Searching 2D-Strings for Matching Frames

Itai Boneh
Reichman University, Herzliya, Israel
University of Haifa, Israel

Dvir Fried
Bar Ilan University, Ramat Gan, Israel

Shay Golan
Reichman University, Herzliya, Israel
University of Haifa, Israel

Matan Kraus
Bar Ilan University, Ramat Gan, Israel

Adrian Miclăuş
Faculty of Mathematics and Computer Science, University of Bucharest, Romania

Arseny Shur
Bar Ilan University, Ramat Gan, Israel

Abstract

We study a natural type of repetitions in 2-dimensional strings. Such a repetition, called a matching frame, is a rectangular substring of size at least 2×2 with equal marginal rows and equal marginal columns. Matching frames first appeared in literature in the context of Wang tiles.

We present two algorithms finding a matching frame with the maximum perimeter in a given $n \times m$ input string. The first algorithm solves the problem exactly in $\tilde{O}(n^{2.5})$ time (assuming $n \geq m$). The second algorithm finds a $(1-\varepsilon)$-approximate solution in $\tilde{O}(\frac{nm}{\varepsilon^4})$ time, which is near linear in the size of the input for constant ε. In particular, by setting $\varepsilon = O(1)$ the second algorithm decides the existence of a matching frame in a given string in $\tilde{O}(nm)$ time. Some technical elements and structural properties used in these algorithms can be of independent interest.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases 2D string, matching frame, LCP, multidimensional range query

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.10

Related Version *Full Version*: `https://arxiv.org/abs/2310.02670` [9]

Funding *Itai Boneh*: supported by Israel Science Foundation grant 810/21.
Dvir Fried: supported by ISF grant no. 1926/19, by a BSF grant 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Shay Golan: supported by Israel Science Foundation grant 810/21.
Matan Kraus: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Adrian Miclăuş: supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS - UEFISCDI, project number PN-III-P1-1.1-TE-2021-0253, within PNCDI III.
Arseny Shur: supported by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064) and by the State of Israel through the Center for Absorption in Science of the Ministry of Aliyah and Immigration.

1 Introduction

Throughout the years, a variety of notions for repetitive structures in strings have been explored; see, e.g., [18, 31, 27, 42, 29]. Even recently, new efficient algorithms regarding palindromes [10, 22, 37], squares [17], runs [6, 16, 33], and powers [4] have been introduced. In the studies on 2-dimensional strings (aka 2*d-strings* or *matrices*), periodic and palindromic structures also attracted definite interest [2, 3, 5, 13, 21, 30, 19, 38].

Matching frame is a natural repetition in 2d-strings, first considered by Wang [40] when introducing Wang tiles. Given a 2d-string M over an alphabet Σ, a *frame* in M is a rectangle defined by a tuple (u, d, ℓ, r) such that $u < d$ and $\ell < r$. This rectangle covers the submatrix $M[u..d][\ell..r]$ and is *matching* if this submatrix has equal marginal rows and equal marginal columns. Formally, (u, d, ℓ, r) is a matching frame if $M[u][\ell..r] = M[d][\ell..r]$ and $M[u..d][\ell] = M[u..d][r]$ (see Figure 1). Wang's *fundamental conjecture*, later disproved by Berger [8], said "a set of tiles is solvable (= tiles the plane) if and only if it admits a cyclic rectangle (= matching frame)". Note that a fast algorithm to find matching frames would simplify a huge computation conducted by Jeandel and Rao [24] to prove that their aperiodic set of tiles is minimal.

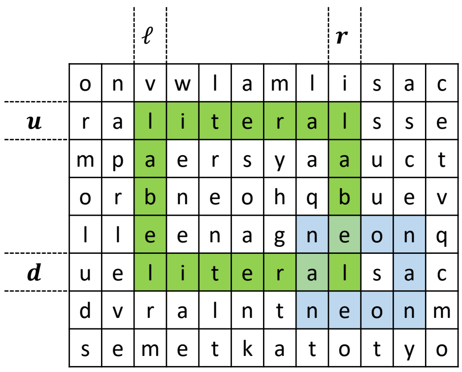

Figure 1 An example of a matching frame $(u, d, \ell, r) = (2, 6, 3, 9)$. The strings on the top and bottom sides of the frame are equal, and the strings on the left and right sides are also equal. The perimeter of the frame is $2 \cdot (6 - 2 + 9 - 3) = 20$. The matrix also contains a smaller matching frame.

Matching frames indicate "potential" periodicity in two dimensions. Namely, if a 2d-string M is built according to some local rule, then any matching frame in M can be extended to a periodic tiling of the plane, *respecting this local rule*. Well-known examples of such local rules are given, in particular, by self-assembly models such as aTAM [36] or 2HAM [11]. Note that matching frame is an *avoidable* repetition: as was first observed by Wang [41], there exist infinite *binary* 2d-strings without matching frames. Avoidable repetitions are interesting, in particular, due to a nontrivial decision problem.

Overall, there is a clear motivation to design efficient algorithms searching for matching frames. Let us specify the exact problem studied in this paper. The *perimeter* of a frame $F = (u, d, \ell, r)$ is the total number of cells in its marginal rows and columns, i.e. $\mathsf{per}(F) = 2(d - u + r - \ell)$. By *maximum* frame (in a set of frames) we mean the frame with the maximal perimeter in this set. In the *maximum matching frame problem*, the goal is to find a maximum matching frame in a given matrix or report that no matching frame exists. We also consider the $(1 - \varepsilon)$-approximation version of this problem, in which the goal is to find a matching frame with a perimeter within the factor $(1 - \varepsilon)$ from the maximum possible.

Our Results. We present $\tilde{O}(nm)$-space algorithms that establish the following bounds on the complexity of the maximum matching frame problem and its approximation version.

▶ **Theorem 1** (Maximum Matching Frame). *The time complexity of the maximum matching frame problem for an $n \times m$ matrix M is $\tilde{O}(n^{2.5})$ in the case $m = \Theta(n)$. In the general case, the complexity is $\tilde{O}(ab \min\{a, \sqrt{b}\})$, where $a = \min\{n, m\}$ and $b = \max\{n, m\}$.*[1]

▶ **Theorem 2** ($(1-\varepsilon)$-Approximation). *The time complexity of the $(1-\varepsilon)$-approximation maximum matching frame problem for an $n \times m$ matrix M is $\tilde{O}(\frac{nm}{\varepsilon^4})$.*

▶ **Corollary 3** (Deciding Matching Frame). *There is an algorithm deciding whether an $n \times m$ matrix M contains a matching frame in $\tilde{O}(nm)$ time and space.*

We remark that our exact and approximation algorithms can be straightforwardly adapted to find matching frames with the maximum area / the minimum perimeter / the minimum area instead of matching frames with the maximum perimeter.

High-Level Overview

Maximum Matching Frame. The algorithm for finding a maximum matching frame follows a heavy-light approach. The parameter used to distinguish between heavy and light frames is the *shorter side* of the frame. A frame $F = (u, d, \ell, r)$ has *height* $d - u$ and *width* $r - \ell$. We assume that there is a maximum matching frame having its height smaller than or equal to its width. (Either the input matrix or its transpose satisfies this assumption and we can apply our algorithm to both matrices and return the better of two results.) For some integer threshold x, we say that a frame with $d - u \leq x$ is *short* (or light); otherwise, it is *tall* (or heavy). We provide two algorithms, one that returns a maximum *short* matching frame in M and another returns a maximum *tall* matching frame in M. The largest of the two answers is the maximum matching frame in M.

The algorithm for short frames iterates over all pairs of rows with distance at most x from each other. Note that there are $O(nx)$ such pairs. Moreover, under the assumption that some matching frame $F = (u, d, \ell, r)$ is short, the rows u and d used by F are processed as a pair. When processing a pair, the algorithm decomposes its rows into maximal equal segments. Every segment is processed in linear time to obtain a maximum matching frame that uses a portion of the segment as top and bottom rows (see Section 5.1). The accumulated size of the segments is bounded by m, so the algorithm runs in $\tilde{O}(n \cdot m \cdot x)$ time.

The algorithm for tall frames (see Section 5.2) first guesses a range $[H/2..H]$ for the height and a range $[W/2..W]$ for the width of a maximum matching frame. As we consider tall frames, the ranges are sufficiently large, so it is easy to find a small set of positions $\mathcal{P}$ in the matrix M such that every frame with the height and width from the given ranges contains a position from $\mathcal{P}$. The algorithm employs a subroutine that, given H, W, and a position (i, j), computes a maximum matching frame among the frames that contain (i, j), have the height in $[H/2..H]$ and the width in $[W/2..W]$. The implementation of this subroutine is the main technical part of the algorithm. This is done by maintaining and querying a range data structure (see Section 4) that allows one to process pairs of columns and pairs of rows with the position (i, j) between them. There are $O(W^2)$ pairs of columns and $O(H^2)$ pairs of rows to be processed, which we do in $\tilde{O}(H^2 + W^2) = \tilde{O}(W^2)$ total time. We also show that $|\mathcal{P}| = O(\frac{nm}{HW})$, and therefore the running time for one pair of ranges is $\tilde{O}(nm\frac{W}{H})$. We

[1] Throughout the paper, $\tilde{O}(f(n)) = O(f(n) \cdot \mathsf{polylog} n)$

further observe that the sum of values $\frac{W}{H}$ over all guessed ranges is $O(\frac{W'}{H'})$ for some single guessed pair (W', H'). Since $x \leq H' \leq W' \leq \max\{n, m\}$, we obtain the running time of $\tilde{O}(nm\frac{\max(n,m)}{x})$.

