

Fast distance multiplication of unit-Monge matrices

Alexander Tiskin*

Abstract

Monge matrices play a fundamental role in optimisation theory, graph and string algorithms. Distance multiplication of two Monge matrices of size n can be performed in time $O(n^2)$. Motivated by applications to string algorithms, we introduced in previous works a subclass of Monge matrices, that we call *simple unit-Monge* matrices. We also gave a distance multiplication algorithm for such matrices, running in time $O(n^{1.5})$. Landau asked whether this problem can be solved in linear time. In the current work, we give an algorithm running in time $O(n \log n)$, thus approaching an answer to Landau's question within a logarithmic factor. The new algorithm implies immediate improvements in running time for a number of algorithms on strings and graphs. In particular, we obtain an algorithm for finding a maximum clique in a circle graph in time $O(n \log^2 n)$, and a surprisingly efficient algorithm for comparing compressed strings. We also point to potential applications in group theory, by making a connection between unit-Monge matrices and Coxeter monoids. We conclude that unit-Monge matrices are a fascinating object and a powerful tool, that deserves further study from both the mathematical and the algorithmic viewpoints.

1 Introduction

A matrix is called *Monge*, if its density matrix is nonnegative. Monge matrices play a fundamental role in optimisation theory, graph and string algorithms. Distance multiplication (also known as min-plus or tropical multiplication) of two Monge matrices of size n can be performed in time $O(n^2)$. Motivated by applications to string comparison, we introduced (using different terminology) in [33, 34] the following subclass of Monge matrices. A matrix is called *unit-Monge*, if its density matrix is a permutation matrix; we further restrict our attention to a subclass of *simple unit-Monge* matrices. In [33, 34], we gave an algorithm for distance multiplication of such matrices, running in time $O(n^{1.5})$. Landau [24] asked (again using different terminology)

whether this problem can be solved in linear time. In the current work, we give an algorithm for distance multiplication of simple unit-Monge matrices, running in time $O(n \log n)$, thus approaching an answer to Landau's question within a logarithmic factor.

Our study of unit-Monge matrices is motivated primarily by applications to string comparison and approximate pattern matching in strings. We presented a number of such algorithmic applications in [34]. Our new distance multiplication algorithm implies immediate improvements in running time for a number of string comparison and graph algorithms: semi-local longest common subsequences between permutations; longest increasing subsequence in a cyclic permutation; longest pattern-avoiding subsequence in a permutation; longest piecewise monotone subsequence; maximum clique in a circle graph; longest common subsequence between a grammar-compressed string and an uncompressed string; subsequence recognition in a grammar-compressed string. In the current work, we give a brief overview of these applications and the improvements brought about by faster distance multiplication of simple unit-Monge matrices.

Due to space constraints, we omit the detailed descriptions of algorithmic applications. These can be found in the full version of this work [36].

2 Terminology and notation

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

$$\{\dots, -2, -1, 0, 1, 2, \dots\}$$
$$\{\dots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots\}$$

For ease of reading, odd half-integer variables will be indicated by hats (e.g. \hat{i} , \hat{j}). We denote integer and odd half-integer *intervals* by

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

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

*Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom. Research supported by the Centre for Discrete Mathematics and Its Applications (DIMAP), University of Warwick.

We will make extensive use of matrices with integer elements, and with integer or odd half-integer indices. Given two index ranges I, J , it will be convenient to denote their Cartesian product by $(I | J)$. We extend this notation to Cartesian products of intervals:

$$\begin{aligned} [i_0 : i_1 | j_0 : j_1] &= ([i_0 : i_1] | [j_0 : j_1]) \\ \langle i_0 : i_1 | j_0 : j_1 \rangle &= (\langle i_0 : i_1 \rangle | \langle j_0 : j_1 \rangle) \end{aligned}$$

Given index ranges I, J , a *matrix over* $(I | J)$ is indexed by $i \in I, j \in J$. A matrix is *nonnegative*, if all its elements are nonnegative.

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

The matrices we consider can be *implicit*, i.e. represented by a compact data structure that allows access to every element in a specified (small, but not necessarily constant) time.

DEFINITION 2.1. Let D be a matrix over $\langle i_0 : i_1 | j_0 : j_1 \rangle$. Its distribution matrix D^Σ over $[i_0 : i_1 | j_0 : j_1]$ is defined by

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

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

DEFINITION 2.2. Let A be a matrix over $[i_0 : i_1 | j_0 : j_1]$. Its density matrix A^\square over $\langle i_0 : i_1 | j_0 : j_1 \rangle$ is defined by

$$\begin{aligned} A^\square(\hat{i}, \hat{j}) &= A(\hat{i} + \frac{1}{2}, \hat{j} - \frac{1}{2}) - A(\hat{i} - \frac{1}{2}, \hat{j} - \frac{1}{2}) - \\ &\quad A(\hat{i} + \frac{1}{2}, \hat{j} + \frac{1}{2}) + A(\hat{i} - \frac{1}{2}, \hat{j} + \frac{1}{2}) \end{aligned}$$

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

Note that for any matrix D as above, and for all \hat{i}, \hat{j} , we have $(D^\Sigma)^\square(\hat{i}, \hat{j}) = D(\hat{i}, \hat{j})$. Also, for any matrix A as above, and for all i, j , we have $(A^\square)^\Sigma(i, j) + b(i) + c(j) = A(i, j)$, where $b(i) = A(i, j_0), c(j) = A(i_1, j)$. An important special case is when b, c are both zero vectors.

