume 38 of *NIC Series*, pages 193–200. John von Neumann Institute for Computing, 2007.

[140] Peter Krusche and Alexander Tiskin. String comparison by transposition networks. In *London Algorithmics 2008 Theory and Practice*, volume 11 of *Texts in Algorithmics*. College Publications, 2009.

[141] Peter Krusche and Alexander Tiskin. Computing alignment plots efficiently. In *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 158–165. 2010.

[142] Peter Krusche and Alexander Tiskin. New algorithms for efficient parallel string comparison. In *Proceedings of SPAA*, pages 209–216, 2010.

[143] G M Landau, E Myers, and M Ziv-Ukelson. Two algorithms for LCS consecutive suffix alignment. *Journal of Computer and System Sciences*, 73:1095—-1117, 2007.

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

[145] G M Landau and U Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[146] G M Landau and M Ziv-Ukelson. On the common substring alignment problem. *Journal of Algorithms*, 41:338—-359, 2001.

[147] V Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8—-17, 1965.

[148] Y Lifshits. Processing Compressed Texts: A Tractability Border. In *Proceedings of CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 228—-240, 2007.

[149] Yury Lifshits and Markus Lohrey. Querying and Embedding Compressed Texts. In *Proceedings of MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 681—-692, 2006.

[150] M Lohrey. Algorithmics on SLP-compressed strings: a survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

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

[152] 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*, 78:7665—-7669, 1981.

[153] Satya N. Majumdar and Sergei Nechaev. Exact asymptotic results for the Bernoulli matching model of sequence alignment. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 72(2):020901, 2005.

[154] A Marzal and S Barrachina. Speeding up the computation of the edit distance for cyclic strings. In *Proceedings of ICPR, part 2*, pages 891—-894, 2000.

[155] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[156] S Masuda, K Nakajima, T Kashiwabara, and T Fujisawa. Efficient Algorithms for Finding Maximum Cliques of an Overlap Graph. *Networks*, 20:157–171, 1990.

[157] Ury Matarazzo, Dekel Tsur, and Michal Ziv-Ukelson. Efficient all path score computations on grid graphs. *Theoretical Computer Science*, 525:138–149, 2014.

[158] Volodymyr Mazorchuk and Benjamin Steinberg. Double Catalan monoids. *Journal of Algebraic Combinatorics*, 36(3):333–354, nov 2012.

[159] W F McColl. Scalable Computing. In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. 1995.

[160] W F McColl. Universal Computing. In L Bougé and Others, editors, *Proceedings of Euro-Par (Part I)*, volume 1123 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 1996.

[161] 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.

[162] E W Myers and W Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[163] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251—-266, 1986.

[164] G Myers. Approximately matching context-free languages. *Information Processing Letters*, 54:85–92, 1995.

[165] G Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[166] G Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[167] Saul B. Needleman and Christian 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.

[168] A. D. Neverov, A. A. Mironov, and M. S. Gelfand. Spliced Alignment and Similarity-based Gene Recognition. In *Handbook of Computational Molecular Biology*, Chapman and Hall/CRC Computer and Information Science Series, chapter 2. Chapman and Hall/CRC, 2006.

[169] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[170] M S Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5(1):75–92, 1990.

[171] Mike Paterson and Vlado Dančík. Longest common subsequences. In *Proceedings of MFCS*, volume 841 of *Lecture Notes in Computer Science*, pages 127—-142, 1994.

[172] Emma Picot, Peter Krusche, Alexander Tiskin, Isabelle Carré, and Sascha Ott. Evolutionary analysis of regulatory sequences (EARS) in plants. *The Plant Journal*, 64(1):165–176, 2010.

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

[174] Robert A Proctor. Classical Bruhat orders and lexicographic shellability. *Journal of Algebra*, 77(1):104–126, jul 1982.

[175] 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.

[176] K R Rasmussen, J Stoye, and E W Myers. Efficient q-Gram Filters for Finding All ϵ-Matches over a Given Length. *Journal of Computational Biology*, 13(2):296–308, 2006.

[177] P Rice, I Longden, and A Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, 2000.

[178] S. V. Rice, H. Bunke, and T. A. Nartker. Classes of Cost Functions for String Edit Distance. *Algorithmica*, 18(2):271–280, 1997.

[179] R.W. Richardson and T.A. Springer. The Bruhat order on symmetric varieties. *Geometriae Dedicata*, 35(1-3):389–436, 1990.