Finally, the algorithm selects the threshold $x = \sqrt{\max\{n, m\}}$ and applies the algorithms for both the short and the tall case to obtain a running time of $\tilde{O}(nm\sqrt{\max\{n, m\}})$. Alternatively, one can run the algorithm for short frames alone, setting $x = \min(n, m)$. Taking the better of these two options proves Theorem 1.

Approximation Algorithm. As a preliminary step in our approach for finding a $(1-\varepsilon)$-approximation to the maximum matching frame, we apply a two-dimensional variant of the so-called *standard trick* [15, 12] from certain one-dimensional pattern matching problems. In pattern matching, we are given a text $T[1..n]$ and a pattern $P[1..m]$ and the goal is to find all the indices $i \in [n-m+1]$ such that $T[i..i+m-1]$ "matches" P. The standard trick refers to partitioning T into $O(n/m)$ overlapping fragments of size $\Theta(m)$, such that every match of P is contained in a fragment. In general, the trick allows one to assume that the length of the text is within a small factor from the length of the pattern. Our two-dimensional variant of this trick (Lemma 15) allows us to assume that both dimensions of the maximum matching frame are within a $\mathsf{poly}(1-\varepsilon)$ factor of the vertical and the horizontal lengths of M.

This assumption allows us to focus on matching frames with sides that are "close" to the boundaries of M; we call such frames *large*. The algorithm uses a carefully selected threshold for being close to the boundaries, guaranteeing that (1) the maximum matching frame is large and (2) the perimeter of every large frame approximates the perimeter of the maximum matching frame. With that, the problem boils down to determine whether there exists a large matching frame. The main technical novelty of the approximation algorithm is solving this decision problem in near-linear time.

The algorithm for the above decision problem consists of two main components. The first component (see Section 6.3) is an $\tilde{O}(1)$ time subroutine that, given a triplet (u, d, ℓ), decides if there is an integer r such that (u, d, ℓ, r) is a large matching frame. However, applying this subroutine to every triplet would cost $\Omega(n^2m)$ time. The second component (see Section 6.2) of the algorithm is the retrieval of a set of $\tilde{O}(nm)$ triplets such that if some large matching frame exists, there must also be a large matching frame derived from one of these triplets.

We conclude by presenting the combinatorial structure that allows us to consider $\tilde{O}(nm)$ triplets in the second component. Consider a triplet (u, d, ℓ) and let k be the largest integer such that $M[u][\ell..k] = M[d][\ell..k]$ (let S denote this string). Assuming there exists an index r such that (u, d, ℓ, r) is a large matching frame, one has $r \leq k$. Observe that if there is an index $d' < d$ that is close to the bottom boundary of M such that $M[d'][\ell..k] = S$, then (u, d', ℓ, r) is also a large matching frame. Therefore, the triplet (u, d, ℓ) can be removed from the set of triplets that have to be processed. We say that a triplet that is not eliminated due to this reasoning is *interesting*. Surprisingly, the number of interesting triplets is bounded by $O(nm \log n)$ (see Section 6.1). This combinatorial observation is the main novelty of the approximation algorithm.

2 Preliminaries

We use range notation for integers and strings. We write $[i..j]$ and $[i..j)$ for the sets $\{i, \ldots, j\}$ and $\{i, \ldots, j-1\}$ respectively (assuming $i \leq j$). Further, we abbreviate $[1..n]$ to $[n]$. A string $S[1..n] = S[1]S[2]\cdots S[n]$ is a sequence of characters from an alphabet Σ. We also write $\overleftarrow{S[1..n]} = S[n]S[n-1]\cdots S[1]$. For every $i \leq j \in [n]$, $S[i..j] = S[i]S[i+1]\cdots S[j]$ is a *substring* of S. The substring is called a *prefix* (resp., a *suffix*) of S if $i = 1$ (resp., $j = n$). We assume Σ to be linearly ordered, inducing a *lexicographic order* (*lex-order*) on strings.

An $n \times m$ *matrix* (or $2d$-*string*) M is a 2-dimensional array of symbols from Σ. We refer to the number of cells in M as the *size* of M, writing $|M| = nm$. We denote a horizontal substring of M as $M[i][j_1..j_2] = M[i][j_1]M[i][j_1+1]\dots M[i][j_2]$. Similarly, we denote a vertical substring as $M[i_1..i_2][j] = M[i_1][j]M[i_1+1][j]\dots M[i_2][j]$.

2.1 Suffix Arrays, Longest Common Prefixes

For a tuple of strings $\mathcal{S} = (S_1, S_2, \dots, S_n)$, the *lexicographically sorted array* $\mathsf{LSA}_{\mathcal{S}}$ is an array of length n that stores the lex-order of the strings in $\mathcal{S}$. Formally, $\mathsf{LSA}_{\mathcal{S}}[i] = j$ if S_j is the ith string in $\mathcal{S}$ according to the lex-order (ties are broken arbitrarily). For a string $S[1..n]$, the *suffix array* SA_S of S is the LSA of all suffixes of S. Formally, for every $i \in [n]$ let $S_i = S[i..n]$ and let $\mathcal{S}_S = (S_1, S_2, \dots, S_n)$; then $\mathsf{SA}_S = \mathsf{LSA}_{\mathcal{S}_S}$. The suffix arrays were introduced by Manber and Myers [32] and became ubiquitous in string algorithms. The array can be constructed in near-linear time and space by many algorithms [25, 26, 28, 34, 35, 42, 39].

▶ **Lemma 4.** *Given a string $S[1..n]$, the suffix array of S can be constructed in $O(n \log n)$ time and space.*

An important computational primitive is a data structure for computing the length of the *longest common prefix* of two strings $S[1..n]$ and $T[1..m]$, given as $\mathsf{LCP}(S,T) = \max\{\ell \in [\min\{n,m\}] \mid S[1..\ell] = T[1..\ell]\}$. An LCP data structure $\mathsf{LCP}_{\mathcal{S}}$ for a set of strings $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ supports queries in the form "given two indices $i, j \in [n]$, report $\mathsf{LCP}(S_i, S_j)$". We denote by $\mathsf{LCP}(S)$ the LCP data structure for the set of suffixes of a given string $S[1..n]$. It is known that the following can be obtained by applying the lowest common ancestor data structure of [23] to the suffix tree of [42].

▶ **Lemma 5.** *There is an* LCP *data structure with $O(n \log n)$ construction time and $O(1)$ query time. The data structure uses $O(n)$ space.*

The following facts are easy. For their proofs, see the full version [9] of this paper.

▶ **Fact 6.** *Given three strings S_1, S_2 and S_3, the condition* $\mathsf{LCP}(S_1, S_2) > \mathsf{LCP}(S_1, S_3)$ *implies* $\mathsf{LCP}(S_1, S_3) = \mathsf{LCP}(S_2, S_3)$.

▶ **Fact 7.** *Let $\mathcal{S} = (S_1, S_2, \dots, S_n)$ be a tuple of strings and let $P[1..m]$ be a string. The set* $\mathsf{Occ}(\mathcal{S}, P) = \{k \mid S_k[1..m] = P\}$ *coincides with the range* $\mathsf{LSA}_{\mathcal{S}}[i..j]$ *for some $i, j \in [n]$.*

Furthermore, there is an $O(\log n)$ time algorithm that given k, m, $\mathsf{LSA}_{\mathcal{S}}$*, and* $\mathsf{LCP}_{\mathcal{S}}$ *computes i and j such that* $\mathsf{Occ}(\mathcal{S}, S_k[1..m]) = \mathsf{LSA}_{\mathcal{S}}[i..j]$.

▶ **Definition 8** (Fingerprint)**.** *For a tuple $\mathcal{S}$ and a string $P = S_k[1..m]$, the* fingerprint *of P in $\mathcal{S}$ is the tuple (i, j, m) such that i and j are the indices specified in Fact 7.*

2.2 Orthogonal Range Queries

Our algorithms use data structures for *orthogonal range queries*. Such a data structure stores, for some positive integer dimension d, a set $\mathcal{P} \subseteq \mathbb{R}^d$ of d-dimensional points. Each point $p \in \mathcal{P}$ has an associated value $v(p) \in \mathbb{R}$. The data structure supports the queries regarding an input d-dimensional orthogonal range $R = [a_1..b_1] \times [a_2..b_2] \times \ldots \times [a_d..b_d]$. For a point $p = (x_1, x_2, \dots, x_d)$ one has $p \in R$ if $x_i \in [a_i..b_i]$ for every $i \in [1..d]$. We need the queries $\mathsf{Maximum}(R) = \mathsf{argmax}_{v(p)}(p \in R \cap \mathcal{P})$ and $\mathsf{Minimum}(R) = \mathsf{argmin}_{v(p)}(p \in R \cap \mathcal{P})$. For this, we use the data structure [43, 14] with the following running times.

