Computing Covers of 2D-Strings

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Israel

Jakub Radoszewski
University of Warsaw, Poland

Wojciech Rytter
University of Warsaw, Poland

Tomasz Waleń
University of Warsaw, Poland

Wiktor Zuba
University of Warsaw, Poland

Abstract

We consider two notions of covers of a two-dimensional string T. A (rectangular) subarray P of T is a 2D-cover of T if each position of T is in an occurrence of P in T. A one-dimensional string S is a 1D-cover of T if its vertical and horizontal occurrences in T cover all positions of T. We show how to compute the smallest-area 2D-cover of an $m \times n$ array T in the optimal $\mathcal{O}(N)$ time, where $N = mn$, all aperiodic 2D-covers of T in $\mathcal{O}(N \log N)$ time, and all 2D-covers of T in $N^{4/3} \cdot \log^{\mathcal{O}(1)} N$ time. Further, we show how to compute all 1D-covers in the optimal $\mathcal{O}(N)$ time. Along the way, we show that the Klee's measure of a set of rectangles, each of width and height at least $\sqrt{n}$, on an $n \times n$ grid can be maintained in $\sqrt{n} \cdot \log^{\mathcal{O}(1)} n$ time per insertion or deletion of a rectangle, a result which could be of independent interest.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases 2D-string, cover, dynamic Klee's measure problem

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.12

Funding *Panagiotis Charalampopoulos*: Supported by the Israel Science Foundation grant 592/17.
Jakub Radoszewski: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.
Tomasz Waleń: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.
Wiktor Zuba: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

1 Introduction

We say that a string C is a *cover* of a string S if each position of S is inside an occurrence of C in S. In other words, S can be obtained from several copies of C by concatenations with possible overlaps. As an example, string *abaababaabaaba* has proper covers *aba* and *abaaba*. The shortest cover and all covers of a string can be computed in linear time; see [2, 7] and [16, 17], respectively.

We consider two notions of covers of 2D-strings, 1D-covers and 2D-covers. We say that a 2D-string C is a *2D-cover* of a 2D-string T if each position of T is inside an occurrence of C in T. For an example, see Figure 1. Let T be an $m \times n$ 2D-string with $N = m \cdot n$. An $\mathcal{O}(N^2)$-time algorithm for computing all 2D-covers of T and an $\mathcal{O}(N)$ *average*-time algorithm for computing the smallest-area 2D-cover of T were shown in [19]. They also present applications of the 2D-covers problem. The problem was also mentioned recently in the context of string recovery in [1].

© Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba;
licensed under Creative Commons License CC-BY 4.0
32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya; Article No. 12; pp. 12:1–12:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An $\mathcal{O}(N)$-time algorithm for computing the 2D-covers of T that are of a square shape $n \times n$, in the case that T is a square matrix itself, was shown in [11]. Let us note that there is a big difference between square-shaped 2D-covers and rectangular 2D-covers: we have only $O(n)$ square-shaped covers of an $n \times n$ 2D-string, while there may be $\Omega(n^2)$ distinct rectangular 2D-covers. This makes the rectangular case much harder.

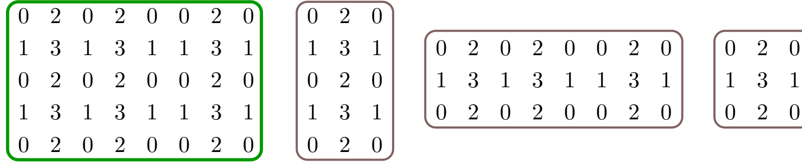

Figure 1 A 2D-string and its proper 2D-covers (the first one is vertically periodic, the two others are aperiodic).

We further say that a (1D) string C is a *1D-cover* of a 2D-string T if each position of T is inside an occurrence of C in a row, read from left to right, or in a column of T, read top-down. For an example, see Figure 2.

Let us define the two types of covers of 2D-strings more formally. A subarray $T[i\mathinner{.\,.}j, i'\mathinner{.\,.}j']$ is called a *2D-substring* of T. For an $m' \times n'$ 2D-string S, we denote:

$$Occ(S,T) = \{(i,j) : T[i\mathinner{.\,.}i+m'-1, j\mathinner{.\,.}j+n'-1] = S\}, \text{ and}$$
$$Cov(S,T) = \bigcup\{[i\mathinner{.\,.}i+m'-1] \times [j\mathinner{.\,.}j+n'-1] : (i,j) \in Occ(S,T)\}.$$

▶ **Definition 1** (2D-cover). *A 2D-string S is a 2D-cover of an $m \times n$ 2D-string T if $Cov(S,T) = [1\mathinner{.\,.}m] \times [1\mathinner{.\,.}n]$.*

A 1D-string (or simply a string) can be considered as a 2D-string of height 1. We denote by $\mathsf{trans}(S)$ the transpose of a 2D-string S. If S is a 1D-string, then $\mathsf{trans}(S)$ is a single column. We say that a position (i,j) of a 2D-string T is *horizontally covered* by a 1D-string S if $(i,j) \in Cov(S,T)$, and it is *vertically covered* by S if $(i,j) \in Cov(\mathsf{trans}(S),T)$.

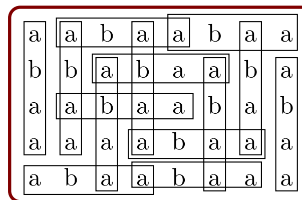

Figure 2 The string *abaa* is a 1D-cover of this 2D-string, i.e., its occurrences as a horizontal strip, read left-to-right, and as a vertical strip, read top-down, cover the 2D-string. Note that *aaba* is not a 1D-cover here.

▶ **Definition 2** (1D-cover). *A 1D-string S is a 1D-cover of a 2D-string T if each position of T is covered horizontally or vertically by S.*

When restricted to 2D-strings T of height 1, i.e. 1D-strings, both definitions yield the standard definition of a cover of a string.

Our Results. For an $m \times n$ 2D-string T of size $N = mn$ over an integer alphabet, we show the following:

- The smallest-area 2D-cover of T can be computed in $\mathcal{O}(N)$ time.
- All aperiodic 2D-covers of T can be computed in $\mathcal{O}(N \log N)$ time.
- All 2D-covers of T can be computed in $\tilde{\mathcal{O}}(N^{4/3})$ time, or in $\tilde{\mathcal{O}}(N\sqrt{\max(m,n)})$ time.[1]
- All 1D-covers of T can be computed in $\mathcal{O}(N)$ time.

The results concerning 2D- and 1D-covers can be found in Sections 3 and 5 and in Section 6, respectively. First, in Section 2 we show several connections between covers and periodicity in 2D. In the intermediate Section 4 we show a solution to an auxiliary geometric problem that employs a solution to dynamic Klee's measure for *fat* rectangles in a grid of size N and is the cornerstone of our algorithm for computing all 2D-covers.

2 Preliminaries

Let T be an $m \times n$ 2D-string of height m and width n, $T = T[1\,..\,m, 1\,..\,n]$. We write $m = \mathsf{height}(T)$ and $n = \mathsf{width}(T)$ and say that $N = mn$ is the *size* of T, which we denote as $\mathsf{size}(T) = N$. We assume that the 2D-string T for which covers are to be computed is over an integer alphabet $[1\,..\,N^{\mathcal{O}(1)}]$.

For an $m \times n$ 2D-string T, by $\mathsf{hstr}(T)$ we denote a length-n 1D-string over alphabet $[1\,..\,n]$ such that $\mathsf{hstr}(T)[i] = \mathsf{hstr}(T)[j]$ if and only if the i-th and j-th columns of T are equal. Similarly we define $\mathsf{vstr}(T)$, a length-m vertical string over alphabet $[1\,..\,m]$, by taking rows instead of columns. Let us denote by $T^{(h)}$, $T^{<h>}$, $T_{(w)}$ the 2D-substrings consisting of the first h rows, the last h rows, and the first w columns of T, respectively.

▶ **Lemma 3.** *For an $m \times n$ 2D-string T, strings $\mathsf{hstr}(T^{(h)})$, $\mathsf{hstr}(T^{<h>})$, $\mathsf{vstr}(T_{(w)})$ for all h, w can be computed in $\mathcal{O}(N)$ time.*

Proof. We consider computing all strings $\mathsf{hstr}(T^{(h)})$ for $h = 1, \ldots, m$; the other cases are symmetric. First of all, we renumber consistently the characters in each row of T separately so that they are in $[1\,..\,n]$. This can be done in $\mathcal{O}(N)$ time using one global radix sort. Thus we have already computed $\mathsf{hstr}(T^{(1)})$. Assume now we have computed $\mathsf{hstr}(T^{(h-1)})$. Then we compose a 2D-string T' of height 2, the first row is $\mathsf{hstr}(T^{(h-1)})$, the second one is the h-th row of T. It is enough now to encode consistently the columns of T', which can be done in $\mathcal{O}(n)$ time using radix sort as each column of T' is a pair of integers in $[1\,..\,n]$. ◀

Let us introduce a function Is2DCover(X, Y) which tests if a 2D-string X is a 2D-cover of a 2D-string Y. In [19], a 2D pattern matching algorithm [3, 6] and 2D dynamic programming were used to show the following result.

