Finding Maximal 2-Dimensional Palindromes

Sara H. Geizhals[1] and Dina Sokol[2]

1 **The Graduate Center of the City University of New York, 365 Fifth Avenue, New York, NY 10016, USA , shgeizhals@gmail.com**
2 **Brooklyn College of the City University of New York, 2900 Bedford Avenue, Brooklyn, NY 11210, USA , sokol@sci.brooklyn.cuny.edu**

Abstract

This paper extends the problem of palindrome searching into a higher dimension, addressing two definitions of 2D palindromes. The first definition implies a square, while the second definition (also known as a *centrosymmetric factor*), can be any rectangular shape. We describe two algorithms for searching a 2D text for maximal palindromes, one for each type of 2D palindrome. The first algorithm is optimal; it runs in linear time, on par with Manacher's linear time 1D palindrome algorithm. The second algorithm searches a text of size $n_1 \times n_2$ ($n_1 \geq n_2$) in $O(n_2)$ time for each of its $n_1 \times n_2$ positions. Since each position may have up to $O(n_2)$ maximal palindromes centered at that location, the second result is also optimal in terms of the worst-case output size.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases palindrome, pattern matching, 2-Dimensional, centrosymmetric factor

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.19

1 Introduction

Palindromes are strings that read the same forwards and backwards. Formally, a string P is a palindrome if it is of the form uau^R, where u is a non-empty string and u^R is its reverse; a is the empty string or a single character. a is called the *gap*, while u and u^R are called respectively the *left arm* and *right arm* of the palindrome. Palindromes have long drawn the attention of computer science researchers. The classical online and linear time palindrome algorithm is due to Manacher [21] in 1975. A palindrome variation called a *palstar*, which is loosely defined as the concatenation of palindromes, was studied as well in the 1970's by [19] and [11]. There is later research concerning searching for palindromes when there is a parallel model [3][4].

Other variations of palindrome search that have been studied more recently include gapped palindromes, complementary palindromes, approximate palindromes, and compressed palindromes. A *gapped palindrome* is when the size of the gap $|a| \geq 2$ [20]. *Complementary palindromes* are relevant in DNA, and it is where a character matches its complementary character instead of itself, e.g. *AACGTT*. [20]'s gapped palindrome algorithm can be adapted to find complementary gapped palindromes (which they refer to as *biological gapped palindromes*). *Approximate palindromes* have an allowed number of variations between the arms, and they have been studied in run-length compressed texts [6] as well as in the online model [2]. An interesting algorithm that searches for palindromes with edit distance of k is presented in [15]. Compressed palindromes have been studied as well under straight line programs [22].

Extending the concept of a palindrome to two dimensions has various applications. For example, face recognition technology exploits symmetry characteristics of the human face

Figure 1 Two examples of *sq2DP*s. The one on the left is written in Latin, while the one on the right is in Hebrew.

in order to extract a set of significant features [7]. Determining the global maximum of local reflectional symmetry in grey level images is related to genetic algorithms [18]. [14] creates palindromic shapes as representations of the intrinsic and extrinsic symmetries of 2D articulated planar shapes.

This paper presents algorithms that work with two different definitions of *2D palindromes*. The first definition dates back to the early Romans, and it can apply only to a square pattern; hence, we refer to it as a *sq2DP*. The second definition is termed a *centrosymmetric factor*[1] in [5]. This type of 2D palindrome can take on any rectangular shape and thus we refer to it as a *rect2DP*. To the best of our knowledge, the problem of searching a 2D text for maximal 2D palindromes has not been previously studied.

▶ **Definition 1.** A *sq2DP* is an $m \times m$ 2D pattern that admits four symmetries: identity, two diagonal reflections, and 180° rotation.

For example, Figure 1 portrays two famous *sq2DP*s. The one on the left is the first dateable representation of this type of 2D palindrome, and it was found in the ruins of Pompeii. The language is Latin, and it means, "the sower [planter] Arepo works with the help of wheel [a plough]" [10][24]. The one on the right is a *sq2DP* formed of five Hebrew words, of five characters each. It was written by Rabbi Abraham ibn Ezra (1089-1164) in response to the question as to whether a fly landing in honey makes the honey not kosher. Its translation is: "We have explained that the glutton [fly] who is in the honey was burned and incinerated [i.e., it disintegrated and therefore does not make it not kosher]" [23].

This problem is important to group theorists, in the field of mathematics. A *sq2DP* is a 2D pattern invariant under the subgroup generated by the two diagonal reflections of the dihedral group known as D_8. The D_8 group is one that is formed by the set of a square's eight symmetries (four rotations and four reflections).

