Two Dimensional Parameterized Matching

Carmit Hazay[1], Moshe Lewenstein[1,⋆], and Dekel Tsur[2,⋆⋆]

[1] Bar-Ilan University
{harelc,moshe}@cs.biu.ac.il
[2] University of California, San Diego
dtsur@cs.ucsd.edu

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 to an $m \times m$ pattern. This models, for example, searching for color images with changing of color maps. We present an algorithm that solves the two dimensional parameterized matching problem in $O(n^2 + m^{2.5} \cdot \text{polylog}(m))$ time.

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, 11], one is given a text T and pattern P and the goal is to find all the substrings of T of length $|P|$ that p-match to 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]).

In [9, 11], an optimal linear time algorithm was given for p-matching. However, it was assumed that the alphabet was of constant size. An optimal algorithm for p-matching in the presence of an unbounded size alphabet was given in [6]. In [10], a novel method was presented for parameterized matching by constructing *parameterized suffix trees*, which also allows for online p-matching. The parameterized suffix tree was further explored by Kosaraju [16] and faster constructions were given by Cole and Hariharan [12].

In [7], approximate parameterized matching was introduced and a solution for binary alphabets was given. In [15], 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 maximum matchings in bipartite graphs.

⋆ The second author was supported by an IBM faculty award grant.

⋆⋆ The third author was a postdoc at Caesarea Rothschild Institute of Computer Science, University of Haifa during the research of these results.

One of the interesting problems in web searching is searching for color images, see [4, 8, 17]. If the colors are fixed, this is exact two dimensional pattern matching [3]. However, images can appear under different color maps. The image is the same image however each color has been recolored with a unique color. Two-dimensional parameterized search is precisely what is needed. The fastest algorithm for solving the two dimensional parameterized matching problem was given in [2], and its time complexity is $O(n^2 \log^2 m)$.

It is an open question whether a linear time algorithm for the two dimensional parameterized matching problem exists. In this paper we show that it is possible to get an algorithm that is linear in the text size, but with a penalty in the preprocessing stage. More precisely, the time complexity of our algorithm is $O(n^2 + m^{2.5} \cdot \text{polylog}(m))$. For alphabets drawn from a large universe (larger than polynomial in n) the time complexity increases to $O(n^2 \log |\Sigma| + m^{2.5} \cdot \text{polylog}(m))$. However, this seems to be unavoidable as it was shown in [6] that the one-dimensional parameterized matching requires $\Omega(n \log |\Sigma|)$ time.

Due to lack of space, some proofs are omitted.

2 Preliminaries and Definitions

Let T and T' be two texts of size $k \times k$. We say that there is a *function matching* between T and T' if there is a mapping f from the alphabet of T to the alphabet of T' such that $T'[i,j] = f(T[i,j])$ for all i and j. If the mapping f is one-to-one, we say that T and T' *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 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 to P.

The algorithms for one dimensional parameterized matching are based on converting the pattern and text strings into predecessor strings. The *predecessor* of location i in a string S is the location containing the previous appearance of the symbol $S[i]$, if there is such a location. The *predecessor string* of S is obtained by replacing each character in S by the distance to its predecessor, or by 0 if the character does not have a predecessor. For example, the predecessor string of *aabbaba* is $0, 1, 0, 1, 3, 2, 2$. A simple and well-known fact is that:

Observation 1 *S and S' p-match iff they have the same predecessor string.*

Observation 1 gives a handle on finding p-matches for 1-dimensional texts. We would like to use a similar observation for 2-dimensional texts as well. In order to do so we define the following.

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

Let A be a $k \times l$ string. We define the *linearization* of A to be the one-dimensional string $A[1,1], \ldots, A[1,l], A[2,1], \ldots, A[2,l], \ldots, A[k,1], \ldots, A[k,l]$.