▶ **Lemma 4** ([19]). Is2DCover(X, Y) *can be computed in $\mathcal{O}(\mathsf{size}(Y))$ time.*

A string B is a *border* of a string S if and only if B is both a prefix and a suffix of S. It is readily verified that each cover of S is a border of S. A similar relation holds for covers of 2D-strings. A *2D-border* of a 2D-string T is a 2D-string U that occurs in each of the four corners of T. A *1D-border* of a 2D-string is a 1D-string which is a prefix of the first row or the first column of T and also a suffix of the last row or the last column of T; see Figure 3.

▶ **Observation 5.** *A 2D-/1D-cover of a 2D-string T is also a 2D-/1D-border of T.*

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylogarithmic factors.

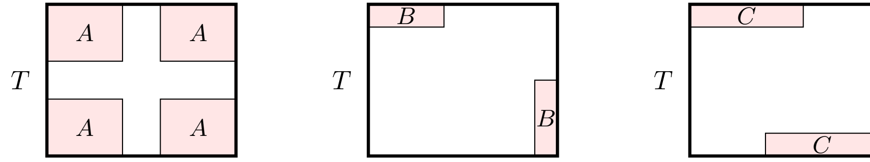

Figure 3 A is a 2D-border of T and B, C are 1D-borders of T (the cases when the top-left corner is covered horizontally; there are two other cases).

All 1D-borders of a 2D-string can be computed in linear time using the Knuth-Morris-Pratt (KMP) algorithm [14] as borders of strings of the form $X\#Y$, for a sentinel letter $\#$ that is not in the alphabet, where X and Y are the first row or column and the last row or column of T, respectively. Moreover, all 2D-borders of a 2D-string can be computed in linear time, as shown in the following lemma (which works for any alphabet with constant-time letter comparisons).

▶ **Lemma 6** ([11, Theorem 3.2]). *All 2D-borders of a 2D-string T of size N can be computed in $\mathcal{O}(N)$ time.*

We say that a string S of length $|S|$ has period p if $S[i] = S[i+p]$ for all $i = 1, \ldots, |S|-p$. By $\mathsf{per}(S)$ we denote the smallest period of S. String S is called *periodic* if $2 \cdot \mathsf{per}(S) \leq |S|$ and *aperiodic* otherwise. Let us state the periodicity lemma, one of the most classical results in combinatorics on strings.

▶ **Lemma 7** (Periodicity Lemma (weak version) [13]). *If a string S has periods p and q such that $p + q \leq |S|$, then $\mathsf{gcd}(p, q)$ is also a period of S.*

We say that p is a vertical (resp. horizontal) period of a 2D-string T if it is a period of each column (resp. row) of T. We denote the smallest vertical and horizontal periods of T by $\mathsf{vper}(T)$ and $\mathsf{hper}(T)$, respectively. A 2D-string T is called *periodic* if $2 \cdot \mathsf{vper}(T) \leq \mathsf{height}(T)$ or $2 \cdot \mathsf{hper}(T) \leq \mathsf{width}(T)$, and *aperiodic* otherwise.

The shortest cover of a string is aperiodic [2]. A similar observation can be made in 2D.

▶ **Observation 8.** *The shortest 1D-cover and the smallest-area 2D-cover of a 2D-string are aperiodic.*

▶ **Lemma 9.** *For two 2D-borders of T of widths w and w' with $w < w' \leq \frac{3}{2}w$, the one with the smaller height is horizontally periodic. Symmetrically, for two 2D-borders of heights h and h' with $h < h' \leq \frac{3}{2}h$, the one with smaller width is vertically periodic.*

Proof. Both 2D-borders occur in the top-left and the top-right corners. This means that, for h equal to the minimum of the heights of the two 2D-borders, $\mathsf{hstr}(T^{(h)})[1\mathinner{.\,.} w]$ appears both as prefix and as suffix of $\mathsf{hstr}(T^{(h)})[1\mathinner{.\,.} w']$.

Hence, $p = w' - w \leq \frac{w}{2}$ is a horizontal period of $T[1\mathinner{.\,.} h, 1\mathinner{.\,.} w']$. Then, p is also a horizontal period of the 2D-border of height h, as this 2D-border is a 2D-substring of $T[1\mathinner{.\,.} h, 1\mathinner{.\,.} w']$. For the symmetric statement, it suffices to transpose all involved 2D-strings. ◀

3 Computing aperiodic 2D-Covers

We first consider computation of the smallest 2D-cover, which is also (automatically) aperiodic and of all aperiodic 2D-covers. Later, in Section 5, we consider the more involved computation of all 2D-covers.

3.1 Computing Smallest 2D-Cover in Linear Time

A 2D-substring of height h and width w is called a *candidate* if it is the smallest-area 2D-cover of height h of both $T^{(h)}$ and $T^{<h>}$, and the smallest-area 2D-cover of width w of $T_{(w)}$; see Figure 4. Note that there is at most one candidate for each height h: the smallest-area 2D-cover of $T^{(h)}$ of height h. Further, note that each candidate is a 2D-border of T.

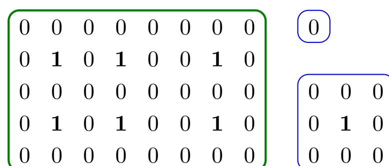

Figure 4 This 2D-string has two candidates, shown on the right. The 3×3 candidate is the smallest-area 2D-cover.

Correctness of the algorithm is based on the following lemma.

▶ **Lemma 10.** *If X, Y are candidates and $\mathsf{height}(X) < \mathsf{height}(Y)$, then*
(a) $\mathsf{width}(X) \le \mathsf{width}(Y)$*;* **(b)** $\mathsf{height}(Y) > \frac{3}{2}\mathsf{height}(X)$*;*
(c) *if X is a 2D-cover of T, then it is also a 2D-cover of Y.*

Proof. If we had $\mathsf{width}(Y) < \mathsf{width}(X)$, then $T^{(\mathsf{height}(X))}$ would have a 2D-cover of height $\mathsf{height}(X)$ and width $\mathsf{width}(Y)$ (composed of the first $\mathsf{height}(X)$ rows of Y), so X would not be a candidate for its height. This proves part (a).

Now, recall that X and Y are 2D-borders of T. If we had $\mathsf{height}(Y) \le \frac{3}{2}\mathsf{height}(X)$, then point (a) and Lemma 9 would imply that X is vertically periodic. Hence, X would not be the smallest-area 2D-cover of $T_{(\mathsf{width}(X))}$ of width $\mathsf{width}(X)$ (Observation 8).

Part (c) follows from the fact that Y is a 2D-border of T, similarly to the fact that a cover C of a 1D-string S is a cover of every border B of S that satisfies $|C| \le |B|$; see [2]. ◀

▶ Remark 11. Lemma 10 (a) implies that there is exactly one smallest-area 2D-cover of T. In particular, [19] also considered the problem of computing an $h \times w$ 2D-cover that is minimal in terms of $h + w$ or $\max(h, w)$, and our algorithm below solves these variants as well.

Algorithm 1 Smallest 2D-cover.

```
CAND := the set of candidates;
X := the element of CAND of smallest height;
remove X from CAND;
foreach Y in CAND, in increasing order of heights do
    if not Is2DCover(X, Y) then X := Y;
return X;
```

▶ **Theorem 12.** *The smallest-area 2D-cover of a 2D-string T of size N can be computed in $\mathcal{O}(N)$ time.*

Proof. We linearize in $\mathcal{O}(N)$ time all 2D-strings $T^{(h)}$, $T^{<h>}$, $T_{(w)}$ using Lemma 3, and compute their shortest covers in the sense of 1D-strings. Then $T[1\,..\,h, 1\,..\,w]$ is a candidate if and only if the shortest covers of $\mathsf{hstr}(T^{(h)})$, $\mathsf{hstr}(T^{<h>})$ are of length w and the shortest cover of $\mathsf{vstr}(T_{(w)})$ is of length h.

We proceed as in Algorithm 1. The most expensive part of the algorithm is the operation Is2DCover(X, Y). It costs $\mathcal{O}(\mathsf{size}(Y))$ time. The sum of these costs is linear since the sizes of Y are geometrically growing (in each step by a factor at least 3/2) due to points (a) and (b) of Lemma 10. The largest among them has size $\mathcal{O}(N)$.

The correctness of the algorithm follows from point (c) of Lemma 10. More precisely, an invariant holds that among all the candidates that were considered in the foreach-loop up to a given point, only X can be the smallest-area 2D-cover of T. Indeed, if Is2DCover(X, Y) is true, then Y is not the smallest-area 2D-cover because it has a 2D-cover itself, and otherwise X cannot be a 2D-cover of T by point (c) of Lemma 10. ◀

3.2 Computing All Aperiodic 2D-Covers

We start with a tight asymptotic bound on the number of aperiodic 2D-covers.

