Two Dimensional Parameterized Matching

Richard Cole* Carmit Hazay† Moshe Lewenstein‡ Dekel Tsur§

Abstract

Two equal length strings, or two equal sized two-dimensional texts, *parameterize match* (*p-match*) if there is a one-one mapping (relative to the alphabet) of their characters. *Two-dimensional parameterized matching* is the task of finding all $m \times m$ substrings of an $n \times n$ text that p-match an $m \times m$ pattern. This models searching for color images with changing of color maps, for example. We present two algorithms that solve the two-dimensional parameterized matching problem. The time complexities of our algorithms are $O(n^2 \log^2 m)$ and $O(n^2 + m^{2.5}\,\mathrm{polylog}(m))$. Our algorithms are faster than the $O(n^2 m \log^2 m \log\log m)$ time algorithm for this problem of Amir et al [2].

A key step in both of our algorithms is to count the number of distinct characters in every $m \times m$ substring of an $n \times n$ string. We show how to solve this problem in $O(n^2)$ time. This result may be of independent interest.

1 Introduction

Let S and S' be two equal length strings. We say that S and S' *parameterize match*, or *p-match* for short, if there is a bijection π from the alphabet of S to the alphabet of S' such that $S'[i] = \pi(S[i])$ for every index i. In the *parameterized matching problem*, introduced by Baker [9, 10], given an input comprising a text T and a pattern P, the goal is to find all the substrings of T of length $|P|$ that p-match P. Baker introduced parameterized matching for applications that arise in software tools for analyzing source code. Other applications for parameterized matching arise in image processing and computational biology (see [2]).

An optimal linear time algorithm for the parameterized matching problem when the alphabet has constant size was given in [9, 10]; an optimal algorithm in the presence of an unbounded size alphabet was given in [5]. In [9,10], the parameterized matching problem was solved by constructing *parameterized suffix trees*, which also allows for online p-matching.

*New York University, cole@cs.nyu.edu. This work was supported in part by NSF grants IIS-0414763, CCF-0515127, and CCF-0830516.

†University of Arhus, carmit@cs.au.dk.

‡Bar-Ilan University, moshe@cs.biu.ac.il. ML was supported by an IBM faculty award grant.

§Ben-Gurion University of the Negev, dekelts@cs.bgu.ac.il.

The parameterized suffix tree was further explored by Kosaraju [17] and a faster construction was given by Cole and Hariharan [13].

In [6], approximate parameterized matching was introduced and a solution for binary alphabets was given. In [16], an $O(nk^{1.5} + mk \log m)$ time algorithm was given for approximate parameterized matching with k mismatches, and a strong relation was shown between this problem and finding maximum matchings in bipartite graphs.

One of the interesting problems in web searching is searching for color images, see [4,8,18]. If the colors are fixed, this is exact two-dimensional pattern matching [3]. However, images can appear under different color maps: this maintains an unchanged partitioning of pixels by color, but each set of equal-colored pixels may have received a changed color. Two-dimensional parameterized search is precisely what is needed. An algorithm for two-dimensional parameterized matching was given in [2]; its time complexity is $O(n^2 m \log^2 m \log \log m)$ for an $n \times n$ text and an $m \times m$ pattern.

It is an open question whether a linear time algorithm for two-dimensional parameterized matching exists. In this paper we show two new algorithms for the problem. The first algorithm is almost linear in the input size, and the second algorithm is linear in the text size, but with a higher cost for preprocessing the pattern. The first algorithm is a convolution-based method that uses a novel reduction of the two-dimensional space to one dimension. The second algorithm is a dueling based solution that uses properties of the two-dimensional form of the problem. The first algorithm has time complexity $O(n^2 \log^2 m)$, and the second algorithm runs in time $O(n^2 + m^{2.5} \operatorname{polylog}(m))$.

A key step in both of our algorithms is to count the number of distinct characters in every $m \times m$ substring of an $n \times n$ string. Amir et al. [4] gave an $O(n^2 \log m)$ time algorithm for this problem; we show how to solve it in $O(n^2)$ time. This result may be of independent interest.

The rest of the paper is organized as follows. In Section 2, we start with some definitions and other preliminaries. Next, in Section 3, we present the $O(n^2 \log^2 m)$ time algorithm, and in Section 4, the $O(n^2 + m^{2.5} \operatorname{polylog}(m))$ algorithm. Finally, in Section 5, we describe the algorithm for substring character counting.

2 Preliminaries

Let S and S' be two-dimensional strings of equal size. We say that there is a *function matching* from S to S' if there is a mapping f from the alphabet of S to the alphabet of S' such that $S'[x, y] = f(S[x, y])$ for all x and y. If the mapping f is one-to-one, we say that S and S' *parameterize match*, or *p-match* for short. Note that the definition of function matching is asymmetric whereas the definition of parameterized matching is symmetric. The two-dimensional parameterized matching problem is defined as follows:

Input: An $n \times n$ text T and an $m \times m$ pattern P.
Output: All substrings of T of size $m \times m$ that p-match P.

Throughout the paper we assume that the alphabet of T is $\{1, \ldots, n^2\}$ and the alphabet

of P is $\{1, \ldots, m^2\}$.

Observation 1. *There is a parameterized matching between S and S' if and only if there is a function matching from S to S', and the number of distinct characters in S is equal to the number of distinct characters in S'.*

Observation 1 allows our algorithms to have the following structure. In their first step, our algorithms create a list L of $m \times m$ substrings of T such that:

1. Either there is a (separate) function matching from P to each string in L, or there is a (separate) function matching from each string in L to P.

2. Every substring of T that p-matches P appears in L.

The second step computes the number of distinct characters in every $m \times m$ substring of T. The strings in L that have the same number of distinct characters as P are precisely the substrings of T that p-match P. This stage takes $O(n^2)$ time using the algorithm in Section 5. Consequently, for each of our algorithms, which will be given in Sections 3 and 4 respectively, it suffices to describe the computation of the list L.

The *left-to-right/top-to-bottom traversal order* of a two-dimensional string is an ordering of the locations inside the string obtained by traversing the first (topmost) row in left to right order, then the second row from left to right, and so on. Other traversal orders are defined analogously.

We let $[a, b] \times [c, d]$ denote the set of all pairs (x, y) of integers with $a \leq x \leq b$ and $c \leq y \leq d$. Such a set is called a *rectangle*. We let $[a] \times [c, d]$ denote $[a, a] \times [c, d]$ and $[a, b] \times [c]$ denote $[a, b] \times [c, c]$. For a rectangle R define $R + (i, j) = \{(x + i, y + j) : (x, y) \in R\}$.

The usual array indexing is used for two-dimensional strings, namely the x coordinate indexes rows, increasing from top to bottom, and the y coordinate indexes columns, increasing from left to right.

Finally, we describe several exact matching problems, which will be used by our algorithms. In the *one-dimensional exact wildcard matching* problem, the input is a pattern P and a text T, both containing wildcard characters. The goal is to find all substrings of T of length $|P|$ that match P, where a wildcard character can match any character. The exact wildcard matching problem can be solved using convolutions.

Lemma 1 (Cole and Hariharan [12], Clifford and Clifford [11])**.** *The exact wildcard matching problem can be solved in $O(|T| \log |P|)$ time.*

In the *two-dimensional exact wildcard matching* problem, the pattern and text are two-dimensional strings.

Lemma 2. *The two-dimensional exact wildcard matching problem can be solved in $O(|T| \log |P|)$ time.*

Proof. Using standard technique, the two-dimensional problem can be reduced to the one-dimensional problem, and the lemma follows from Lemma 1. ∎

Next, we consider an extension of the previous problem, called *two-dimensional exact wildcard matching with witnesses*. The input to this problem is a pattern P and a text T. The goal is to find for each substring T' of T, whose size is the same as P and that doesn't match P, a *witness* to the mismatch, namely, a location (x, y) such that $T'[x, y]$ does not match $P[x, y]$.

Lemma 3. *The two-dimensional exact wildcard matching with witnesses problem can be solved in $O(|T| \operatorname{polylog} |P|)$ time.*

Proof. The theorem follows from combining the algorithm of Alon and Naor [1] (see also [7]) with the algorithm of Lemma 1. ■

Next, we consider the following problem, which we call *region matching with witnesses*: The input are two strings P_1 and P_2 with wildcard characters, and the goal is to compare substrings of these strings of equal sizes and find witnesses to the mismatches. We consider two variants of the problem: In the first variant, P_1 and P_2 are of size $n \times n$, and the substrings that are compared are $P_1[1\,..\,n-i, 1\,..\,n-j]$ and $P_2[i+1\,..\,n, j+1\,..\,n]$ for all i and j. In the second variant, P_1 is of size $(2n-1) \times n$ and P_2 is of size $n \times m$, where $m \leq n$. The substrings that are compared are all substrings of P_1 and P_2 of size $n \times n$.

Lemma 4. *The region matching with witnesses problem can be solved in $O(|P_2| \operatorname{polylog} |P_1|)$ time.*

Proof. The problem can be reduced to the exact wildcard matching with witnesses problem as follows. For the first variant, construct a string P_2' of size $(2n-1) \times (2n-1)$ by adding $n-1$ rows and $n-1$ columns of wildcard characters to P_2. More precisely, $P_2'[x, y] = P_2[x, y]$ if $x \leq n$ and $y \leq n$, and otherwise $P_2'[x, y]$ is a wildcard. Now, $P_1[1\,..\,n-i, 1\,..\,n-j]$ matches $P_2[i+1\,..\,n, j+1\,..\,n]$ if and only if P_1 matches $P_2'[i+1\,..\,i+m, j+1\,..\,j+m]$.

For the second variant, construct a string P_2' from P_2 by adding $2n-2$ rows of wildcards, where $n-1$ rows are added above the original rows of P_2, and $n-1$ rows are added below the original rows. The substring $P_1[i+1\,..\,i+n, 1\,..\,n]$ matches the substring $P_2[1\,..\,n, j+1\,..\,j+n]$ if and only if P_1 matches $P_2'[1\,..\,n-i, j+1\,..\,j+n]$. ■

3 An $O(n^2 \log^2 m)$ algorithm

In this section we present our first algorithm for solving the two-dimensional parameterized matching problem. The key idea of this algorithm is to encode the pattern and text by replacing each character with the "distance" to other occurrences of the same character in the pattern or text. Using this encoding, the parameterized matching problem is reduced to exact wildcard matching. This approach was also used in the algorithm for one-dimensional parameterized matching due to Amir et al [5]. However, the encoding in the two-dimensional case is much more complex. We will begin this section with a description of the algorithm of Amir et al. Then, we will give a simple inefficient extension of this approach to two dimensions. Finally, we will describe our more efficient algorithm.

The algorithm of Amir et al. encodes each character of the pattern (or text) by the distance to the nearest occurrence of the same character to the left, or by 0 if there is no such occurrence. Except for the first occurrence of each character, a parameterized match in the original text and pattern corresponds to a standard match in the recoded text and pattern. The algorithm compares the encoded strings using a variant of the Knuth-Morris-Pratt algorithm. For our purpose, consider the following less efficient variant of the algorithm. Encode P and T into strings P_2 and T_2 as described above, except that a character in P which has no occurrence of the same character to its left is encoded by ϕ. Treating ϕ as a wildcard, it is now easy to see the following.

Lemma 5. *For every substring T' of T of length $|P|$, and for the corresponding substring T'_2 of T_2,*

1. *If there is a parameterized matching between P and T' then there is an exact wildcard matching between P_2 and T'_2.*
2. *If there is an exact wildcard matching between P_2 and T'_2 then there is a function matching from P to T'.*

Example 1. *Let $P =$* abacb *and $T =$* yxyxxy*. Then, $P_2 = \phi\phi 2\phi 3$ and $T_2 = 002213$. There is a function matching from P to $T[2\,..\,6]$, and an exact wildcard matching between P_2 and $T_2[2\,..\,6]$.*

By Lemma 5, the one-dimensional parameterized matching problem can be solved by solving the wildcard matching problem between P_2 and T_2 and computing the number of distinct characters in P and in every substring of T of length $|P|$.

A straightforward extension of this algorithm into two dimensions is to encode each character in the pattern or text using several characters. For each location (x, y) in the pattern (or text) define $2m-1$ disjoint rectangles $R_{(x,y),-(m-1)}, R_{(x,y),-(m-2)}, \ldots, R_{(x,y),m-1}$, where $R_{(x,y),0} = [x] \times [1, y-1]$, and for $i \neq 0$, $R_{(x,y),i} = [x+i] \times [1, y]$. Note that if $y = 1$ the rectangle $R_{(x,y),0}$ is empty. A rectangle in the pattern (resp., text) is *active* if it is non-empty and all the locations of the rectangle fit inside the pattern (resp., text), namely, the rectangle is contained in $[1, m] \times [1, m]$ (resp., $[1, n] \times [1, n]$). The remaining rectangles are said to be *inactive*. Each active rectangle is traversed from right to left so as to identify the first location (x', y'), if any, that contains the same character as (x, y). The location (x', y') is called a *neighbor* of (x, y). The inactive rectangle will not generate neighbors. The text and pattern are encoded as follows: Each location (x, y) in the pattern is encoded by a sequence of $2m-1$ characters $c_1 \cdots c_{2m-1}$. The character c_i is induced by the i-th rectangle of (x, y). If the i-th rectangle of (x, y) produces a neighbor (x', y'), then c_i is the rank of the location (x', y') in the traversal order of the rectangle. If the i-th rectangle does not produce a neighbor, then $c_i = \phi$. The text is encoded similarly, except that when a rectangle yields no neighbor, $c_i = 0$. Let P_2 and T_2 denote the encoded pattern and text. See Figure 1 for an example.

We now prove that Lemma 5 also holds for the two-dimensional case. We first give some definitions that will be used in the proof.