▶ **Lemma 9.** *For any integer d, a set of n points in $\mathbb{R}^d$ can be preprocessed in $O(n\log^{d-1} n)$ time and space to support* Maximum *and* Minimum *range queries in $O(\log^{d-1} n)$ time.*

In Section 6.3, we use a very particular type of 2-dimensional Maximum/Minimum queries, where $v(p)$ is one of the coordinates of p. Though faster data structures are known in this case [7, 20], using these data structures cannot improve the asymptotics of our results.

3 Data Structures

When looking for matching frames in an $n \times m$ matrix M, we make use of the following data structures, which all our algorithms create during their preprocessing phase.

- For each column $\ell \in [m]$ we use
  1. a lex-sorted array $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$ of the strings $\{M[i][\ell..m] \mid i \in [n]\}$ (see Figure 2a);
  2. an LCP structure $\mathsf{LCP}^{\ell}_{\mathsf{rows}}$ over $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$;
  3. a range query structure D^{ℓ}_{rows}, containing all pairs $\{(i, I^{i,\ell}_{\mathsf{rows}}) \mid i \in [n]\}$, where $I^{i,\ell}_{\mathsf{rows}}$ is the index of the string $M[i][\ell..m]$ in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$ (see Figure 2b).

  In addition, we build the same three structures for the set of all strings of the form $\overleftarrow{M[i][1..\ell]}$, denoted as $\mathsf{LSA}^{\ell}_{\overleftarrow{\mathsf{rows}}}$, $\mathsf{LCP}^{\ell}_{\overleftarrow{\mathsf{rows}}}$ and $D^{\ell}_{\overleftarrow{\mathsf{rows}}}$.
- Symmetrically, for each row $u \in [n]$ we use
  1. a lex-sorted array $\mathsf{LSA}^{u}_{\mathsf{columns}}$ of the strings $\{M[u..n][i] \mid i \in [m]\}$;
  2. an LCP structure $\mathsf{LCP}^{u}_{\mathsf{columns}}$ over $\mathsf{LSA}^{u}_{\mathsf{columns}}$;
  3. a range query structure D^{u}_{columns}, containing all pairs $\{(i, I^{u,i}_{\mathsf{columns}}) \mid i \in [m]\}$, where $I^{u,i}_{\mathsf{columns}}$ is the index of the string $M[u \ldots n][i]$ in $\mathsf{LSA}^{u}_{\mathsf{columns}}$.

  In addition, we build the same three structures for the set of all strings of the form $\overleftarrow{M[1..u][i]}$, denoted as $\mathsf{LSA}^{u}_{\overleftarrow{\mathsf{columns}}}$, $\mathsf{LCP}^{u}_{\overleftarrow{\mathsf{columns}}}$ and $D^{u}_{\overleftarrow{\mathsf{columns}}}$.

In the full version [9] of this paper we show how to construct all these structures in $O(nm\log(nm))$ time and space.

4 The Segment Compatibility Data Structure

In this section we present the *segment compatibility data structure* (SCDS), which is at the core of our maximum matching frame algorithm (see Section 5.2). We start with technical definitions.

Segment, aligned pair, compatible pairs. A horizontal (resp. vertical) *segment* is a triplet (i, j_1, j_2) (resp. (i_1, i_2, j)) with $j_1 < j_2$ (resp. $i_1 < i_2$). It represents the horizontal (resp. vertical) segment in the plane connecting the points (i, j_1) and (i, j_2) (resp. (i_1, j) and (i_2, j)). A pair (s_1, s_2) of horizontal segments is *aligned* if $s_1 = (i_1, j_1, j_2)$ and $s_2 = (i_2, j_1, j_2)$ for some $i_1 < i_2, j_1 < j_2 \in \mathbb{N}$. Such a pair has *distance* $|i_2 - i_1|$. Symmetrically, a pair of vertical segments (s_1, s_2) is aligned if $s_1 = (i_1, i_2, j_1)$ and $s_2 = (i_1, i_2, j_2)$ for some $i_1 < i_2, j_1 < j_2 \in \mathbb{N}$. Such a pair has *distance* $|j_2 - j_1|$.

An aligned pair of horizontal segments (i_1, j_1, j_2) and (i_2, j_1, j_2) and an aligned pair of vertical segments (a_1, a_2, b_1) and (a_1, a_2, b_2) are *compatible* if and only if $a_1 \le i_1 \le i_2 \le a_2$, and $j_1 \le b_1 \le b_2 \le j_2$.

The SCDS stores a set of aligned pairs of vertical segments and supports the query

- MaxCompatible(h_1, h_2): given an aligned pair (h_1, h_2) of horizontal segments, return a pair (v_1, v_2) with the maximum distance among the stored pairs compatible with (h_1, h_2), or return null if no stored pair is compatible with (h_1, h_2).

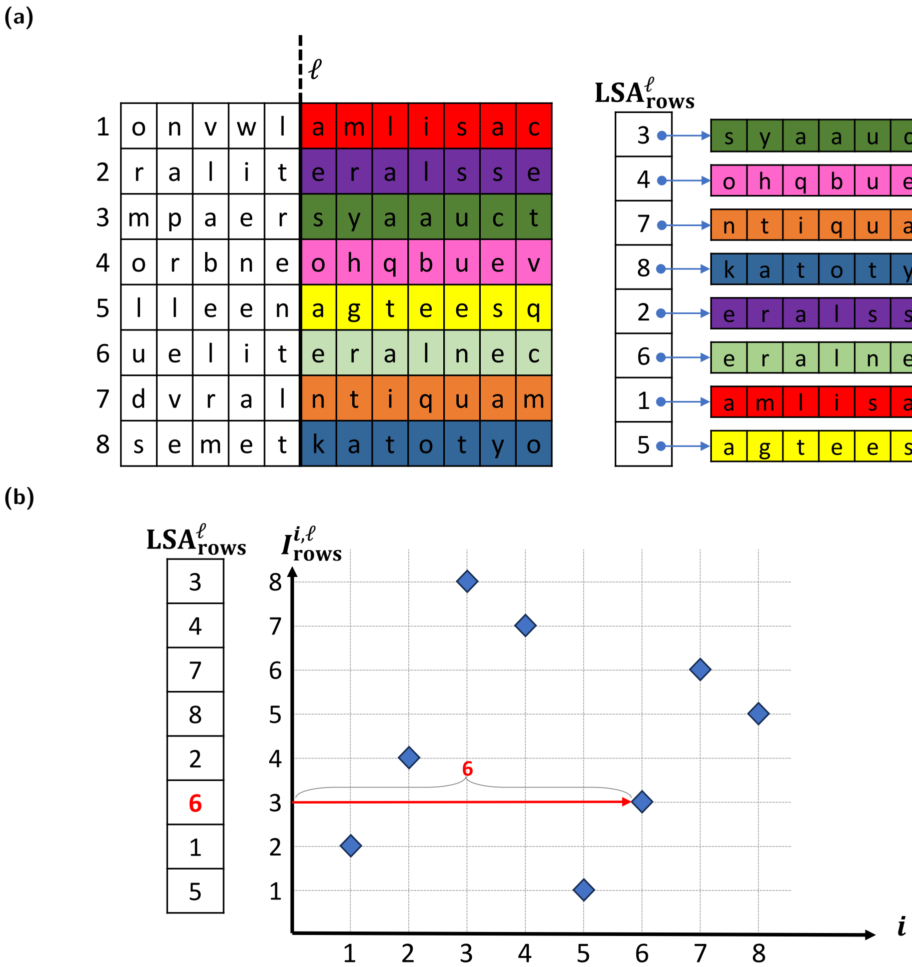

Figure 2 (a) An example of $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$. Every cell in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$ contains an index corresponding to a horizontal word in the matrix starting in column ℓ. The (indices representing the) words appear bottom-up in lex-order.
(b) A visualization of the points stored in D^{ℓ}_{rows}. Every point corresponds to a horizontal word. The height of every point corresponds to the location of the corresponding word in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$. The horizontal location of a point represents the index of its appearance in the string.

▶ **Lemma 10.** *Given a set T of t aligned pairs of vertical segments, the* SCDS *with $O(\log^3 t)$ query time can be built in $O(t \log^3 t)$ time.*

Proof. For each aligned pair $P = \big((a_1, a_2, b_1), (a_1, a_2, b_2)\big)$, we define a 4-dimensional point $\mathsf{point}(P) = (a_1, a_2, b_1, b_2)$ with the value $v(\mathsf{point}(P)) = b_2 - b_1$. Then we build, for the set of points $\{\mathsf{point}(P) \mid P \in T\}$, a 4-dimensional range data structure D with $\mathsf{Maximum}$ queries.

Let $(h_1, h_2) = \big((i_1, j_1, j_2), (i_2, j_1, j_2)\big)$ be a pair of aligned horizontal segments and let $R = ([-\infty, i_1], [i_2, \infty], [j_1, j_2], [j_1, j_2])$. It is clear that a pair P is compatible with (h_1, h_2) if and only if $\mathsf{point}(P) \in R$. Hence, to perform the query $\mathsf{MaxCompatible}(h_1, h_2)$, we query D with $\mathsf{Maximum}(R)$ and return the output.