▶ **Lemma 13.** *A 2D-string T of size N has $\mathcal{O}(\log^2 N)$ distinct aperiodic 2D-covers. Moreover, there is an infinite family of binary 2D-strings with $\Omega(\log^2 N)$ distinct aperiodic 2D-covers.*

Proof. Lemma 9 implies that there are $\mathcal{O}(\log^2 N)$ distinct aperiodic 2D-borders of T: at most one of height in $[(\frac{3}{2})^i \mathinner{.\,.} (\frac{3}{2})^{i+1})$ and width in $[(\frac{3}{2})^j \mathinner{.\,.} (\frac{3}{2})^{j+1})$ for each pair of non-negative integers i, j. The same bound applies to 2D-covers due to Observation 5.

For $i = 1, 2$, let $\mathcal{C}_i$ be the set of lengths of covers of a string S_i of length n_i over alphabet Σ_i. Then, an $n_1 \times n_2$ 2D-string T over alphabet $\Sigma_1 \times \Sigma_2$ defined as $T[i, j] = (S_1[i], S_2[j])$ has 2D-covers of all dimensions in $\mathcal{C}_1 \times \mathcal{C}_2$.

A binary Fibonacci string of length n has $\Theta(\log n)$ aperiodic covers [10]. Hence, with the above construction we can obtain a 2D-string of size $N = n^2$ over the alphabet with 4 letters with $\Omega(\log^2 N)$ aperiodic 2D-covers. We can encode each letter by its 2-digit binary representation, obtaining a binary 2D-string with $\Omega(\log^2 N)$ aperiodic 2D-covers. ◀

A direct application of the bound from Lemma 13 and the Is2DCover routine would yield an $\mathcal{O}(N \log^2 N)$-time algorithm. The theorem below, whose proof can be found in Appendix B, shaves a $\log N$ factor from this complexity. Let us note that it uses the same order of inspecting candidates as the algorithm Simple-All-2D-covers below.

▶ **Theorem 14.** *All aperiodic 2D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N \log N)$ time.*

4 Rectangle Cover Problem

In the Klee's measure problem in 2D, we are given M rectangles in the plane and are to output the area of the union of these rectangles. This problem can be solved in $\mathcal{O}(M \log M)$ time in the offline setting [4, 8]. In the dynamic variant of the problem, where rectangles can be inserted and deleted, Klee's measure can be maintained in $\tilde{\mathcal{O}}(\sqrt{M})$ time per update [18]. Below, we consider a special version of the Klee's measure problem with *fat* rectangles in a grid of size bounded by N and use it to solve an auxiliary problem called Rectangle Cover. Details omitted due to space constraints can be found in Appendix A.

For a set $\mathcal{F}$ of rectangles let us denote by $\mathcal{F}_{h,w}$ the subset of rectangles with height at least h and width at least w.

Rectangle Cover
Input: A set $\mathcal{F}$ consisting of M rectangles in $[0 \mathinner{.\,.} m] \times [0 \mathinner{.\,.} n]$, where $N = m \cdot n$, $n \geq m$.
Output: All pairs (h, w) (called *good* pairs) for which $|\bigcup \mathcal{F}_{h,w}| = N$.

We will reduce Rectangle Cover to a restricted variant of the following problem.

Dynamic Klee's Measure
Input: An initially empty set of rectangles $\mathcal{R}$ in $[0\,..\,m] \times [0\,..\,n]$; $N = mn$, $n \geq m$.
Updates: Insert or delete a rectangle to $\mathcal{R}$ and return the area of $\bigcup \mathcal{R}$.

We start by a direct reduction that we then refine in Lemma 16.

▶ **Lemma 15.** *The Rectangle Cover problem reduces in $\mathcal{O}(M+N)$ time to the Dynamic Klee's Measure problem with a grid of the same dimensions and $\mathcal{O}(M)$ updates.*

Proof. Note that if (h, w) is a good pair, then $(h-1, w)$ and $(h, w-1)$ are good pairs as well. Hence, we will compute for each h, the maximum w for which (h, w) is good, as shown in Algorithm 2. We use the following equality and a symmetric one.

$$\mathcal{F}_{h,w} = \mathcal{F}_{h,w+1} \cup \{X \in \mathcal{F} \,:\, \mathsf{width}(X) = w,\, \mathsf{height}(X) \geq h\}.$$

Algorithm 2 Rectangle Cover via Dynamic Klee's Measure.

```
w := n + 1; R := ∅;
for h := 1 to m do
    while w > 0 and |⋃ R| ≠ N do
        // Invariant: R = F_{h,w}
        w := w − 1;
        R := R ∪ {X ∈ F : width(X) = w, height(X) ≥ h};
    Report (h, 1), . . . , (h, w);
    R := R \ {X ∈ F : height(X) = h};
```

Let us note that each rectangle from $\mathcal{F}$ is inserted at most once to $\mathcal{R}$ and deleted at most once from $\mathcal{R}$. The condition in the while-loop can be checked by computing $\mathcal{R}$'s Klee's measure. Rectangles that are to be inserted to $\mathcal{R}$ and deleted from $\mathcal{R}$ in subsequent steps can be easily determined if all rectangles in $\mathcal{F}$ are pre-sorted by their widths and heights, via bucket sort, in $\mathcal{O}(M+N)$ time. ◀

A rectangle will be called a **fat rectangle** if both its width *and* height are at least $\sqrt{n}$.

▶ **Lemma 16.** *The Rectangle Cover problem reduces in $\tilde{\mathcal{O}}(M \min\{m, \sqrt{n}\} + N)$ time to the Dynamic Klee's Measure problem on a grid of the same dimensions and $\mathcal{O}(M)$ insertions and deletions of fat rectangles.*

Proof. For every $h \leq \min\{m, \sqrt{n}\}$, we compute the maximum w such that (h, w) is a good pair using binary search. In order to test a candidate (h, w), we compute the area of the union of rectangles from $\mathcal{F}_{w,h}$ (i.e., solve static Klee's measure problem in 2D) using one of the classic $\mathcal{O}(M \log M)$-time approaches [4, 8]. If $m > \sqrt{n}$, the same approach is then used for every $w \leq \sqrt{n}$. Finally, for $h, w > \sqrt{n}$ we use Algorithm 2 from the proof of Lemma 15. More precisely, we start the for-loop with $h = \lceil \sqrt{n} \rceil$ and break when w drops below $\lceil \sqrt{n} \rceil$. Thus, the set $\mathcal{R}$ contains only fat rectangles throughout the execution of the algorithm. ◀

A proof of the following lemma can be found in Appendix A.

▶ **Lemma 17.** *The Dynamic Klee's Measure problem with fat rectangles can be solved in $\tilde{\mathcal{O}}(\sqrt{n})$ time per operation, after $\tilde{\mathcal{O}}(N)$-time preprocessing.*

▶ **Lemma 18.** *The* Rectangle Cover *problem can be solved in* $\tilde{\mathcal{O}}(M \cdot \min\{m, \sqrt{n}\} + N)$ *time.*

Proof. We use Lemma 16 to reduce the problem to Dynamic Klee's Measure problem on fat rectangles with $\mathcal{O}(M)$ updates in $\tilde{\mathcal{O}}(M \cdot \min\{m, \sqrt{n}\} + N)$ time, and Lemma 17 to solve the latter problem with $\tilde{\mathcal{O}}(\sqrt{n})$ update time. Note that the Dynamic Klee's Measure problem is void if $m \leq \sqrt{n}$ (no fat rectangle exists). ◀

▶ Remark 19. To solve the Rectangle Cover problem it is not necessary to compute the exact area of union of the fat rectangles but just to check if they cover the whole grid. This would slightly simplify the solution, as shown in Appendix A.1, but the main idea would stay intact. We decided to state the auxiliary problem as a variant of dynamic Klee's measure since it could find other applications.

5 Computing All 2D-Covers

Let us start with simple but less efficient algorithms. We say that (h, w) is a weak candidate if $T[1 \mathinner{.\,.} h, 1 \mathinner{.\,.} w]$ is a 2D-cover of $T^{(h)}$ and of $T_{(w)}$. Let us denote by L_h the sorted list of widths of weak candidates of height h.

For a list L of integers we denote by $cut(L, y)$ the list of elements $x \in L$ such that $x \leq y$. We next present two preliminary algorithms for computing all 2D-covers of T.

Algorithm 3 Simple-All-2D-covers.

```
Result := ∅;
for h := 1 to m do
    foreach z ∈ L_h, in decreasing order do
        if Is2DCover(T[1..h, 1..z], T) then
            Result := Result ∪ {(h, w) : w ∈ cut(L_h, z)}; break;
        else remove z from all lists L_t;
return Result;
```

Algorithm 4 Binary-Search-All-2D-covers.

```
Result := ∅;
for h := 1 to m do
    // Binary search with O(log n) instances of Is2DCover
    z := max ({w ∈ L_h : Is2DCover(T[1..h, 1..w], T)} ∪ {0}));
    if z > 0 then Result := Result ∪ {(h, w) : w ∈ cut(L_h, z)};
return Result;
```

▶ **Proposition 20.** *The algorithm Simple-All-2D-covers outputs all 2D-covers of* T *in* $\mathcal{O}(N^{3/2})$ *time if* T *is a square matrix, and* $\mathcal{O}(N \max(m, n))$ *time in general. The algorithm Binary-Search-All-2D-covers computes all 2D-covers in* $\mathcal{O}(N^{3/2} \log N)$ *time.*

Proof. In the algorithm Simple-All-2D-covers in each column we make at most one negative test of Is2DCover and in each row at most one positive test. Hence, the time complexity is $\mathcal{O}(N \max(m, n))$.