▶ **Definition 2.** A *rect2DP* is a rectangular block of m_1 rows and m_2 columns that admits the two symmetries of identity and 180° rotation.

Each 2D palindrome has a *center*, which is the point that results in an equal number of columns to the left and right, as well as an equal number of rows above and below. The technical definition of the center differs slightly depending on the type and size of the 2D palindrome. Given an $m \times m$ *sq2DP*, if m is odd, the center is at location $(\lceil \frac{m}{2} \rceil, \lceil \frac{m}{2} \rceil)$. If m

[1] The paper studies the complexity of 2D Sturmian sequences in terms of the number of centrosymmetric factors that can occur in a 2D Sturmian sequence. Although their definition uses a binary alphabet due to its context, this paper assumes any bounded alphabet.

Table 1 Two examples of *rect2DP*s. The left one's center is between the two 3's.

1	0	2	4
0	3	3	0
4	2	0	1

n	e	v	e	r	o	d
d	o	r	e	v	e	n

is even, the center is in between rows and columns. Similarly for a *rect2DP*, if the number of rows (resp. columns) is even, the center is placed between rows (resp. columns). For example, the center of the left *rect2DP* in Table 1 is in the second row between the two 3's.

We present one algorithm for *sq2DP*, and one for *rect2DP*. Both algorithms consider each possible position of a center, and then locate the 2D palindrome(s) centered there. As with 1D palindromes, we are interested only in the 2D palindromes that are *maximal*. A *sq2DP* of size $m \times m$ is *maximal* if enlarging it by one on all sides – to size $(m+2) \times (m+2)$ – results in a pattern that is not a *sq2DP*. There is exactly one maximal *sq2DP* centered at each possible center position. Similarly, a *rect2DP* is maximal if it is not contained within a larger *rect2DP* with the same center. For a given text position, a maximal *rect2DP* is the highest *rect2DP* for its width or the widest *rect2DP* for its height. Thus, there may be several maximal *rect2DP* centered at a given position.

The remainder of this paper is organized as follows: Section 2 presents an algorithm for locating all maximal *sq2DP* in a given 2D text. Its input is T of size $n \times n$, and its runtime is linear, i.e. $O(n^2)$. This is on par with Manacher's linear palindrome algorithm and stems from the fact that there is exactly one maximal palindrome centered at each position. In Section 3, we describe a different algorithm that searches for maximal *rect2DP*. Its input is T of size $n_1 \times n_2$ (where $n_1 \geq n_2$), and its runtime is $O(n_1 n_2^2)$. We conclude in Section 4 with our plans for future work.

2 Square 2D Palindrome

The input to the algorithm is a 2D text T over a bounded alphabet Σ. For simplicity, we assume T is of size $n \times n$, however, the algorithm can be used for any rectangular text. The algorithm searches T for all maximal *sq2DP* that occur in T.

The basis of the algorithm is that the symmetry property of palindromes in one dimension also applies to *sq2DP*. In 1D, the palindromes that are substrings of the left arm of a palindrome will appear as well in the right arm. To illustrate, consider the lengths of the maximal palindromes centered at each position in the string *abacaba*: 1,3,1,7,1,3,1, and note the symmetry of this numerical list (around its center).

Henceforth we distinguish between the two diagonals of a square as follows. The diagonal that extends from the upper left corner to the lower right corner is called the *main diagonal*, and the diagonal that extends from the upper right corner to the lower left corner is called the *anti-diagonal*. Assume that P is a maximal *sq2DP* in T centered at position (C_i, C_j). Suppose we are considering location (i, j) of T as a possible center for a palindrome, and (i, j) is contained in the bottom right triangle of the larger palindrome P that is centered at (C_i, C_j). Further assume that both the maximal palindrome centered at (i, j) and at the mirror position of (i, j) over the anti-diagonal of P are completely contained within P. We can conclude the following:

▶ Observation 1. The maximal palindrome centered at a location (i, j) is identical to the maximal palindrome centered at the mirror position of (i, j) over the anti-diagonal of P.

The observation follows directly from the symmetry that the palindrome has over its anti-diagonal. As in 1D, the smaller palindromes contained in the upper triangle are mirrored exactly in the lower triangle. Note that this is true whether we use either diagonal, but our algorithm uses only the values over the anti-diagonal.