Definition 2 (predecessor). *Let T be an $n \times n$ text, T' be a strip of T, and (i,j) be a location in T. The* predecessor of (i,j) w.r.t. *T' is the location (i',j') in T such that $(i'-1)m+j'$ is the predecessor of location $(i-1)m+j$ in the linearization of T'.*

A pair of a location in T and its predecessor will be called a *location-predecessor pair.*

3 Overview

We begin by giving an overview of the algorithm. As in many pattern matching algorithms, we assume w.l.o.g. that $n = 2m$. Larger texts can be cut into $2m \times 2m$ pieces which are handled separately.

The outline of the algorithm follows the "duel-and-sweep"-paradigm that appeared in [3] (there it is named "consistency and verification" and was used for 2-dimensional exact matching) and is based upon ideas of duelling techniques [18, 19]. The idea of the "duel-and-sweep"-paradigm is to maintain a list of $m \times m$ substrings of T, called *candidates*, that might match to P. Starting with all the $m \times m$ substrings of T, the list is pruned in two stages to a list of all the candidates which actually match. The two stages are called the *duelling stage* and the *sweeping stage*. However, exact matching turns out to be much simpler than parameterized matching. This is mainly because of the "witness" which is used in the duelling stage to differentiate between two pattern alignments. In exact matching a witness is simply one location with a mismatch between the two alignments. However, in parameterized matching one location is not sufficient to rule out a match. The following definition captures the concept of a witness in parameterized matching.

Definition 3 (witness). *Let P be a pattern of size $m \times m$. Consider an alignment of P with itself, starting at location $(a+1,b+1)$ (namely, $P[x,y]$ in the first copy of P is aligned with $P[x+a,y+b]$ in the second copy of P). A* witness *relative to the offset (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* $P[x+a,y+b] = P[x'+a,y'+b]$.

In the duelling stage, we use the fact that if two p-matches of P in T overlap, then there is a p-matching between the two substrings of P that correspond to the overlapping area of the two matches. Using a witness array for P, we can check pairs of overlapping candidates which do not agree on their overlapping area, and rule out at least one candidate from the pair in constant time per pair. After this stage we remain only with candidates that agree with each over.

In the sweeping stage, we need to check the remaining candidates. This is done by going over all the strips of T from left to right, and for each strip, checking the candidates that are contained in the current strip.

Suppose that the current strip starts at column y, and consider some candidate $T' = T[x\,..\,x+m-1, y\,..\,y+m-1]$ in the strip. Suppose that we know the predecessor for every location in the current strip of T.

Before we make the next observation, a central concept which has a similar flavor to the witness defined above is as follows.

Definition 4 (mismatch pair). *Let R and S be two equal sized 2-dimensional strings. A* mismatch pair *between R and S is a pair of locations $(i,j),(i',j')$ such that one of the following holds:*

1. *$R[i,j] = R[i',j']$ and $S[i,j] \neq S[i',j']$.*
2. *$R[i,j] \neq R[i',j']$ and $S[i,j] = S[i',j']$.*

We now observe that:

Observation 2 *1. If T' p-matches to P, then every location-predecessor pair inside T' (namely, both the location and its predecessor are in T') is not a mismatch pair for T' and P.*

2. If there is no location-predecessor pair inside T' which is a mismatch pair for T' and P, then there is a function matching between T' and P.

3. If there is a function matching between T' and P, then there is a p-matching between T' and P if and only if the number of distinct characters in T' is equal to the number of distinct characters in P.

By Observation 2, the algorithm for checking the candidates will have two steps: In the first step, we go over all the location-predecessor pairs inside the candidate, and check whether one of these pairs is a mismatch pair. In the second step we compute the number of distinct characters in each candidate and compare it to the number of distinct characters in P.

In order to implement the first step, we go over al strips of T (from left to right) while maintaining the predecessor of every location in the current strip. Computing the predecessor of every location in the first strip of T takes $O(m^2)$ time. When going from one strip to the next strip, only at most $6m$ predecessors are changed, and we will show how to do the update in $O(m)$ time.