DEFINITION 2.3. Matrix A will be called *simple*, if $(A^\square)^\Sigma = A$.

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

DEFINITION 2.4. Matrix A is called a *Monge matrix*, if

$$A(i, j) + A(i', j') \leq A(i, j') + A(i', j)$$

for all $i \leq i', j \leq j'$. Equivalently, matrix A is a *Monge matrix*, if A^\square is nonnegative.

DEFINITION 2.5. A permutation (respectively, subpermutation) matrix is a zero-one matrix containing exactly one (respectively, at most one) nonzero in every row and every column.

Typically, permutation and subpermutation matrices will be indexed by odd half-integers.

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

DEFINITION 2.6. A square matrix A is called a *unit-Monge matrix*, if A^\square is a permutation matrix.

By Definitions 2.4, 2.6, a unit-Monge matrix over any index range is Monge.

Matrices that are both simple and unit-Monge will be the main focus of this paper. Note that a square matrix A is simple unit-Monge, if and only if $A = P^\Sigma$, where P is a permutation matrix.

Example. The following matrix is simple unit-Monge:

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

THEOREM 2.1. Given a permutation matrix P of size n , and the value $P^\Sigma(i, j), i, j \in [0 : n]$, the values $P^\Sigma(i \pm 1, j), P^\Sigma(i, j \pm 1)$, where they exist, can be queried in time $O(1)$.

Proof. Straightforward from the definitions; see [36] for details.

3 Matrix distance multiplication

3.1 The algebraic structure We will make extensive use of the $(\min, +)$ -algebra on integer or real numbers, where the operators \min and $+$ play the role of addition and multiplication, respectively. This algebra is often called *distance* (or *tropical*) algebra.

DEFINITION 3.1. Let A, B, C be matrices over $[i_0 : i_1 \mid j_0 : j_1]$, $[j_0 : j_1 \mid k_0 : k_1]$, $[i_0 : i_1 \mid k_0 : k_1]$ respectively. The matrix distance product $A \odot B = C$ is defined by

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

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

Like any multiplication of matrices over a semiring, matrix distance multiplication is associative. The set of all square matrices with elements in $[0 : \infty]$ over a given index range forms a monoid with respect to distance multiplication.

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

THEOREM 3.1. Let A, B, C be matrices, such that $A \odot B = C$. If A, B are Monge (respectively, simple unit-Monge), then C is also Monge (respectively, simple unit-Monge).

Proof. Straightforward from the definitions; see [36] for details.

Since both these sets also contain a (different) identity element, they each form a submonoid in the multiplicative monoid of the distance semiring of integer matrices (where, in the case of Monge matrices, the range of matrix elements is extended by $+\infty$).

Theorem 3.1 gives us the basis for performing distance multiplication of simple unit-Monge matrices implicitly, by taking the density permutation matrices as input, and producing a density permutation matrix as output. It will be convenient to introduce special notation for implicit distance multiplication of this kind.

DEFINITION 3.2. Let P_A, P_B, P_C be permutation matrices. The implicit matrix distance product $P_A \square P_B = P_C$ is defined by $P_A^\Sigma \odot P_B^\Sigma = P_C^\Sigma$.

Example. In Figure 1, Subfigure 1a shows a triple of 6×6 permutation matrices P_A, P_B, P_C , with nonzeros indicated by green¹ bullets, such that $P_A \square P_B = P_C$.

Further understanding of the distance multiplication monoid of simple unit-Monge matrices can be gained by the following construction. Given a permutation matrix P over $(I \mid J)$, we represent the indices in sets I and J by nodes on two parallel lines, respecting the order of indices within each set. We represent

¹For colour illustrations, the reader is referred to the online version of this work. If the colour version is not available, all references to colour can be ignored.

every nonzero $P(\hat{i}, \hat{j}) = 1$ by connecting node $\hat{i} \in I$ with node $\hat{j} \in J$ by a continuous strictly monotone line called a *seaweed*. We call the resulting configuration a *seaweed braid*. Unless P is the identity matrix Id , some of the seaweeds in the seaweed braid will have to cross. However, due to monotonicity of seaweeds, no “unnecessary” crossings can occur; in other words, a given pair of seaweeds can only cross at most once.

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

Example. In Figure 2, Subfigure 1b shows the seaweed braids representing the implicit matrix distance product in Subfigure 1a.