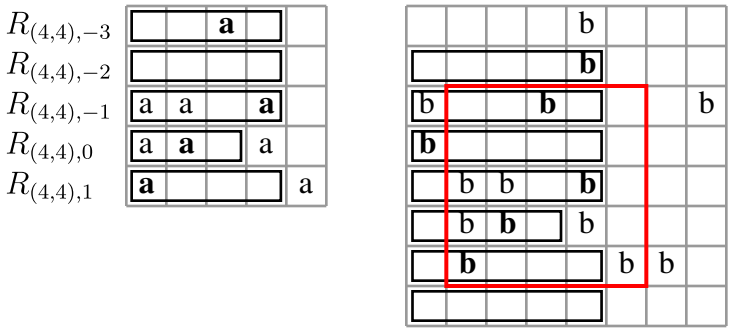


Figure 1: An example of the simple algorithm for a pattern of size 5×5 (left) and a text of size 8×8 (right). Assume that in this example, P p-matches $T' = T[3\,..\,7, 2\,..\,6]$ (the substring T' is marked in the figure). The active rectangles for location $(4, 4)$ in P and location $(6, 5)$ in T are shown (the active rectangles for these locations are $R_{(4,4),-3}, \ldots, R_{(4,4),1}$ and $R_{(6,5),-4}, \ldots, R_{(6,5),2}$, respectively), and the selected neighbors in the active rectangles are marked in bold. Location $(4, 4)$ is encoded by $\phi2\phi124\phi\phi\phi$, location $(6, 5)$ is encoded by 125124000, and these two strings match. The neighbors of $(4, 4)$ in P are aligned with the neighbors of $(6, 5)$ in T.

Definition 1 (linked locations)**.** *Two locations (x, y) and (x', y') in P (or T) are* linked *if there is a series of locations $(w_0, z_0) = (x, y), (w_1, z_1), (w_2, z_2), \ldots, (w_l, z_l) = (x', y')$ such that for all i, either (w_i, z_i) is a neighbor of (w_{i+1}, z_{i+1}), or (w_{i+1}, z_{i+1}) is a neighbor of (w_i, z_i) (or possibly both).*

Definition 2 (aligned neighbors)**.** *The neighbors of location (x, y) in P are* aligned *with the neighbors of location $(x + a, y + b)$ in T if for all i, if the i-th rectangle of (x, y) in P produces a neighbor (x', y') then the i-th rectangle of $(x+a, y+b)$ in T produces the neighbor $(x' + a, y' + b)$.*

Observation 2. *Every two locations in P holding the same characters are linked.*

Lemma 6. *Let $T' = T[a + 1\,..\,a + m, b + 1\,..\,b + m]$ be some substring of T.*

1. *If P p-matches T' then for every location (x, y) in P, the neighbors of (x, y) in P are aligned with the neighbors of $(x + a, y + b)$ in T.*
2. *If for every location (x, y) in P, the neighbors of (x, y) in P are aligned with the neighbors of $(x + a, y + b)$ in T, then there is a function matching from P to T'.*

Proof. Part 1 of the observation follows from the construction. To prove Part 2, consider two locations (x_1, y_1) and (x_2, y_2) in P with $P[x_1, y_1] = P[x_2, y_2]$. By Observation 2, (x_1, y_1) is linked with (x_2, y_2) in P. Due to the assumption that the neighbors of a location (x, y) in P are aligned with the neighbors of $(x + a, y + b)$ in T for every (x, y), it follows that (x_1+a, y_1+b) is linked with (x_2+a, y_2+b) in T. In particular, $T[x_1+a, y_1+b] = T[x_2+a, y_2+b]$. Since this holds for every two locations in P that contain the same character, we conclude that there is a function matching from P to T'. ■

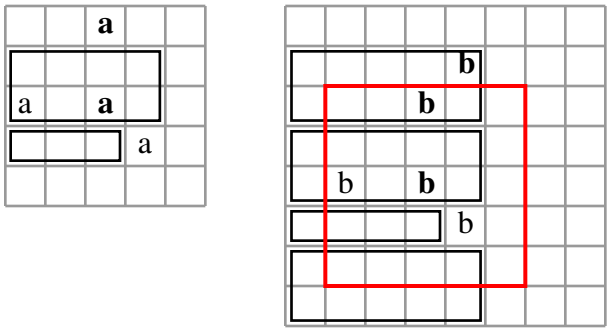


Figure 2: An example of using rectangles of height 2. The pattern P is shown on left and the text T on the right. Suppose that P p-matches $T[3\,..\,7, 2\,..\,6]$. The rectangles of location $(4,4)$ in P are $[0,1]\times[1,4]$, $[2,3]\times[1,4]$, $[4]\times[1,3]$, $[5,6]\times[1,4]$, $[7,8]\times[1,4]$, where only the second and third rectangles are active. Location $(4,4)$ is encoded by $\phi 4\phi\phi\phi$, and location $(6,5)$ in T is encoded by 14000. The first rectangle of $(4,4)$ cannot be active, otherwise it would generate the neighbor $(1,3)$. However, the first rectangle of $(6,5)$ generates the neighbor $(2,5)$, and therefore, the neighbors of $(4,4)$ would not be aligned with the neighbors of $(6,5)$.

We note that the converse of Part 1 of the lemma does not hold, namely, if P p-matches T' then the neighbors of a location $(x+a, y+b)$ in T' need not be aligned with neighbors of (x,y) in P. For example, in Figure 1, location $(6,5)$ in T has a neighbor $(4,1)$ which has no corresponding neighbor in P.

The correctness of Lemma 5 now follows directly from Lemma 6. The encoded pattern P_2 and the encoded text T_2 have sizes $O(m^3)$ and $O(mn^2)$, respectively. Thus, by Lemma 2, solving the wildcard matching problem on P_2 and T_2 takes $O(mn^2 \log m)$ time.

How can the above algorithm be improved? Since the time complexity depends on the number of neighbors chosen for each location, a natural improvement is to replace the rectangles of height 1 by taller rectangle, say of height 2. Each active rectangle will be scanned in a top-to-bottom/right-to-left order. Recall that a non-empty rectangle is active if all the locations of the rectangle fit inside the pattern. This is essential as otherwise Lemma 6 will not hold (see Figure 2). However, this creates another problem, as two locations in P holding the same character are not necessarily linked (namely, Observation 2 does not hold), as shown in Figure 3. Note that this happens when the leftmost location is near the boundary of the pattern. To overcome this problem, we need a better choice of rectangles that ensures the linked characters property.

Our solution has the following form. Let $t = \lceil \log_2 m \rceil$. For each location (x,y) in the pattern we define $4t+4$ disjoint rectangles. This construction is illustrated in Figure 4. The first four rectangles comprise row x and column y partitioned at location (x,y), i.e., the rectangles are $[x]\times[y+1,m]$, $[x]\times[1,y-1]$, $[x+1,m]\times[y]$, and $[1,x-1]\times[y]$. Next, we define t disjoint rectangles $R_{(x,y),0},\ldots,R_{(x,y),t-1}$ that cover the quadrant below and to the

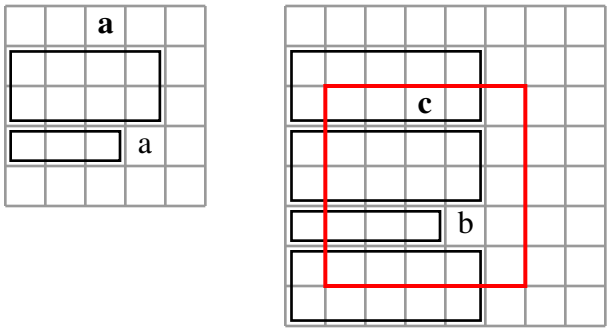


Figure 3: An example showing that using rectangles of height 2 does not ensure that locations holding the same character are linked. In the pattern P (left), the locations $(1,3)$ and $(4,4)$ are not linked as the rectangle of $(4,4)$ that contains $(1,3)$ is not active. Thus, the character of this rectangle is ϕ. Assuming that the non-shown characters of $T' = T[3\,..\,7, 2\,..\,6]$ are equal to the corresponding characters of P, the encoded string P_2 matches the substring of T_2 corresponding to T', while there is no function matching from P to T'.

right of (x,y). For $i = 0,\ldots,t-2$,

$$R_{(x,y),i} = [x+m-2^{i+1}+1, x+m-2^i] \times [y+1, m]$$

and

$$R_{(x,y),t-1} = [x+1, x+m-2^{t-1}] \times [y+1, m].$$

For the quadrant above and to the left of (x,y) we define the rectangles

$$S_{(x,y),i} = [x-m+2^i, x-m+2^{i+1}-1] \times [1, y-1]$$

for $i = 0,\ldots,t-2$, and

$$S_{(x,y),t-1} = [x-m+2^{t-1}, x-1] \times [1, y-1].$$

Analogous rectangles are defined for the remaining two quadrants.

The rectangles of a location (x,y) in the text are defined similarly. The only difference is that the rectangles that are to the right of column y now extend until column n, and the rectangle in column y below x now extends to row n. Note that the number of rectangles and their rows are still defined in terms of m.

We now define the scan order of the rectangles. Each active rectangle is traversed in a direction away from (x,y) until the first occurrence of $P[x,y]$ is found. More precisely, the traversal order is top-to-bottom/left-to-right in the rectangles to the right of column y, top-to-bottom/right-to-left in the rectangles to the left of column y, top-to-bottom in the rectangle $[x+1, m] \times [y]$, and bottom-to-top in the rectangle $[1, x-1] \times [y]$.

The following lemma shows that our construction has the linked characters property.

Lemma 7. *Let (x,y) and (x',y') be two locations in the pattern both containing the same character. Then these two locations are linked.*

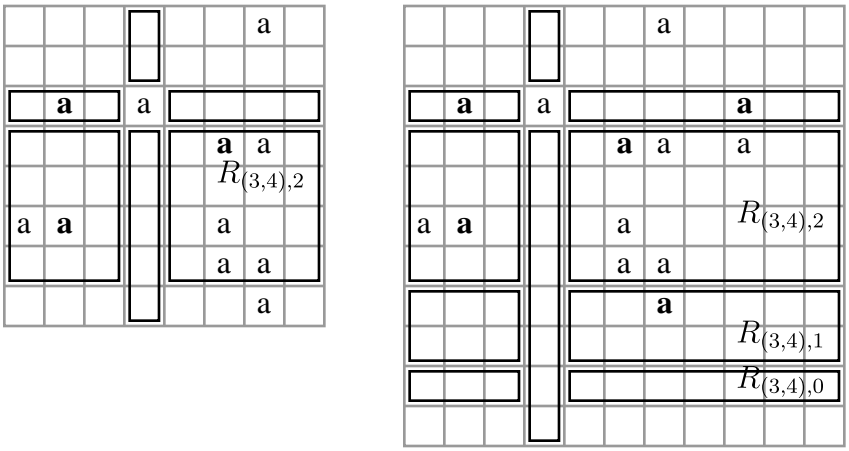


Figure 4: An example of neighbor selection for a pattern of size 8×8 (left) and a text of size 11×11 (right). The neighbors of $(3, 4)$ in P and T are marked in bold. The active rectangles for location $(3, 4)$ in P and T are shown in the figure. The rectangles for location $(3, 4)$ in P in the bottom-right quadrant are $R_{(3,4),0} = [10] \times [5, 8]$, $R_{(3,4),1} = [8, 9] \times [5, 8]$, and $R_{(3,4),2} = [4, 7] \times [5, 8]$. Only the last rectangle is active. The corresponding rectangles for location $(3, 4)$ in T are $[10] \times [5, 12]$, $[8, 9] \times [5, 12]$, and $[4, 7] \times [5, 12]$. All these rectangles are active. In this example P p-matches $T[1\,..\,8, 1\,..\,8]$, so by Lemma 6, the neighbors of $(3, 4)$ in P are aligned with the neighbors of $(3, 4)$ in T.

Proof. Clearly, if $x = x'$ then there is a series of locations along row x linking the locations (x, y) and (x', y'). Similarly, these two locations are linked if $y = y'$. We now consider the case that $x \neq x'$ and $y \neq y'$. W.l.o.g. suppose that $x < x'$ and $y < y'$.

We claim that either (x, y) lies in one of the active rectangles of (x', y') or (x', y') lies in one of the active rectangles of (x, y) (or possibly both).

W.l.o.g. suppose that $x \leq m/2$ (otherwise the roles of the two locations can be exchanged). If (x', y') is inside one the active rectangles of (x, y) then the claim holds. Otherwise, there must be at least one inactive rectangle among $R_{(x,y),0}, \ldots, R_{(x,y),t-1}$. Let $R_{(x,y),j}$ be the topmost inactive rectangle. As $x \leq m/2$, it follows that $R_{(x,y),t-1}$ is active, so $j \leq t-2$. The location (x', y') must be inside $R_{(x,y),j}$ (since the index of the last row of $R_{(x,y),j}$ is larger than m). As easy calculation shows that (x, y) is inside $S_{(x',y'),j}$ also.

Now we will show that $S_{(x',y'),j}$ is active. Since $(x', y') \in R_{(x,y),j}$, $x' \geq x + m - 2^{j+1} + 1$. Moreover, the fact that $R_{(x,y),j}$ does not fit inside the pattern implies that $x + m - 2^j > m$, namely that $x > 2^j$. Thus, $x' > 2^j + m - 2^{j+1} + 1 = m - 2^j + 1$, so $x' - m + 2^j > 1$. The last inequality implies that the rectangle $S_{(x',y'),j}$ is active. This shows that (x, y) is inside an active rectangle of (x', y').