Clearly, if we check each location-predecessor pair for each candidate separately, then the algorithm will not be efficient (the time complexity will be $\Theta(m^4)$ in the worst case). However, we can use the fact that after the duelling stage, all the remaining candidates agree with each other. Therefore, for each location-predecessor pair in T, the pair is either a mismatch pair for P and every candidate that contains the pair, or the pair is not a mismatch pair for any candidate. Thus, for each location-predecessor pair we need to check only two characters in P, and if there is a mismatch, we can rule out all the remaining candidates that contain the pair.

The second step is done by computing the number of distinct characters in every $m \times m$ substring of T. Amir et al. [4] gave an algorithm for this problem whose time complexity is $O(m^2 \log m)$. We can solve the problem in $O(m^2)$ time, but omit the details for space reasons. (Amir and Cole [5] also have solved the problem in the same time bounds).

4 Algorithm Details

From Observation 2, we need to check for each candidate whether the location-predecessor pairs inside it are mismatch pairs. As discussed in Section 3, we will check every location-predecessor pair only once during the entire algorithm.

The main idea is as follows: We go over the strips of T from left to right. When going from the $(y-1)$-th strip to the y-th strip, there are at most $3n = 6m$ new location-predecessor pairs: (1) Every location in column $y+m-1$ and its predecessor form a new pair, (2) Every location in column $y+m-1$ may become the predecessor of some location in columns $y,\dots,y+m-2$, and (3) Every location in columns $y,\dots,y+m-2$ whose predecessor was in column $y-1$ will form with its new predecessor a new pair. After a preprocessing step on T that will be described in Section 5, we can find all the new location-predecessor pairs in time $O(m)$.

Now, we only need to check whether there is a mismatch pair among the new location-predecessor pairs, as we already checked the old pairs in the previous iterations. Let $(r,c),(r',c')$ be some new location-predecessor pair for the y-th strip. All the candidates that contain the locations (r,c) and (r',c') agree with each other, so $(r,c),(r',c')$ is a mismatch pair for all of these candidate, or for none. Therefore, we only need to find one candidate (z,w) (with $w \geq y$) that contains both locations, and check whether $P[r-z+1, c-w+1] = P[r'-z+1, c'-w+1]$. If these two character are not equal, then (z,w) cannot be a match, and moreover, every candidate that contains (r,c) and (r',c') cannot be a match. In other words, we can rule out every candidate whose top-left corner is in the rectangle $\{r-m+1,\dots,r'\} \times \{y,\dots,\min(c,c')\}$. Note that it is possible that the predecessor of (r,c) changes at some strip y' for $y < y' \leq \min(c,c')$, so $(r,c),(r',c')$ is not a location-predecessor pair for some of the candidates that we rule out. However, this does not affect the correctness of the algorithm.

We now need to handle two issues: How to find a candidate that contains the pair $(r,c),(r',c')$, and in a case of a mismatch, how to rule out the candidates in the corresponding rectangle.

We first deal with the first issue. Given a location-predecessor pair (r,c), (r',c'), we will find the highest candidate that can contain the pair $(r,c),(r',c')$, which will be denoted (z,w). More precisely, among all the candidates in $\{r-m+1,\dots,n\} \times \{y,\dots,\min(c,c')\}$ (note that the rectangle here ends in row n), (z,w) is the candidate with smallest row number (ties are broken arbitrarily). Clearly, if (z,w) does not contain the pair $(r,c),(r',c')$ (that is, if $z > r'$), then there is no candidate (that begins in a column greater than y) that contains the pair $(r,c),(r',c')$.

We now describe how to find the highest candidate in constant time. To do that, we build an $n \times n$ array A, where $A[i,j]$ is the smallest row in which a candidate starts, among all the candidates with start row at least $i-m+1$ and start column j. If there are no such candidates, then $A[i,j] = 2m$. The array A can be easily computed by scanning the text column by column, from bottom to top. Now, given a location-predecessor pair $(r,c),(r',c')$, in order to find the highest candidate that can contain the pair, we need to find the minimum value among the subrow $A[r, y], A[r, y+1], \ldots, A[r, \min(c, c')]$. This is the range minima problem [13], so after preprocessing each row of A in $O(m)$ time per row, we can find the minimum element in some subrow of A in constant time.