The idea of the algorithm is to use Observation 1 as follows. When searching for a palindrome centered at location (i, j) that is contained in a larger palindrome P, we first consider the value from the mirror image of position (i, j) over the anti-diagonal of P. If the maximal palindrome at the mirror image extends beyond the left boundary of P, then we take the minimum of the value at the mirror image and the boundary of P. Following this initial setting, we use the naive method to check whether it is possible to extend the palindrome centered at (i, j) beyond the right boundary of the containing palindrome. This mimics the algorithm of Manacher in two dimensions, but of course additional techniques are needed to render the algorithm linear time.

2.1 Preprocessing Stage

The text T is preprocessed by constructing a generalized suffix tree (GST) for the columns of T, from bottom to top and from top to bottom, and for the rows of T, from left to right and right to left. Then, it is preprocessed to allow $O(1)$-time longest common prefix (LCP) queries.

We define forward subcolumns and subrows (resp.) beginning at any location (i, j) as: $c(i, j) = T[i, j] \dots T[n, j]$, $r(i, j) = T[i, j] \dots T[i, n]$. Similarly, the reverse subcolumns and subrows are denoted by: $c'(i, j) = T[i, j] \dots T[1, j]$, $r'(i, j) = T[i, j] \dots T[i, 1]$.

Using these subcolumns and subrows, we can define four directions of L's cornered at a particular location (i, j), as subcolumn-subrow pairs[2].

1. A "backwards L," denoted $\text{⅃}_{i,j} = \langle c'(i, j), r'(i, j)\rangle$, consists of a pair of T's reverse subcolumn and reverse subrow.
2. An "upside down L," denoted $\Gamma_{i,j} = \langle c(i, j), r(i, j)\rangle$ consists of a pair of T's forward subcolumn and forward subrow.
3. An L, denoted $\text{L}_{i,j} = \langle c'(i, j), r(i, j)\rangle$, consists of a pair of T's reverse subcolumn and forward subrow.
4. An "upside down backwards L," denoted $\text{⅂}_{i,j} = \langle c(i, j), r'(i, j)\rangle$, consists of a pair of T's forward subcolumn and reverse subrow.

We also define constant time symmetry checking between Γ and ⅃. This can be done by taking the minimum value of the LCP of the corresponding sides of the L's when reflected over the anti-diagonal. Specifically, LCP $(\Gamma_{i,j}, \text{⅃}_{p,q}) = min(LCP(c'(p, q), r(i, j)), LCP(c(i, j), r'(p, q)))$. Similarly, in the other direction, the longest symmetric prefix between L and ⅂, reflected over the main diagonal, can be found in constant time.

2.2 Scanning Stage

In the scanning stage of the algorithm, we define a set of forward diagonals in the text, parallel to the main diagonal, $d = -(n - 1)$ to $(n - 1)$. This is similar to the method used by Amir and Farach [1] for multiple pattern matching of square patterns. We number each

[2] These L's are similar to the L's defined by Amir and Farach in [1]; the L-suffix tree of Giancarlo [12] uses a similar concept.

forward diagonal $d = i - j$, the difference between its row and column coordinates. Note that $d = 0$ is the main diagonal, $d > 0$ are the diagonals below the main diagonal, and $d < 0$ are the diagonals above the main diagonal. Each diagonal contains $n - |d|$ positions, where $|d|$ represents the absolute value of d.

Since the same procedure is performed on each forward diagonal, we describe the algorithm for a single forward diagonal d. The goal of the algorithm is to fill d's integer array $pals$ which corresponds to the $n-|d|$ positions on diagonal d in T. Each element in $pals$ will contain a value representing the maximal *sq2DP* centered at the corresponding position in T. Value v indicates that it consists of the position itself, plus v in the four directions (up, down, left, and right) – i.e., a *sq2DP* of size $(v * 2 - 1)$, with this position as its center.

We explain how Algorithm 1 works on diagonal $d \geq 0$. For $d < 0$, the same algorithm works with minor modifications to indices. The variable $maxCenter$ is the center of the *sq2DP* that has extended the farthest; $maxCenter + pals[maxCenter]$ is the rightmost (and lowest) position it reaches. j is a pointer that moves along the positions on the diagonal one at a time, and at each position we determine the size of the maximal *sq2DP* centered at the position pointed to by j.