We run the algorithm Binary-Search-All-2D-covers for T if $m \le n$ and otherwise for T transposed. For each h we execute $\mathcal{O}(\log n)$ instances of Is2DCover, which require time $\mathcal{O}(N \log n)$ in total for a given h. Summing over all heights, this yields $\mathcal{O}(Nm \log n) = \mathcal{O}(N^{3/2} \log N)$ time. ◀

We proceed with faster computation of all 2D-covers. Henceforth let us assume w.l.o.g. that $n \ge m$. Let T be an $m \times n$ 2D-string of size $N = mn$. We say that a 2D-string U is an (x, y)-array if

$$\mathsf{height}(U) \in [(\tfrac{3}{2})^x \mathinner{..} (\tfrac{3}{2})^{x+1}) \text{ and } \mathsf{width}(U) \in [(\tfrac{3}{2})^y \mathinner{..} (\tfrac{3}{2})^{y+1}).$$

We call a 2D-border of T that is an (x, y)-array an (x, y)-border of T.

We say that a 2D-string U is of *periodic type* (p, q, a, b) if $\mathsf{vper}(U) = p$, $\mathsf{hper}(U) = q$, $\mathsf{height}(U) \bmod p = a$, and $\mathsf{width}(U) \bmod q = b$.

▶ **Lemma 21.** *For given x, y, all (x, y)-borders of T are of the same periodic type.*

Proof. Assume there are two (x, y)-borders U and V of T. Let us show that $\mathsf{vper}(U) = \mathsf{vper}(V)$. If $\mathsf{height}(U) = \mathsf{height}(V)$, then each column of U occurs as a column of V and vice versa. Hence, $\mathsf{vper}(U) = \mathsf{vper}(V)$. If $\mathsf{height}(U) \ne \mathsf{height}(V)$, then $p := |\mathsf{height}(U) - \mathsf{height}(V)| \in [1 \mathinner{..} \frac{1}{2} \cdot (\frac{3}{2})^x)$ is a vertical period of both U and V, so both are vertically periodic with period p (c.f. Lemma 9). Let $p' = \mathsf{vper}(U)$ and $p'' = \mathsf{vper}(V)$. Then $p' \le p$ and both p and p' are periods of each column of U. Hence, by the periodicity lemma (Lemma 7), $p' \mid p$. Similarly for p''. Hence, p' (p'') is the least common multiple of smallest periods of the set of length-p prefixes of columns of U (resp. V). The sets of length-p prefixes of columns of U and V are the same. We thus have $\mathsf{vper}(U) = p' = p'' = \mathsf{vper}(V)$. Moreover, p' divides $|\mathsf{height}(U) - \mathsf{height}(V)|$, which shows that $\mathsf{height}(U) \bmod p = \mathsf{height}(V) \bmod p$. The proof for the horizontal period is symmetric. ◀

Let us recall that all 2D-borders of T can be computed in $\mathcal{O}(N)$ time (Lemma 6). For each non-empty group of (x, y)-borders, we can compute the periodic type of one of its representatives U in $\mathcal{O}((\frac{3}{2})^{x+y})$ time using the KMP algorithm for each column and row. This gives $\mathcal{O}(N)$ time in total. Below we show how to compute all 2D-covers of T of a given periodic type (p, q, a, b) in $\tilde{\mathcal{O}}(N \cdot \min\{m, \sqrt{n}\})$ time.

Let us henceforth fix a periodic type (p, q, a, b) and denote $X = T[1 \mathinner{..} p + a, 1 \mathinner{..} q + b]$. We call X the *root* of all 2D-covers of this periodic type. We denote by V_X the set of top-left corners of occurrences of X in T. It can be computed using 2D pattern matching for X.

A grid of points $\{(u + ip, v + jq) : 0 \le i < \alpha, 0 \le j < \beta\}$ for any u, v will be called an (α, β)-*r-grid*, or simply an *r-grid*. If a 2D-string U of height h and width w has a periodic type (p, q, a, b), then its occurrence in T *implies* a $(\lfloor h/p \rfloor, \lfloor w/q \rfloor)$-grid subset of V_X. See Figure 5. Moreover, such an r-grid in V_X always *generates* an occurrence of U in T. An (α, β)-r-grid D in V_X is called *maximal* if there is no (α', β')-r-grid D' in V_X such that D is a proper subset of D'. The next lemma follows from [9, Section 5].

▶ **Lemma 22.** *The set of maximal r-grids in V_X can be computed in $\mathcal{O}(N)$ time.*

Every r-grid in V_X can be extended to a maximal r-grid in V_X. This suggests the following important observation.

▶ **Observation 23.** *A 2D-string of height h and width w and of periodic type (p, q, a, b) is a 2D-cover of T if and only if T is covered by substrings of T that are generated by maximal (α, β)-r-grids satisfying $\alpha \ge \lfloor h/p \rfloor, \beta \ge \lfloor w/q \rfloor$.*

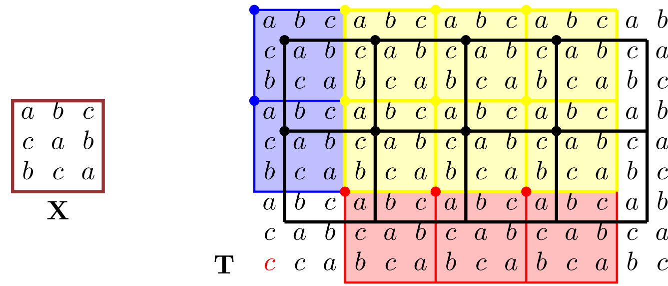

Figure 5 Let $(p, q, a, b) = (3, 3, 0, 0)$. Some (not all) occurrences of X in T are shown, with the corresponding elements of V_X drawn as dots. The $(2, 4)$-r-grid in the black frame is maximal, whereas the yellow $(2, 3)$-r-grid is not; it is a part of two maximal r-grids: blue+yellow and red+yellow.

From now on we will treat 2D-substrings generated by maximal r-grids in T as rectangles. The following lemma follows directly from the above discussion.

▶ **Lemma 24.** *The problem of computing all 2D-covers of T of a given periodic type reduces in $\mathcal{O}(N)$ time to an instance of* Rectangle Cover *with $M = \mathcal{O}(N)$ and the same N.*

▶ **Theorem 25.** *All 2D-covers of a 2D-string of size $N = m \times n$, $n \geq m$, can be computed in $\tilde{\mathcal{O}}(N \cdot \min\{m, \sqrt{n}\})$ time or $\tilde{\mathcal{O}}(N^{4/3})$ time.*

Proof. For each of the $\tilde{\mathcal{O}}(1)$ groups of (x, y)-borders, we apply Lemma 24 to reduce the problem in scope to Rectangle Cover in $\mathcal{O}(N)$ time, then we apply the algorithm from Lemma 18. In total we obtain $\tilde{\mathcal{O}}(N \cdot \min\{m, \sqrt{n}\})$ time, and we can use the fact that $\min\{m, \sqrt{n}\} \leq (m \cdot n)^{1/3} = N^{1/3}$. ◀

6 Computing 1D-Covers

Let us recall that a 1D-cover of an $m \times n$ 2D-string T must be a 1D-border of T (see Observation 5 and Figure 3). We will only show how to compute 1D-covers of T which are prefixes of the first row and suffixes of the last row. The three other cases can be treated analogously. Henceforth we denote $W = T[1, 1\,..\,n]\#T[m, 1\,..\,n]$, where # is some sentinel letter not in the alphabet.

6.1 An $\mathcal{O}(N \log N)$-Time Algorithm Computing All 1D-Covers

We will partition the 1D-borders in scope in $\mathcal{O}(\log n)$ groups based on their periods. Then, we will show how to process each group in $\mathcal{O}(N)$ time, relying on the following property: each element of a group G can be covered by each of the elements of G that are shorter than it. Hence, it will suffice to compute the longest element of each group that covers T.

For a (1D-)string S, let us denote by $\mathbf{B}(S)$ all borders of S which are either aperiodic or are the longest ones with a given smallest period. Let us denote by $\mathbf{A}(S)$ the set of all aperiodic borders of S; $\mathbf{A}(S) \subseteq \mathbf{B}(S)$.

▶ **Fact 26** ([12]). *The size of $\mathbf{B}(S)$ is $\mathcal{O}(\log |S|)$.*

By a *big border* of a string X, we mean X itself or any border Z of X such that $|Z| \geq 2 \cdot \mathsf{per}(X)$. Given a string S of length n, due to Fact 26, we can partition its borders into $\mathcal{O}(\log n)$ groups, such that the following holds for each group G: G is the set of big

borders of its longest element $X \in \mathbf{B}(S)$. We denote such a group by $\mathsf{Group}(X)$ and call $\mathsf{per}(X)$ the *period of the group*. In particular, for a string $X \in \mathbf{B}(S)$ of the form U^kV with $k \geq 2$, $\mathsf{per}(X) = |U|$ and $0 \leq |V| < \mathsf{per}(X)$, we have $\mathsf{Group}(X) = \{U^jV : j \in [2\mathinner{.\,.} k]\}$.