Due to Lemma 9, the construction time and the query time are as required. ◀

5 Maximum Matching Frame

In this section we prove Theorem 1, describing an algorithm with the announced time complexity. We assume that the input matrix M contains a maximum matching frame (u, d, ℓ, r) whose *height* $d - u$ is smaller than or equal to its *width* $r - \ell$. To cover the complementary case, the algorithm is applied both to the original matrix M and to its transpose $M^\top$ and then the maximum result is reported.

Our algorithm chooses a parameter x and distinguishes between *short* frames of height at most x and *tall* frames with height larger than x. It processes the two types of frames separately and returns the maximum between two solutions.

5.1 Algorithm for Short Frames

In this section we prove the following lemma:

▶ **Lemma 11.** *There is an algorithm that for a given $x \in [n]$ finds, in $\tilde{O}(n \cdot m \cdot x)$ time and $O(n)$ additional space, a maximum matching frame of height at most x.*

Proof. For every two rows $u', d' \in [n]$ such that $d' \in [u' + 1..u' + x]$ the algorithm works as follows. First, the algorithm finds all maximal ranges $[a..b]$ such that $M[u'][a..b] = M[d'][a..b]$. By "maximal" we mean that a range can not be extended to the right or to the left while keeping equality. Note that all maximal ranges are disjoint. For $k \in [m]$ we denote the vertical string $M[u'..d'][k]$ by S_k.

Let $[a..b]$ be a maximal range. For every vertical string S_k with $k \in [a..b]$ we find its leftmost and rightmost occurrences in the range $[a..b]$. This is achieved by initializing an empty dictionary $D_{a,b}$ and scanning the range $[a..b]$ left to right. For each $k \in [a..b]$ the algorithm computes the fingerprint f in $\mathsf{LSA}^{u'}_{\mathsf{columns}}$ of the string S_k (see Definition 8). If f is not in $D_{a,b}$, we add f to $D_{a,b}$ and update both the leftmost and rightmost occurrence of S_k to be k. If f is already in $D_{a,b}$, we update the rightmost occurrence of S_k to be k.

After completing the scan, the algorithm finds a vertical string S_k such that the distance between the leftmost occurrence ℓ' and the rightmost occurrence r' of S_k is maximal. If $\ell' < r'$, we call the frame (u', d', ℓ', r') the (a, b)-range candidate of (u', d'); otherwise, there is no such candidate. Among all maximal ranges $[a..b]$, an (a, b)-range candidate with the maximal perimeter is the (u', d')-candidate (if there are no (a, b)-range candidates for (u', d'), there is no (u', d') candidate). The algorithm outputs a (u', d')-candidate with the maximal perimeter over all pairs of rows (u', d') or returns null if there are no such candidates.

Correctness. Let $F' = (u', d', \ell', r')$ be the frame returned by the algorithm. Then F' is the (a, b)-range candidate of (u', d') for some range $[a..b]$ such that $a \le \ell' < r' \le b$. Then, the equality $M[u'][a..b] = M[d'][a..b]$ implies $M[u'][\ell'..r'] = M[d'][\ell'..r']$, while $M[u'..d'][\ell'] = M[u'..d'][r']$ by the choice of ℓ', r'. Hence, F' is matching.

Let $F = (u, d, \ell, r)$ be a maximum matching frame among the frames of height at most x. When the algorithm iterates over the rows u, d, it identifies a range $[a..b]$ such that $a \le \ell < r \le b$. Let $\hat{F} = (u, d, \hat{\ell}, \hat{r})$ be the (a, b)-range candidate of (u, d). Since F is a valid choice for this candidate, the inequality $r - \ell \le \hat{r} - \hat{\ell}$ holds, implying $\mathsf{per}(F) \le \mathsf{per}(\hat{F}) \le \mathsf{per}(F')$.

Complexity. For a pair of rows (u', d'), identifying the maximal ranges takes $O(m)$ time. A maximal range $[a..b]$ requires $O(b - a)$ dictionary operations, each taking $O(\log n)$ time using, for example, an AVL tree [1]. Since all the maximal ranges of (u', d') are disjoint, their lengths sum to at most m, leading to the running time $\tilde{O}(m)$ for (u', d').

Since $d' \in [u'+1..u'+x]$, there are $O(n \cdot x)$ pairs of rows to process. Therefore, the total running time of the algorithm is $\tilde{O}(n \cdot m \cdot x)$. Since the algorithm considers every pair of rows (u', d') separately, the (additional) space usage of the algorithm is $O(n)$. ◀

5.2 Algorithm for Tall Frames

In this section, we prove the following lemma:

▶ **Lemma 12.** *There is an algorithm that for a given $x \in [n]$ finds, in $\tilde{O}(\frac{n \cdot m^2}{x})$ time and $\tilde{O}(m^2)$ additional space, a maximum matching frame of height at least x.*

Given a frame $F = (u, d, \ell, r)$ and a position $p = (i, j)$ such that $i \in [u..d]$ and $j \in [\ell..r]$, we say that p is *contained* in F and F *contains* p. We say that F is a (p, H, W)-frame if $d - u \in [H/2..H]$, $r - \ell \in [W/2..W]$, and F contains p. We introduce an algorithm that finds a maximum matching (p, H, W)-frame and use it as a subroutine of the algorithm finding the maximum matching tall frame.

▶ **Lemma 13.** *Given a position (i, j) in M and a pair of positive integers $(H, W) \in [n] \times [m]$, there is an algorithm finding a maximum matching $((i, j), H, W)$-frame in $\tilde{O}(H^2+W^2)$ time and $\tilde{O}(W^2)$ additional space.*

Proof. For every pair $(\ell, r) \in [m]^2$ such that $r - \ell \in [W/2..W]$ and $j \in [\ell..r]$, the algorithm finds the maximal aligned agreement between the columns ℓ and r intersecting the ith row by executing two LCP queries. First the algorithm queries $\mathsf{LCP}^i_{\mathsf{columns}}$ to obtain the maximal d' such that $M[i..d'][\ell] = M[i..d'][r]$. Similarly, the algorithm queries $\mathsf{LCP}^i_{\overleftarrow{\mathsf{columns}}}$ to obtain the minimal u' such that $M[u'..i][\ell] = M[u'..i][r]$. Then the algorithm stores the pair of segments $s_1 = (u', d', \ell)$ and $s_2 = (u', d', r)$. To conclude this part, the algorithm constructs an SCDS over all stored pairs.

Next, the algorithm iterates over all pairs $(u, d) \in [n]^2$ such that $d - u \in [H/2..H]$ and $i \in [u..d]$. For each such pair, the algorithm queries the data structures $\mathsf{LCP}^j_{\mathsf{rows}}$ and $\mathsf{LCP}^j_{\overleftarrow{\mathsf{rows}}}$ (similar to the above computation of vertical agreements), obtaining the minimal ℓ' and the maximal r' such that $M[u][\ell'..r'] = M[d][\ell'..r']$. The algorithm then constructs the horizontal aligned pair of segments $s^h_1 = (u, \ell', r')$ and $s^h_2 = (d, \ell', r')$. The algorithm queries SCDS for $(s^v_1, s^v_2) \leftarrow \mathsf{MaxCompatible}(s^h_1, s^h_2)$. Let $s^v_1 = (t_1, t_2, \ell)$ and $s^v_2 = (t_1, t_2, r)$. We call the frame (u, d, ℓ, r) the (u, d)-*optimal frame*. If the query $\mathsf{MaxCompatible}(s^h_1, s^h_2)$ returns null, there is no (u, d)-optimal frame. The algorithm reports the (u, d)-optimal frame with the maximum perimeter among all pairs (u, d), or returns null if no such frames were found.

Correctness. By construction, each frame (u, d, ℓ, r) identified by the algorithm is a (p, H, W)-frame. We proceed to show that it is a matching frame. Recall that (u, d, ℓ, r) was obtained from two compatible pairs of segments s^v_1, s^v_2 and s^h_1, s^h_2. Notice that for the pair $s^v_1 = (u_v, d_v, \ell)$ and $s^v_2 = (u_v, d_v, r)$ to be compatible with $s^h_1 = (u, \ell_h, r_h), s^h_2 = (d, \ell_h, r_h)$, the inequalities $u_v \le u$ and $d_v \ge d$ must hold. By the construction of s^v_1 and s^v_2 we have $M[u_v..d_v][\ell] = M[u_v..d_v][r]$ and then $M[u..d][\ell] = M[u..d][r]$. In a similar way, one can prove $M[u][\ell..r] = M[d][\ell..r]$, showing that (u, d, ℓ, r) is a matching frame as required.

To conclude the correctness of our algorithm, we need to show that some maximum matching (p, H, W)-frame is (u, d)-optimal for some (u, d). Let (u_t, d_t, ℓ_t, r_t) be a maximum matching (p, H, W)-frame. For (u_t, d_t), the algorithm creates the horizontal aligned pair $s^h_1 = (u_t, \ell_h, r_h), s^h_2 = (d_t, \ell_h, r_h)$. Since $M[u_t][\ell_t..r_t] = M[d_t][\ell_t..r_t]$, we have $\ell_h \le \ell_t$ and $r_h \ge r_t$. By a similar argument, when constructing the SCDS, the algorithm creates a vertical aligned pair $s^v_1 = (u_v, d_v, \ell_t)$, $s^v_2 = (u_v, d_v, r_t)$ with $u_v \le u_t$ and $d_v \ge d_t$. Denote the output of $\mathsf{MaxCompatible}(s_1^h, s_2^h)$ by $\big((u', \ell', r'), (d', \ell', r')\big)$. One has $r' - \ell' \geq r_t - \ell_t$ since the pair (s_1^v, s_2^v) is compatible with (s_1^h, s_2^h). Then (u_t, d_t, ℓ', r') is a matching frame with perimeter $2(d_t - u_t + r' - \ell') \geq 2(d_t - u_t + r_t - \ell_t)$. Due to the maximality of the perimeter of (u_t, d_t, ℓ_t, r_t), we have that (u_t, d_t, ℓ', r') is a maximum matching (p, H, W)-frame.