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

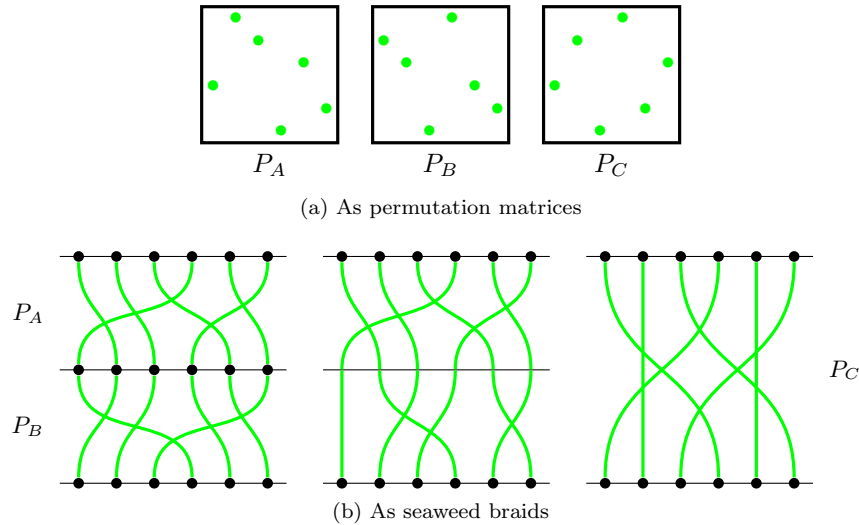


Figure 1: Implicit matrix distance product $P_A \square P_B = P_C$

of monoid \mathcal{T}_n consists of the *idempotence relations*

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

and the *braid relations*

$$\begin{aligned} g_t g_u &= g_u g_t & t, u \in [1 : n - 1], u - t \geq 2 \\ g_t g_u g_t &= g_u g_t g_u & t, u \in [1 : n - 1], u - t = 1 \end{aligned}$$

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

THEOREM 3.2. *The distance multiplication monoid of $n \times n$ simple unit-Monge matrices is isomorphic to the seaweed monoid \mathcal{T}_n .*

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

The classical *positive braid monoid* (see e.g. [22, Section 6.5]) on generators $id, g_1, g_2, \dots, g_{n-1}$, is

defined by the braid relations alone. Therefore, the seaweed monoid is isomorphic to the quotient of the positive braid monoid by the idempotence relations. A generalisation of the seaweed monoid is given by *Coxeter monoids*, which arise naturally as subgroup monoids in groups. The theory of Coxeter monoids can be traced back to Bourbaki [7], and has been developed by Tsaranov [37] and Richardson and Springer [29]. The contents of this and the following subsections can be regarded as the first step in the algorithmic study of Coxeter monoids.

3.2 The algorithm For generic matrices, direct application of Definition 3.1 gives an algorithm for matrix distance 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 [9], running in time $O\left(\frac{n^3(\log \log n)^3}{\log^2 n}\right)$.

For Monge matrices, distance multiplication can be easily performed in quadratic time, using the standard technique of Aggarwal et al. [1]. When matrices are represented explicitly, this running time is clearly optimal. However, for implicit simple unit-Monge matrices, the distance multiplication time can be reduced further. In [32, 34], we gave an algorithm running in time $O(n^{1.5})$. We now show that still further improvement is possible.

THEOREM 3.3. *Let P_A, P_B, P_C be $n \times n$ permutation matrices, such that $P_A \square P_B = P_C$. Given the nonzeros of P_A, P_B , the nonzeros of P_C can be computed in time $O(n \log n)$.*

Proof. Without loss of generality, let P_A, P_B, P_C be over $\langle 0 : n \mid 0 : n \rangle$. The algorithm is defined by recursion

on n .

Recursion base. If $n = 1$, the computation is trivial.

Recursive step. Assume without loss of generality that $n > 1$ is even. Informally, the idea is to split the range of index j in the definition of matrix distance product (Definition 3.1) into two subranges of size $\frac{n}{2}$. For each of these subranges of j , we use the sparsity of the input permutation matrix P_A (respectively, P_B) to partition the range of index i (respectively, k) into two disjoint, not necessarily contiguous, subsets of size $\frac{n}{2}$. We then call the algorithm recursively on the two resulting half-sized subproblems, and use the two returned half-sized permutation matrices to reconstruct the output permutation matrix P_C , relying on the Monge properties of the respective distribution matrices.

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

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

Let the first subproblem be

$$P_{A,lo} = P_A(* | 0 : \frac{n}{2}) \quad P_{B,lo} = P_B(0 : \frac{n}{2} | *)$$

$$P_{A,lo}^\Sigma \odot P_{B,lo}^\Sigma = P_{C,lo}^\Sigma$$

and the second subproblem be

$$P_{A,hi} = P_A(* | \frac{n}{2} : n) \quad P_{B,hi} = P_B(\frac{n}{2} : n | *)$$

$$P_{A,hi}^\Sigma \odot P_{B,hi}^\Sigma = P_{C,hi}^\Sigma$$

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

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

$$P_C^\Sigma(i, k) = \min_{j \in [0:n]} (P_A^\Sigma(i, j) + P_B^\Sigma(j, k)) =$$

$$\min \left(\min_{j \in [0:\frac{n}{2}]} (P_A^\Sigma(i, j) + P_B^\Sigma(j, k)), \right.$$

$$\left. \min_{j \in [\frac{n}{2}:n]} (P_A^\Sigma(i, j) + P_B^\Sigma(j, k)) \right)$$