We now turn to the problem of eliminating candidates. As discussed above, during the algorithm we find rectangles that do not contain a match. Instead of eliminating the candidates at the time each rectangle is discovered, we store the rectangle in a list L, and after obtaining all the rectangles, we perform a candidates elimination stage.

In the candidates elimination stage, we need to remove each candidate whose top-left corner is in some rectangle of L. This is done by moving a vertical sweep line from left to right. We maintain two vectors V and B, where $V[i]$ (resp., $B[i]$) is the number of rectangles in L that intersect the current sweep line, and whose top (resp., bottom) row is i. Using these vectors, we can compute the number of rectangles that contain each point on the sweep line, and eliminate the candidates whose top-left corner is contained in at least one rectangle. The time complexity of this step is linear in the number of rectangles in L, which is at most $6m^2$.

5 Text Preprocessing

In this section, we show how to compute an array of pointers, which will be used to maintain the predecessors of the current strip.

Definition 5 (left predecessor). *For a location (i, j) in T with $j > m$, the* left predecessor of (i, j) *is the predecessor of location (i, j) w.r.t. the $(j-m+1)$-th strip of T.*

Given the left predecessors of all the locations in T, it is straightforward to maintain the predecessor of every location w.r.t. the current strip.

We now show how to compute the left predecessor of every location in T. We scan the entire text bottom-up left-to-right. For each symbol σ, we keep a list L_σ of all the locations (i, j) with $j > m$ and $T[i, j] = \sigma$ for which we haven't computed a left predecessor yet. The elements of L_σ are ordered according to their scan order. When the scan of T reaches a location (x, y), we need to check for which elements of $L_{T[x,y]}$ the left predecessor is (x, y), and these elements will be removed from $L_{T[x,y]}$. Moreover, if $y > m$ and the left predecessor of (x, y) is not in row x, then (x, y) is added to the end of $L_{T[x,y]}$. For efficient implementation of the algorithm above, we use the following properties of the L_σ lists.

Claim 1 *If (i, j) and (i', j') are two elements of some list L_σ, where (i, j) appears in the list before (i', j'), then $i > i'$ and $j < j'$.*

Claim 2 *Let $(i_1, j_1), \ldots, (i_s, j_s)$ be the locations in $L_{T[x,y]}$ according to their order. A location (x, y) is either the left predecessor of $(i_1, j_1), \ldots, (i_f, j_f)$ for some $1 \le f \le s$, the left predecessor of $(i_f, j_f), \ldots, (i_s, j_s)$ for some $1 < f \le s$, or the left predecessor of none of these locations.*

From Claim 2, the text preprocessing stage can be implemented in $O(n^2)$ time.

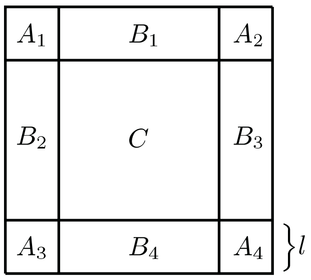

Fig. 1. The subsets of D.

6 Pattern Preprocessing

Let P be an $m \times m$ string. We will show how to compute a witness for every offset (a, b) that has a witness.

Let l be some integer that will be specified later. Consider an alignment of P against itself, with offset (a, b). We say that (a, b) is a *source* if there is a p-match between the overlapping areas in the two copies of P. If (a, b) is not a source, then a there is at least one witness for (a, b). A witness $(x, y), (x', y')$ is called a *witness of type 1* if it satisfies condition 1 in Definition 3, and otherwise, it is called a witness of type 2. There are four cases that we need to consider, according to the signs of a and b. In the following, we will handle the case when $a \geq 0$ and $b \geq 0$. The other cases are symmetrical, and thus omitted.