Complexity. It can be easily shown that there are $O(W^2)$ pairs (ℓ, r) satisfying $r - \ell \leq W$ and $j \in [\ell..r]$. Similarly, there are $O(H^2)$ pairs (u, d) satisfying $d - u \leq H$ and $i \in [u..d]$. By Lemma 10, the construction of the SCDS takes $\tilde{O}(W^2)$ time. The algorithm then applies $O(H^2)$ queries to the SCDS and the overall complexity is $\tilde{O}(W^2 + H^2)$. The additional space usage of the algorithm is dominated by the SCDS data structure of size $\tilde{O}(W^2)$. ◀

Proof of Lemma 12. The algorithm iterates over all pairs $H, W \in \{x \cdot 2^k \mid k \geq 1\}$ such that $H \leq W < 2m$. For a pair (H, W), the algorithm runs the subroutine from Lemma 13 for every position $(i, j) \in [n] \times [m]$ such that $i \bmod H/2 = 0$ and $j \bmod W/2 = 0$. Finally, the algorithm reports the maximum matching frame among all outputs of this subroutine.

Correctness. Since every instance of the subroutine from Lemma 13 reports a matching frame or a null, the algorithm also reports a matching frame (or a null). Let $F = (u, d, \ell, r)$ be a maximum matching frame of height at least x. Let W (resp. H) be the smallest number in $\{x \cdot 2^k \mid k \geq 1\}$ which is at least $r - \ell$ (resp. $d - u$). Then there exist $i \in [u..d]$ and $j \in [\ell..r]$ such that $i \bmod H/2 = 0$ and $j \bmod W/2 = 0$. Hence the algorithm ran the subroutine for $((i, j), H, W)$-frames and got reported a matching frame F' with $\mathsf{per}(F') \geq \mathsf{per}(F)$. Therefore, the algorithm returns a maximum matching frame.

Complexity. For a given pair (H, W), the subroutine of Lemma 13 was called for $\lfloor \frac{2n}{H} \rfloor \cdot \lfloor \frac{2m}{W} \rfloor$ points (i, j). In total, these calls cost $\tilde{O}\big(\frac{nm}{HW}(W^2 + H^2)\big) = \tilde{O}\big(nm\frac{W}{H}\big)$ time. Therefore, the algorithm runs in $\tilde{O}(nm) \cdot \sum_{H,W} \frac{W}{H}$ time, where the summation is over all possible pairs. Let $t = \lceil \log \frac{m}{x} \rceil$. Since $x \leq H \leq W < 2m$, we have $\sum_{H,W} \frac{W}{H} = 2^t + 2 \cdot 2^{t-1} + 3 \cdot 2^{t-2} + \cdots \leq 4 \cdot 2^t = O(\frac{m}{x})$. The time bound from the lemma now follows. The additional space usage of the algorithm is dominated by the space of the largest instance of Lemma 13, which is $\tilde{O}(W^2)$ for some W. Since $W < 2m$, we have the required bound $\tilde{O}(m^2)$. ◀

5.3 Combining the Short and Tall Algorithms

In this section, we combine the results of Section 5.1 and Section 5.2 to prove Theorem 1.

Proof of Theorem 1. Applying the algorithm of Lemma 11 and the algorithm of Lemma 12 with the same threshold $x = \sqrt{m}$ and reporting the maximum frame between both outputs yields an algorithm with running time $\tilde{O}(nm \cdot \sqrt{m})$. We run the same scheme for the transposed matrix $M^\top$ and $x = \sqrt{n}$, which takes $\tilde{O}(nm \cdot \sqrt{n})$ time. In total, processing both M and $M^\top$ takes $\tilde{O}(nm \cdot \sqrt{\max\{n, m\}})$ time. The space usage of the algorithm is dominated by the preprocessed data, which takes $\tilde{O}(nm)$ space.

Notice that $d - u \leq r - \ell$ for all considered frames, yielding $d - u \leq \min\{n, m\}$. Therefore, applying Lemma 11 to both M and $M^\top$ with $x = \min\{n, m\}$ provides an alternative algorithm that outputs the maximum matching frame within $\tilde{O}(nm \cdot \min\{n, m\})$ time. Choosing the faster between the two above algorithms implies Theorem 1. ◀

6 Approximation Version

In the $(1-\varepsilon)$-approximation version of the problem, the goal is to find, given a matrix M with a maximum matching frame F, a matching frame F' in M with $\mathsf{per}(F') \geq (1-\varepsilon)\mathsf{per}(F)$. Our algorithm reduces the problem to multiple instances of a decision problem defined below. The reduction is shown in Lemma 15 below and the decision problem is solved in Section 6.3.

Decision problem. The input for this problem is a matrix M, and an *inner rectangle* $(u_\square, d_\square, \ell_\square, r_\square)$ in M. A frame (u, d, ℓ, r) in M is *surrounding* if $(u_\square, d_\square, \ell_\square, r_\square)$ is strictly inside it; formally, if $u < u_\square \leq d_\square < d$ and $\ell < \ell_\square \leq r_\square < r$. The goal in this version of the problem is to output a surrounding matching frame (u, d, ℓ, r) or report that no such frame exists in M. In Section 6.3, we show that this problem can be solved in near-linear time, by proving the following lemma.

▶ **Lemma 14.** *Given an $n \times m$ matrix M with an inner rectangle $(u_\square, d_\square, \ell_\square, r_\square)$, there is an algorithm that finds, in $\tilde{O}(nm)$ time and space, a surrounding matching frame in M or reports that no such frame exists.*

Via an application of a 2-dimensional variant of the so-called *standard trick* [15, 12], we obtain the following reduction.

▶ **Lemma 15.** *Let $a = 1 + \varepsilon/3$. For every $(h, w) \in [\log_a n] \times [\log_a m]$ such that $a^h, a^w \geq 2$, there is a set $\mathcal{M}_{h,w}$ of sub-matrices, each associated with an inner rectangle, such that the following properties are satisfied:*

1. *$|\mathcal{M}_{h,w}| = O(\frac{nm}{\varepsilon^2 a^{h+w}})$.*
2. *For every sub-matrix $M' \in \mathcal{M}_{h,w}$, $|M'| = O(a^{h+w})$.*
3. *For every frame (u, d, ℓ, r) with $d - u \in [a^h..a^{h+1} - 1]$ and $r - \ell \in [a^w..a^{w+1} - 1]$ there is a sub-matrix $M' \in \mathcal{M}_{h,w}$ such that (u, d, ℓ, r) is a surrounding frame in M' with respect to its inner rectangle.*
4. *For every surrounding frame F in any $M' \in \mathcal{M}_{h,w}$, $\mathsf{per}(F) \geq (1-\varepsilon)\big(2(a^{w+1} + a^{h+1})\big)$.*

The inner rectangles and the corners of the sub-matrices in $\mathcal{M}_{h,w}$ can be obtained in $O(|\mathcal{M}_{h,w}|)$ time and space given h and w.

Proof. Fix $(h, w) \in [\log_a n] \times [\log_a m]$. We define several numeric values that are used repeatedly by our reduction, namely $\delta_w = \left\lfloor \frac{\varepsilon a^{w+1}}{3} \right\rfloor$, $\delta_h = \left\lfloor \frac{\varepsilon a^{h+1}}{3} \right\rfloor$, $W_w = \lceil a^{w+2} \rceil$, and $H_h = \lceil a^{h+2} \rceil$. For convenience, assume without loss of generality that both $\frac{n - H_h}{\delta_h}$ and $\frac{m - W_w}{\delta_w}$ are integers. Otherwise, the algorithm adds dummy rows and columns to the right and to the bottom sides of the matrix with distinct unique characters not in Σ until δ_h divides $n - H_h$ and δ_w divides $m - W_w$. The set $M_{h,w}$ of sub-matrices of M is defined as follows:

$$\mathcal{M}_{h,w} = \left\{ M[\alpha\delta_h + 1..\alpha\delta_h + H_h][\beta\delta_w + 1..\beta\delta_w + W_w] \mid \alpha \in [0..\tfrac{n-H_h}{\delta_h}] \text{ and } \beta \in [0..\tfrac{m-W_w}{\delta_w}] \right\}.$$

In words, those are all sub-matrices with width $W_w - 1$ and height $H_h - 1$, having their upper left corner in a cell (x', y') of M such that $x' \bmod \delta_h = y' \bmod \delta_w = 1$. Note that Properties 1 and 2 are trivially satisfied. Additionally, it is clear that the corners of each sub-matrix can be obtained in constant time.

Property 3 is obtained by combining the following two claims.