We have shown that either (x, y) lies in one of the active rectangles of (x', y'), or (x', y') lies in one of the active rectangles of (x, y). W.l.o.g. suppose that the latter occurs. It need not be that (x', y') is a neighbor of (x, y), however. Nonetheless, by induction on $y' - y$, we show that they are linked. The base case, with $y = y'$, has already been demonstrated. Let

(x'', y'') denote the neighbor of (x, y) in the rectangle containing (x', y'). Then $y < y'' \leq y'$. By induction, (x'', y'') and (x', y') are linked and the inductive claim follows. ■

Using the new neighbor selection scheme, the algorithm constructs strings P_2 and T_2 as described above. The sizes of these strings are $O(m^2 \log m)$ and $O(n^2 \log m)$, respectively. Thus, by Lemma 2, the wildcard matching problem on P_2 and T_2 can be solved in time $O(n^2 \log^2 m)$.

It remains to show how to identify the neighbors. This is readily done in $O(m^2 \log^2 m)$ time in the pattern and $O(n^2 \log^2 m)$ time in the text. We describe the approach for the pattern. The method for the text is the same.

Finding the neighbors in the rectangles $[x] \times [y+1, m]$, $[x] \times [1, y-1]$, $[x+1, m] \times [y]$, and $[1, x-1] \times [y]$ for all locations (x, y) is done by means of simple scans of the rows and columns of the pattern.

Next, we describe how to find the neighbors in the remaining rectangles. The idea is to maintain, for each character c, windows $w_1, \ldots, w_{t-1}$ over the pattern. Window w_i has a width of m columns and a height of 2^i rows (except for window w_{t-1} which has height $m - 2^{t-1}$). For each i, window w_i is slid down the pattern, row by row. During the movement of w_i, the occurrences of c inside w_i are kept in a balanced search tree in top-to-bottom/left-to-right order. Let (x, y) be a location in P that contains the character c. Its neighbor in rectangle $R_{(x,y),i}$ is found when the window w_i covers the rows of $R_{(x,y),i}$, by searching the binary search tree for w_i.

Each search takes $O(\log m)$ time. Thus, over all characters and neighbors, the searches take $O(m^2 \log^2 m)$ time. To slide a window one row down entails deleting some character instances and adding others. This takes time $O(\log m)$ per change, and as each character instance is added once and deleted once from a window of each size, this takes time $O(m^2 \log^2 m)$ over all characters and windows.

We have shown:

Theorem 8. *There is an algorithm for two-dimensional parameterized matching that runs in time $O(n^2 \log^2 m)$ on an $n \times n$ text and an $m \times m$ pattern.*

4 An $\mathbf{O(n^2 + m^{2.5}\,\mathrm{polylog}(m))}$ algorithm

In this section we present another algorithm for parameterized matching which follows the "duel-and-sweep" paradigm. This paradigm appeared in [3], where it was named "consistency and verification" and was used for two-dimensional exact matching; it is based on the dueling technique [19, 20].

We begin by giving an overview of the "duel-and-sweep" algorithm for two-dimensional *exact* matching. The algorithm maintains a list of *candidates* that initially contains all $m \times m$ substrings of T. The list is pruned in two stages, called the *dueling stage* and the *sweeping stage*, after which the list will contain exactly those substrings of T that match P.

The following notation will be helpful. $T_{x,y}$ denotes $T[x\,..\,x+m-1, y\,..\,y+m-1]$, the $m \times m$ substring of T with top left corner at location (x, y), also called its *start* location.

Consider two overlapping candidate substrings $T_1 = T_{x,y}$ and $T_2 = T_{x+a,y+b}$, with $a \geq 0$.

- If both candidates match P it follows that when P is aligned with itself with offset (a, b), the overlapping areas in the two copies of P match. In other words, $P_1 = P[a+1\,..\,m, b+1\,..\,m]$ matches $P_2 = P[1\,..\,m-a, 1\,..\,m-b]$.
- If P_1 does not match P_2, then T_1 and T_2 cannot both match P. If T_1 matches, P_1 matches when aligned with T_1's bottom right corner, and if T_2 matches, P_2 matches when aligned with T_2's top left corner.

So if P_1 and P_2 do not match, at least one of the candidates T_1 and T_2 can be ruled out. This is done in constant time using the following process, called *dueling*.

Let (x', y') be a location in P_1 such that $P_1[x', y'] \neq P_2[x', y']$. The location (x', y') is called a *witness* for (a, b). Let $c = T[x + a + x' - 1, y + b + y' - 1]$. Clearly, either $c = P_1[x', y']$, $c = P_2[x', y']$, or neither of these equalities hold, but both equalities cannot hold. In the first case $c \neq P_2[x', y'] = P[x', y']$, so T_2 does not match P. In the second case $c \neq P_1[x', y'] = P[x' + a, y' + b]$, so T_1 does not match P. In the last case, neither T_1 nor T_2 match P.

In order to perform duels between candidates, the algorithm precomputes a *witness table* that contains a witness for every mismatch offset (a, b). After performing all possible duels between the candidates, the remaining candidates are pairwise consistent. Amir et al. [3] showed how to perform the duels in $O(n^2)$ time. The dueling stage of Amir et al. relies on a transitivity property of the consistency relation, which follows from the fact that exact matching is a transitive relation. We note that parameterized matching is also transitive; this fact allows us to use the dueling stage of Amir et al. in our algorithm.

In the sweeping stage, the algorithm checks for each remaining candidate whether it matches the pattern. By using the fact that the candidates are pairwise consistent, this stage too can be implemented in $O(n^2)$ time.

As in many pattern matching algorithms, we assume w.l.o.g. that $n = 2m$. Larger texts can be cut into overlapping pieces of size $2m \times 2m$ which are handled independently, where successive pieces overlap in $m - 1$ rows or columns.

In the next sections we describe how to adapt the "duel-and-sweep" algorithm to two-dimensional parameterized matching. In Section 4.1 we describe the dueling stage, and in Section 4.2 we describe the sweeping stage. In Sections 4.3 and 4.4 we describe preprocessing stages that are required for the sweeping stage and dueling stage, respectively.

4.1 Dueling stage

In this section we describe the dueling stage of our algorithm. In exact matching a witness is simply a location with a mismatch between the two aligned copies of the pattern. However, in parameterized matching two locations are needed to rule out a match, as specified in the following definitions. See figure 5.

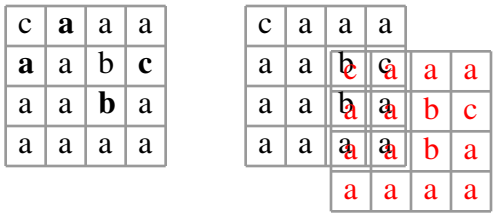


Figure 5: An example of mismatch offset and witnesses. For the pattern P on the left, the offset $(1,2)$ is a mismatch offset. The alignment of P with itself with this offset is shown on the right. The pair $(2,1),(1,2)$ is a type 1 witness for $(1,2)$ as $P[2,1] = P[1,2]$ and $P[3,3] \neq P[2,4]$. The pair $(1,1),(2,1)$ is a type 2 witness for $(1,2)$.

Definition 3 (mismatch offset)**.** *Let P be a pattern of size $m \times m$. An offset (a,b) with $b \geq 0$ is called a* mismatch offset of P *if when P is aligned with itself with offset (a,b), the overlapping areas in the two copies of P do not p-match (in other words, if $a \geq 0$ then (a,b) is a mismatch offset if $P[1\,..\,m-a, 1\,..\,m-b]$ does not p-match $P[a+1\,..\,m, b+1\,..\,m]$, and if $a < 0$ then (a,b) is a mismatch offset if $P[a+1\,..\,m, 1\,..\,m-b]$ does not p-match $P[1\,..\,m-a, b+1\,..\,m]$.)*

Definition 4 (witness)**.** *Let (a,b) be a mismatch offset of pattern P. A* witness *for (a,b) is a pair of locations $(x,y),(x',y')$ such that one of the following holds:*

1. *$P[x,y] = P[x',y']$ and $P[x+a, y+b] \neq P[x'+a, y'+b]$.*
2. *$P[x,y] \neq P[x',y']$ and $P[x+a, y+b] = P[x'+a, y'+b]$.*

The witness is called a type 1 witness *if the first condition holds, and otherwise it is a* type 2 witness*.*

Given a witness table for P, the dueling stage for parameterized matching is performed by using the dueling algorithm of Amir et al. [3]. In Section 4.4 we explain how to construct the witness table for P.

4.2 Sweeping stage

After the dueling stage, we are left with a list of candidate locations in T that are pairwise consistent, namely for every two candidates $T_{x,y}$ and $T_{x',y'}$ with $x \geq x'$, the offset $(x-x', y-y')$ is not a mismatch offset. The goal of the sweeping stage is to check for each candidate $T_{x,y}$, whether there is a function matching from P to $T_{x,y}$. Checking one candidate can be trivially done in $O(m^2)$ time, but the number of candidates can be $\Theta(n^2)$, so in order to obtain $O(n^2)$ time for this stage, we need to design a way to check all candidates in parallel.

The main ideas of this step are as follows. First, we handle each character of the alphabet separately. That is, for every character c, we check for each candidate $T_{x,y}$ whether if has the following property, which we call *consistency w.r.t. c*: For all locations (x',y') inside the

candidate (i.e., locations of T inside the rectangle $[x, x+m-1] \times [y, y+m-1]$) holding the character c, the corresponding locations $(x'-x+1, y'-y+1)$ in P hold the same character. There is a function matching from P to $T_{x,y}$ if and only if $T_{x,y}$ has the consistency property w.r.t. every character c.

Fix a character c. If we consider a single candidate T_{x_1,y_1}, we can check whether the candidate has the consistency property w.r.t. c by arbitrarily selecting an ordering of all locations holding c inside T_{x_1,y_1}, and then for each two consecutive locations (x'_1, y'_1) and (x'_2, y'_2) in the ordering, check whether the corresponding locations $(x'_1 - x_1 + 1, y'_1 - y_1 + 1)$ and $(x'_2 - x_1 + 1, y'_2 - y_1 + 1)$ in P hold the same character. Now, in order to check whether another candidate T_{x_2,y_2} has the consistency w.r.t. c property, we need to perform character equality tests for pairs of locations in P corresponding to pairs of locations holding c inside T_{x_2,y_2}. If the candidates T_{x_1,y_1} and T_{x_2,y_2} overlap, the equality tests due to pairs inside the overlapping area are the same (assuming the ordering selected for the locations holding c in these two candidate are consistent), and thus can be performed only once. In case of an inequality in one of the tests, both T_{x_1,y_1} and T_{x_2,y_2} can be ruled out.

Now suppose that we want to check all candidates for consistency w.r.t. c. We extend the idea presented above by using a partition of the text into strips of width m, and ordering all locations holding c inside a strip according to a left-to-right/top-to-bottom traversal order. For each two consecutive locations holding c in the strip, we will perform a character equality test in P, and in case of an inequality, we will rule out the candidates containing the two locations.

We now formalized the ideas described above.

Definition 5 (strip)**.** *Let T be an $n \times n$ text.* Strip *i of T is the rectangle $[1, n] \times [i, i+m-1]$.*

Definition 6 (predecessor)**.** *Let T be an $n \times n$ text, S a strip of T, and (x, y) a location inside S. The* predecessor of (x, y) w.r.t. S*, if any, holds the same character as $T[x, y]$ and is the first such location encountered when traversing S in right-to-left/bottom-to-top order starting from (x, y).*

See Figure 6 for an example illustrating the above definitions. A location and its predecessor in T will be called a *location-predecessor pair*, or a *predecessor-location pair* if the predecessor is listed first. Recall that a candidate is an $m \times m$ substring of T that survived the dueling stage. We note that candidate $T_{x,y}$ is contained in Strip y.

Definition 7 (mismatch pair)**.** *Let $T_{x,y}$ be a candidate. A* mismatch pair *for $T_{x,y}$ is a pair of locations $(x_1, y_1), (x_2, y_2)$ contained in $T_{x,y}$, such that $T[x_1, y_1] = T[x_2, y_2]$ and $P[x_1 - x + 1, y_1 - y + 1] \neq P[x_2 - x + 1, y_2 - y + 1]$.*

Observation 3. *1. If P p-matches $T_{x,y}$ then there are no mismatch pairs for $T_{x,y}$.*

2. If there is no predecessor-location pair w.r.t. Strip y that is a mismatch pair for $T_{x,y}$ then there is a function matching from $T_{x,y}$ to P.

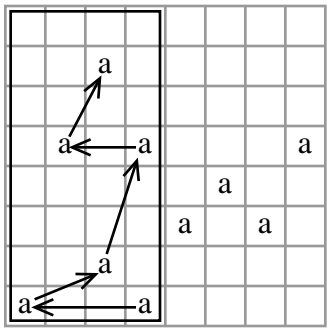


Figure 6: An example for the definition of predecessor. Here $m = 4$ and $n = 8$, so Strip 1 of T is the rectangle $[1, 8] \times [1, 4]$. The predecessors w.r.t. Strip 1 of the locations that contain the character 'a' are shown in the figure ($(8, 1)$ is the predecessor of $(8, 4)$, $(7, 3)$ is the predecessor of $(8, 1)$ etc.).

A *surviving candidate* is a candidate that wasn't ruled out previously by the algorithm. By Observation 3, it suffices to check for each surviving candidate $T_{x,y}$ whether the predecessor-location pairs it contains (w.r.t. to Strip y) are mismatch pairs for $T_{x,y}$. The candidates produced by the dueling stage are pairwise consistent. Therefore, for each predecessor-location pair w.r.t. Strip y, the pair either is a mismatch pair for every candidate (contained in Strip y or in some other strip) that contains the pair, or is a mismatch pair for no such candidate. Thus, for each predecessor-location pair w.r.t. Strip y it suffices to check just two characters in P, and if there is a mismatch this rules out all the surviving candidates containing the pair. By processing the pairs in an appropriate order, on finding a mismatch it will be enough to rule out the candidates with start location in columns $y' \geq y$ (and, in fact, these will be all the surviving candidates containing the pair).