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

$$\min_{j \in [0:\frac{n}{2}]} (P_A^\Sigma(i, j) + P_B^\Sigma(j, k)) =$$

$$\min_{j \in [0:\frac{n}{2}]} (P_{A,lo}^\Sigma(i, j) + P_{B,lo}^\Sigma(j, k) + P_{B,hi}^\Sigma(\frac{n}{2}, k)) =$$

$$\min_{j \in [0:\frac{n}{2}]} (P_{A,lo}^\Sigma(i, j) + P_{B,lo}^\Sigma(j, k)) + P_{B,hi}^\Sigma(\frac{n}{2}, k) =$$

$$P_{C,lo}^\Sigma(i, k) + P_{C,hi}^\Sigma(0, k)$$

The second argument can be rewritten analogously, so we have

$$P_C^\Sigma(i, k) = \min(P_{C,lo}^\Sigma(i, k) + P_{C,hi}^\Sigma(0, k),$$

$$P_{C,hi}^\Sigma(i, k) + P_{C,lo}^\Sigma(i, n))$$

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

$$\delta(i, k) =$$

$$(P_{C,lo}^\Sigma(i, k) + P_{C,hi}^\Sigma(0, k)) - (P_{C,hi}^\Sigma(i, k) + P_{C,lo}^\Sigma(i, n)) =$$

$$(P_{C,hi}^\Sigma(0, k) - P_{C,hi}^\Sigma(i, k)) - (P_{C,lo}^\Sigma(i, n) - P_{C,lo}^\Sigma(i, k)) =$$

$$\sum_{\hat{i} \in \langle 0:i \rangle, \hat{k} \in \langle 0:k \rangle} P_{C,hi}(\hat{i}, \hat{k}) - \sum_{\hat{i} \in \langle i:n \rangle, \hat{k} \in \langle k:n \rangle} P_{C,lo}(\hat{i}, \hat{k})$$

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

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

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

$$P_C^\Sigma(\hat{i} \pm \frac{1}{2}, \hat{k} \pm \frac{1}{2}) = P_{C,lo}^\Sigma(\hat{i} \pm \frac{1}{2}, \hat{k} \pm \frac{1}{2})$$

for all four sign choices made consistently on both sides of the equation. Hence, $P_C(\hat{i}, \hat{k}) = P_{C,lo}(\hat{i}, \hat{k})$.

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

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

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

Therefore,

$$(3.1) \quad P_C^\Sigma(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) = P_{C,hi}^\Sigma(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) <$$

$$P_{C,lo}^\Sigma(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2})$$

and, furthermore,

$$(3.2) \quad P_C^\Sigma(\hat{i} \pm \frac{1}{2}, \hat{k} \pm \frac{1}{2}) = P_{C,lo}^\Sigma(\hat{i} \pm \frac{1}{2}, \hat{k} \pm \frac{1}{2})$$

for the remaining three sign combinations chosen consistently on both sides of the equation. By Definitions 2.1, 2.2, we have

$$\begin{aligned} P_C(\hat{i}, \hat{k}) &= P_C^\Sigma(\dots) && - P_C^\Sigma(\dots) \\ &- P_C^\Sigma(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) && + P_C^\Sigma(\dots) \\ P_{C,lo}(\hat{i}, \hat{k}) &= P_{C,lo}^\Sigma(\dots) && - P_{C,lo}^\Sigma(\dots) \\ &- P_{C,lo}^\Sigma(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) && + P_{C,lo}^\Sigma(\dots) \end{aligned}$$

By (3.1), we have a strict inequality between the two additive terms fully shown above, and by (3.2), the abbreviated additive terms are pairwise equal between the two expressions. Hence, $P_C(\hat{i}, \hat{k}) > P_{C,lo}(\hat{i}, \hat{k})$. Since both $P_C, P_{C,lo}$ are zero-one matrices, this implies that $P_C(\hat{i}, \hat{k}) = 1$ and $P_{C,lo}(\hat{i}, \hat{k}) = 0$ (symmetrically, also $P_{C,hi}(\hat{i}, \hat{k}) = 0$).

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

$$(3.3) \quad P_{C,lo}(\hat{i}, \hat{k}) = 1 \text{ and } \delta(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) \leq 0$$

$$(3.4) \quad P_{C,hi}(\hat{i}, \hat{k}) = 1 \text{ and } \delta(\hat{i} - \frac{1}{2}, \hat{k} - \frac{1}{2}) \geq 0$$

$$(3.5) \quad \delta(\hat{i} - \frac{1}{2}, \hat{k} - \frac{1}{2}) < 0 \text{ and } \delta(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) > 0$$