Let (a, b) be an offset which is not a source, and let $D = \{1, \ldots, m - a\} \times \{1, \ldots, m - b\}$ be the set of locations in the overlapping area of P when P is aligned against itself with offset (a, b). We partition D into subsets (see Figure 1): $A_1, \ldots, A_4$ are the $l \times l$ squares in the corners of D. $B_1, \ldots, B_4$ are rectangles of width or height l in the borders of D excluding the corners, and C is the remaining part of D (namely, $C = D \setminus (A_1 \cup \cdots \cup A_4 \cup B_1 \cup \cdots \cup B_4)$). Formally,

$$A_1 = \{(x, y) \in \mathbb{Z}^2 : 1 \leq x \leq l \text{ and } 1 \leq y \leq l\},$$
$$A_2 = \{(x, y) \in \mathbb{Z}^2 : 1 \leq x \leq l \text{ and } m - b - l + 1 \leq y \leq m - b\},$$

etc.

We say that a witness w for (a, b) is *simple* if it does not satisfy any of the following conditions:

1. w is of type 1, and one of the locations of w is in A_1.
2. w is of type 1, one of the locations of w is in $A_2 \cup B_3 \cup A_4$, and the other location is in $A_3 \cup B_4 \cup A_4$.
3. w is of type 2, and one of the locations of w is in A_4.
4. w is of type 2, one of the locations of w is in $A_1 \cup B_1 \cup A_2$, and the other location is in $A_1 \cup B_2 \cup A_3$.

The algorithm consists of three stages:

1. Find simple witnesses.
2. Find witnesses that satisfy conditions 1 or 3 above.
3. Find witnesses that satisfy conditions 2 or 4 above.

The three stages of the algorithm are described in the next sub-sections. In each stage, we will handle only offsets do not contain witnesses that were handled by the previous stages.

6.1 Stage 1

This stage is similar to the algorithm of Amir et al. [2]: We create new strings P_1 and P_2 by replacing every character $P[x,y]$ in P by $8\frac{m}{l}+4$ characters, where each of these characters is either a pointer to some location (x',y') in P such that $P[x',y']=P[x,y]$, or a null pointer.

For each location (x,y) in P, we define rectangles on the grid $\mathbb{Z}^2$ in following way: For $i=-\frac{m}{l},\ldots,\frac{m}{l}-1$, let

$$H_{i,(x,y)} = \{(x',y')\in\mathbb{Z}^2 : y'\le y \text{ and } x+il\le x'<x+(i+1)l\},$$
$$\hat{H}_{i,(x,y)} = \{(x',y')\in\mathbb{Z}^2 : y'\ge y \text{ and } x+il\le x'<x+(i+1)l\},$$
$$V_{i,(x,y)} = \{(x',y')\in\mathbb{Z}^2 : x'\le x \text{ and } y+il\le y'<y+(i+1)l\},$$

and

$$\hat{V}_{i,(x,y)} = \{(x',y')\in\mathbb{Z}^2 : x'\ge x \text{ and } y+il\ge y'<y+(i+1)l\}.$$

Furthermore, let $H_{(x,y)}=\{(x,y')\in\mathbb{Z}^2 : y'\le y\}$, and we similarly define the rectangles $\hat{H}_{(x,y)}$, $V_{(x,y)}$, and $\hat{V}_{(x,y)}$. We say that a rectangle $H_{i,(x,y)}$ (or other H rectangle) is *inside the square of* P if there is at least one column of $H_{i,(x,y)}$ that is contained in $\{1,\ldots,m\}\times\{1,\ldots,m\}$, namely, if $1\le y+il$ and $y+(i-1)l-1\le m$. A V rectangle is inside the square of P if there is at least one row of the rectangle that is contained in $\{1,\ldots,m\}\times\{1,\ldots,m\}$.