▷ Claim 16. Every frame (u, d, ℓ, r) with $d - u \in [a^h..a^{h+1} - 1]$ and $r - \ell \in [a^w..a^{w+1} - 1]$ is contained in some $M' \in \mathcal{M}_{h,w}$.

Proof. Let x (resp. y) be the largest integer multiple of δ_h (resp. δ_w) that is smaller than u (resp. ℓ). By definition, $\mathcal{M}_{h,w}$ contains a sub-matrix $M' = M[x+1..x+H_h][y+1..y+W_w]$. In order to prove that (u,d,ℓ,r) is fully contained inside M', we need to show that (1) $x < u$, (2) $y < \ell$, (3) $x + H_h \geq d$ and (4) $y + W_w \geq r$. Conditions (1), (2) are immediate from the choice of x and y. Let us show (3). The choice of x also implies $x + \delta_h \geq u$. Therefore,

$$x + H_h \geq u - \delta_h + H_h = u - \left\lfloor \frac{\varepsilon a^{h+1}}{3} \right\rfloor + \lceil a^{h+2} \rceil$$
$$\geq u - \frac{\varepsilon a^{h+1}}{3} + a^{h+2} = u + a^{h+1}\left(a - \frac{\varepsilon}{3}\right) = u + a^{h+1}.$$

By conditions of the lemma, $d - u < a^{h+1}$, so we obtain $x + H_h > d$ as required. Condition (4) can be shown in the same way. ◁

For each sub-matrix $M' = M[x+1..x+H_h][y+1..y+W_w]$ we define the inner rectangle $R_\square = (u_\square, d_\square, \ell_\square, r_\square) = (x + H_h - \lceil a^h \rceil + 1, x + \lceil a^h \rceil - 1, y + W_w - \lceil a^w \rceil + 1, y + \lceil a^w \rceil - 1)$. As the further argument does not depend on x, y, we assume $x = y = 0$ for simplicity.

▷ Claim 17. If (u,d,ℓ,r) is a frame in M' with $r - \ell \in [a^w..a^{w+1}-1]$ and $d - u \in [a^h..a^{h+1}-1]$, then (u,d,ℓ,r) is a surrounding frame.

Proof. Since $d \leq H_h$ and $d - u \geq a^h$, one has $u \leq d - a^h \leq H_h - a^h < u_\square$, as required. Since $u \geq 1$, one also has $d \geq a^h + 1 > d_\square$ as required. The inequalities $\ell < \ell_\square$ and $r > r_\square$ are proved in the same way, so (u,d,ℓ,r) is surrounding by definition. ◁

To prove Property 4, we note that the perimeter of a surrounding frame in M' is at least $2((d_\square - u_\square + 2) + (r_\square - \ell_\square + 2))$. We show that $d_\square - u_\square + 2 \geq (1-\varepsilon) \cdot a^{h+1}$. It can be similarly argued that $r_\square - \ell_\square + 2 \geq (1-\varepsilon) \cdot a^{w+1}$; the two inequalities together yield Property 4. Recall that $u_\square = H_h - \lceil a^h \rceil + 1$, $d_\square = \lceil a^h \rceil - 1$, $H_h = \lceil a^{h+2} \rceil$. Then

$$d_\square - u_\square + 2 = \lceil a^h \rceil - 1 - H_h + \lceil a^h \rceil - 1 + 2 \geq 2a^h - a^{h+2} = a^{h+1}\left(\tfrac{2}{a} - a\right)$$

It remains to show that $\frac{2}{a} - a \geq 1 - \varepsilon$. Indeed,

$$\frac{2}{a} - a = \frac{2 - (1 + 2\varepsilon/3 + \varepsilon^2/9)}{1 + \varepsilon/3} = \frac{1 - 2\varepsilon/3 - \varepsilon^2/3 + 2\varepsilon^2/9}{1 + \varepsilon/3} = 1 - \varepsilon + \frac{2\varepsilon^2/9}{1 + \varepsilon/3} > 1 - \varepsilon,$$

as required. The lemma is proved. ◀

With Lemmas 14 and 15, we are ready to prove Theorem 2.

Proof of Theorem 2. The algorithm first processes frames of height 1 or width 1, applying the algorithm of Lemma 11 with $x = 1$ to both M and $M^\top$. After that, the algorithm proceeds as follows. For every pair $(h,w) \in [\log_a n] \times [\log_a m]$ such that $a^w, a^h \geq 2$, it creates the set $\mathcal{M}_{h,w}$ with the corresponding inner rectangles (see Lemma 15) and applies Lemma 14 on every $M' \in \mathcal{M}_{h,w}$ with its inner rectangle. The algorithm returns the maximum frame among the matching frames returned by algorithms of Lemma 11 and Lemma 14. If neither of these two algorithms reported a frame, then a "no frames" answer is reported.

Correctness. Let $F = (u, d, \ell, r)$ be a maximum matching frame in M. If $d = u + 1$ or $r = \ell + 1$, then F is found by the algorithm of Lemma 11. Otherwise, consider the pair $(h, w) \in [\log_a n] \times [\log_a m]$ such that $d - u \in [a^h..a^{h+1} - 1]$ and $r - \ell \in [a^w..a^{w+1} - 1]$. By Property 3 of Lemma 15, there is a sub-matrix $M' \in \mathcal{M}_{h,w}$ that contains F as a surrounding frame. The algorithm in Lemma 14 returns a surrounding matching frame F' in M', and by Property 4 of Lemma 15, $\mathsf{per}(F') \geq (1-\varepsilon)\big(2(a^{w+1}+a^{h+1})\big)$. Since $\mathsf{per}(F) < 2(a^{w+1}+a^{h+1})$, the approximation guarantee is fulfilled.

Complexity. Given h and w, the running time of the algorithm that obtains $\mathcal{M}_{h,w}$ and the suitable $R_\square$ is $O(|\mathcal{M}_{h,w}|) \subseteq O(nm/\varepsilon^2)$ by Property 1 of Lemma 15.

Due to Properties 1 and 2 of Lemma 15, the sum of the sizes of the matrices in $\mathcal{M}_{h,w}$ is $O\left(\frac{nm}{\varepsilon^2}\right)$. Hence, applying Lemma 14 on all $M' \in \mathcal{M}_{h,w}$ takes $\tilde{O}\left(\frac{nm}{\varepsilon^2}\right)$ time. Recall that there are $O(\log_{1+\varepsilon} n \cdot \log_{1+\varepsilon} m) = O(\frac{1}{\varepsilon^2} \log n \cdot \log m)$ values of h and w. Thus, the total running time of the algorithm is $\tilde{O}(\frac{nm}{\varepsilon^4})$. Each matrix in $\mathcal{M}_{h,w}$ is processed separately. The space complexity of processing a matrix is $\tilde{O}(a^{h+w}) = \tilde{O}(nm)$. The space is reused when each matrix is processed, so the overall space complexity of the algorithm is $\tilde{O}(nm)$. ◀

6.1 Interesting Pairs and Interesting Triplets

In order to prove Lemma 14, we introduce and study the following notion, illustrated by Figure 3.

▶ **Definition 18.** *Given a tuple $(S_1, \ldots, S_n)$ of strings, we call a pair (i, j) interesting if $i < j$ and for any ℓ such that $\ell \in [i + 1, j - 1]$ one has $\mathsf{LCP}(S_i, S_\ell) < \mathsf{LCP}(S_i, S_j)$.*

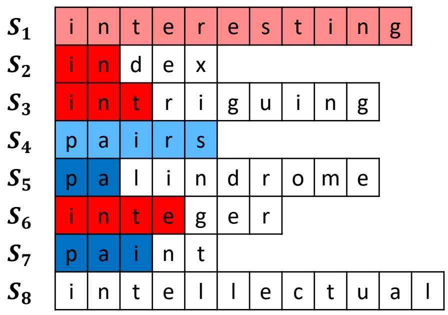

Figure 3 An example of interesting pairs where the first component of the pair is S_1 or S_4. The rows beginning in red form interesting pairs with S_1 and the rows beginning in blue form interesting pairs with S_4. The color indicates the LCP of the components of the pair. Notice that (S_1, S_8) is not an interesting pair because of S_6.

Trivially, all pairs of the form $(i, i + 1)$ are interesting for any tuple. The next lemma bounds the number of interesting pairs. This bound is tight as shown in the full version [9] of this paper.

▶ **Lemma 19.** *For each n-tuple of strings, there are $O(n \log n)$ interesting pairs.*

Proof. For a given tuple $(S_1, \ldots, S_n)$, fix an integer $\ell \in [1..\lceil \log n \rceil]$ and consider the set $\mathcal{I}_\ell = \{(i, j) \mid (i, j)$ is interesting and $j - i \in [2^{\ell-1}..2^\ell - 1]\}$. We say that a pair $(i, j) \in \mathcal{I}_\ell$ is of the *first type* if $i = \max\{i' \mid (i', j) \in \mathcal{I}_\ell\}$ and of the *second type* otherwise. The following claim is crucial.

▷ Claim 20. All pairs of the first type from $\mathcal{I}_\ell$ have different second components; all pairs of the second type from $\mathcal{I}_\ell$ have different first components.