In order to perform the checks (3.3)–(3.5) efficiently, it is sufficient to find for each $d \in [-n+1 : n-1]$ a value $r(d) \in [1 : 2n-1]$, such that $r(d) + d$ is odd, and

$$\begin{aligned} \delta(i, k) &\leq 0 && \text{if } i + k < r(k - i) \\ \delta(i, k) &\geq 0 && \text{if } i + k > r(k - i) \end{aligned}$$

for all $i, k \in [0 : n]$ (note that the above list of two cases is exhaustive, since $r(k-i) - (i+k) = r(k-i) + (k-i) - 2k$ must be odd, and therefore $i + k \neq r(k - i)$). Such a value $r(d)$ is guaranteed to exist by the monotonicity of function δ . Furthermore, values $r(d)$ can be chosen so that $|r(\hat{d} + \frac{1}{2}) - r(\hat{d} - \frac{1}{2})| = 1$ for all $\hat{d} \in \langle -n+1 : n-1 \rangle$. Informally, array r defines a monotone rectilinear path, consisting of points $(\frac{r(d)-d}{2}, \frac{r(d)+d}{2})$, from the bottom-left to the top-right corner of $\langle 0 : n \mid 0 : n \rangle$.

By definition of array r , for each d we have

$$\begin{aligned} w^-(d) &= \delta(\frac{r(d)-d-1}{2}, \frac{r(d)+d-1}{2}) \in [-1, 0] \\ w^+(d) &= \delta(\frac{r(d)-d+1}{2}, \frac{r(d)+d+1}{2}) \in [0, 1] \end{aligned}$$

We call the values $w^-(d), w^+(d)$ *witnesses* for $r(d)$.

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

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

Now assume that for a current value of d , we have the value $r(d)$, and the witnesses $w^-(d), w^+(d)$. Our next goal is to compute $r(d+1)$, along with its two witnesses. Let

$$w^* = \delta(\frac{r(d)-d-1}{2}, \frac{r(d)+d+1}{2}) \in [-1 : 1]$$

Value w^* can be obtained from either $w^-(d)$ or $w^+(d)$ by Theorem 2.1 in time $O(1)$. We now let

$$r(d+1) = r(d) + \begin{cases} 1 & \text{if } w^* \in [-1 : 0] \\ -1 & \text{if } w^* \in [0 : 1] \end{cases}$$

If $w^* = 0$, then the choice between 1 and -1 is made arbitrarily. Following this choice, we obtain the new witnesses as

$$\begin{aligned} w^-(d+1) &= \begin{cases} \delta(\frac{r(d)-d-3}{2}, \frac{r(d)+d-1}{2}) & \text{if } w^* \in [-1 : 0] \\ w^* & \text{if } w^* \in [0 : 1] \end{cases} \\ w^+(d+1) &= \begin{cases} w^* & \text{if } w^* \in [-1 : 0] \\ \delta(\frac{r(d)-d+1}{2}, \frac{r(d)+d+3}{2}) & \text{if } w^* \in [0 : 1] \end{cases} \end{aligned}$$

In each case, the value for the new witness can be obtained from respectively $w^-(d), w^+(d)$ by Theorem 2.1 in time $O(1)$. If $w^* = 0$, then the choices are made consistently with the arbitrary choice made in the definition of $r(d+1)$.

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

Given arrays r, w^-, w^+ , conditions (3.3)–(3.5) can now be expressed as follows:

$$(3.6) \quad P_{C,lo}(\hat{i}, \hat{k}) = 1 \text{ and } \hat{i} + \hat{k} < r(\hat{k} - \hat{i})$$

$$(3.7) \quad P_{C,hi}(\hat{i}, \hat{k}) = 1 \text{ and } \hat{i} + \hat{k} > r(\hat{k} - \hat{i})$$

$$(3.8) \quad \begin{aligned} &\hat{i} + \hat{k} = r(\hat{k} - \hat{i}) \text{ and} \\ &w^-(\hat{k} - \hat{i}) = -1 \text{ and } w^+(\hat{k} - \hat{i}) = 1 \end{aligned}$$

The nonzeros of P_C satisfying either of the conditions (3.6), (3.7) can be found in time $O(n)$ by checking directly each of the nonzeros in matrices $P_{C,lo}$ and $P_{C,hi}$. The nonzeros of P_C satisfying condition (3.8) can be found in time $O(n)$ by a linear sweep of the values $r(d)$ for all $d \in [-n+1 : n-1]$. For each d , we let $\hat{i} = \frac{r(d)+d}{2}, \hat{k} = \frac{r(d)-d}{2}$, and substitute these values into (3.8). We have now obtained all the nonzeros of matrix P_C .

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 $O(n \log n)$.

Example. Figure 2 illustrates the proof of Theorem 3.3 on a problem instance with a solution generated by the Mathematica 7 software. Subfigure 2a shows a pair of input 20×20 permutation matrices P_A, P_B , with nonzeros indicated by green bullets. Subfigure 2b 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 red crosses and blue diamonds. Subfigure 2c shows a recursive step. The boundaries separating sets $\delta^{-1}([-10 : -1])$, $\delta^{-1}(\{0\})$, $\delta^{-1}([1 : 10])$ are indicated by the red and the blue line. Function r corresponds to an arbitrary monotone rectilinear path within $\delta^{-1}(\{0\})$, inclusive of the boundaries. In particular, either of the boundaries itself can be taken to define r . The nonzeros in the output matrix P_C satisfying (3.6), (3.7), (3.8) are indicated respectively by red crosses, blue diamonds and green bullets; note that overall, there are 20 such nonzeros, and that they define a permutation matrix.