First, we explain how to compute all predecessor-location pairs. This is done by means of a sweep across T's strips from left to right. For the current strip, for each character value c, the algorithm maintains a doubly linked list of locations in the current strip than contain c, ordered by the predecessor relation. Predecessor-location pairs are simply list neighbors. Computing the predecessor of every location in the first strip takes $O(m^2)$ time. Going from Strip $(y - 1)$ to Strip y creates at most $3n = 6m$ new predecessor-location pairs of the following types:

(1) Every location in column $y + m - 1$ and its predecessor, if any, form a new pair.

(2) Similarly, every location in column $y + m - 1$ and its successor, if any, form a new pair.

(3) The Strip $(y - 1)$ predecessor and successor of a location in column $y - 1$, if both are present, may form a new predecessor-location pair.

After a preprocessing step on T that will be described in Section 4.3, all the new predecessor-location pairs for Strip y are found in $O(m)$ time. Over all strips, this takes $O(m^2)$ time.

To find mismatch pairs for the current strip, it suffices to check those predecessor-location pairs which are new (i.e., did not appear in the previous strip), as the old pairs were already checked in previous iterations. Let $(x_1, y_1), (x_2, y_2)$ be a new predecessor-location pair for Strip y. As noted above, $(x_1, y_1), (x_2, y_2)$ is either a mismatch pair for all the candidates that contain the pair, or for none. Therefore, it suffices to find just one candidate $T_{x',y'}$ that contains the pair. Then checking whether $P[x_1 - x' + 1, y_1 - y' + 1] = P[x_2 - x' + 1, y_2 - y' + 1]$ suffices. If these two characters are not equal, then $T_{x',y'}$ cannot be a match, and moreover, no candidate that contains the pair $(x_1, y_1), (x_2, y_2)$ can be a match.

We need to handle two issues: how to find a candidate that contains the pair $(x_1, y_1), (x_2, y_2)$, and in the event of a mismatch, how to rule out the candidates starting in the corresponding rectangle.

First, we explain how to find a candidate. The following lemma gives the possible start locations of the candidates that we need to consider when handling some predecessor-location pair.

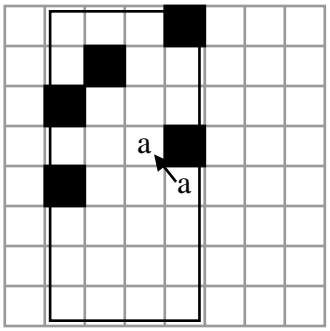
Lemma 9. *Let $(x_1, y_1), (x_2, y_2)$ be a predecessor-location pair in Strip y, which is not a predecessor-location pair in Strip $(y-1)$. Let $T_{x',y'}$ be a surviving candidate such that $(x_1, y_1), (x_2, y_2)$ is a mismatch pair for $T_{x',y'}$. Then, $(x', y') \in [x_2 - m + 1, x_1] \times [y, \min(y_1, y_2)]$.*

Proof. The rectangle of $T_{x',y'}$ is $[x', x' + m - 1] \times [y', y' + m - 1]$. Therefore, the fact $T_{x',y'}$ contains both (x_1, y_1) and (x_2, y_2) implies that $(x', y') \in [\max(x_1, x_2) - m + 1, \min(x_1, x_2)] \times [\max(y_1, y2) - m + 1, \min(y_1, y_2)]$. From the definition of a predecessor-successor pair, $x_1 \leq x_2$, so $[\max(x_1, x_2) - m + 1, \min(x_1, x_2)] = [x_2 - m + 1, x_1]$.

To finish the proof of the lemma, we argue that $y' \geq y$. For if the pair $(x_1, y_1), (x_2, y_2)$ was created by (1) or (2) above, then $\max\{y_1, y_2\} = y + m - 1$ and the claim follows as it was shown above that $y' \geq \max(y_1, y2) - m + 1$. If the pair was created by (3) and $y' < y$, then, in Strip $(y-1)$, at least one of the predecessor-location pairs including (x_1, y_1) or (x_2, y_2) would have been a mismatch pair for candidate $T_{x',y'}$, which would therefore have been eliminated as a candidate already. ■

In order to find a surviving candidate that contains a predecessor-location pair $(x_1, y_1), (x_2, y_2)$, we use Lemma 9. Among all the candidates that survived the dueling stage with start locations inside the rectangle $[x_2 - m + 1, 2m] \times [y, \min(y_1, y_2)]$, we find a candidate $T_{x',y'}$ that minimizes x' (ties are broken arbitrarily). Clearly, if no such candidate exists, or if the candidate $T_{x',y'}$ exists but does not contain the predecessor-location pair (that is, if $x' > x_1$), then there are no candidates for which the pair could be a mismatch pair, and we can continue to the next predecessor-location pair. Note that $T_{x',y'}$ may have been already ruled out during the sweeping stage, but we can still use this candidate to check whether the predecessor-location pair is a mismatch pair for the surviving candidates that contain it.

Next, we describe how to find the candidate $T_{x',y'}$ in constant time. This process uses a $2m \times 2m$ array A, where $A[i, j]$ is the smallest integer $r \geq i$ such that (r, j) is the start location of a candidate. If there is no such integer then $A[i, j] = 2m$. The array A is computed at the beginning of the sweeping stage by scanning the text column by column, from bottom to top (for the candidates are already at hand and can be associated with their start locations



(a) T

8	3	2	8	1	8	8	8
8	3	2	8	4	8	8	8
8	3	8	8	4	8	8	8
8	5	8	8	4	8	8	8
8	5	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8

(b) The array A

Figure 7: An example for selecting a candidate containing a predecessor-location pair. The text T is shown in Figure (a), and the start locations of the candidates generated by the dueling stage are marked with black squares. The array A is shown in Figure (b). Suppose that the current strip is Strip 2, and the algorithm handles the predecessor-location pair $(4,4),(5,5)$. To find a candidate containing the pair, the algorithm computes the minimum value in the subrow $A[2, 2\,..\,4]$, which is 2. This value corresponds to the candidate $T_{2,3}$.

in the text). Thus, given a predecessor-location pair $(x_1, y_1), (x_2, y_2)$, the candidate (x', y') can be obtained by finding the minimum value in the subrow $A[x_2 - m + 1, y\,..\,\min(y_1, y_2)]$. This amounts to a range minima query [15], so after preprocessing each row of A in $O(m)$ time per row, the minimum element in a subrow of A can be found in constant time. The candidate that generates the minimum value is the candidate $T_{x',y'}$ (see Figure 7).

We turn to the problem of eliminating candidates. As discussed above, the algorithm finds mismatch pairs, and for each mismatch pair $(x_1, y_1), (x_2, y_2)$ it eliminates the candidates that contain this pair. These candidates are precisely the candidates whose start locations are inside the rectangle $[x_2 - m + 1, x_1] \times [y, \min(y_1, y_2)]$. Instead of eliminating candidates at the time such a rectangle is discovered, the rectangle is added to a list L, and after all the rectangles are found, a separate stage is used for candidate elimination.

In the candidate elimination stage, the algorithm needs to remove each candidate whose start location lies in some rectangle in L. Again, the algorithm sweeps across the columns of T from left to right. It will maintain two vectors B and C, where $B[i]$ (resp., $C[i]$) is the number of rectangles in L that intersect the current sweep line, and whose top (resp., bottom) row is i. Using these vectors, it is straightforward to compute, for each location on the sweep line, the number of rectangles in L that contain it, and if the location is contained in at least one rectangle, to eliminate the corresponding candidate. This takes $O(1)$ time per location and $O(1)$ time per rectangle. Since there is at most one rectangle per predecessor-location pair, and there are at most $6m^2$ such pairs, it follows that the time complexity of this stage is $O(m^2)$.

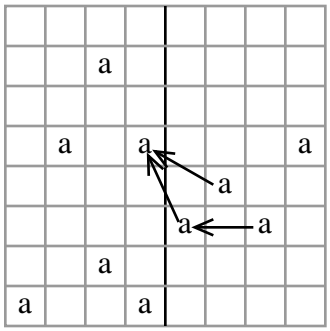


Figure 8: An example for the definition of left predecessor. Location $(6, 5)$ is the left predecessor of $(6, 7)$, and location $(4, 4)$ is the left predecessor of $(6, 5)$ and $(5, 6)$. Location $(4, 8)$ does not have a left predecessor.

4.3 Text Preprocessing

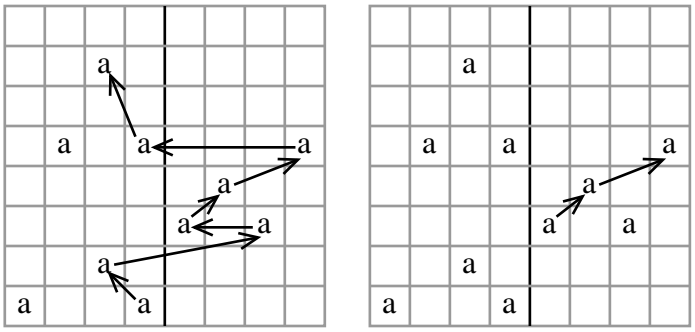
In this section, we show how to compute an array of pointers, called *left predecessors*, that will be used to maintain the predecessor of every location w.r.t. the current strip (as describe in Section 4.2). The predecessor and successor pointers for the current strip form a collection of doubly linked lists, one per character, which are stored in place in T, i.e., each location holds its predecessor and successor pointers. The left predecessors are used in updating these lists when moving from Strip $(y-1)$ to Strip y, as follows. First, each location $(x, y-1)$ in column $y-1$, in top-to bottom order say, is removed from its list by connecting its neighbors. Second, in top-to-bottom order, each location $(x, y+m-1)$ is inserted to its correct place in the list of locations that contain the same character as $(x, y+m-1)$. If $(x, y+m-1)$ has a left predecessor, $(x, y+m-1)$ is inserted right after its left predecessor; otherwise $(x, y+m-1)$ becomes the head of the list.

We now give the definition of left predecessor.

Definition 8 (left predecessor)**.** *For a location (x, z) in T with $z > m$, the* left predecessor of (x, z) *is the predecessor of location (x, z) w.r.t. Strip $(z-m+1)$.*

See Figure 8 for an example of left predecessor. Next, we explain how the algorithm computes the left predecessor for each location (x, z) in T with $z > m$. First, for each character c, it forms a list L_c^1 of candidate left predecessors (see Figure 9(a)). This consists of the locations of c in right-to-left/bottom-to-top order except that in each row only the following locations are kept: the rightmost location containing c in columns 1 to m, together with those locations containing c in columns $m+1$ to $2m$. Next, the algorithm traverses L_c^1. It maintains a second list L_c^2 which holds locations (x, z) with $z > m$ which were traversed in L_c^1, and for which the left predecessor has not yet been traversed. The order of the locations in L_c^2 is according to the traversal order (see Figure 9(b)). Our implementation of this process uses the following two properties of the L_c^2 lists.

Lemma 10. *If (x, z) and (x', z') are two locations in L_c^2, where (x, z) precedes (x', z') in L_c^2, then $x > x'$ and $z < z'$.*



(a) L^1_{a} (b) L^2_{a}

Figure 9: An example of the algorithm for computing the left predecessor. Figure (a) shows the list L^1_{a}. Figure (b) shows the list L^2_{a} when the traversal of L^1_{a} has reached location $(4, 8)$. At this point, out of the 6 traversed locations of L^1_{a}, the locations $(8, 4)$ and $(7, 3)$ are in columns at most $m = 4$, and the left predecessor of location $(6, 7)$ has been traversed. Thus, L^2_{a} contains the remaining 3 locations: $(6, 5)$, $(5, 6)$, and $(4, 8)$. The next step of the algorithm is to assign $(4, 4)$ as the left predecessor of $(6, 5)$ and $(5, 6)$, and remove $(6, 5)$ and $(5, 6)$ from L^2_{a}.

Proof. Suppose, for a contradiction, that either $x \leq x'$ or $z \geq z'$. From the fact that (x, z) appears before (x', z') in the right-to-left/bottom-to-top scan we have that either (1) $x = x'$ and $z > z'$, or (2) $x > x'$ and $z \geq z'$. Since $|z - z'| < m$ (this follows from the inequalities $m < z \leq 2m$ and $m < z' \leq 2m$), in both cases (x', z') is the left predecessor of (x, z). Thus L^2_c cannot contain both (x, z) and (x', z'), yielding a contradiction. ■

Lemma 11. *Suppose that the traversal of L^1_c has reached location (x, z). Let $(x_1, z_1), \ldots, (x_s, z_s)$ be the locations in L^2_c in list order. The location (x, z) is either the left predecessor of $(x_1, z_1), \ldots, (x_f, z_f)$ for some $1 \leq f \leq s$, the left predecessor of $(x_g, z_g), \ldots, (x_s, z_s)$ for some $1 < g \leq s$, or the left predecessor of none of these locations.*

Proof. There are three cases.

1. If $z > z_s$ or if $z \leq z_1 - m$ then (x, z) is the left predecessor of none of the locations in L^2_c.

2. If $z_1 \leq z \leq z_s$ then let g be the minimum integer such that $z \leq z_g$. Then, (x, z) is the left predecessor of $(x_g, z_g), \ldots, (x_s, z_s)$.