[180] C Rick. Simple and fast linear space computation of longest common subsequences. *Information Processing Letters*, 75:275—-281, 2000.

[181] G de B Robinson. On the representations of the symmetric group. *American Journal of Mathematics*, 60:745—-760, 1938.

[182] G Rote. Path Problems in Graphs. *Computing Supplementum*, 7:155–189, 1990.

[183] D. Rotem and J. Urrutia. Finding maximum cliques in circle graphs. *Networks*, 11(3):269–278, 1981.

[184] W Rytter. Algorithms on Compressed Strings and Arrays. In *Proceeedings of SOFSEM*, volume 1725 of *Lecture notes in Computer Science*, pages 48–65, 1999.

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

[186] Bruce Sagan. *The Symmetric Group*. Springer, 2001.

[187] Y Sakai. An Almost Quadratic Time Algorithm for Sparse Spliced Alignment. *Theory of Computing Systems*, 48(1):189–210, 2011.

[188] Yoshifumi Sakai. A fast algorithm for multiplying min-sum permutations. *Discrete Applied Mathematics*, 159:2175—-2183, 2011.

[189] Yoshifumi Sakai. A substring–substring LCS data structure. *Theoretical Computer Science*, 753(2):16–34, 2019.

[190] 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.

[191] J Seiferas. Sorting Networks of Logarithmic Depth, Further Simplified. *Algorithmica*, 53(3):374–384, 2009.

[192] Peter H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[193] H Shi and J Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[194] J. Sieckmann and P Szabó. A Noetherian and confluent rewrite system for idempotent semigroups. *Semigroup Forum*, 25:83—-110, 1982.

[195] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[196] 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.

[197] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. Dphil thesis, University of Oxford, 1998.

[198] A. Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158:759—-769, 2009.

[199] Alexander Tiskin. All-Pairs Shortest Paths Computation in the BSP Model. In *Proceedings of ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 178—-189, 2001.

[200] Alexander 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.

[201] Alexander 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 270–281, 2006.

[202] Alexander Tiskin. Communication-efficient parallel generic pairwise elimination. *Future Generation Computer Systems*, 23:179—-188, 2007.

[203] Alexander Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008.

[204] Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.

[205] Alexander Tiskin. Periodic String Comparison. In *Proceedings of CPM*, volume 5577 of *Lecture Notes in Computer Science*, pages 193–206, 2009.

[206] Alexander Tiskin. Parallel Selection by Regular Sampling. In *Proceedings of Euro-Par*, volume 6272 of *Lecture Notes in Computer Science*, pages 393—-399, 2010.

[207] Alexander Tiskin. Towards Approximate Matching in Compressed Strings: Local Subsequence Recognition. In *Proceedings of CSR*, volume 6651 of *Lecture Notes in Computer Science*, pages 401–414. 2011.

[208] Alexander Tiskin. Efficient high-similarity string comparison. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops on - EDBT '13*, page 358, New York, New York, USA, 2013. ACM Press.

[209] Alexander Tiskin. Fast Distance Multiplication of Unit-Monge Matrices. *Algorithmica*, 71:859–888, 2015.

[210] S.V. Tsaranov. Representation and Classification of Coxeter Monoids. *European Journal of Combinatorics*, 11(2):189–204, mar 1990.

[211] E Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.

[212] L G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[213] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, jun 1977.

[214] A M Vershik, S Nechaev, and R Bikbov. Statistical Properties of Locally Free Groups with Applications to Braid Groups and Growth of Random Heaps. *Communications in Mathematical Physics*, 212(2):469–501, 2000.

[215] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[216] Max Ward, Andrew Gozzard, and Amitava Datta. A Maximum Weight Clique Algorithm For Dense Circle Graphs With Many Shared Endpoints. *Journal of Graph Algorithms and Applications*, 21(4):547–554, 2017.

[217] T A Welch. A Technique for High-Performance Data Compression. *Computer*, 17:8—-19, 1984.

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

[219] S Wu, U Manber, G Myers, and W Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317—-323, 1990.

[220] T Yamamoto, H Bannai, S Inenaga, and M Takeda. Faster Subsequence and Don't-Care Pattern Matching on Compressed Texts. In *Proceedings of CPM*, volume 6661 of *Lecture Notes in Computer Science*, pages 309—-322, 2011.

[221] Yufei Zhao. On the Bruhat order of the symmetric group and its shellability. Technical report, 2007.

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

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