▶ **Observation 27.** *The defined groups form a partition of the set of borders of S.*

▶ **Example 28.** For $S = (abaab)^5a$, we have $\mathbf{B}(S) = \{a, aba, abaaba, S\}$ and $\mathsf{Group}(S) = \{(abaab)^ia : i = 2, \dots, 5\}$. The groups of the remaining elements of $\mathbf{B}(S)$ are singletons.

The following lemma will be used in Section 6.3, but we state it here as the required definitions have been already introduced. Its proof can be found in Appendix B.

▶ **Lemma 29.** *For $X, Y \in \mathbf{B}(S)$ with $|X| < |Y|$, we have $\mathsf{per}(Y) \geq \mathsf{per}(X) \cdot |\mathsf{Group}(X)|$. Moreover, for $X, Y, Z, W \in \mathbf{B}(S)$ with $|X| < |Y| < |Z| < |W|$, we have $\mathsf{per}(W) \geq \frac{9}{8}\mathsf{per}(X)$.*

Every 1D-cover of T that is a prefix of the first row of T and a suffix of the last row of T is in $\mathsf{Group}(X)$ for some $X \in \mathbf{B}(W)$. The following lemma gives us the monotonicity property that was mentioned before, which will allow us to handle each group efficiently.

▶ **Lemma 30.** *If some $Z \in \mathsf{Group}(X)$ for $X \in \mathbf{B}(W)$ is a 1D-cover of T, then all elements of $\mathsf{Group}(X)$ that are shorter than Z are also 1D-covers of T.*

Proof. If $|\mathsf{Group}(X)| > 1$, then X is periodic. Thus, the elements of $\mathsf{Group}(X)$ are of the form U^jV, where V is a (possibly empty) prefix of U and $j \geq 2$. It is then readily verified that $Z = U^kV$, and hence each of its occurrences in T, can be covered by U^jV for all $j \in [2\mathinner{.\,.} k)$. ◀

Let us introduce two arrays that are building blocks of the algorithm. For a family $\mathcal{F}$ of subintervals of $[1\mathinner{.\,.} n]$, for each $k \in [1\mathinner{.\,.} n]$, let us denote by $\mathcal{F}\text{-}Cov[k]$ the length of the longest interval in $\mathcal{F}$ containing k. If there is no such interval for a given k, then $\mathcal{F}\text{-}Cov[k] = 0$. The table $\mathcal{F}\text{-}Cov$ can be computed in $\mathcal{O}(|\mathcal{F}| + n)$ time by sorting the intervals using radix sort, and then performing a standard line sweeping algorithm.

For two strings X and Y, we define the table $LBB_{X,Y}$ such that $LBB_{X,Y}[i]$ is the length of the longest big border in $\mathsf{Group}(X)$ which starts at position i in Y; $LBB_{X,Y}[i] = 0$ if there is no such big border.

Algorithm 5 1D-Covers from a group.

Input: A 2D-string T and $X \in \mathbf{B}(W)$, where $W = T[1, 1\mathinner{.\,.} n]\#T[m, 1\mathinner{.\,.} n]$.
Output: The elements of $\mathsf{Group}(X)$ that cover T.

processing rows:

for $i := 1$ **to** m **do**
  $Y :=$ i-th row of T; $\mathcal{F} := \{\, [\, j\mathinner{.\,.} j + LBB_{X,Y}[j] - 1\,] \;:\; j \in [1\mathinner{.\,.} n]\}$;
  for $j := 1$ **to** n **do** $maxCov[i, j] := \mathcal{F}\text{-}Cov[j]$;

processing columns:

for $j := 1$ **to** n **do**
  $Y :=$ j-th column of T; $\mathcal{F} := \{\, [\, i\mathinner{.\,.} i + LBB_{X,Y}[i] - 1\,] \;:\; i \in [1\mathinner{.\,.} m]\}$;
  for $i := 1$ **to** m **do** $maxCov[i, j] := \max(maxCov[i, j], \mathcal{F}\text{-}Cov[i])$;

$min := \min\{maxCov[i, j] \;:\; 1 \leq i \leq m, 1 \leq j \leq n\}$;

return $\{Z \in \mathsf{Group}(X) \;:\; |Z| \leq min\}$;

▶ **Lemma 31.** *Given strings X and Y and integers $|X|$ and $\mathsf{per}(X)$, $LBB_{X,Y}$ can be computed in time $\mathcal{O}(|Y|)$.*

Proof. If X is not periodic, $\mathsf{Group}(X) = \{X\}$ and it suffices to find all occurrences of X in Y using the KMP algorithm. Otherwise, let $p = \mathsf{per}(X)$. We use the KMP algorithm to find all occurrences of $U := X[1\,..\,2p + (|X| \bmod p)]$ in Y. We compute the $LBB_{X,Y}$ array right-to-left. If there is an occurrence of U in Y starting at position i, then $LBB_{X,Y}[i] = \max(|U|, p + LBB_{X,Y}[i+p])$. Otherwise, $LBB_{X,Y}[i] = 0$. ◀

The algorithm processing a single group is specified in the pseudocode above. Each row and column of T is processed separately. The goal is to compute, for each position of T, the longest element of $\mathsf{Group}(X)$ that covers it, eventually stored in $maxCov[i,j]$. Then, by taking the minimum over all positions of T, due to Lemma 30, we can identify the elements of $\mathsf{Group}(X)$ that cover T. Each row/column Y is processed according to the following observation, in time proportional to its length due to efficient computation of $\mathcal{F}\text{-}Cov$ and LBB arrays (Lemma 31).

▶ **Observation 32.** *Let X and Y be two strings and k be an integer, such that $X \in \mathbf{B}(Y)$ and $k \in [i\,..\,i + LBB_{X,Y}[i] - 1]$. Then, all elements of $\{Z \in \mathsf{Group}(X) : |Z| \le LBB_{X,Y}[i]\}$ cover the position k in Y. If $[i\,..\,i + LBB_{X,Y}[i] - 1]$ is the maximal (over all i's) fragment in Y containing position k, then $LBB_{X,Y}[i]$ is the length of the longest element of $\mathsf{Group}(X)$ covering position k in Y.*

▶ **Lemma 33.** *Algorithm 5 computes all the 1D-borders from a single group $\mathsf{Group}(X)$ that cover T in $\mathcal{O}(N)$ time.*

We thus obtain an $\mathcal{O}(N \log N)$-time algorithm for the problem in scope. We first compute all borders of W and organize them in groups $\mathsf{Group}(X)$ for $X \in \mathbf{B}(W)$ in $\mathcal{O}(n)$ time. Then, we process each group separately, obtaining the following preliminary result.

▶ **Proposition 34.** *All 1D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N \log N)$ time.*

6.2 A Linear-Time Algorithm Computing Aperiodic 1D-Covers

We will now show how to compute in linear time all aperiodic 1D-covers of T. The developed tools will also be useful in obtaining a general algorithm that computes all 1D-covers. A shortest 1D-cover is aperiodic (Observation 8), so in this section we also obtain a linear-time algorithm for computing all shortest 1D-covers.

We first prove a lemma that will allow us to efficiently process aperiodic borders of W.

▶ **Lemma 35.** *We have $\sum_{B \in \mathbf{A}(W)} |Occ(B,T)| = \mathcal{O}(N)$.*

Proof. Let $B_1, \dots, B_k$ be all aperiodic 1D-borders of W. First, two occurrences of an aperiodic string S can overlap by up to $|S|/2 - 1$ letters. Hence, $|Occ(B_i,T)| = \mathcal{O}(N/|B_i|)$. Further, for each $i \in [1\,..\,k)$, B_i is a border of B_{i+1} and hence, since the two occurrences of B_i overlap by less than $|B_i|/2$ letters, we have $|B_{i+1}| > 3|B_i|/2$. This implies the lemma. ◀

By the next lemma, we can compute all sets $Occ(B,T)$, for $B \in \mathbf{A}(W)$, in time $\mathcal{O}(N)$. The lemma can be proved using the $PREF$ array [12], as shown in Appendix B, or the more heavyweight internal pattern matching queries [15].

▶ **Lemma 36.** *Let $S_1, \dots, S_k$ be prefixes of W. Then we can compute all the sets $Occ(S_i,T)$ in $\mathcal{O}(N + \sum_{i=1}^{k} |Occ(S_i,T)|)$ time.*

We reduce our problem to the following auxiliary problem.

Coloured Strips Problem
Input: $\mathcal{O}(N)$ horizontal/vertical line segments on an $m \times n$ grid, where $N = mn$. Each line segment has a colour in $1, \ldots, \lfloor \log N \rfloor$.
Output: For each colour answer YES iff segments of this colour cover the whole grid.