For each j, the value in $pals[j]$ is set in a way similar to Manacher [21]: if the position is past $maxCenter + pals[maxCenter]$, then it has never been seen yet, and therefore its value in $pals$ is initialized to 1. If the position is before $maxCenter + pals[maxCenter]$, then it is known to be part of a palindrome, and therefore its value in $pals$ is initialized to the value of its mirror image over $maxCenter$; but if that value, when added to j, would extend beyond $maxCenter + pals[maxCenter]$, then the value is reduced so that it doesn't extend. Following the initial setting of the value in $pals[j]$, a while loop continually performs constant-time symmetry checking between the L's of different orientation to check how far the current palindrome extends.

One such square is demonstrated in Figure 2. Diagonal $d > 0$ is depicted and location j on diagonal d is depicted as the large dot. The LCP queries start one beyond $j + pals[j]$ and $j - pals[j]$: one involves ⅃ (straight vertical and horizontal lines) with Γ (dashed vertical and horizontal lines), and the other query involves L (dashed vertical and straight horizontal lines) with ⅂ (straight vertical and dashed horizontal lines).

Although both reflectional symmetries must be checked individually, it is not necessary to explicitly check the 180° rotation, since it is implied by transitivity from the reflectional symmetries. Specifically, location (i, j) must match its symmetric location over the main diagonal, which is (j, i). By the anti-diagonal symmetry, $T[j, i] = T[n - i, n - j]$ which is exactly the location symmetric to (i, j) by the 180° rotation.

Note that the algorithm works with *sq2DP*s of odd × odd dimensions; for even × even ones, include the following modifications: before the preprocessing stage, add a row to the top and the bottom of T, plus a row between every two rows. Also add a column on the left and the right of T, plus a column between every two columns. The added rows and columns are filled with a character that does not appear in T; and T of size $n \times n$ is now of size $(2n + 1) \times (2n + 1)$. When the scanning stage outputs a *sq2DP* of size $(2v + 1) \times (2v + 1)$, where v is even, that is indicative of a *sq2DP* of size $v \times v$, once the added rows and columns are removed.

2.3 Example

Using Table 2, we will demonstrate the scanning stage with an example, at the point where $d = 0$ and $j = 6$ (for position $T[6, 6]$; it is underlined). This position is contained in a palindrome, as $maxCenter + pals[maxCenter] = 5 + 5 = 10$ extends beyond it. Therefore,

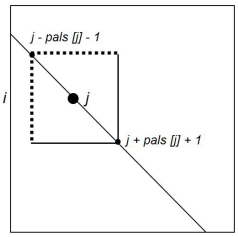

Figure 2 LCP queries on position j (large dot) of diagonal $d > 0$. One query involves a backwards L (straight vertical and horizontal lines) with an upside down L (dashed vertical and horizontal lines), and the other involves an L (dashed vertical and straight horizontal lines) with an upside down backwards L (straight vertical and dashed horizontal lines).

Algorithm 1: Algorithm for *sq2DP*.

input : GST of the columns and rows of T in forward and reverse order, diagonal d

output: diagonal d's integer array $pals$, of size $n - |d|$, containing the values of the maximal *sq2DP* centered at the corresponding positions in T

$maxCenter = 1$
$pals[1] = 1$
for $j = 2$ *to* $n - |d|$ **do** //for positions j on diagonal d
  $i = d + j$ //jth position on diagonal d is at $T[d + j, j]$ (if $d \geq 0$)
  if $maxCenter + pals[maxCenter] \leq j$ `/* position not known to be part of palindrome */`
  then
    $pals[j] = 1$
  else
    $pals[j] = \min\{pals[2 * maxCenter - j], maxCenter + pals[maxCenter] - j\}$
  while $(j + pals[j] < n)$ *and* $(j - pals[j] > 1)$ *and* `/* in bounds */`
  `/* The following two LCP queries check each of the diagonal symmetries, verifying whether the current palindrome can be enlarged by one layer all around. */`
  $LCP\ (\Gamma_{i-pals[j]-1, j-pals[j]-1}, \lrcorner_{i+pals[j]+1, j+pals[j]+1}) \geq 2 \times pals[j]$ *and*
  $LCP\ (\mathrm{L}_{i+pals[j]+1, j-pals[j]-1}, \urcorner_{i-pals[j]-1, j+pals[j]+1}) \geq 2 \times pals[j]$
  do
    $pals[j]{+}{+}$
  end
  if $j + pals[j] > maxCenter + pals[maxCenter]$ **then**
    $maxCenter = j$
end

Table 2 Text T (left) and $d = 0$'s *pals* array (right), at the point where the algorithm will calculate the value for $T[6,6]$ in $pals[6]$.