3. If neither case 1 nor 2 occurs, then $z_1 - m + 1 \leq z < z_1$. Let f be the maximum integer such that $z_f - m + 1 \leq z$. In this case, (x, z) is the left predecessor of $(x_1, z_1), \ldots, (x_f, z_f)$. ■

Lemma 11 leads to the following algorithm for computing left predecessors. Suppose that currently $L^2_c = (x_1, z_1), (x_2, z_2), \ldots, (x_s, z_s)$, and that (x, z) is the location being traversed. Then:

Case 1 $z < z_1$. Make (x, z) the left predecessor of all of $(x_1, z_1), \ldots, (x_f, z_f)$, where f is the largest index such that $z_f - z < m$. Then remove all of $(x_1, z_1), \ldots, (x_f, z_f)$ from L_c^2.

Case 2 $z > z_m$. Add (x, z) to the right end of L_c^2.

Case 3 $z_1 \leq z \leq z_m$. Make (x, z) the left predecessor of all of $(x_g, z_g), \ldots, (x_m, z_m)$, where g is the least index such that $z \leq z_g$. Then remove all of $(x_g, z_g), \ldots, (x_m, z_m)$ from L_c^2 and finally add (x, z) to the right end of L_c^2.

Clearly, the time complexity for computing the left predecessors is $O(m^2)$.

4.4 Pattern Preprocessing

In this section we show how to compute the witness table for the pattern, namely how to compute a witness for every mismatch offset (a, b). The main idea is similar to the algorithm in Section 3: The algorithm selects neighbors for each location in P and then compare the neighbors of regions of P by solving exact wildcard matching problems. However, there are several important aspects in which the algorithm of this section differs from the algorithm of Section 3. Throughout this section we will discuss these aspects.

To illustrate the main ideas of this stage, we begin by describing an inefficient algorithm for constructing the witness table. For each location (x, y) in the pattern define $2m - 1$ disjoint rectangles $R_{(x,y),-(m-1)}, R_{(x,y),-(m-2)}, \ldots, R_{(x,y),m-1}$, where $R_{(x,y),0} = [x] \times [1, y-1]$, and for $i \neq 0$, $R_{(x,y),i} = [x-m+i] \times [1, y]$. Each active rectangle is traversed from right to left and may generate a neighbor. Then create two strings P_1 and P_2 as follows. The string P_1 is obtained by replacing each location (x, y) in the pattern with a sequence of $2m-1$ characters $c_1 \cdots c_{2m-1}$, corresponding to the rectangles of (x, y), as described in Section 3. If the i-th rectangle does not produce a neighbor, $c_i = \phi$. The string P_2 is obtained similarly, except that when an active rectangle yields no neighbor, $c_i = 0$ (note that if the i-th rectangle is inactive, $c_i = \phi$). Now, let (a, b) be some mismatch offset, and suppose that it has at least one type 1 witness. In this case, the string $P_1' = P_1[1 \,..\, m-a, 1 \,..\, (m-b)(2m-1)]$ does not match $P_2' = P_2[a+1 \,..\, a+m, b(2m-1)+1 \,..\, (b+m)(2m-1)]$. Moreover, we can find a witness to the mismatch of P_1' and P_2', and obtain from it a witness for the offset (a, b). An example is given in Figure 10.

What about type 2 witnesses? As shown in Figure 11, if an offset has only type 2 witnesses, the algorithm described above may not find a witness to the offset. Thus, we need to handle the type 1 and type 2 witnesses separately. Handling type 2 witnesses is done in an analogous way: For each location (x, y) in the pattern define $2m - 1$ disjoint rectangles $\hat{R}_{(x,y),-(m-1)}, \hat{R}_{(x,y),-(m-2)}, \ldots, \hat{R}_{(x,y),m-1}$, where $\hat{R}_{(x,y),0} = [x] \times [y+1, m]$, and for $i \neq 0$, $\hat{R}_{(x,y),i} = [x-m+i] \times [y, m]$. These rectangles are used to construct strings $\hat{P}_1$ and $\hat{P}_2$. As before, finding mismatches between corresponding substrings of $\hat{P}_1$ and $\hat{P}_2$ and witnesses to these mismatches yields witnesses for mismatch offsets that have type 2 witnesses.

The algorithm described above defines $\Theta(m)$ rectangles for each location in the pattern, and thus its time complexity is $\Omega(m^3)$. To reduce the time complexity, we will use "horizon-

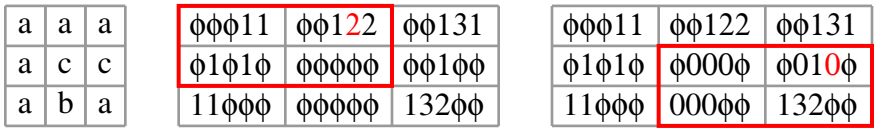


Figure 10: An example of the simple algorithm for witness computation. The pattern P is shown on the left, and the strings P_1 and P_2 are shown on the middle and on the right, respectively. The offset $(1, 1)$ is a mismatch offset, and the strings $P'_1 = P_1[1\,..\,2, 1\,..\,10]$ and $P'_2 = P_2[2\,..\,3, 6\,..\,15]$ do not match. A witness to this mismatch is, for example, $(1, 9)$ as $P'_1[1, 9]$ does not match $P'_2[1, 9]$. The character $P'_1[1, 9]$ was generated from the rectangle $R_{(1,2),1}$, and the value '2' of the character is due to the neighbor $(2, 1)$ selected in this rectangle. Thus, $(1, 2), (2, 1)$ is a type 1 witness for the offset $(1, 1)$. Other witnesses for the mismatch of P'_1 and P'_2 are $(1, 4)$ and $(2, 2)$, and each generates a type 1 witness for $(1, 1)$.

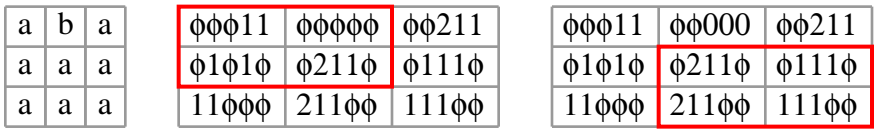


Figure 11: An example showing that type 2 witnesses require special handling. In this example, the offset $(1, 1)$ is a mismatch offset whose witnesses are all of type 2. The substrings $P'_1 = P_1[1\,..\,2, 1\,..\,10]$ and $P'_2 = P_2[2\,..\,3, 6\,..\,15]$ match.

tal" rectangles of height more than one, as in Section 3. We will also use "vertical" rectangles of width more than one. In order to be able to detect some type 1 witness, we need its two locations to be linked. As shown in Section 3 (in Figure 3), when using a rectangles scheme in which the rectangles have heights and widths greater than one, the two locations of a witness may not necessarily be linked. In the rectangles scheme of this section, which will be described below, this can occur in two cases: (1) when one location is near the upper left corner of the pattern, or (2) when the two locations are near the bottom and right boundaries of the relevant region of the pattern (the region $[1, m-a] \times [1, m-b]$ when computing a witness for a mismatch offset (a, b)). We will, therefore, use several rectangles schemes, each designed to find witnesses of different types. We will first describe the easy case which handles witnesses whose locations do not satisfy one of the two cases above. Such witnesses are called *simple*. Afterward, we will show how to handle the non-simple witnesses.

We note that the rectangles scheme of Section 3 does not work here due to the following differences between the problems.

1. Here we need to find witnesses to mismatches of regions of P of various sizes, whereas in Section 3 the goal was to find mismatches between corresponding strings of fixed size $m \times m$.

2. The scheme of Section 3 uses four types of rectangles which we characterize as *left*, *up*, *right*, and *down*. The characterization is according to the scan order of the rectangle: top-to-bottom/right-to-left, left-to-right/bottom-to-top, top-to-bottom/left-to-right, and left-to-right/top-to-bottom, respectively. When finding type 1 witnesses, we can only use left and up rectangles.

We now give a detailed description of the algorithm. Let (a, b) be a mismatch offset. There are two cases to consider: when $a \geq 0$ and when $a < 0$. In the following, we will handle the former case. The latter case is symmetrical, and thus omitted. Let $l = \lfloor \sqrt{m} \rfloor$. We will assume throughout this section that $a < m - 4l$ and $b < m - 4l$. At the end of this section we will show how to handle mismatch offsets (a, b) with either $a \geq m - 4l$ or $b \geq m - 4l$.

Let $D^{a,b} = [1, m-a] \times [1, m-b]$ be the set of locations in the overlap area when P is aligned with itself with offset (a, b). We partition $D^{a,b}$ into subregions (see Figure 12): $D_1^{a,b}$, $D_3^{a,b}$, $D_7^{a,b}$, and $D_9^{a,b}$ are the $l \times l$ squares in the corners of $D^{a,b}$. $D_2^{a,b}$, $D_4^{a,b}$, $D_6^{a,b}$, and $D_8^{a,b}$ are the rectangles of width or height l forming the borders of $D^{a,b}$ excluding the corners, and $D_5^{a,b}$ is the remaining part of $D^{a,b}$. Formally,

$$\begin{aligned} D_1^{a,b} &= [1, l] \times [1, l], \\ D_2^{a,b} &= [1, l] \times [l+1, m-b-l], \\ D_3^{a,b} &= [1, l] \times [m-b-l+1, m-b], \end{aligned}$$

and so on. We define $D^{a,b}[i_1, i_2, \ldots, i_k] = D_{i_1}^{a,b} \cup D_{i_2}^{a,b} \cup \cdots \cup D_{i_k}^{a,b}$.

We say that a witness w for (a, b) is *non-simple* if it satisfies one of the following conditions (see Figure 13):

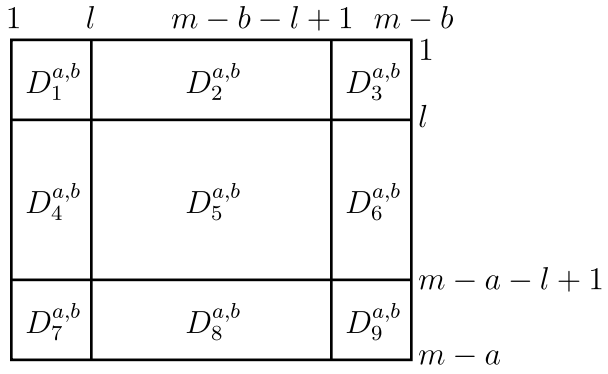


Figure 12: The subsets of $D^{a,b}$.

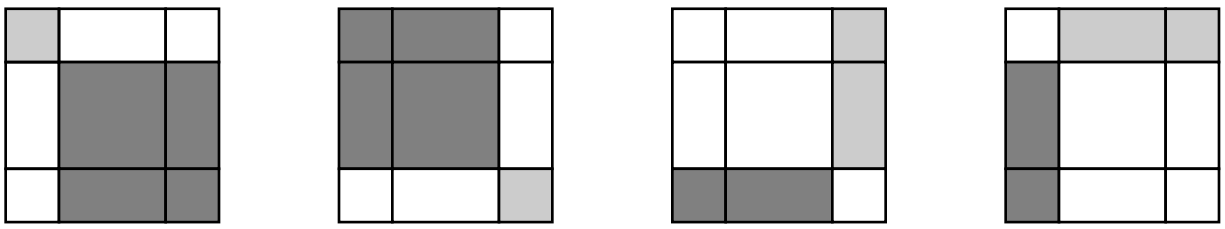


Figure 13: The four conditions in the definition of non-simple witness. In each condition, one of the locations of the witness is in the light gray area, and the other location is in the dark gray area.

1. w is of type 1, one of the locations of w is in $D^{a,b}[1]$, and the other location is in $D^{a,b}[5,6,8,9]$.

2. w is of type 2, one of the locations of w is in $D^{a,b}[9]$, and the other location is in $D^{a,b}[1,2,4,5]$.

3. w is of type 1, one of the locations of w is in $D^{a,b}[3,6]$ and the other location is in $D^{a,b}[7,8]$.

4. w is of type 2, one of the locations of w is in $D^{a,b}[2,3]$ and the other location is in $D^{a,b}[4,7]$.

If a witness w does not satisfy any of the above conditions, it is called *simple*. The algorithm is comprised of three stages:

1. Find simple witnesses.

2. Find non-simple witnesses that satisfy Conditions 1 or 2 above.

3. Find non-simple witnesses that satisfy Conditions 3 or 4 above.

The three stages of the algorithm are described in the rest of this section. We design each stage in a way that guarantees that it finds a witness for every mismatch offset that has a witness of the specified type and no witness for this offset has been found in the previous stages. We note that the witness found for the offset may not be of the specified type. Moreover, even if a mismatch offset does not have a witness of the specified type, the stage may find a witness for the offset.

In the following stages, we will describe only how to find witnesses of type 1, as handling the witnesses of type 2 is symmetrical.

Stage 1

Stage 1 is based on choosing $4\lceil\frac{m}{l}\rceil + 4l - 4$ neighbors for each location (x, y) in P. Similar to Section 3, we define rectangles for each location in P, where each rectangle can provide one neighbor (note that here the rectangles are not disjoint).

For a location (x, y) we define the following rectangles (see Figure 14 for an example). For $i = -\lceil\frac{m}{l}\rceil, \ldots, \lceil\frac{m}{l}\rceil - 1$, let

$$H^1_{(x,y),i} = [x + il, x + (i + 1)l - 1] \times [1, y - 1]$$

and

$$V^1_{(x,y),i} = [1, x - 1] \times [y + il, y + (i + 1)l - 1].$$

Furthermore, for $i = -l + 1, \ldots, l - 1$, we define $H^2_{(x,y),i} = [x + i] \times [1, y - 1]$ and $V^2_{(x,y),i} = [1, x-1] \times [y+i]$. Recall that a non-empty rectangle is active if it is contained in $[1, m] \times [1, m]$.