▶ **Lemma 37.** *Computing all aperiodic 1D-covers of a 2D-string of size $N = mn$ can be reduced in $\mathcal{O}(N)$ time to an instance of Coloured Strips Problem with the same m, n.*

Proof. We construct an instance of the Coloured Strips Problem by choosing line segments of colour $i \in [1 \mathinner{.\,.} k]$ to be inclusion-maximal fragments of rows/columns of T covered by B_i, where $\mathbf{A}(W) = \{B_1, \ldots, B_k\}$.

The set of horizontal line segments of colour i is $\{[x, y \mathinner{.\,.} y+|B_i|-1] : (x, y) \in \mathit{Occ}(B_i, T)\}$, and the set of vertical line segments is $\{[x \mathinner{.\,.} x + |B_i| - 1, y] : (x, y) \in \mathit{Occ}(\mathsf{trans}(B_i), T)\}$. Lemmas 35 and 36 show that the total number of line segments is indeed $\mathcal{O}(N)$ and that they can be computed in linear time, respectively. ◀

▶ **Lemma 38.** *The Coloured Strips Problem can be solved in $\mathcal{O}(N)$ time.*

Proof. We first use radix sort to merge horizontal line segments of the same colour as long as any two intersect. We then repeat this for vertical line segments. Now no two line segments of the same direction and colour intersect.

We will assign in $\mathcal{O}(N)$ time to each grid cell (i, j) a bitmask $\mathit{hcol}(i, j)$, such that its kth bit is set iff cell (i, j) is covered by a horizontal line segment of colour k. We explicitly store $\mathit{hcol}(i, j)$ in $\mathcal{O}(1)$ words of space. We process each row using a line sweeping algorithm, updating the maintained bitmask whenever we encounter the endpoint of a horizontal line segment. We similarly compute an analogously defined bitmask $\mathit{vcol}(i, j)$ for vertical line segments, for each cell (i, j).

In the end, we return all colours in the bitmask $\bigwedge_{i,j} (\mathit{hcol}(i, j) \vee \mathit{vcol}(i, j))$. ◀

The last two lemmas imply immediately the following preliminary result.

▶ **Proposition 39.** *All aperiodic 1D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N)$ time.*

Next, we will present a more involved algorithm that computes *all* 1D-covers in linear time. It will rely on several additional technical concepts.

6.3 A Linear-Time Algorithm Computing All 1D-Covers

We will first show how to efficiently process all groups with large periods, using the fact that a pattern P in a text Y has $\mathcal{O}(|Y|/\mathsf{per}(P))$ occurrences. For each group with period p greater than $\log N$, we will compute all occurrences of its shortest element in each row/column in $\mathcal{O}(N/p)$ total time, after a global $\mathcal{O}(N)$-time preprocessing, using Lemma 36. Then, using this representation, we will be able to compute the elements of the group that cover T in $\mathcal{O}(N \log N/p)$ time. The periods grow geometrically by Lemma 29 and are at least $\log N$, so this yields $\mathcal{O}(N)$ time in total.

Then, we could employ Lemma 33 for each group with period at most $\log N$. There are only $\mathcal{O}(\log \log N)$ such groups, as the period of each group is at least a constant factor larger than the period of the previous group. Thus, we could process these groups in $\mathcal{O}(N \log \log N)$ time in total.

We will conclude this section by showing how to process in linear time all groups with periods not larger than $\log N$ using a variant of the Coloured Strips Problem, thus obtaining a linear-time algorithm for the problem in scope.

6.3.1 Handling Groups with Large Periods

We will rely on the fact that the total number of essential intervals in the families $\mathcal{F}$ in Algorithm 5 is of linear size.

For each group $\mathsf{Group}(X)$ with period greater than $\log N$, we first invoke Lemma 36 for the shortest element $S \in \mathsf{Group}(X)$. Then we perform the following *strip-merging routine*: In each row/column, we merge any two occurrences of S that are exactly $\mathsf{per}(S)$ positions apart as long as we can. If occurrences are treated as segments, this produces $\mathcal{O}(N/\mathsf{per}(S))$ horizontal/vertical line segments; each segment stores a weight equal to its length. We have thus reduced the problem in scope to the following problem.

> Restricted 2D Manhattan Skyline Problem
> **Input:** M horizontal/vertical line segments with positive weights in an $m \times n$ grid.
> **Output:** A point of the grid with minimum weight; the weight of a point is equal to the maximum weight of a line segment that covers it or 0 if the point is not covered by any segment.

▶ **Lemma 40.** *The Restricted 2D Manhattan Skyline Problem can be solved in time* $\mathcal{O}(M \log M)$.

Proof. We first sort the endpoints of line segments in $\mathcal{O}(M \log M)$ time; first by their x coordinate and then by their y coordinate. We now present a top-to-bottom line sweeping algorithm. The broom stores, for each point in $[1\,..\,n]$, the maximum weight of a vertical segment that contains it provided that the weight is positive. We implement the broom as a balanced BST. When processing a row with k horizontal segments, $[1\,..\,n]$ can be split into $\mathcal{O}(k)$ pairwise-disjoint intervals with equal maximum weight of a horizontal segment covering them. Then, by asking a query to the broom for each of the $\mathcal{O}(k)$ pairwise-disjoint intervals, we are able to compute the maximum weight of a vertical segment that intersects each of the intervals; consequently, a point of this row with minimum weight. Each query to the balanced BST is answered in $\mathcal{O}(\log M)$ time, so a row with k horizontal line segments is processed in $\mathcal{O}(k \log M)$ time. The total number of updates to the broom is upper bounded by the number of endpoints of vertical line segments, and each of them is processed in $\mathcal{O}(\log M)$ time. The stated complexity follows. ◀

▶ **Lemma 41.** *All 1D-covers with period greater than* $\log N$ *of a 2D-string* T *of size* N *can be computed in time* $\mathcal{O}(N)$.

Proof. We have $\mathcal{O}(\log n)$ groups to process, each with period greater than $\log N$. We showed that we can reduce, in $\mathcal{O}(N)$ time, the problem in scope to $\mathcal{O}(\log n)$ instances of the Restricted 2D Manhattan Skyline Problem; for each group with period p an instance with $M = \mathcal{O}(N/p)$. The periods grow geometrically (Lemma 29) and are at least $\log N$, so the time needed to solve all these instances is $\mathcal{O}(N/\log N \cdot \log N) = \mathcal{O}(N)$ by Lemma 40. ◀

6.3.2 Handling Groups with Small Periods

Let us consider all groups with periods at most $\log N$. If any of the considered groups contains at least $\log n$ strings, we treat it in $\mathcal{O}(N)$ time, invoking Algorithm 5 (see Lemma 33). Note that, due to Lemma 29, we can have at most one such group.

▶ **Lemma 42.** *The remaining groups with periods at most* $\log N$ *have* $\mathcal{O}(\log N)$ *elements in total.*

Proof. The number of groups is $\mathcal{O}(\log\log N)$ since their periods are geometrically increasing by Lemma 29. Moreover, if $n_1, \dots, n_k$ are sizes of the non-singleton groups, then by Lemma 29 we have $\prod_{i=1}^{k} n_i = \mathcal{O}(\log N)$, which implies that $\sum_{i=1}^{k} n_i = \mathcal{O}(\log N)$. ◀

We now have only $\mathcal{O}(\log N)$ strings to test. Unfortunately, we cannot use the Coloured Strips Problem directly, because the reduction of Lemma 37 could yield $\omega(N)$ line segments, as now we take several strings from each group. However, we can treat the elements of each group as a batch. Let us consider the following variant of the Coloured Strips Problem.

Coloured Strips Problem with Shades
Input: $\mathcal{O}(N)$ horizontal/vertical line segments on an $m \times n$ grid, where $N = mn$. Each line segment has a colour i and a shade in $[1\,..\,s_i]$, such that $\sum_i s_i = \mathcal{O}(\log N)$. The shades of a colour i are sorted from the lightest (1) to the darkest (s_i). No three line segments of the same colour intersect and none is contained in another line segment of the same colour.
Output: For each colour i, its darkest shade x such that line segments of colour i and shades non-darker than x cover the grid, if any.

Note that the above problem with one colour is a variant of the Restricted 2D Manhattan Skyline Problem.

For each group that we are to process, we invoke Lemma 36 for its shortest element, and then perform the strip-merging routine, outlined in Section 6.2 – the group number is the colour of the line segment and the length of a line segment is now its shade. Over all groups this takes $\mathcal{O}(N)$ time. We summarize the above discussion in the following statement.

▶ **Lemma 43.** *Computing all 1D-covers of a 2D-string of size* $N = m \times n$ *with period not larger than* $\log N$ *can be reduced in* $\mathcal{O}(N)$ *time to an instance of* Coloured Strips Problem with Shades *with the same* m, n.

The Coloured Strips Problem with Shades could be solved in $\mathcal{O}(N)$ time even without the assumptions on the intersections of segments of the same colour, but their presence makes the solution simpler.

▶ **Lemma 44.** *The* Coloured Strips Problem with Shades *can be solved in time* $\mathcal{O}(N)$.

Proof. The proof mimics that of Lemma 38, with a few differences. We compute for each grid cell (i, j) a bitmask $hcol(i, j)$ of size $\sum_i s_i$ that specifies, for each colour k and shade a, if the cell is covered by a horizontal line segment of colour k and a shade of darkness at least a. Now a horizontal line segment of shade a sets all bits of shades $1, \dots, a$ in the bitmask, which can be done in $\mathcal{O}(1)$ time with standard word-RAM operations.