	1	2	3	4	5	6	7	8	9	10
1	a	b	b	b	b	a	a	b	b	e
2	b	c	c	c	b	c	c	c	b	e
3	b	c	c	c	b	c	c	c	a	e
4	b	c	c	c	b	c	c	c	a	e
5	b	b	b	b	a	b	b	b	b	a
6	a	c	c	c	b	c	c	c	b	c
7	a	c	c	c	b	c	c	c	b	c
8	b	c	c	c	b	c	c	c	b	c
9	b	b	a	a	b	b	b	b	a	b
10	e	e	e	e	a	c	c	c	b	c

index	1	2	3	4	5	6	7	8	9	10
value	1	1	3	1	5	?	?			

its value in *pals* is that of its mirror image over *maxCenter*: 1. The two LCP queries indicate no extensions. Then $j = 7$, and that refers to $T[7,7]$ (underlined). Its value in *pals* is that of its mirror image over *maxCenter* – $T[3,3]$'s value of 3. LCP queries are performed, in an effort for a larger *sq2DP*, and they start with a square of size 7×7 (as sizes 3×3 and 5×5 are already known to be part of the *sq2DP*). They do indicate a *sq2DP* of size 7×7, but then the algorithm cannot continue as it would go out of bounds. Thus, *maxCenter* is set to point to this position and $pals[7]$ is set to 4. Then j is 8, and the algorithm will calculate the value of $pals[8]$ for $T[8,8]$.

2.4 Runtime

▶ **Theorem 3.** *The time complexity for finding all maximal* sq2DP *in a text of size* $n \times n$ *is* $O(n^2)$.

Proof. The runtime of the preprocessing stage is as follows: the construction of the GST is in time linear to the size of T [8]. Then it takes $O(n)$-time to preprocess to allow for constant-time LCP queries [13].

The scanning stage runs in $O(n^2)$ time. This is because Algorithm 1 is run on each of the $2n - 1$ diagonals. There are $O(n)$ positions j per diagonal (as seen by the number of iterations of the **for** loop in Algorithm 1). As in Manacher's algorithm, the initial value in *pals* is copied from the mirror image around *maxCenter*, and therefore each matching L is compared exactly once. Each mismatching L can be charged to the center for which it mismatched since each center encounters at most one mismatch. ◀

3 Rectangle 2D Palindrome

Working with *rect2DP*s is different than working with *sq2DP*s, as the mirror image property of Observation 1 is unique to squares and does not hold for general rectangles. Therefore, we use a different approach to finding them. The input is a 2D text T over a bounded alphabet Σ, with n_1 rows and n_2 columns. We assume $n_1 \geq n_2$; otherwise, it is possible to first rotate T by 90°. The preprocessing and scanning stages are described in this section, and all maximal *rect2DP* in T are reported.

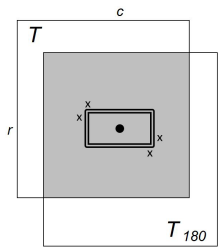

Figure 3 Placing of T_{180} on top of T, with $T[r, c]$ (represented as the dot) as the anchor. Some mismatches are demarcated with x's, and a *rect2DP* (enclosed in double lines) is placed within their bounds.

3.1 Preprocessing Stage

In the preprocessing stage, we construct a GST for the columns in T, both from top to bottom and from bottom to top. Then the GST is preprocessed to allow for constant-time LCP queries.

3.2 Scanning Stage

The scanning stage is run on each position of T. We describe the algorithm for a given position (r, c). The scanning stage outputs integer tuples whose values represent the height and width of maximal *rect2DP*(s) centered at position $T[r, c]$.

The underlying idea is visually depicted in Figure 3. Place T_{180}, which is T rotated by 180°, on top of T, with $T[r, c]$ as the *anchor* (that is, $T[r, c]$ must be placed on top of itself). Let T_{ov} be the overlapping region, which is shaded. For each column in T_{ov}, find the first mismatch between T and T_{180} that is above row r in T, and the first mismatch that is below row r of T. In Figure 3, some mismatches are demarcated with x's. Then, for each width possible, attempt to place a *rect2DP* whose center is position $T[r, c]$ and which is bounded on top and on bottom by mismatches. In Figure 3, such a *rect2DP* (enclosed with double lines) is drawn.