4 Algorithmic applications

The longest common subsequence (LCS) problem is a classical problem in computer science. Given two strings a, b of lengths m, n respectively, the LCS problem asks for the length of the longest string that is a subsequence of both a and b . This length is called the strings' LCS score. We refer the reader to monographs [10, 17] for the background and further references.

The semi-local LCS problem is a generalisation of the LCS problem, arising naturally in the context of string comparison. Given two strings a, b as before, the semi-local LCS problem asks for the LCS score of each string against all substrings of the other string, and of all prefixes of each string against all suffixes of the other string.

In [33, 34], we introduced the semi-local LCS problem, described its connections with unit-Monge matrices, and a number of its algorithmic applications. Many of these applications use distance multiplication of simple unit-Monge matrices as a subroutine. In these cases, we can immediately obtain improved algorithms by plugging in the new multiplication algorithm given by Theorem 3.3. Due to space limitations, we cannot give here a detailed description of these improvements. Instead, we give their brief overview, and refer the reader to [36] for more details.

4.1 Semi-local comparison of permutations An important special case of string comparison is where each of the input strings a, b is a *permutation string*, i.e. a string that consists of all distinct characters. Without loss of generality, we may assume that $m = n$, and that both strings are permutations of a given totally ordered alphabet of size n . The semi-local LCS problem on permutations is equivalent to finding the length of the longest increasing subsequence (LIS) in every substring of a given permutation string.

In [34], we gave an algorithm for the semi-local LCS problem on permutation strings, running in time $O(n^{1.5})$. After plugging in the algorithm of Theorem 3.3, the running time of this algorithm improves to $O(n \log^2 n)$.

The cyclic LCS problem on permutation strings is equivalent to the LIS problem on a circular string. This problem has been considered by Albert et al. [3], who gave a Monte Carlo randomised algorithm, running in time $O(n^{1.5} \log n)$ with small error probability. Our new method improves the running time to deterministic $O(n \log^2 n)$.

Let a, b be permutation strings, both of length n , but generally over different alphabets. Strings a, b are *isomorphic*, if they have the same relative order of characters, i.e. $\alpha_i < \alpha_j$ iff $\beta_i < \beta_j$ for all i, j . Given a target permutation string T of length n and a set of *antipattern* permutation strings Y , the *longest Y -avoiding subsequence problem* is to find the longest subsequence of T that *does not* contain a subsequence isomorphic to any string in Y . For example, the LIS problem on a permutation string can be interpreted as the longest $\{“21”\}$ -avoiding subsequence problem. For a detailed introduction into these problems and their connections, see the work by Albert et al. [2] 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. [2] 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 [2] run in polynomial time, ranging from $O(n \log n)$ to $O(n^5)$. Two particular antipattern sets considered in [2] are (in that work's original notation) $C_3 = \{“132”, “213”, “321”\}$, $C_4 = \{“132”, “213”, “312”\}$. For both these antipattern sets, algorithms given in [2] run in time $O(n^2 \log n)$. Our new method improves the running time to $O(n \log^2 n)$.