Proof. The first statement stems directly from the definition of the first type. Let us prove the second one. Assume by contradiction that $(i,j),(i,j') \in \mathcal{I}_\ell$ are pairs of the second type, with $j' < j$. As (i,j) is not of the first type, $\mathcal{I}_\ell$ contains a pair (i',j) with $i' > i$. We prove the following sequence of inequalities, leading to a contradiction.

$$\mathsf{LCP}(S_i,S_{i'}) \overset{(1)}{<} \mathsf{LCP}(S_i,S_{j'}) \overset{(2)}{=} \mathsf{LCP}(S_{j'},S_j) \overset{(3)}{=} \mathsf{LCP}(S_{i'},S_{j'}) \overset{(4)}{<} \mathsf{LCP}(S_{i'},S_j) \overset{(5)}{=} \mathsf{LCP}(S_i,S_{i'}),$$

Since $2^{\ell-1} \le j-i'$, $2^{\ell-1} \le j'-i$ and $j-i < 2^\ell \le j-i' + j'-i$, we have $i' < j'$. Since (i,j') is an interesting pair and $i' \in [i{+}1..j'{-}1]$, we obtain (1) by Definition 18. Since (i,j) is an interesting pair, every $k \in [i{+}1..j{-}1]$ satisfies $\mathsf{LCP}(S_i,S_k) < \mathsf{LCP}(S_i,S_j)$. Hence, by Fact 6 we have $\mathsf{LCP}(S_i,S_k) = \mathsf{LCP}(S_k,S_j)$. We obtain (2) and (5) by setting $k = j'$ and $k = i'$ respectively. Finally, (i',j) is an interesting pair, and $j' \in [i'+1..j-1]$. So, Definition 18 gives us (4) and then Fact 6 implies (3). ◁

Claim 20 says that $\mathcal{I}_\ell$ contains at most n pairs of the first type and at most n pairs of the second type. As ℓ takes $\lceil \log n \rceil$ values, the lemma follows. ◀

To relate interesting pairs to our decision problem we need one more notion.

▶ **Definition 21.** *Let M be an $n \times m$-matrix and $\ell \in [m]$. A triplet (u,d,ℓ) is called* interesting *if the pair (u,d) is interesting for the tuple $(M[1][\ell..m],\dots,M[n][\ell..m])$.*

6.2 Finding all interesting triplets

▶ **Lemma 22.** *All interesting triplets for an $n \times m$ matrix M can be found in $\tilde{O}(nm)$ time.*

We assume that the data structures described in Section 3 are constructed. We process each $\ell \in [m]$ independently, computing all interesting triplets of the form (u,d,ℓ). By Definition 21, such a triplet is interesting if the pair (u,d) is interesting for the tuple $\mathcal{S} = (S_1,\dots,S_n)$, where $S_i = M[i][\ell..m]$. Below we work with this fixed tuple $\mathcal{S}$. The algorithm scans $\mathcal{S}$ string by string; while processing S_i, the algorithm finds all the interesting pairs (i,j).

For $i < j \in [n]$, let $L(i,j)$ be the maximum LCP value between S_i and any S_k for $k \in [i+1\dots j]$. Let $I(i,j) = \min\{k \in [i+1\dots j] \mid \mathsf{LCP}(S_i,S_k) = L(i,j)\}$ be the minimum index k with this maximum LCP value. Using the function $I(i,j)$ we characterize the set of interesting pairs that share the first index i.

▶ **Lemma 23.** *For $i \in [n]$, let $j_1 > j_2 > \dots > j_z$ be the second coordinates of all interesting pairs of the form (i,j). Then $j_1 = I(i,n)$ and $j_k = I(i,j_{k-1}-1)$ for every $k \in [2..z]$.*

Proof. First we need to prove that $(i,I(i,n))$ is interesting and that there is no interesting pair (i,j') with $j' > I(i,n)$. By the definitions of $L(i,n)$ and $I(i,n)$, for every $j' < I(i,n)$ we have $\mathsf{LCP}(S_i,S_{j'}) < L(i,n) = \mathsf{LCP}(S_i,S_{I(i,n)})$, so $(i,I(i,j))$ is interesting. Now consider a pair (i,j') with $j' > I(i,n)$. The same definitions imply $\mathsf{LCP}(S_i,S_{j'}) \le L(i,n) = \mathsf{LCP}(S_i,S_{I(i,n)})$, so the pair $(S_i,S_{j'})$ is not interesting and we have $j_1 = I(i,n)$ as required.

Let $k \in [2..z]$ and consider the second statement. Similar to the above, we argue that the pair $(i,I(i,j_{k-1}-1))$ is interesting and no pair (i,j') such that $I(i,j_{k-1}-1) < j' < j_{k-1}$ is interesting. Hence $I(i,j_{k-1}-1)$ follows j_{k-1} in the list of second coordinates of interesting pairs of the form (i,j), i.e., $j_k = I(i,j_{k-1}-1)$. ◀

We proceed to show how to compute $I(i,j)$ and $L(i,j)$ efficiently.

▶ **Lemma 24.** *Given i and j, $L(i,j)$ can be computed in $O(\log n)$ time.*

Proof. Note that if we lex-sort the tuple $(S_i, \ldots, S_j)$, then the maximum LCP value with S_i would be reached by one of its neighbors $S_{j_{\mathsf{left}}}$ and $S_{j_{\mathsf{right}}}$ in the sorted tuple; we assume $S_{j_{\mathsf{left}}} < S_i < S_{j_{\mathsf{right}}}$ (one neighbor may absent). Thus, $L(i,j) = \max\{\mathsf{LCP}(S_i, S_{j_{\mathsf{left}}}), \mathsf{LCP}(S_i, S_{j_{\mathsf{right}}})\}$. The algorithm retrieves j_{left} and j_{right} using range queries on D^ℓ_{rows} as detailed below.

Recall that $I^{x,\ell}_{\mathsf{rows}}$ denotes the index of S_x in $\mathsf{LSA}^\ell_{\mathsf{rows}}$. Note that $I^{j_{\mathsf{right}},\ell}_{\mathsf{rows}}$ is the minimal index satisfying $I^{x,\ell}_{\mathsf{rows}} > I^{i,\ell}_{\mathsf{rows}}$ with $x \in [i+1..j]$. Hence, in order to get j_{right} one queries D^ℓ_{rows} for a point $(x, I^{x,\ell}_{\mathsf{rows}})$ in the range $[i+1..j] \times [I^{i,\ell}_{\mathsf{rows}}+1..\infty]$ that minimizes $I^{x,\ell}_{\mathsf{rows}}$; the first coordinate of this point is j_{right}. Symmetrically, in order to get j_{left} one queries D^ℓ_{rows} for a point $(x, I^{x,\ell}_{\mathsf{rows}})$ in the range $[i+1..j] \times [1..I^{i,\ell}_{\mathsf{rows}}-1]$ that maximizes $I^{x,\ell}_{\mathsf{rows}}$; the first coordinate of this point is j_{left}. After retrieving j_{right} and j_{left}, one queries the $\mathsf{LCP}^\ell_{\mathsf{rows}}$ structure for $\mathsf{LCP}(S_i, S_{j_{\mathsf{right}}})$ and $\mathsf{LCP}(S_i, S_{j_{\mathsf{left}}})$, and outputs the maximum as $L(i,j)$.

Two range queries take $O(\log n)$ time (Lemma 9 for $d = 2$) while two LCP queries take $O(1)$ time (Lemma 5). The lemma now follows. ◀

▶ **Lemma 25.** *Given i and j, $I(i,j)$ can be computed in $O(\log n)$ time.*

Proof. The algorithm starts by applying Lemma 24 to obtain $L(i,j)$ in $O(\log n)$ time. Let $P = S_i[1..L(i,j)]$ be the prefix of length $L(i,j)$ of S_i. Recall that by definition, $I(i,j)$ is the minimal index $k \in [i+1..j]$ such that $S_k[1..L(i,j)] = P$. Using Fact 7, the algorithm finds, in $O(\log n)$ time, a pair of indices i_P, j_P such that $S_z[1..L(i,j)] = P$ if and only if $I^{z,\ell}_{\mathsf{rows}} \in [i_P..j_P]$. After that, the algorithm retrieves $I(i,j)$ by querying D^ℓ_{rows} for the point $(k, I^{k,\ell}_{\mathsf{rows}})$ in the range $[i+1..j] \times [i_P..j_P]$ with the minimal first coordinate. This coordinate k is then reported as $I(i,j)$. As this query takes $O(\log n)$ time by Lemma 9 for $d = 2$, the lemma follows. ◀

Proof of Lemma 22. Let ℓ be fixed and $\mathcal{S} = \{S_1, \ldots, S_n\}$ be defined as above. For each S_i, the algorithm finds $j_1 = I(i,n)$ using Lemma 25, reports (i, j_1) as an interesting pair (see Lemma 23), and then iterate. As long as $j_k \neq i+1$, the algorithm finds $j_{k+1} = I(i, j_k - 1)$ using Lemma 25 and reports the interesting pair (i, j_{k+1}). Note that the algorithm is guaranteed to finish the iteration, since the pair $(i, i+1)$ is interesting.

The algorithm spends $O(\log n)$ time per interesting pair by Lemma 22; the number of such pairs is $O(n \log n)$ by Lemma 19. Multiplying this by m choices for ℓ, we obtain the required time bound $\tilde{O}(nm)$. ◀

6.3 Algorithm for the Decision Variant

In this section we prove Lemma 14, presenting the required algorithm.