With the presence of shades, we can no longer merge intersecting horizontal line segments of the same colour. However, when processing each row using a line sweeping algorithm, the simplifying assumptions in the problem guarantee that we may have at most two active line segments of the same colour. We maintain their shades explicitly, which lets us update the maintained bitmask.

We then compute *vcol*-bitmasks and return the darkest shade of each colour whose corresponding bit is set in the bitmask $\bigwedge_{i,j} (hcol(i, j) \vee vcol(i, j))$. ◀

By combining Lemmas 41, 43 and 44 we arrive at the main result of this section.

▶ **Theorem 45.** *All 1D-covers of a 2D-string of size* N *can be computed in time* $\mathcal{O}(N)$.

References

1. Amihood Amir, Ayelet Butman, Eitan Kondratovsky, Avivit Levy, and Dina Sokol. Multidimensional period recovery. In *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020*, volume 12303 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2020. doi:10.1007/978-3-030-59212-7_9.
2. Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.
3. Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978. doi:10.1137/0207043.
4. Jon Louis Bentley. Algorithms for Klee's rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.
5. Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980. doi:10.1145/358841.358850.
6. Richard S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977. doi:10.1016/0020-0190(77)90017-5.
7. Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.
8. Timothy M. Chan. Klee's measure problem made easy. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013*, pages 410–419. IEEE Computer Society, 2013. doi:10.1109/FOCS.2013.51.
9. Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. The number of repetitions in 2D-strings. In *28th Annual European Symposium on Algorithms, ESA 2020*, volume 173 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.32.
10. Michalis Christou, Maxime Crochemore, and Costas S. Iliopoulos. Quasiperiodicities in Fibonacci strings. *Ars Combinatoria*, 129:211–225, 2016.
11. Maxime Crochemore, Costas S. Iliopoulos, and Maureen Korda. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 20(4):353–373, 1998. doi:10.1007/PL00009200.
12. Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology.* World Scientific, 2002. doi:10.1142/4838.
13. Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.
14. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.
15. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.
16. Dennis W. G. Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.
17. Dennis W. G. Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Information Processing Letters*, 54(2):101–103, 1995. doi:10.1016/0020-0190(94)00235-Q.
18. Mark H. Overmars and Chee-Keng Yap. New upper bounds in Klee's measure problem. *SIAM Journal on Computing*, 20(6):1034–1045, 1991. doi:10.1137/0220065.
19. Alexandru Popa and Andrei Tanasescu. An output-sensitive algorithm for the minimization of 2-dimensional string covers. In *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019*, volume 11436 of *Lecture Notes in Computer Science*, pages 536–549. Springer, 2019. doi:10.1007/978-3-030-14812-6_33.

A Special Case of Klee's Measure Problem

In this section we show a solution to a special case of a dynamic version Klee's measure problem. Let us start with an auxiliary lemma.

We say that a rectangle is a *corner rectangle* in a grid if one of its corners coincides with a corner of the grid. We say that a rectangle is a *side rectangle* of a grid if its width or height is equal to k and one of its sides coincides with one of the sides of the grid. Our solution is based on the following key lemma.

▶ **Lemma 46.** *Dynamic Klee's Measure problem on a $k \times k$ grid with only corner and side rectangles inserted to $\mathcal{R}$ can be solved in $\tilde{\mathcal{O}}(k)$ time per insertion/deletion of a corner rectangle and $\tilde{\mathcal{O}}(1)$ time per insertion/deletion of a side rectangle.*

Proof. Let us first consider operations on corner rectangles. Each corner rectangle covers a top or bottom interval of positions in each grid column. Hence, the total area that is *not* covered by corner rectangles consists of at most one interval of positions in each column; see Figure 6. This representation of the non-covered area can be computed in $\mathcal{O}(k)$ time if, for each type of a corner rectangle (top-left, top-right etc.) we store the maximum-height corner rectangle of each possible width. These values can be updated in $\tilde{\mathcal{O}}(k)$ time per insertion or deletion of a corner rectangle if all corner rectangles are stored in a balanced binary search tree (BST).

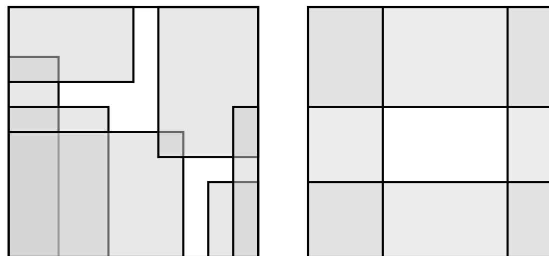

Figure 6 The intersection of the region that is not covered by corner rectangles with each column consists of at most one interval (left). The region that is not covered by side rectangles is a rectangle (right). The non-covered regions are shown in white.

The area not covered by side rectangles is a rectangle that only depends on the two highest side rectangles that contain the horizontal sides of grid and the two widest side rectangles that contain the vertical sides; see Figure 6 again. The rectangle can be updated in $\tilde{\mathcal{O}}(1)$ time after an insertion or a deletion of a side rectangle if all side rectangles are stored in a balanced BST. We construct a data structure that is recomputed from scratch after each insertion or deletion of a corner rectangle and allows us to compute the area of the intersection of the region that is not covered by corner rectangles with a query rectangle.

The area of each column covered by corners can be represented by (at most) two disjoint vertical strips (i.e., width-1 rectangles), each adjacent to the boundary of the grid. Let us focus on handling vertical strips with opposite corners $(x, 0)$ and $(x+1, y)$, as vertical strips with corners (x, y) and $(x+1, k)$ can be handled symmetrically. No two vertical strips in our collection intersect, and hence we can indeed handle each case separately.

We will maintain a set of (weighted) points $\mathcal{P}$ in 2D, such that a vertical strip with corners $(x, 0)$ and $(x+1, y)$ will be represented by point (x, y) with weight y. We will store a 2D range query data structure over $\mathcal{P}$, capable of returning the number and the total weight of points inside any queried axes-parallel rectangle. Consider such a rectangle R with corners (i, a) and (j, b), $i \le j$, $a \le b$.

We have the following cases for the intersection of a vertical strip β (as above) and R.

- Case I: $b \leq y$. In this case, the intersection of β and R has area $b - a$. The total area of R covered by such vertical strips is equal to the product of $b - a$ and the number of points of $\mathcal{P}$ in $[i \mathinner{.\,.} j-1] \times [b \mathinner{.\,.} \infty)$.
- Case II: $a < y < b$. In this case, the intersection of β and R has area $y - a$. The total area of R covered by such vertical strips is equal to the total weight of points of $\mathcal{P}$ in $[i \mathinner{.\,.} j-1] \times [a \mathinner{.\,.} b)$ minus the product of their number and a.

We can implement the 2D range query data structures with $\tilde{\mathcal{O}}(1)$ query time and $\tilde{\mathcal{O}}(k)$ space and construction time using range trees [5]. ◀

Let us recall that a rectangle is called a fat rectangle if its width *and* height are at least $\sqrt{n}$.

▶ **Lemma 17.** *The* DYNAMIC KLEE'S MEASURE *problem with fat rectangles can be solved in $\tilde{\mathcal{O}}(\sqrt{n})$ time per operation, after $\tilde{\mathcal{O}}(N)$-time preprocessing.*

Proof. Let us partition the grid into unit squares of height and width $\lceil\sqrt{n}\rceil$. Each fat rectangle in the grid is thus partitioned into: corner rectangles in at most 4 unit squares, side rectangles in $\mathcal{O}((n+m)/\sqrt{n}) = \mathcal{O}(\sqrt{n})$ unit squares, and several consecutive full unit squares in each column; see Figure 7.

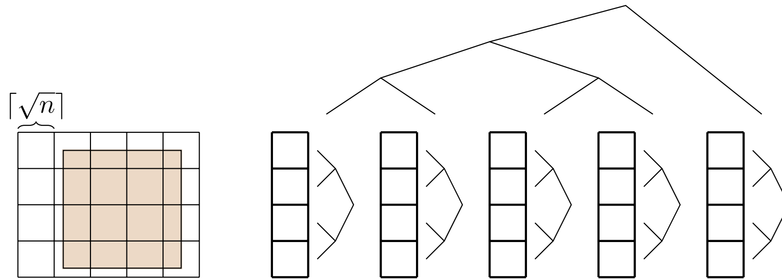

Figure 7 To the left: a fat rectangle decomposed into 4 corner rectangles, 8 side rectangles and 4 full unit squares. To the right: the structure of range trees that forms a kd-tree.

Each unit square stores the data structure of Lemma 46 that returns the covered area in this unit square. We also build range trees over each column of unit squares, and one range tree over the set of columns (see Figure 7 again), in order to support intervals of full unit squares in rectangle decompositions. We can use the range tree by Bentley [4] that allows insertions and deletions of intervals and counting the length of the union of intervals in $\mathcal{O}(\log N)$ time. This construction of range trees is similar to the kd-tree used in [18]. Let us mention that the range trees need to be slightly adjusted in order to account for the values returned by the data structure of Lemma 46 for unit squares that are not covered in full by any rectangle. ◀

A.1 Checking if the Whole Grid is Covered

If we are only interested in checking if the union of all rectangles in the dynamic problem covers the whole grid, which is the case in our application, the approach can be simplified as follows.