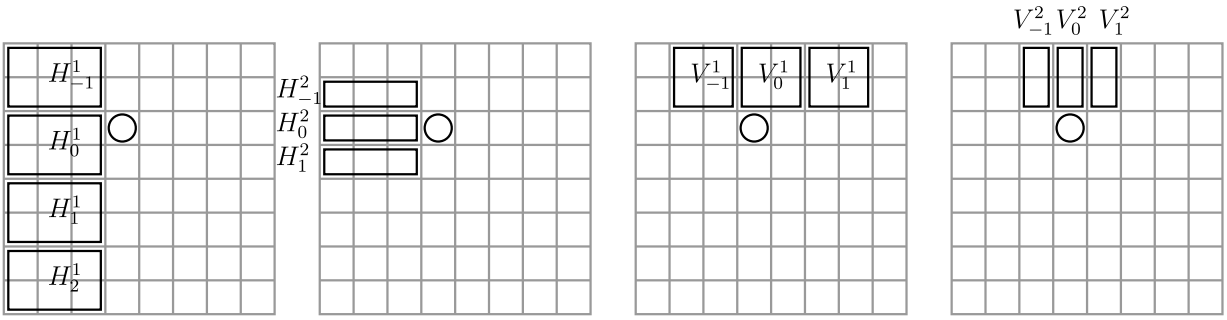


Figure 14: The active rectangles of location $(3, 4)$ in an 8×8 pattern, where $l = 2$. The rectangle $V^1_{(3,4),2} = [1, 2] \times [8, 9]$ is not active as its rightmost column is 9.

Every active rectangle of (x, y) is scanned in a direction away from (x, y), namely, the scan order of $H^1_{(x,y),i}$ is top-to-bottom/right-to-left, and the scan order of $V^1_{(x,y),i}$ is left-to-right/bottom-to-top. The first occurrence of the character $T[x, y]$ in a rectangle is the neighbor generated by the rectangle.

In Section 3 we gave the definition of linked locations, and our goal was to design rectangles ensuring that all locations holding the same character are linked. Here, the linked locations property does not suffice. To see why, consider some mismatch offset (a, b), and a type 1 witness $(x, y), (x', y')$ with $x < m - a - l$, $x' = m - a$, and $y' < y$. Suppose that (x', y') is the neighbor selected in a rectangle $H^1_{(x,y),i}$, and the last row of this rectangle is row $m - a + 1$. Clearly, (x, y) and (x', y') are linked. However, the rectangle $H^1_{(x+a,y+b),i}$ of (x, y) is not active, as its last row is row $m + 1$. Thus, the corresponding character of (x', y') is ϕ, and this character matches the character of (x, y) when comparing the respected substrings of P_1 and P_2. Therefore, we need a definition of a new property which will be used instead of the linked locations property.

Definition 9 ($D^{a,b}$-neighbor)**.** *A location (x', y') is a $D^{a,b}$-*neighbor *of location (x, y) if (x', y') is a neighbor that is generated by an active rectangle of (x, y) that is contained in $D^{a,b}$.*

Definition 10 ($D^{a,b}$-linked)**.** *Two locations (x, y) and (x', y') in $D^{a,b}$ are $D^{a,b}$-*linked *if there is a series of locations $(w_0, z_0) = (x, y), (w_1, z_1), \ldots, (w_l, z_l), (w_{l+1}, z_{l+1}) = (x', y')$ such that for all i, at least one of the locations (w_i, z_i) and (w_{i+1}, z_{i+1}) is a $D^{a,b}$-neighbor of the other location.*

We will shortly show that the rectangles we have defined will be sufficient for obtaining the desired for Stage 1. In the next stages we will need to define additional rectangles for locations in P. The definitions of $D^{a,b}$-neighbor and $D^{a,b}$-linked will also apply to the neighbors generated by these rectangles.

While in Section 3 all locations holding the same character were linked, here for each choice of a and b, only some of the locations are $D^{a,b}$-linked. This does not cause a problem, as this stage only needs to find simple witnesses. The following lemma describes the $D^{a,b}$-linked locations property.

Lemma 12. *Let (a, b) be an offset. Let (x, y) and (x', y') be two locations inside $D^{a,b}$ containing the same character. Suppose that neither of the following conditions hold*

- *One of the locations is in $D^{a,b}[1]$ and the other location is in $D^{a,b}[5, 6, 8, 9]$.*
- *One of the locations is in $D^{a,b}[3, 6]$ and the other location is in $D^{a,b}[7, 8]$.*

Then (x, y) and (x', y') are $D^{a,b}$-linked.

Proof. We first claim that at least one case among the following cases must occur.

Case 1 $|y - y'| < l$.

Case 2 $|x - x'| < l$.

Case 3 The topmost location among (x, y) and (x', y') is in $D^{a,b}[2, 5, 8]$.

Case 4 The leftmost location among (x, y) and (x', y') is in $D^{a,b}[4, 5, 6]$.

For a contradiction, suppose that Cases 1–4 do not occur. W.l.o.g. we assume that $x > x'$, so (x', y') is the topmost location among (x, y) and (x', y'). Since Case 3 does not occur, either $y' \leq l$ or $y' \geq m - b - l + 1$.

Suppose first that $y' \leq l$. If $y \leq y'$ then $|y - y'| < l$ which contradicts the assumption that Case 1 does not occur. Therefore $y > y'$, so (x', y') is the leftmost location among (x, y) and (x', y'). Since Case 4 does not occur, we have that either $x' \leq l$ or $x' \geq m - a - l + 1$. If $x' \geq m - a - l + 1$ then $m - a - l + 1 \leq x' < x \leq m - a$. Therefore $|x - x'| < l$, but this contradicts the assumption that Case 2 does not occur. On the other hand, if $x' \leq l$ then $(x', y') \in D^{a,b}[1]$. As we assumed that Cases 1 and 2 do not occur, $(x, y) \notin D^{a,b}[1, 2, 3, 4, 7]$. Therefore, $(x, y) \in D^{a,b}[5, 6, 8, 9]$ which contradicts an assumption of the lemma.

From the above we conclude that $y' \geq m - b - l + 1$. Moreover, $y < y'$ otherwise Case 1 occurs. The location (x, y) is the leftmost location among (x, y) and (x', y'), so either $x \leq l$ or $x \geq m - a - l + 1$. The former case cannot occur otherwise Case 2 occurs. If the latter case occurs then from the assumption that Cases 1 and 2 do not occur we obtain that $(x', y') \in D^{a,b}[3, 6]$ and $(x, y) \in D^{a,b}[7, 8]$ which contradicts an assumption of the lemma. Therefore, at least one case among Cases 1–4 must occur.

We now show that (x, y) and (x', y') are $D^{a,b}$-linked when Case 1 occurs and when Case 3 occurs. The proof for Cases 2 and 4 is symmetric. W.l.o.g. we assume that $x \geq x'$.

Case 1 First, if $y = y'$ then (x, y) and (x', y') are $D^{a,b}$-linked by a series of locations on column y (due to the $V^2_{\cdot,0}$ rectangles). Moreover, if $0 < |y - y'| < l$ then we have that $(x', y') \in V^2_{(x,y),y'-y}$, so the rectangle $V^2_{(x,y),y'-y}$ generates a $D^{a,b}$-neighbor (x'', y') for (x, y). From the above, (x'', y') is $D^{a,b}$-linked to (x', y') and therefore (x, y), and (x', y') are $D^{a,b}$-linked.

Case 3 We prove the lemma for Case 3 using induction on $x - x'$. The base of the induction (when $x = x'$) was already shown. Suppose now that $x > x'$. We claim that (x', y') is contained in some rectangle $V^1_{(x,y),i}$ of (x, y), and this rectangle is contained in $D^{a,b}$. The first part of the claim is true since by construction, the $V^1_{(x,y),\cdot}$ rectangles cover all the locations of $D^{a,b}$ above m. The second part of the claim is true since $l + 1 \leq y' \leq m - b - l$. Therefore, (x, y) has a $D^{a,b}$-neighbor (x'', y'') with $x < x'' \leq x'$. By induction (x'', y'') and (x', y') are $D^{a,b}$-linked, and therefore (x, y) and (x', y') are $D^{a,b}$-linked. ■

From Lemma 12 we obtain the following observation.

Observation 4. *Let (a, b) be an offset.*

1. *If (x, y) is a location in $D^{a,b}$ such that the neighbors of (x, y) are not aligned with the neighbors of $(x + a, y + b)$ then (a, b) is a mismatch offset. Moreover, if some rectangle of (x, y) produces a neighbor (x', y') while the corresponding rectangle of $(x + a, y + b)$ does not produce the neighbor $(x' + a, y' + b)$, then $(x, y), (x', y')$ is a type 1 witness for (a, b).*

2. *If (a, b) is a mismatch offset and it has a simple type 1 witness, then there is a location (x, y) in $D^{a,b}$ whose neighbors are not aligned with the neighbors of $(x + a, y + b)$.*

As in Section 3, we transform the problem of comparing the neighbor structures into an exact matching problem. We define strings P_1 and P_2 as follows. The string P_1 is obtained from P by replacing each character $P[x, y]$ in P with $L = 4\lceil m/l \rceil + 4l - 4$ characters that encode the locations of the neighbors of (x, y) relative to (x, y). If a rectangle does not yield a neighbor then the character for this rectangle is the wildcard character ϕ. The string P_2 is built in the same way, except that the character 0 is used when a rectangle yields no neighbor. The strings P_1 and P_2 are both of size $m \times mL$.

After building P_1 and P_2 the algorithm solves the region matching with witnesses problem on P_1 and P_2. The following lemma shows the correctness of Stage 1.

Lemma 13. *Let (a, b) be an offset.*

1. *If P_1 does not match $P'_2 = P_2[a+1\,..\,a+m, bL+1\,..\,(b+m)L]$ then (a, b) is a mismatch offset. Moreover, a type 1 witness for (a, b) can be obtained from every witness to the mismatch of P_1 and P'_2 in constant time.*

2. *If (a, b) has a simple type 1 witness then P_1 does not match P'_2.*

Proof. The lemma follows from Observation 4. ■

The bottleneck in the time complexity of Stage 1 is the time for solving the region matching with witnesses problem. By Lemma 4, the time complexity of this step is $O(|P_2| \operatorname{polylog}(m)) = O((\frac{m^3}{l} + l) \operatorname{polylog}(m)) = O(m^{5/2} \operatorname{polylog}(m))$.

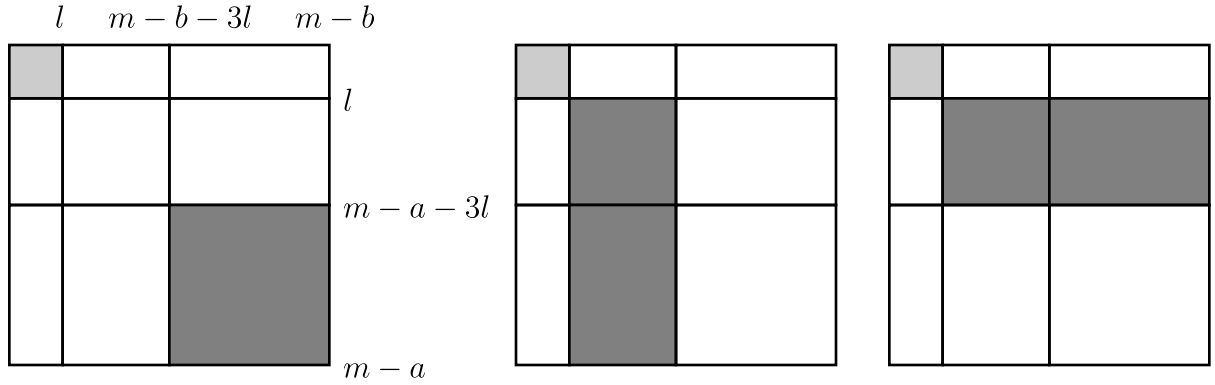


Figure 15: The three substages of Stage 2. Each substage finds witnesses in which one of the locations of the witness is in the light gray area, and the other location is in the dark gray area.

Stage 2

Recall that the goal of Stage 2 is to find type 1 witnesses w for mismatch offsets (a, b) (which do not have simple witnesses) such that one location of w is in $D^{a,b}[1]$ and the other location is in $D^{a,b}[5, 6, 8, 9] = [l + 1, m - a] \times [l + 1, m - b]$ (this stage also finds type 2 witnesses but we omit the details). Stage 2 is composed of three substages (see Figure 15).

1. Stage 2a finds witnesses w such that one location of w is in $D^{a,b}[1]$, and the other location is in $[m - a - 3l + 1, m - a] \times [m - b - 3l + 1, m - b]$.

2. Stage 2b finds witnesses w such that one location of w is in $D^{a,b}[1]$ and the other location is in $[l + 1, m - a] \times [l + 1, m - b - 3l]$.

3. Stage 2c finds witnesses w such that one location of w is in $D^{a,b}[1]$ and the other location is in $[l + 1, m - a - 3l] \times [l + 1, m - b]$.

We denote the rectangles $[m - a - 3l + 1, m - a] \times [m - b - 3l + 1, m - b]$ and $[l + 1, m - a] \times [l + 1, m - b - 3l]$ appearing above by $D^{a,b}[10]$ and $D^{a,b}[11]$, respectively.

We next describe stages 2a and 2b. Stage 2c is analogous to Stage 2b, and thus we omit its description.

Stage 2a

Recall that this stage finds a witness for mismatch offsets (a, b) in which one location of the witness is in $D^{a,b}[10]$ and the other location is in $D^{a,b}[1]$. To accomplish this, we will define new rectangles that will $D^{a,b}$-link every pair of locations (x, y) and (x', y') holding the same character such that $(x, y) \in D^{a,b}[10]$ and $(x', y') \in D^{a,b}[1] = [1, l] \times [1, l]$. The difficulty of handling such pairs is the proximity of (x, y) to both the first row and column of the pattern. While there are rectangles $H^1_{(x,y),i}$ and $V^1_{(x,y),i'}$ (defined on Stage 1) that contain (x', y'), both these rectangles may be inactive, and thus (x, y) and (x', y') may not be $D^{a,b}$-linked by the

rectangles defined on Stage 1. The key idea here is that there are only $\Theta(l)$ rows in which the locations (x, y) and (x', y') can appear. Thus, for every location (x, y) we can define $\Theta(l)$ horizontal rectangles of height 1 that cover $D^{a,b}[1]$. Since each rectangle has height 1, the rectangle that contains (x', y') is necessarily active.