The algorithm starts by modifying M as follows. For every $(i,j) \in [u_\square \ldots d_\square] \times [\ell_\square \ldots r_\square]$, we set $M[i][j] = \$_{i,j}$ with $\$_{i,j}$ being a unique symbol not in Σ. Since neither of the changed symbols belongs to a marginal row/column of a surrounding frame, this modification preserves surrounding matching frames. The following claim clarifies the role of interesting triplets.

▶ **Lemma 26.** *If a matrix M with an inner rectangle $(u_\square, d_\square, \ell_\square, r_\square)$ contains a surrounding matching frame (u, d, ℓ, r), then it contains a surrounding matching frame (u', d', ℓ, r) such that (u', d', ℓ) is an interesting triplet.*

Proof. Let (u, d, ℓ, r) be a surrounding matching frame in M. We denote $S_h = M[u][\ell..r] = M[d][\ell..r]$. Let u' be the maximal index in $[u..u_\square - 1]$ such that $M[u'][\ell..r] = S_h$ and let d' be the minimal index in $[d_\square + 1..d]$ such that $M[d'][\ell..r] = S_h$. The frame (u', d', ℓ, r) is surrounding by definition and matching by construction (note that $M[u..d][\ell] = M[u..d][r]$ implies $M[u'..d'][\ell] = M[u'..d'][r]$). Finally, for arbitrary $d'' \in [u'+1..d'-1]$ one has $M[d''][\ell..r] \neq S_h$. If $d'' < u_\square$ or $d'' > d_\square$, this condition holds by the choice of u' and d' respectively. Otherwise the condition is guaranteed by uniqueness of the symbols of the inner rectangle. Hence $\mathsf{LCP}(M[u'][\ell..m], M[d''][\ell..m]) < |S_h| \leq \mathsf{LCP}(M[u'][\ell..m], M[d'][\ell..m])$, and the triplet (u', d', ℓ) is interesting by definition. ◀

The Algorithm. After setting $M[i][j] = \$_{i,j}$ for each $(i,j) \in [u_\square \ldots d_\square] \times [\ell_\square \ldots r_\square]$, the algorithm applies the preprocessing described in Section 3 and finds all interesting triplets in $O(nm \log^2 n)$ time by applying Lemma 22. The final ingredient we need is a mechanism verifying, given an interesting triplet (u, d, ℓ), if there is a surrounding matching frame (u, d, ℓ, r). For this purpose, we present the following lemma.

▶ **Lemma 27.** *There is an algorithm that, given an interesting triplet (u, d, ℓ) of M, outputs an integer r such that (u, d, ℓ, r) is a surrounding matching frame or reports* null *if no such r exists. The algorithm runs in $O(\log n)$ time.*

Proof. The algorithm reports null if $u \geq u_\square$, or $d \leq d_\square$, or $\ell \geq \ell_\square$. Otherwise, it seeks for a value r such that (i) $r \geq r_\square + 1$, (ii) $M[u][\ell..r] = M[d][\ell..r]$, and (iii) $M[u..d][r] = M[u..d][\ell]$.

The algorithm queries $\mathsf{LCP}^{\ell}_{\mathsf{rows}}$ for $L_{u,d} = \mathsf{LCP}(M[u][\ell..m], M[d][\ell..m])$. By definition of LCP, we have $M[u][\ell..r] = M[d][\ell..r]$ if and only if $r \leq \ell + L_{u,d} - 1$. Hence, conditions (i) and (ii) are satisfied if and only if $r \in [r_\square + 1 \ldots \ell + L_{u,d} - 1]$. To check (iii), let $S_v = M[u..d][\ell]$. Using Fact 7, the algorithm finds the pair of indices i_v, j_v such that $M[u..d][r] = S_v$ if and only if $r \in \mathsf{LSA}^u_{\mathsf{columns}}[i_v..j_v]$. Now the algorithm checks the existence of a value r satisfying (i)–(iii) by querying D^u_{columns} for a point within the range $[r_\square + 1..\ell + L_{u,d} - 1] \times [i_v..j_v]$. If the queried structure returns a point $(r, I^{u,r}_{\mathsf{columns}})$, the algorithm outputs r; otherwise, it reports null, as there is no value of r such that (u, d, ℓ, r) is a surrounding matching frame.

The algorithm performs a single LCP query ($O(1)$ time by Lemma 5), finds i_v and j_v ($O(\log n)$ time by Fact 7), queries D^u_{columns} ($O(\log n)$ time by Lemma 9), and compares a constant number of integers. The lemma follows. ◀

We are finally ready to prove Lemma 14.

Proof of Lemma 14. After finding all interesting triplets, the algorithm applies the subroutine from Lemma 27 to every interesting triplet (u, d, ℓ). If this subroutine outputs r, the algorithm outputs the surrounding matching frame (u, d, ℓ, r). If the subroutine outputs null for all interesting triplets, then, relying on Lemma 26, the algorithm reports that no surrounding matching frame exists.

The algorithm spends $O(nm \log^2(nm))$ for each of three tasks it performs: preprocessing (Section 3), finding interesting triplets (Lemma 22), and verifying interesting triplets (Lemma 19 and Lemma 27). Thus, its time (and therefore, space) complexity is $\tilde{O}(nm)$, as required. ◀

References

1 Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

2 Amihood Amir and Gary Benson. Two-dimensional periodicity and its applications. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 440–452, 1992.

3 Amihood Amir and Gary Benson. Two-dimensional periodicity in rectangular arrays. *SIAM Journal on Computing*, 27(1):90–106, 1998.

4 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA*, volume 144 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

5 Amihood Amir, Gad M Landau, Shoshana Marcus, and Dina Sokol. Two-dimensional maximal repetitions. *Theoretical Computer Science*, 812:49–61, 2020.

6 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

7 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and lempel-ziv parsing. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.

8 Robert Berger. *The undecidability of the domino problem.* Amer. Math. Soc., 1966.

9 Itai Boneh, Dvir Fried, Shay Golan, Matan Kraus, Adrian Miclaus, and Arseny Shur. Searching 2d-strings for matching frames. *arXiv preprint arXiv:2310.02670*, 2023.

10 Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic length in linear time. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.23.

11 Sarah Cannon, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Matthew J. Patitz, Robert T. Schweller, Scott M. Summers, and Andrew Winslow. Two hands are better than one (up to constant factors): Self-assembly in the 2HAM vs. aTAM. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 172–184. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPICS.STACS.2013.172.

12 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate circular pattern matching. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022*, volume 244 of *LIPIcs*, pages 35:1–35:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ESA.2022.35.

13 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. The number of repetitions in 2d-strings. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA*, volume 173 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.32.

14 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.

15 Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007. doi:10.1016/j.ipl.2006.08.002.

16 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.63.

17 Jonas Ellert, Pawel Gawrychowski, and Garance Gourdel. Optimal square detection over general alphabets. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*, pages 5220–5242. SIAM, 2023. doi:10.1137/1.9781611977554.ch189.

18 Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

19 Guilhem Gamard, Gwénaël Richomme, Jeffrey O. Shallit, and Taylor J. Smith. Periodicity in rectangular arrays. *Inf. Process. Lett.*, 118:58–63, 2017. doi:10.1016/J.IPL.2016.09.011.

20 Younan Gao, Meng He, and Yakov Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 54:1–54:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.54.

21 Pawel Gawrychowski, Samah Ghazawi, and Gad M. Landau. Lower bounds for the number of repetitions in 2d strings. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE*, volume 12944 of *Lecture Notes in Computer Science*, pages 179–192. Springer, 2021. doi:10.1007/978-3-030-86692-1_15.

22 Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznanski. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. doi:10.1007/S00453-019-00591-8.

23 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

24 Emmanuel Jeandel and Michaël Rao. An aperiodic set of 11 Wang tiles. *CoRR*, abs/1506.06492, 2015. arXiv:1506.06492.

25 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.

26 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 2003. doi:10.1007/3-540-44888-8_14.

27 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

28 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. doi:10.1016/j.jda.2004.08.002.

29 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

30 Manasi S Kulkarni and Kalpana Mahalingam. Two-dimensional palindromes and their properties. In *International Conference on Language and Automata Theory and Applications*, pages 155–167. Springer, 2017.

31 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.

32 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990*, pages 319–327. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

33 Oleg Merkurev and Arseny M. Shur. Searching runs in streams. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2019. doi:10.1007/978-3-030-32686-9_15.

34 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In James A. Storer and Michael W. Marcellin, editors, *2009 Data Compression Conference (DCC 2009)*, pages 193–202. IEEE Computer Society, 2009.

35 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using d-critical substrings. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 54–67. Springer, 2009.

36 Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 459–468. ACM, 2000. doi:10.1145/335305.335358.

37 Mikhail Rubinchik and Arseny M. Shur. Palindromic k-factorization in pure linear time. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 81:1–81:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.MFCS.2020.81.

38 Taylor Smith. Properties of two-dimensional words. Master's thesis, University of Waterloo, 2017.

39 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

40 Hao Wang. Proving theorems by pattern recognition II. *Bell System Tech. J.*, 40:1–41, 1961.

41 Hao Wang. Games, logic and computers. *Scientific American*, 213(5):107, 1965.

42 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

43 Dan E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14(1):232–253, 1985. doi:10.1137/0214019.