The string P_1 is constructed by replacing every character $P[x,y]$ in P by the characters $c_1,\ldots,c_{8m/l+4}$ (P_1 is an $m\times((8\frac{m}{l}+4)m)$ string) which are defined as follows:

- For $i=-\frac{m}{l},\ldots,\frac{m}{l}-1$, if the rectangle $H_{i,(x,y)}$ is inside the square of P, traverse over the elements of $H_{i,(x,y)}$ in a column major order (from right to left), until reaching a location (x',y') for which $P[x',y']=P[x,y]$, if there is such a location. If a location (x',y') was found, then set $c_{m/l+i+1}=(x-x',y-y')$, and we say in this case that $c_{m/l+i+1}$ *points to* (x',y'). Otherwise (namely, the rectangle $H_{i,(x,y)}$ does not contain a character equal to $P[x,y]$), set $c_{m/l+i+1}=\phi$.
  If the rectangle $H_{i,(x,y)}$ is not inside the square of P, then set $c_{m/l+i+1}=\phi$.
- For $i=-\frac{m}{l},\ldots,\frac{m}{l}-1$, the character $c_{3m/l+i+1}$ is built from the rectangle $\hat{H}_{i,(x,y)}$ in the same way described above, except that the rectangle is traversed from right to left, and furthermore, if the rectangle does not contain a character equal to $P[x,y]$, then $c_{3m/l+i+1}=0$.

– For $i = -\frac{m}{l}, \ldots, \frac{m}{l} - 1$, the characters $c_{5m/l+i+1}$ and $c_{7m/l+i+1}$ are built from the rectangles $V_{i,(x,y)}$ and $\hat{V}_{i,(x,y)}$ the same as above, except that the rectangles are traversed in a row major order (if the corresponding rectangle does not contain a character equal to $P[x, y]$, we use ϕ for $c_{5m/l+i+1}$ and 0 for $c_{7m/l+i+1}$).
– The character $c_{8m/l+1}$ corresponds to the rectangle $H_{(x,y)}$, and it is handled the same as the characters $c_1, \ldots, c_{2m/l}$. Furthermore, the characters $c_{8m/l+2}$, $c_{8m/l+3}$, and $c_{8m/l+4}$ correspond to the rectangles $\hat{H}_{(x,y)}$, $V_{(x,y)}$, and $\hat{V}_{(x,y)}$, respectively.

The string P_2 is built in two steps. The first step is the same as building P_1, except that we replace the roles of ϕ and 0, that is, we use ϕ for characters that correspond to the rectangles $\hat{H}_{i,(x,y)}$ and $\hat{V}_{i,(x,y)}$, and 0 for characters that correspond to the rectangles $H_{i,(x,y)}$ and $V_{i,(x,y)}$. After the first step, P_2 is an $m \times ((8\frac{m}{l}+4)m)$ string. In the second step, we expand P_2 into a $2m \times 2((8\frac{m}{l}+4)m)$ string, where all the new characters are ϕ.

After building P_1 and P_2, we solve the (standard) matching problem with don't care symbols on P_1 and P_2, where P_1 is the pattern, P_2 is the text, and ϕ is the don't care symbol. Moreover, using the algorithm of Alon and Naor [1], we find witnesses for every mismatch between P_1 and P_2, namely, for every (a, b) such that P_1 does not match to $P_2[a+1\mathinner{..}a+m, b+1\mathinner{..}b+m]$, we find a location (x, y) such that $P_1[x, y] \neq P_2[x + a, y + b]$.

Consider some fixed pair (a, b), and define $P_2' = P_2[a+1\mathinner{..}a+m, b+1\mathinner{..}b+m]$.

Claim 3 *If P_1 does not match to P_2', then (a, b) is not a source. Moreover, from every witness to the mismatch of P_1 and P_2' we can obtain a witness for (a, b) in constant time.*

The converse of Claim 3 is not true. A weaker result is given in the following lemma.

Lemma 1. *If (a, b) has a simple witness then P_1 does not match to P_2'.*

Proof. Suppose that $w = \{(x, y), (x', y')\}$ is a simple witness for (a, b). W.l.o.g. we assume that $x \geq x'$, and moreover, if $x = x'$ we assume that $y > y'$. We will deal with the case when w is a witness of type 1, and omit the proof for the case when w is of type 2. We consider 3 cases.