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

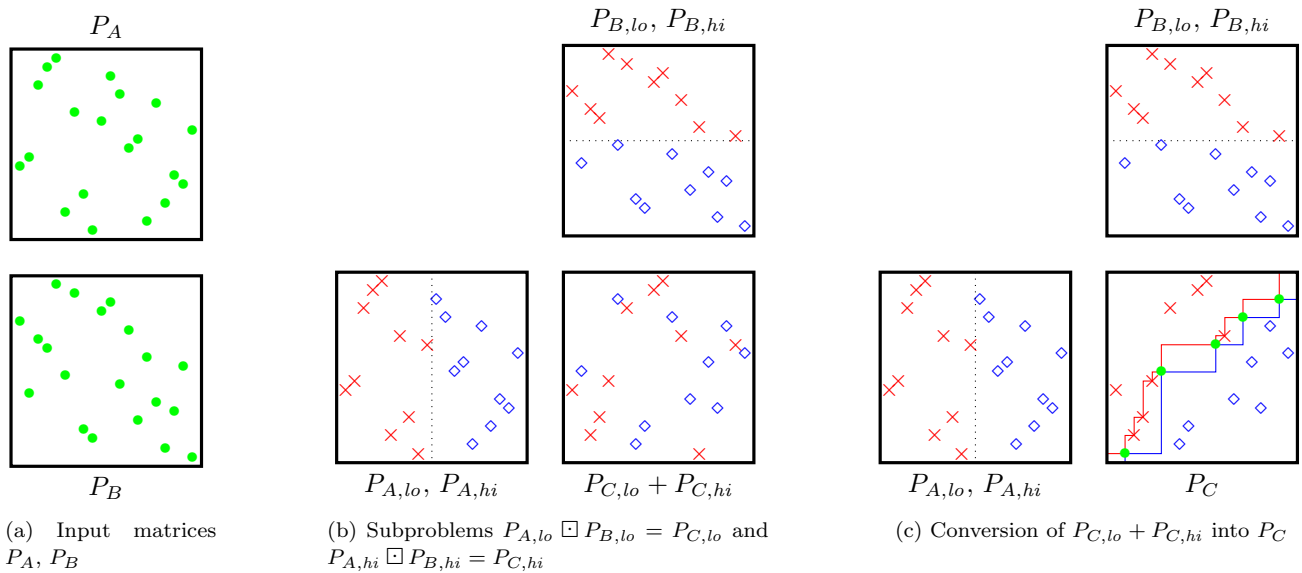


Figure 2: Proof of Theorem 3.3: $P_A \square P_B = P_C$

constant number of monotone pieces. In particular, given a permutation string a of size n , the *longest k -increasing subsequence* (respectively, *longest k -modal subsequence*) problem is to find the longest subsequence in a that is a concatenation of at most k sequences, all of which are increasing (respectively, alternate between increasing and decreasing). Using standard sparse LCS algorithms [20, 6], such an instance of the LCS problem can be solved in time $O(nk \log n)$. Demange et al. [11] gave a similar algorithm for the longest k -modal subsequence problem, also running in time $O(nk \log n)$. Our new method gives algorithms for both the k -increasing and k -modal subsequence problems, running in time $O(n \log^2 n)$. This is an improvement, as long as $k = \omega(\log n)$.

A *circle graph* [14, 16] is defined as the intersection graph of a set of chords in a circle, i.e. the graph where each node represents a chord, and two nodes are adjacent, whenever the corresponding chords intersect. We consider the maximum clique problem on a circle graph. The *interval model* of a circle graph is obtained by cutting the circle at an arbitrary point and laying it out on a line, so that the chords become (closed) intervals. The original circle graph is isomorphic to the overlap graph of its interval model, i.e. the graph where each node represents an interval, and two nodes are adjacent, whenever the corresponding intervals intersect but do not contain one another.

It has long been known that the maximum clique problem in a circle graph on n nodes is solvable in polynomial time [15]. A number of algorithms have

been proposed in [30, 19, 27, 5]; the problem has also been studied in the context of line arrangements in the hyperbolic plane [21, 13]. 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 [34], we gave an algorithm running in time $O(n^{1.5})$. Our new method improves the running time to $O(n \log^2 n)$.

4.2 Compressed string comparison Algorithms on compressed strings are an area of steadily increasing importance in algorithm theory. It has long been known that certain algorithmic problems can be solved directly on a compressed string, without first decompressing it. Early examples of such algorithms were given e.g. by Amir et al. [4] and by Rytter [31].

Let t be a string of length n (typically large). We call t a *grammar-compressed string* (*GC-string*), when it is represented implicitly by a special type of context-free grammar, called a *straight-line program* (*SLP*). An SLP of length \bar{n} , $\bar{n} \leq n$, is a sequence of \bar{n} *statements*. A statement numbered k , $1 \leq k \leq \bar{n}$, has one of the following forms:

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

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

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

classical LZ78 and LZW compression schemes by Ziv, Lempel and Welch [39, 38].

The LCS problem on two GC-strings has been considered by Lifshits and Lohrey [26], and proven to be NP-hard.

We consider the LCS problem on two input strings, one of which is a GC-string and the other a plain string. This problem can be regarded as a special case of computing the edit distance between a context-free language given by a grammar of size \bar{n} , and a string of size m . For this more general problem, Myers [28] gave an algorithm running in time $O(m^3\bar{n} + m^2 \cdot \bar{n} \log \bar{n})$. In [35], we gave an algorithm for another generalisation of the LCS problem, running in time $O(m^{1.5}\bar{n})$. Lifshits [25] asked whether the LCS problem can be solved in time $O(m\bar{n})$. Our new method gives an algorithm for the LCS problem between a GC-string and an uncompressed string running in time $O(m \log m \cdot \bar{n})$, thus approaching an answer to Lifshits' question within a logarithmic factor.

Hermelin et al. [18] gave a more detailed analysis of this problem's complexity, by considering the weighted alignment problem on a pair of GC-strings a, b of total compressed length $\bar{r} = \bar{m} + \bar{n}$, parameterised by the strings' total plain length $r = m + n$. They gave an algorithm running in time $O(r^{1.34}\bar{r}^{1.34})$ for general weights, and in time $O(r^{1.2}\bar{r}^{1.4})$ for rational weights. In the case of rational weights, the parameterised running time of weighted GC-string alignment can be improved by the following straightforward algorithm. First, we uncompress one of the input strings — say, string a . Then, we solve the LCS problem on the plain version of string a against the GC-string b . The resulting running time is $O(m \log m \cdot \bar{n}) = O(r \log r \cdot \bar{r})$.

The *minimal-window subsequence recognition problem* (the counting version) consists in counting all inclusion-minimal substrings in a text string of length n , containing a pattern string of length m as a subsequence. Cégielski et al. [8] give an algorithm for this problem on a GC-text and an uncompressed pattern, running in time $O(m^2 \log m \cdot \bar{n})$. Our new method improves the running time to $O(m \log m \cdot \bar{n})$. The same improvement is achieved for the problem variant with windows of fixed length.