The idea is implemented in Algorithm 2 as follows: create T_{ov} as a subtext of T. It has $T[r, c]$ as its center, and it has the coordinates

top left: $T[r - min_r, c - min_c]$ top right: $T[r - min_r, c + min_c]$
bottom left: $T[r + min_r, c - min_c]$ bottom right: $T[r + min_r, c + min_c]$

where $min_r = \min(r - 1, n_1 - r)$ and $min_c = \min(c - 1, n_2 - c)$. Thus, T_{ov} has $min_r * 2 + 1$ rows and $min_c * 2 + 1$ columns. Note that since position $T[r, c]$ is the center of T_{ov}, it is at position $T_{ov}[min_r + 1, min_c + 1]$.

Then, finding mismatches between T and T_{180} is performed as constant-time LCP queries on the GST of the columns of T. Specifically, let $0 \leq k \leq min_c$. Every query involves row r; it is between the column that is k to the left of $T[r, c]$, from row r and above, with the column that is k to the right of $T[r, c]$, from row r and below. The results are stored in the *colLcp* array.

Finally, for every width, beginning with the widest possible, perform a range minimum query (RMQ) on *colLcp*. The resulting value bounds a rectangle on top and on bottom.

Algorithm 2: Algorithm for *rect2DP*.

input : T, GST of T's columns in forward and reverse order, r, c

output: integer tuples whose values represent the maximal *rect2DP*(s) centered at position $T[r, c]$

```
1  for k = 1 to (min_c * 2 + 1) do //for columns k
2      colLcp[k] = LCP(T[r, c − min_c + k], ..., T[1, c − min_c + k];
3                      T[r, c + min_c − k], ..., T[n_1, c + min_c − k])
4  end
5  maxHeight = 0
6  for w = min_c ... 0 do //for widths w, in decreasing order
7      height = RMQ (colLcp, (min_c + 1) − w, (min_c + 1) + w)
8      if height > maxHeight    /* if height ≤ maxHeight, then there is no rect2DP; or
       there is, but it's not maximal */
9      then
10         maxheight = height
11         output ⟨(height × 2 − 1), (w × 2 + 1)⟩
12 end
```

Table 3 The algorithm is at the point of locating the *rect2DP*s for position $T[3, 5]$ (underlined). The T_{ov} subtext (bold) is centered at that position. The *colLcp* array is also shown.

	1	2	3	4	5	6	7	8
1	e	**e**	**e**	**e**	**e**	**e**	**e**	**e**
2	e	**e**	**d**	**d**	**b**	**b**	**e**	**e**
3	e	**d**	**c**	**c**	**a**	**c**	**c**	**b**
4	e	**e**	**e**	**b**	**b**	**d**	**e**	**e**
5	e	**e**	**e**	**e**	**e**	**e**	**e**	**e**
6	e	e	e	e	e	e	e	e

	1	2	3	4	5	6	7
colLcp	0	1	3	3	3	3	0

If the height is less than or equal to a previously found height then the rectangle is not a *rect2DP* (as it is not maximal). Otherwise, the algorithm outputs an integer tuple – height and width – representing this maximal *rect2DP*.

The algorithm above works with *rect2DP*s of odd × odd dimensions. For the case where one or both of the dimensions is even, similar modifications can be done to the text as provided in the *sq2DP* case. Alternatively, each possible center, including in between rows and columns, can be considered.

3.3 Example

In Table 3, T has $n_1 = 6$ rows and $n_2 = 8$ columns. We will demonstrate the scanning stage, at the point of the algorithm where r = 3 and c = 5 (position $T[3, 5]$, which is underlined). That position is the center of T_{ov} (which is bold), by having 5 rows (since $min_r = \min(3-1, 6-3)*2+1 = 5$) and 7 columns (since $min_c = \min(5-1, 8-5)*2+1 = 7$).

In particular, they are T's rows 1-5 and columns 2-8.

The *colLcp* array is shown. In detail: *colLcp*[1] is the result of the LCP query between *dee* and *bee* (which is 0), *colLcp*[2] is from the LCP query between *cde* and *cee* (which is 1), *colLcp*[3] is from the LCP query between *cde* and *cde* (which is 3), and so on.

Then, we set $maxHeight$ to 0. We will look for *rect2DP*s in w widths, in decreasing order. When $w = 3$, we are looking for a *rect2DP* that is centered at this position and is of width 7. No such *rect2DP* exists, and this is found by the algorithm ($height = 0$; since $height \not> maxHeight$, there is no output). When $w = 2$, $height = 1$. $1 > maxHeight$, and so this *rect2DP* is maximal: $maxHeight$ is set to 1, and $\langle(1 \times 2 - 1), (2 \times 2 + 1)\rangle = \langle 1, 5\rangle$ is outputted. It refers to the *rect2DP* of size 1×5: *ccacc*. When $w = 1$, we are looking for a *rect2DP* that is centered at this position and is of width 3. $height = 3$, and $3 > maxHeight$, which means that there is such a *rect2DP*. It is of size 5×3 – from $T[1, 4]$ through $T[5, 6]$. Lastly, when $w = 0$, $height = 3$. Because $3 \not> 3$ there is no output. This correlates, as the *rect2DP* of size 3×3 isn't maximal.