Suppose first that we only need to handle offsets (a, b) for some fixed a. As location (x', y') is in rows $1, \ldots, l$ and location (x, y) is in rows $m - a - 3l + 1, \ldots, m - a - 1$, the difference $x - x'$ is between $m - a - (4l - 1)$ and $m - a$. Thus, for each location $(x, y) \in D^{a,b}[10]$ we define rectangles $[x - (m - a - i)] \times [1, y - 1]$ for $i = 1, \ldots, 4l - 1$. Note that the rows of the rectangles depend on a. We need to handle offsets (a, b) for all values of a without defining too many rectangles. If we look at two values a and $a + 1$, the rectangles for a location (x, y) needed for offsets (a, b) (for all b) are the same as the rectangles needed for offsets $(a + 1, b)$, except for the rectangles $[x - (m - a - 1)] \times [1, y - 1]$ and $[x - (m - a - 4l)] \times [1, y - 1]$. More generally, we can group together l consecutive values of a, and define rectangles for each group separately.

Formally, we partition the offsets into sets $O_{a'} = \{(a, b) | a' \leq a \leq a' + l - 1\}$, where a' is a multiple of l. Consider some set $O_{a'}$. For a location (x, y) in $[m - a' - 4l + 2, m - a'] \times [1, m]$ we define the rectangles

$$H^{3,a'}_{(x,y),i} = [x - (m - a' - i)] \times [1, y - 1]$$

for every $1 \leq i \leq 5l - 2$. As before, each rectangle may generate a neighbor. The rectangles are scanned in right-to-left order.

Lemma 14. *Let (a, b) be an offset in $O_{a'}$. Let $(x, y) \in D^{a,b}[10]$ and $(x', y') \in D^{a,b}[1]$ be two locations holding the same character. Then (x, y) and (x', y') are $D^{a,b}$-linked.*

Proof. Let $i = x' - x + m - a'$. Since $m - a - 3l + 1 \leq x \leq m - a$ and $1 \leq x' \leq l$ we have that $i \geq 1 - (m - a) + m - a' \geq 1$ and $i \leq l - (m - a - 3l + 1) + m - a' = 4l - 1 + a - a' \leq 5l - 2$. Therefore, the rectangle $H^{3,a'}_{(x,y),i} = [x'] \times [1, y - 1]$ is defined. Since $y' \leq l$ and $y \geq m - a - 3l + 1 > l + 1$ (due to the assumption that $a < m - 4l$), it follows that $y' \leq y - 1$, hence $(x', y') \in H^{3,a'}_{(x,y),i}$. Therefore, (x, y) has a $D^{a,b}$-neighbor (x', y'') in $H^{3,a'}_{(x,y),i}$ and the lemma follows. ∎

In order to find a witness for offset (a, b), we need to compare the neighbors of locations (x, y) inside $D^{a,b}[10]$ with the neighbors of $(x + a, y + b)$. The locations of the form $(x + a, y + b)$ are in $D^{a,b}[10] + (a, b) = [m - 3l + 1, m] \times [m - 3l + 1, m]$. Therefore, for locations $(x, y) \in [m - 3l + 1, m] \times [m - 3l + 1, m]$ we define rectangles $H^{3,a'}_{(x,y),i}$ as before (namely, $H^{3,a'}_{(x,y),i} = [x - (m - a' - i)] \times [1, y - 1]$). We now transform the problem of comparing neighbors to an exact wildcard matching with witnesses problem. Let $P^{a'}_1$ and $P^{a'}_2$ be strings that encode the neighbors of all $H^{3,a'}_{(x,y),i}$ rectangles for locations in $[m - 3l + 1, m] \times [m - 3l + 1, m]$ and $[m - a' - 4l + 2, m - a'] \times [1, m]$, respectively. The following lemma follows from Lemma 14.

Lemma 15. *Let (a, b) be an offset in $O_{a'}$.*

1. *If $P_1^{a'}$ does not match $P_2 = P_2^{a'}[l-(a \bmod l)\,..\,2l-1-(a \bmod l), m-b-l+1\,..\,m-b]$ then (a,b) is a mismatch offset. Moreover, a type 1 witness for (a,b) can be obtained from every witness to the mismatch of $P_1^{a'}$ and P_2 in constant time.*

2. *If (a,b) is a mismatch offset for which no witness was found is Stage 1 and there is a type 1 witness w for (a,b) such that one location of w is in $D^{a,b}[1]$ and the other location is in $D^{a,b}[10]$, then $P_1^{a'}$ does not match P_2.*

In Stage 2a the algorithm solves $O(m/l)$ instances of the exact wildcard matching with witnesses problem, each of size $O(ml^2)$. By Lemma 3, the time complexity of this stage is $O(m^{5/2}\,\mathrm{polylog}(m))$.

Stage 2b

In this stage we define additional rectangles in order to $D^{a,b}$-link pairs of locations $(x,y),(x',y')$ holding the same character in which $(x,y) \in D^{a,b}[1]$ and $(x',y') \in D^{a,b}[11]$. The approach of Stage 2a does not work here since the rectangle $D^{a,b}[11]$ can contain $\Theta(m^2)$ locations and therefore defining rectangles for the locations in $D^{a,b}[11]$ would generate too many rectangles. Instead, we will define rectangles for the locations in $D^{a,b}[1]$.

We partition the set of offsets into sets $O_{a,b'} = \{(a,b'),(a,b'+1),\ldots,(a,b'+l-1)\}$ where b' is a multiple of l. The offsets in each set $O_{a,b'}$ will be handled together. Consider some set $O_{a,b'}$. For a location (x,y) in $[1,l]\times[1,l]$ we define the rectangle

$$H^{4,a,b'}_{(x,y)} = [l+1, m-a] \times [y+1, y+m-b'-3l].$$

From each rectangle a neighbor is generated by scanning the rectangle in top-to-bottom/right-to-left order.

Lemma 16. *Let (a,b) be an offset in $O_{a,b'}$. Let $(x,y) \in D^{a,b}[1]$ and $(x',y') \in D^{a,b}[11]$ be two locations containing the same character. Then (x,y) and (x',y') are $D^{a,b}$-linked.*

Proof. We will show that (x,y) has a $D^{a,b}$-neighbor (x'',y'') such that (x',y') and (x'',y'') are $D^{a,b}$-linked. It is easy to verify that $(x',y') \in H^{4,a,b'}_{(x,y)}$. Moreover, the rectangle $H^{4,a,b'}_{(x,y)}$ is contained in $D^{a,b}$. Thus, the rectangle $H^{4,a,b'}_{(x,y)}$ generates a $D^{a,b}$-neighbor; let (x'',y'') denote that neighbor. Due to the scan order, $y'' \geq y' \geq l+1$. Moreover, $y'' \leq y+m-b'-3l \leq m-b-l-1$ and $l+1 \leq x'' \leq m-a$. Therefore, $(x'',y'') \in D^{a,b}[2,5,8]$. We also have $(x',y') \in D^{a,b}[5,8]$. By Lemma 12, (x',y') and (x'',y'') are $D^{a,b}$-linked. Thus, (x,y) and (x',y') are $D^{a,b}$-linked. ∎

In order to compare the neighbors of locations $(x,y) \in D^{a,b}[1]$ with neighbors of $(x+a, y+b)$, we define rectangles for locations (x,y) in $[a+1, a+l]\times[b'+1, b'+2l-1]$:

$$H^{5,a,b'}_{(x,y)} = [l+1+a, m] \times [y+1, y+m-b'-3l].$$

We next define two strings $P_1^{a,b'}$ and $P_2^{a,b'}$. The string $P_1^{a,b'}$ encodes the neighbors of the $H^{4,a,b'}_{(x,y)}$ rectangles, and $P_2^{a,b'}$ encodes the neighbors of the $H^{5,a,b'}_{(x,y)}$ rectangles.

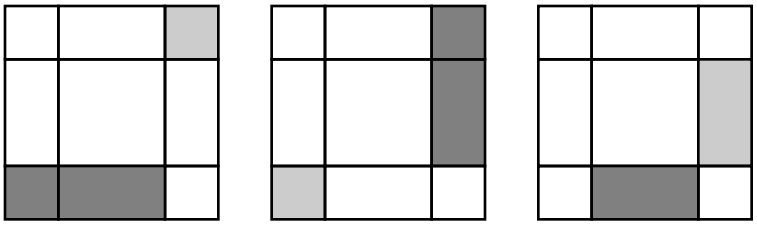


Figure 16: The three substages of Stage 3. Each substage finds witnesses in which one of the locations of the witness is in the light gray area, and the other location is in the dark gray area.

Lemma 17. *Let (a, b) be an offset in $O_{a,b'}$.*

1. *If $P_1^{a,b'}$ does not match $P_2 = P_2^{a,b'}[1\,..\,m, 1 + (b \bmod l)\,..\,m + (b \bmod l)]$ then (a, b) is a mismatch offset. Moreover, a type 1 witness for (a, b) can be obtained from every witness to the mismatch of $P_1^{a,b'}$ and P_2 in constant time.*
2. *If (a, b) is a mismatch offset for which no witness was found in Stage 1 and there is a type 1 witness w for (a, b) such that one location of w is in $D^{a,b}[1]$ and the other location is in $D^{a,b}[11]$ then $P_1^{a,b'}$ does not match P_2.*

For each set $O_{a,b'}$, the algorithm solves the exact wildcard matching with witnesses problem on $P_1^{a,b'}$ and $P_2^{a,b'}$. Since there are $O(m^2/l)$ sets, and the size of each string $P_2^{a,b'}$ is $O(l^2)$, the time complexity of this stage is $O(m^2 l \operatorname{polylog}(m)) = O(m^{5/2} \operatorname{polylog}(m))$.

Stage 3

Again this stage is composed of several sub-stages (see Figure 16).

1. Stage 3a finds witnesses w such that one location of w is in $D^{a,b}[3]$ and the other location is in $D^{a,b}[7, 8]$.
2. Stage 3b finds witnesses w such that one location of w is in $D^{a,b}[7]$ and the other location is in $D^{a,b}[3, 6]$.
3. Stage 3c finds witnesses w such that one location of w is in $D^{a,b}[6]$ and the other location is in $D^{a,b}[8]$.

Stage 3b is similar to Stage 3a, and thus we omit the details of this stage.

Stage 3a

In this stage we use the partitioning of offsets into the sets defined in Stage 2a, namely $O_{a'} = \{(a, b) | a' \leq a \leq a' + l - 1\}$, where a' is a multiple of l. Consider some set $O_{a'}$. For a

location (x, y) in $[1, l] \times [1, m]$ we define rectangles

$$H^{6,a'}_{(x,y),i} = [x + m - a' - i] \times [1, y - 1]$$

for $i = 1, \ldots, 3l - 2$. Neighbors are selected from the rectangles by scanning the rectangles in right-to-left order.

Lemma 18. *Let (a, b) be an offset in $O_{a'}$. Let $(x, y) \in D^{a,b}[3]$ and $(x', y') \in D^{a,b}[7, 8]$ be two locations with the same character. Then (x, y) and (x', y') are $D^{a,b}$-linked.*

As before, the problem of comparing the neighbors is transformed into a region matching with witnesses problem. The algorithm builds strings $P^{a'}_1$ and $P^{a'}_2$ as follows: For locations $(x, y) \in [a' + 1, a' + 2l - 1] \times [m - l + 1, m]$ we define rectangles $H^{6,a'}_{(x,y),i}$ as above, and $P^{a'}_1$ encode the neighbors of these rectangles. The string $P^{a'}_2$ encodes the neighbors of locations $(x, y) \in [1, l] \times [1, m]$. By Lemma 4, the time complexity of this stage is $O(m^{5/2} \operatorname{polylog}(m))$.

Stage 3c

We now need to find witnesses for mismatch offsets for which no witness was found during the previous stages. The algorithm of this stage uses different approach than the previous stages. In previous stages, multiple offsets were handled in parallel, by reduction to exact matching problems. Here, each offset is handled separately. For each relevant mismatch offset, the algorithm computes an almost minimal rectangle that contains a witness, and then finds the actual witness by examining locations near the corners of the rectangle. The computation of the rectangle is based on counting characters in substrings of P, using the colored intersection counting data-structure of Gupta et al. [14].

Consider some mismatch offset (a, b) for which no witness was found during the previous stages. Without loss of generality we assume that (a, b) has a type 1 witness. The stage is based on the following lemma.

Lemma 19. *Let (a, b) be a mismatch offset, and let $R \subseteq D^{a,b}$ be a rectangle such that there is no type 2 witness for (a, b) whose both locations are inside R. Then, the number of distinct characters inside the region R of P is less than or equal to the number of distinct characters inside the region $R + (a, b)$ of P with strict inequality if and only if there is a type 1 witness w for (a, b) for which both locations are in R.*

Proof. Since there are no type 2 witnesses for (a, b) with both locations inside R, there is a function matching from the substring of P induced by the region $R + (a, b)$ to the substring induced by the region R. Therefore, the first part of the lemma holds. The second part of the lemma follows from Observation 1. ■

In other words, Lemma 19 states that we can tell whether a rectangle R contains a witness by computing the number of distinct characters in R and in $R + (a, b)$. Thus, if these computations can be done fast, a minimal rectangle containing a witness can be found using binary search.