5 Conclusion

In this work, we have given a fast algorithm for distance multiplication of simple unit-Monge matrices, running in time $O(n \log n)$. The only known lower bound is trivial $\Omega(n)$. Therefore, Landau's question whether the problem can be solved in time $O(n)$ is still open, although we have now approached an answer within a logarithmic factor.

Our approach unifies and gives improved solutions to a number of algorithmic problems. It is likely that the scope of the applications can be widened even further, e.g. by considering new kinds of approximate matching and approximate repeat problems in strings.

The algebraic structure underlying our method is the distance-multiplication monoid of simple unit-Monge matrices, which we call the seaweed monoid. As a special case of Coxeter monoids, this structure certainly merits further study. There are likely deeper connections with semigroup and group theories, beginning with an algorithmic study of the general Coxeter monoids and Coxeter groups. One may also expect connections with tropical mathematics, e.g. the tropical rank theory (see e.g. [12]).

In [23], we used distance multiplication of simple unit-Monge matrices to obtain the first parallel LCS algorithm with scalable communication. It would be interesting to see whether the fast distance multiplication algorithm given in the current work can be efficiently parallelised, and whether this can be used to achieve further improvement in the communication efficiency of parallel LCS computation.

References

- [1] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987.
- [2] 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.
- [3] M. H. Albert, M. D. Atkinson, D. Nussbaum, J.-R. Sack, and N. Santoro. On the longest increasing subsequence of a circular list. *Information Processing Letters*, 101:55–59, 2007.
- [4] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.
- [5] A. Apostolico, M. J. Atallah, and S. E. Hambrusch. New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 36:1–24, 1992.
- [6] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987.
- [7] N. Bourbaki. *Groupes et algèbres de Lie. Chapitres 4, 5 et 6*. Hermann, 1968.
- [8] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 127–136, 2006.
- [9] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the 39th ACM STOC*, pages 590–598, 2007.

- [10] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
- [11] 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.
- [12] M. Develin, F. Santos, and B. Sturmfels. On the rank of a tropical matrix. In *Combinatorial and computational geometry*, volume 52 of *MSRI Publications*, pages 213–242. Cambridge University Press, 2005.
- [13] A. Dress, J. H. Koolena, and V. Moulton. On line arrangements in the hyperbolic plane. *European Journal of Combinatorics*, 23(5):549–557, 2002.
- [14] S. Even and A. Itai. Queues, stacks and graphs. In *Theory of Machines and Computations*, pages 71–86. Academic Press, 1971.
- [15] F. Gavril. Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3:261–273, 1973.
- [16] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Number 57 in *Annals of Discrete Mathematics*. Elsevier, second edition, 2004.
- [17] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [18] D. Hermelin, G. M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *Proceedings of the 26th STACS*, pages 529–540, 2009.
- [19] W.-L. Hsu. Maximum weight clique algorithms for circular-arc graphs and circle graphs. *SIAM Journal on Computing*, 14(1):224–231, 1985.
- [20] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.
- [21] A. Karzanov. Combinatorial methods to solve cut-determined multi-flow problems. In *Combinatorial Methods for Flow Problems*, volume 3, pages 6–69. VNIISI, 1979. In Russian.
- [22] C. Kassel and V. Turaev. *Braid Groups*, volume 247 of *Graduate Texts in Mathematics*. Springer, 2008.
- [23] 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.
- [24] G. Landau. Can DIST tables be merged in linear time? An open problem. In *Proceedings of the Prague Stringology Conference*, page 1. Czech Technical University in Prague, 2006.
- [25] Y. Lifshits. Processing compressed texts: A tractability border. In *Proceedings of CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 228–240, 2007.
- [26] Y. Lifshits and M. Lohrey. Querying and embedding compressed texts. In *Proceedings of MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 681–692, 2006.
- [27] S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Efficient algorithms for finding maximum cliques of an overlap graph. *Networks*, 20:157–171, 1990.
- [28] G. Myers. Approximately matching context-free languages. *Information Processing Letters*, 54:85–92, 1995.
- [29] R. W. Richardson and T. A. Springer. The Bruhat order on symmetric varieties. *Geometriae Dedicata*, 35(1–3):389–436, 1990.
- [30] D. Rotem and J. Urrutia. Finding maximum cliques in circle graphs. *Networks*, 11:269–278, 1981.
- [31] W. Rytter. Algorithms on compressed strings and arrays. In *Proceedings of SOFSEM*, volume 1725 of *Lecture notes in Computer Science*, pages 48–65, 1999.
- [32] A. Tiskin. All semi-local longest common subsequences in subquadratic time. In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 352–363, 2006.
- [33] A. Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008.
- [34] A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.
- [35] A. Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158(5):759–769, 2009.
- [36] A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. Technical Report 0707.3619, arXiv, 2009.
- [37] S. Tsaranov. Representation and classification of Coxeter monoids. *European Journal of Combinatorics*, 11(2):189–204, 1990.
- [38] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.
- [39] G. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.