3.4 Runtime

▶ **Lemma 4.** *The time complexity for finding all maximal* rect2DP *in a text of size $n_1 \times n_2$ (where $n_1 \geq n_2$) is $O(n_1 n_2^2)$.*

Proof. The runtime of the preprocessing stage is as follows: the construction of the GST is in time linear to the size of T [8]. Then it takes $O(n)$-time to preprocess to allow for constant-time LCP queries [13].

The scanning stage takes $O(n_1 n_2^2)$-time. This is because there are $n_1 \times n_2$ positions, and each takes $O(n_2)$ time for each **for** loop in Algorithm 2 (they run $O(min_c)$ times and $min_c < n_2$). Note that, following linear time preprocessing, a RMQ takes $O(1)$-time [9]. ◀

▶ **Lemma 5.** *A text T of size $n_1 \times n_2$ can have $O(n_1 n_2^2)$ maximal* rect2DP*.*

Proof. We prove by providing one such example. See Table 4 for an $n \times n$ text that has $O(n^3)$ maximal *rect2DP*. It contains a diamond composed of 0's, and the rest of the text has *'s which indicate unique, unused characters. On the right is a partial table of counts of how many *rect2DP*s are centered at the corresponding text positions. The other three quadrants of the diamond (whose counts are not shown) are reflections and have the same values. Beginning at the center ($T[\lceil n/2 \rceil, \lceil n/2 \rceil]$) and moving outward, an element's count is one less than that of its neighbor. Let i represent a row and j a column; summing the top left quadrant (in this case, from [1, 1] through [7, 6]) of the counts table is: $\sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=1}^{i-1} j = \sum_{i=1}^{\lfloor n/2 \rfloor} \frac{i(i+1)}{2} = O(\sum_{i=1}^{n} i^2)$. Since the sum of the squares of 1 to n is $(n)(n+1)(2n+1)/6 = O(n^3)$, this $n \times n$ input text has $O(n^3)$ maximal *rect2DP*. ◀

▶ **Theorem 6.** *Algorithm 2 has worst case running time proportional to the worst case output size.*

Proof. Combining Lemmas 4 and 5 results in the proof. ◀

4 Conclusion

In this paper, we discussed two types of 2D palindromes and presented efficient algorithms to find them. By unlocking the world of 2D palindromes, we released many research opportunities. Essentially all of the variations of the 1D palindrome problem can now be applied

Table 4 Shown on the left is a text that contains a cubic number of maximal *rect2DP*. The *'s indicate unique, unused characters. On the right is a partial table with counts, which are the exact number of *rect2DP* that are centered at the corresponding text positions.

	1	2	3	4	5	6	7	8	9	0	1	2	3
1	*	*	*	*	*	*	0	*	*	*	*	*	*
2	*	*	*	*	*	0	0	0	*	*	*	*	*
3	*	*	*	*	0	0	0	0	0	*	*	*	*
4	*	*	*	0	0	0	0	0	0	0	*	*	*
5	*	*	0	0	0	0	0	0	0	0	0	*	*
6	*	0	0	0	0	0	0	0	0	0	0	0	*
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	*	0	0	0	0	0	0	0	0	0	0	0	*
9	*	*	0	0	0	0	0	0	0	0	0	*	*
0	*	*	*	0	0	0	0	0	0	0	*	*	*
1	*	*	*	*	0	0	0	0	0	*	*	*	*
2	*	*	*	*	*	0	0	0	*	*	*	*	*
3	*	*	*	*	*	*	0	*	*	*	*	*	*

	1	2	3	4	5	6	7	8	9	0	1	2	3
1							1						
2						1	2						
3					1	2	3						
4				1	2	3	4						
5			1	2	3	4	5						
6		1	2	3	4	5	6						
7	1	2	3	4	5	6	7	6	5	4	3	2	1
8							6						
9							5						
0							4						
1							3						
2							2						
3							1						

to the 2D setting. First, we would like to look at how both types of 2D palindromes relate to palstars and gapped palindromes. Additionally, searching for approximate palindromes is something that would be interesting in 2D. Yet another extension is from [16] and [17], who study *pal-equivalence*. Two strings of the same length are pal-equivalent iff the length of the maximal palindrome at every center in the strings is equal.