We say that a rectangle $D \subseteq D^{a,b}$ is a *mismatch rectangle* if the number of distinct characters inside the region D of P is strictly less than the number of distinct characters inside the region $D+(a,b)$ of P. A mismatch rectangle D is *minimal* if there is no mismatch rectangle that is contained in D.

Since no witness for (a,b) was found during the previous stages, then for every type 1 witness for (a,b), one location of the witness is in $D^{a,b}[6]$ and the other location is in $D^{a,b}[8]$. Symmetrically, for every type 2 witness for (a,b), one location of the witness is in $D^{a,b}[2]$ and the other location is in $D^{a,b}[4]$. In particular, there are no type 2 witnesses for (a,b) with both locations inside $D^{a,b}[5,6,8,9]$. Thus, $D^{a,b}[5,6,8,9]$ contains at least one mismatch rectangle.

Observation 5. *Let $D \subseteq D^{a,b}[5,6,8,9]$ be a minimal mismatch rectangle. Then, the locations at the bottom left corner and the top right corner of D form a witness for (a,b).*

Proof. By Lemma 19 there is a type 1 witness w for (a,b) for which both locations are in D. From the minimality of D the two locations of w must be at opposite corners of D. Since one location of w must be in $D^{a,b}[6]$ and the other location in $D^{a,b}[8]$, it follows that the locations of w are at the bottom left corner and top right corner of D. ■

We do not know how to efficiently find a minimal mismatch rectangle. Instead, we will present an algorithm that finds an "almost minimal" mismatch rectangle. The corners of the rectangle will not necessarily give a witness for (a,b), so an additional search for the witness will be required.

Consider the following intersection counting problem: Given a set S of points in the plane where each point has a color, build a data-structure for S that can answer queries of the form "what is the number of distinct colors for the points inside the 3-sided rectangle $\{(x,y) \in \mathbb{R}^2 : x \leq x_2, y_1 \leq y \leq y_2\}$?". Denote by n the number of points in S. Gupta et al. [14] showed a data-structure for this problem with preprocessing time $O(n^2 \log^2 n)$ that answers queries in $O(\log^2 n)$ time. Using this result, we obtain the following lemma:

Lemma 20. *Let P be an $m \times m$ string, and let l be an integer. After preprocessing of P in $O(\frac{m^3}{l} \log^2 m)$ time, the following queries can be answered in $O(\log^2 m)$ time: "what is the number of distinct characters in the substring $P[x_1 \mathinner{..} x_2, y_1 \mathinner{..} y_2]$?", where x_1, x_2, y_1, and y_2 are integers with either $x_1 \equiv 1 \pmod{l}$ or $x_2 = m$.*

Proof. The preprocessing is as follows: Create a set of points S containing a point (x,y) for every $x = 1, \ldots, m$ and $y = 1, \ldots, m$. The color of (x,y) is $P[x,y]$. For every integer $i \leq m/l$, build the data structure of Gupta et al. on the set of points $S_i = \{(x,y) \in S : x \geq il + 1\}$. Also build a data structure on $S' = \{(-x,y) : (x,y) \in S\}$.

Given a query (x_1, x_2, y_1, y_2), if $x_1 \equiv 1 \pmod{l}$, return the number of distinct colors in the points of $S_{(x_1-1)/l}$ that are inside the 3-sided rectangle $\{(x,y) \in \mathbb{R}^2 : x \leq x_2, y_1 \leq y \leq y_2\}$. If $x_2 = m$, return the number of distinct colors in the points of S' that are inside the 3-sided rectangle $\{(x,y) \in \mathbb{R}^2 : x \leq -x_2, y_1 \leq y \leq y_2\}$. ■

Assume that the data-structure of Lemma 20 was built over P. Our algorithm constructs an "almost minimal" mismatch rectangle D by starting with $D = D^{a,b}[5, 6, 8, 9]$ and then one by one moving the left, right, and top edges of D to the right, left, and down, respectively, as long as the rectangle remains a mismatch rectangle. More precisely, the algorithm constructs D as follows.

1. Find the maximum y such that the rectangle $[l+1, m-a] \times [y, m-b]$ is a mismatch rectangle. Note that from Lemma 19, $[l+1, m-a] \times [y', m-b]$ is a mismatch rectangle for every $y' < y$. Therefore, the algorithm finds the value of y using binary search and queries to the data-structure of Lemma 20.

2. Using binary search, find the minimum z such that the rectangle $[l+1, m-a] \times [y, z]$ is a mismatch rectangle.

3. Using binary search, find the maximum x such that $x-1$ is a multiple of l and $[x, m-a] \times [y, z]$ is a mismatch rectangle. Let $D = [x, m-a] \times [y, z]$.

From Lemma 19 and the definition of D we obtain that there is a type 1 witness for (a, b) with one endpoint in $D_1 = [m-a-l+1, m-a] \times [y]$ and the other endpoint in $D_2[x, x+l-1] \times [z]$. Finding such a witness is implemented as follows. Note that if (x_1, y) and (x'_1, y) are two locations in D_1 containing the same character then $P[x_1+a, y+b] = P[x'_1+a, y+b]$ (this is true since both (x_1, y) and (x'_1, y) are in $D^{a,b}[8]$, and thus these two locations cannot form a type 1 witness for (a, b)). Therefore, the algorithm scans the locations in D_1, and for each location (x_1, y), if no location containing the character $P[x_1, y]$ was already encountered, the algorithm stores (x_1, y) as a *candidate*. The candidates are stored in an array $L[1 \,..\, m^2]$, indexed by the characters contained in the candidates. Next, the algorithm scans the locations in D_2. For each location $(x_2, z) \in D_2$, if there is a candidate (x_1, y) that contains the character $P[x_2, z]$, the algorithm checks whether the pair $(x_1, y), (x_2, z)$ is a witness for (a, b).

By Lemma 20, the time complexity for handling a single offset (a, b) is $O(\log^3 m + l)$. Thus, the time complexity of this stage is $O(\frac{m^3}{l} \log^2 m + m^2 \log^3 m + m^2 l) = O(m^{5/2} \log^2 m)$.

Handling special offsets

We now describe how to handle a mismatch offset (a, b) with $a \geq m - 4l$. For each location (x, y) in P we define the following rectangles: For $i = -4l+1, \ldots, 4l-1$, let $H_{(x,y),i} = [x+i] \times [1, y-1]$ and $\hat{H}_{(x,y),i} = [x+i] \times [y+1, m]$. For each location the algorithm chooses neighbors using these rectangles and then constructs an exact wildcard matching problem as described in Stage 1. The size of the strings in this problem is $O(m^2 l)$, so this stage takes $O(m^{5/2} \operatorname{polylog}(m))$ time. Handling mismatch offsets (a, b) with $b \geq m - 4l$ is similar.

5 Substrings character counting

In this section we show an $O(n^2)$ time algorithm that counts the number of distinct characters in every $m \times m$ substring of an $n \times n$ string T. W.l.o.g. we can assume that $n = 2m$ (if

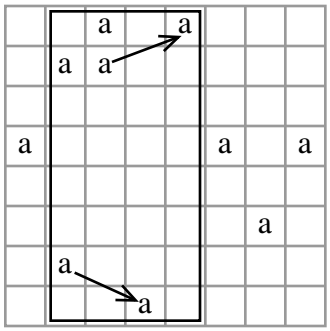


Figure 17: An example of the definition of upper and lower chains. The figure shows the chains when the current strip is Strip 2. The upper chain contains the locations $(2, 3)$ and $(1, 5)$, and the lower chain contains the locations $(7, 2)$ and $(8, 4)$.

needed, by partitioning a larger text into overlapping $2m \times 2m$ pieces).

The algorithm sweeps across the strips of string T (see Definition 5). When at Strip y, it will determine for each location (x, y) how many distinct characters are contained in the $m \times m$ substring $T_{x,y} = T[x \,..\, x + m - 1, y \,..\, y + m - 1]$.

To this end, for each character c, it computes values $D_c[x, y]$ such that $\sum_{x'=1}^{x} D_c[x', y]$ is 1 if $T_{x,y}$ contains the character c and 0 otherwise. As we shall see, each column will have at most three non-zero entries for character c, and the number of changes to these values as y increases is linear in the number of occurrences of c in T. Consequently, the array D_c is maintained just for the current value of y and is updated as the sweep advances. Henceforth, we let $D_c[x]$ denote $D_c[x, y]$, where y is understood to be the current value of y.

In fact, it is more convenient to record just $D[x] = \sum_c D_c[x]$. For then $\sum_{x'=1}^{x} D[x']$ equals the number of distinct characters in $T_{x,y}$.

Next, we explain how to compute D_c. To help with this, two chains of locations in T containing c are maintained, an upper and a lower chain. These chains are illustrated in Figure 17. The upper chain comprises locations in $T[1 \,..\, m, y \,..\, y + m - 1]$. It is the maximal series of locations $(x_1^u, y_1^u), (x_2^u, y_2^u), \ldots, (x_{k_u}^u, y_{k_u}^u)$, where $x_1^u > x_2^u > \cdots > x_{k_u}^u$, $y_1^u < y_2^u < \cdots < y_{k_u}^u$, and there is no location (x', y') in $T[1 \,..\, m, y \,..\, y + m - 1]$ containing a c "below" the chain, i.e., for which $x' > x_i^u$ and $y' > y_{i-1}^u$ for some $1 < i \leq k_u$, or $x' > x_1^u$ or $y' > y_{k_u}^u$. If the chain is empty, we define $x_1^u = 0$.

The upper chain is readily maintained as y advances by deletion at the left end of the chain and insertion, preceded by necessary deletions, at the right end. More specifically, all locations violating the chain order with respect to the new item to be added are removed from the right end of the chain and only then is the new item added. As each item is added and removed at most once, this takes time $O(\text{number of occurrences of } c \text{ in } T)$.

The lower chain is defined analogously for locations in $T[m + 1 \,..\, 2m, y \,..\, y + m - 1]$, but now $x_1^l < x_2^l < \cdots < x_{k_l}^l$ and $y_1^l < y_2^l < \cdots < y_{k_l}^l$. If the chain is empty, we define $x_1^l = 2m + 1$.

The values of x for which the texts $T_{x,y}$ contain c are given by the intervals $[1, x_1^u] \cup [x_1^l -$

$m+1, m+1]$. The definition of D_c is now clear:

Case 1 $x_1^u = 0$, $x_1^l = 2m+1$.
$D_c[x] = 0$ for all x.

In the remaining cases we indicate only the non-zero entries for D_c.

Case 2. $x_1^u \neq 0$, $x_1^l = 2m+1$.
$D_c[1] = 1$, $D_c[x_1^u + 1] = -1$.

Case 3. $u_1^u = 0$, $x_1^l \neq 2m+1$.
$D_c[x_1^l - m + 1] = 1$.

Case 4. $x_1^u \neq 0$, $x_1^l \neq 2m+1$, and $x_1^u \geq x_1^l - m$.
$D_c[1] = 1$.

Case 5. $x_1^u \neq 0$, $x_1^l \neq 2m+1$, and $x_1^u < x_1^l - m$.
$D_c[1] = 1$, $D_c[x_1^u + 1] = -1$, and $D_c[x_1^l - m + 1] = 1$.

Clearly, whenever x_1^l or x_1^u changes, the corresponding update to D_c can be made in $O(1)$ time. In fact, the updates are made directly as increments or decrements to $D[x]$, as appropriate.

Theorem 21. *There is an $O(n^2)$ time algorithm that reports the number of distinct characters in each $m \times m$ substring of an $n \times n$ text.*

Proof. It suffices to show this result for $n = 2m$, as already noted. The cost of the algorithm has two parts. The first is the cost of updating D as the sweep occurs: this takes $O(\sum_c \text{number of occurrences of } c \text{ in } T) = O(m^2)$. The second is to compute $\sum_{x'=1}^{x} D[x']$ for each value of x ($1 \leq x \leq n - m + 1$) for each value of y is in turn ($n - m + 1$ values again). But this is readily done in $O(m^2)$ time. ■

References

[1] N. Alon and M. Naor. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

[2] A. Amir, Y. Aumann, M. Lewenstein, and E. Porat. Function matching. *SIAM J. on Computing*, 35(5):1007–1022, 2006.

[3] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. on Computing*, 23(2):313–323, 1994.

[4] A. Amir, K. W. Church, and E. Dar. The submatrices character count problem: an efficient solution using separable values. *Information and Computation*, 190(1):100–116, 2004.

[5] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994.

[6] A. Apostolico, P. L. Erdös, and M. Lewenstein. Parameterized matching with mismatches. *J. of Discrete Algorithms*, 5(1):135–140, 2007.

[7] Y. Aumann, M. Lewenstein, N. Lewenstein, and D. Tsur. Finding witnesses by peeling. *ACM Transactions on Algorithms*, 7(2):24:1–24:15, 2011.

[8] G. P. Babu, B. M. Mehtre, and M. S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4):327–348, 1995.

[9] B. S. Baker. Parameterized string pattern matching: Algorithms and applications. *J. of Computer and Systems Sciences*, 52(1):28–42, 1996.

[10] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. on Computing*, 26(5):1343–1362, 1997.

[11] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

[12] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34th ACM Symposium on Theory Of Computing (STOC)*, pages 592–601, 2002.

[13] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. on Computing*, 33(1):26–42, 2003.

[14] P. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *J. of Algorithms*, 19(2):282–317, 1995.

[15] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–355, 1984.

[16] C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms*, 3(3), 2007.

[17] S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *Proc. 36th Symposium on Foundation of Computer Science (FOCS)*, pages 631–637, 1995.

[18] M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[19] U. Vishkin. Optimal parallel pattern matching in strings. In *Proc. 12th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 91–113, 1985.

[20] U. Vishkin. Deterministic sampling — a new technique for fast pattern matching. *SIAM J. on Computing*, 20:303–314, 1991.