In Lemma 17, the data structure for each unit square returns a single bit of information, so simply no modification is needed in the range tree by Bentley [4].

The 2D range query data structure from the proof of Lemma 46 can be substituted with range minimum queries on a 1D array as follows. The complement of the union of

corner rectangles consists of horizontal and vertical strips, at most one vertical strip in a column and at most one horizontal strip in a row. Consequently, the complement of union of corner rectangles can be described in $\mathcal{O}(k)$ space using $dist_N$, $dist_E$, $dist_S$, $dist_W$ arrays that measure the distance (respectively in the direction *North, South, East, West*) from each half-integral point in the corresponding side of the whole grid to the first non-covered position in a given direction (see Figure 8).

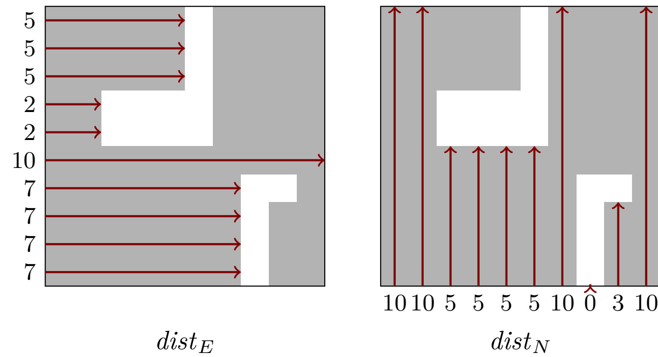

Figure 8 Representation of the regions not covered by corner rectangles from Figure 6. The sequences of numbers in the left/bottom sides are the arrays $dist_E, dist_N$. The (similar) arrays $dist_S, dist_W$ are not shown here.

Using four range minimum queries over *dist* arrays we can check in $\mathcal{O}(1)$ time if there exists some non-covered grid cell in a query rectangle. More precisely, a rectangle with corners (i, a) and (j, b), $i \le j$, $a \le b$, does *not* contain any non-covered cell if and only if at least one of the following conditions is satisfied:

- $\min dist_E[i \mathinner{.\,.} j-1] \ge b$,
- $\min dist_W[i \mathinner{.\,.} j-1] \ge k-a$,
- $\min dist_N[a \mathinner{.\,.} b-1] \ge j$,
- $\min dist_S[a \mathinner{.\,.} b-1] \ge k-i$.

B Remaining Proofs

▶ **Theorem 14.** *All aperiodic 2D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N \log N)$ time.*

Proof. By Lemma 9, aperiodic 2D-borders have only $\mathcal{O}(\log m)$ and $\mathcal{O}(\log n)$ possible heights and widths, respectively. All periodic 2D-borders can be filtered out in $\mathcal{O}(N)$ time using the border arrays of $\mathsf{hstr}(T^{(h)})$ and $\mathsf{vstr}(T_{(w)})$ for all h, w. Let us leave only those aperiodic 2D-borders as candidates which cover both $T^{(h)}$ and $T_{(w)}$ for their respective height h and width w. They can be identified using an algorithm for finding all covers in a 1D-string [16, 17] applied to $\mathsf{hstr}(T^{(h)})$ and $\mathsf{vstr}(T_{(w)})$.

If a candidate $T[1 \mathinner{.\,.} h, 1 \mathinner{.\,.} w]$ is a 2D-cover of T, then candidates $T[1 \mathinner{.\,.} h', 1 \mathinner{.\,.} w]$ and $T[1 \mathinner{.\,.} h, 1 \mathinner{.\,.} w']$ for $h' \le h$, $w' \le w$ are also 2D-covers of T. We check the candidates sorted by non-increasing height (and by non-decreasing width in case of draws), using the Is2DCover(C, T) routine for each candidate C. If the candidate in question turns out to be a 2D-cover, then we approve all the remaining ones with the same width as well, and if it is not, then we can remove all the remaining ones of the same height. Hence, we can charge each test to a unique width or height, which means that we perform $\mathcal{O}(\log n + \log m) = \mathcal{O}(\log N)$ tests in total. Each test can be performed in $\mathcal{O}(N)$ time due to Lemma 4. ◀

▶ **Lemma 29.** *For $X, Y \in \mathbf{B}(S)$ with $|X| < |Y|$, we have $\mathsf{per}(Y) \geq \mathsf{per}(X) \cdot |\mathsf{Group}(X)|$. Moreover, for $X, Y, Z, W \in \mathbf{B}(S)$ with $|X| < |Y| < |Z| < |W|$, we have $\mathsf{per}(W) \geq \frac{9}{8}\mathsf{per}(X)$.*

Proof. We first state a definition and a few facts. A string U is called *primitive* if it cannot be expressed as V^k for a string V and an integer $k > 1$. The *synchronization property* states that a non-empty string U is primitive if and only if it occurs only twice in UU: as a prefix and as a suffix. Moreover, any $|\mathsf{per}(U)|$-length fragment of a string U is primitive [12].

Let us now prove the first statement of the lemma. If X is aperiodic, then the conclusion is clear. Henceforth let us assume that X is periodic. In this case we have $\mathsf{per}(Y) > \mathsf{per}(X)$. Towards a contradiction, suppose that

$$\mathsf{per}(Y) < \mathsf{per}(X) \cdot |\mathsf{Group}(X)| \leq |X| - \mathsf{per}(X).$$

First, it cannot be that $\mathsf{per}(X)$ divides $\mathsf{per}(Y)$, since this would contradict the primitivity of $S[1 \mathinner{.\,.} \mathsf{per}(Y)]$.

In the complementary case that $\mathsf{per}(X)$ does not divide $\mathsf{per}(Y)$, we have that

$$\mathsf{per}(Y) + \mathsf{per}(X) < |X| < |Y| \text{ implies } [1 \mathinner{.\,.} \mathsf{per}(X)] = S[\mathsf{per}(Y) + 1 \mathinner{.\,.} \mathsf{per}(Y) + \mathsf{per}(X)].$$

We thus have an occurrence of $U = S[1 \mathinner{.\,.} \mathsf{per}(X)]$ in X that starts in a position that is not a multiple of $\mathsf{per}(X)$. This contradicts the primitivity of U, due to the synchronization property.

We now move to the proof of the second statement of the lemma. First, let us note that for $U, V \in \mathbf{B}(S)$ with $|U| < |V|$ we have $|V| \geq \frac{3}{2}|U|$. Let us distinguish between two cases.

- **Case I:** $\mathsf{per}(W) = \mathsf{per}(Z)$.
  In this case, Z must be aperiodic by the definition of $\mathbf{B}(S)$, i.e. $\mathsf{per}(Z) > |Z|/2$. Then, Z's longest border cannot be longer than $|Z| - \mathsf{per}(Z) < \mathsf{per}(Z)$. As Y must be a border of Z, we have $|Y| < \mathsf{per}(Z)$. Then, the fact that $\frac{3}{2}\mathsf{per}(X) \leq \frac{3}{2}|X| \leq |Y|$ proves the statement.
- **Case II:** $\mathsf{per}(W) > \mathsf{per}(Z)$.
  In this case, by the periodicity lemma (Lemma 7) we have $|Z| \leq 2\mathsf{per}(W)$. We thus have $\frac{9}{8}\mathsf{per}(X) \leq \frac{9}{8}|X| \leq \frac{3}{4}|Y| \leq |Z|/2 \leq \mathsf{per}(W)$. ◀

▶ **Lemma 36.** *Let $S_1, \ldots, S_k$ be prefixes of W. Then we can compute all the sets $Occ(S_i, T)$ in $\mathcal{O}(N + \sum_{i=1}^{k} |Occ(S_i, T)|)$ time.*

Proof. Let us recall the *PREF* array of a string S that stores, for each $i \in [2 \mathinner{.\,.} |S|]$, the length of the longest common prefix of $S[i \mathinner{.\,.} |S|]$ and S. This array can be computed in linear time [12]. We use the *PREF* array to compute the sets $Occ(S_i, T)$ in time linear in their total size plus $\mathcal{O}(N)$ as follows. Let us assume that S_i are sorted by increasing lengths. We store a doubly-linked list L of pairs, initially containing all pairs from $[1 \mathinner{.\,.} m] \times [1 \mathinner{.\,.} n]$. Moreover, each pair stores a pointer to its corresponding element in the list L, if such an element exists. For each row r, we compute the table $PREF_r$ as the *PREF* array of $T[1, 1 \mathinner{.\,.} n]\#T[r, 1 \mathinner{.\,.} n]$. We construct n buckets and store in the j-th bucket all pairs (r, c) such that $PREF_r[n + 1 + c] = j$. Then, for each $j = 1, \ldots, n$, we report L as $Occ(S_i, T)$ if $j = |S_i|$ and then remove from L all pairs from the j-th bucket. ◀