Another angle for further research, in terms of *rect2DP*, is to reduce the runtime. Although we proved that the output size is potentially asymptotically larger than the input, an optimal algorithm would take time proportional to the actual number of non-trivial palindromes reported.

Finally, it would be interesting to define and study additional geometric shapes of 2D palindromes, e.g. triangular, circular and perhaps a hexagonal 2D palindrome.

References

1 Amihood Amir and Martin Farach. Two-dimensional Dictionary Matching. *Information Processing Letters*, 44(5):233–239, 1992. doi:10.1016/0020-0190(92)90206-B.

2 Amihood Amir and Benny Porat. Approximate On-line Palindrome Recognition, and Applications. In *number 8486 in Lecture Notes in Computer Science*, pages 21–29. Springer International Publishing, 2014. doi:10.1007/978-3-319-07566-2_3.

3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Optimal parallel algorithms for periods, palindromes and squares. In *Automata, Languages and Programming*, number 623 in Lecture Notes in Computer Science, pages 296–307. Springer Berlin Heidelberg, 1992.

4 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel Detection of all Palindromes in a String. *Comput. Sci*, pages 497–506, 1994.

5 Valérie Berthé and Laurent Vuillon. Palindromes and two-dimensional sturmian sequences. *Journal of Automata, Languages and Combinatorics*, 6(2):121–138, 2001.

6 Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao. Efficient retrieval of approximate palindromes in a run-length encoded string. *Theor. Comput. Sci.*, 432:28–37, 2012.

7 F.G.B. De Natale, Daniele D. Giusto, and Fabrizio Maccioni. A symmetry-based approach to facial features extraction. In *1997 13th International Conference on Digital Signal Processing Proceedings*, volume 2, pages 521–525, 1997. doi:10.1109/ICDSP.1997.628390.

8 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS'97, pages 137–143. IEEE Computer Society, 1997.

9 Johannes Fischer and Volker Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

10 fun-with words. History of Palindromes, 1999. URL: http://fun-with-words.com/palin_history.html.

11 Zvi Galil and Joel Seiferas. A Linear-Time On-Line Recognition Algorithm for "Palstar". *Journal of the ACM (JACM)*, 25(1):102–111, 1978. doi:10.1145/322047.322056.

12 Raffaele Giancarlo. A Generalization of the Suffix Tree to Square Matrices, with Applications. *SIAM Journal on Computing*, 24(3):520–562, 1995. doi:10.1137/S0097539792231982.

13 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2), 1984.

14 Amit Hooda, Michael M. Bronstein, Alexander M. Bronstein, and Radu P. Horaud. Shape Palindromes: Analysis of Intrinsic Symmetries in 2d Articulated Shapes. In *Lecture Notes in Computer Science*, volume 6667, pages 665–676, Israel, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-24785-9_56.

15 Ping-Hui Hsu, Kuan-Yu Chen, and Kun-Mao Chao. Finding All Approximate Gapped Palindromes. In *Algorithms and Computation*, number 5878 in Lecture Notes in Computer Science, pages 1084–1093. Springer Berlin Heidelberg, 2009.

16 Tomohiro I, Shunsuke Inenaga, and Masayuki Takeda. Palindrome Pattern Matching. In *Lecture Notes in Computer Science*, pages 232–245. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-21458-5_21.

17 Hwee Kim and Yo-Sub Han. Online Multiple Palindrome Pattern Matching. In *String Processing and Information Retrieval*, pages 173–178. Springer International Publishing, 2013. doi:10.1007/978-3-319-11918-2_17.

18 Nahum Kiryati and Yossi Gofman. Detecting Symmetry in Grey Level Images: The Global Optimization Approach. *International Journal of Computer Vision*, 29(1):29–45, 1998. doi:10.1023/A:1008034529558.

19 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.

20 Roman Kolpakov and Gregory Kucherov. Searching for Gapped Palindromes. In *Combinatorial Pattern Matching*, number 5029 in Lecture Notes in Computer Science, pages 18–30. Springer Berlin Heidelberg, 2008.

21 Glenn 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.

22 Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009. doi:10.1016/j.tcs.2008.12.016.

23 Wikipedia.org. Palindrome, 2015. URL: http://en.wikipedia.org/wiki/Palindrome.

24 Wikipedia.org. Sator Square, 2015. URL: http://en.wikipedia.org/wiki/Sator_Square.