Case 1. In the first case, suppose that either $x = x'$ or (x', y') is in $B_1 \cup C \cup B_4$, or in other words, $l + 1 \leq y' \leq m - b - l$. We prove the lemma using induction on $x - x'$.

If $x = x'$, we have that $(x, y') \in H_{(x,y)}$. Thus, the character of P_1 that corresponds to $H_{(x,y)}$ points to some location (x, y'') such that $y' \leq y'' < y$ and $P[x, y''] = P[x, y]$. If $P[x + a, y'' + b] \neq P[x + a, y + b]$, then the character of P_2 that corresponds to $H_{(x+a,y+b)}$ either points to some location other than $(x + a, y'' + b)$, or is equal to 0. In both cases, we conclude that P_1 does not match to P_2'. If $P[x + a, y'' + b] = P[x + a, y + b]$, then $w' = \{(x, y''), (x, y')\}$ is a simple witness for (a, b) of type 1. We can now use the argument above on w' and either obtain that P_1 does not match to P_2', or obtain a new witness w''. We can continue this process until obtaining a mismatch between P_1 and P_2'.

Now, suppose that $x > x'$, and we proved case 1 of the lemma for every witness in which the difference between the rows in the two locations is less than $x - x'$. Since $l+1 \leq y' \leq m-b-l$, it follows that the rectangle $V_{i,(x,y)}$ that contains the location (x', y') is inside the square of P. Therefore, the character of P_1 that corresponds to $V_{i,(x,y)}$ points to some location (x'', y''), where $x \leq x'' < x'$. If $P[x'' + a, y'' + b] \neq P[x + a, y + b]$ then we are done since the character of P_2 that corresponds to $V_{i,(x+a,y+b)}$ either points to some location other than $(x'' + a, y'' + b)$, or is equal to 0. Otherwise, we use the induction hypothesis on $w' = \{(x'', y''), (x', y')\}$. Note that it is possible that $(x'', y'') \in A_1$ and therefore w' is not simple. However, the arguments we used above still work on w', so we still obtain that P_1 does not match to P_2'.

Case 2. The second case is when either $y = y'$ or leftmost location among (x, y) and (x', y') is in $B_2 \cup C \cup B_3$. The proof for the case is symmetrical to the proof of case 1 (we use the rectangles $V_{(x^*,y^*)}$ and $H_{i,(x^*,y^*)}$ instead of $H_{(x,y)}$ and $V_{i,(x,y)}$, where (x^*, y^*) is the rightmost location).

Case 3. Assume that cases 1 and 2 do not occur. Since case 1 does not occur, we have that either $y' \leq l$ or $y' \geq m - b - l + 1$. We consider 4 sub-cases:

1. $y' \leq l$ and $y > y'$. Since (x', y') is the leftmost location and case 2 does not occur, we have that either $x' \leq l$ or $x' \geq m - a - l + 1$. If $x' \leq l$ then $(x', y') \in A_1$, and therefore w is not simple, a contradiction.
   If $x' \geq m - a - l + 1$, then the rectangle $H_{-1,(x,y)}$ is inside the square of P, and it contains the location (x', y'). Therefore, using the same arguments as in case 1, we obtain that P_1 does not match to P_2'.
2. $y' \leq l$ and $y < y'$. In this case, the rectangle $V_{0,(x,y)}$ is inside the square of P, and it contains the location (x', y'). Thus, P_1 does not match to P_2'.
3. $y' \geq m - b - l + 1$ and $y > y'$. The rectangle $V_{-1,(x,y)}$ is inside the square of P, and it contains the location (x', y'), so P_1 does not match to P_2'.
4. $y' \geq m - b - l + 1$ and $y < y'$. Since (x, y) is the leftmost location, it follows that either $x \leq l$ or $x \geq m - a - l + 1$. In the former case, the rectangle $H_{0,(x',y')}$ is inside the square of P, and it contains the location (x, y). It follows that P_1 does not match to P_2'. In the latter case, we obtain that w is not simple, a contradiction. □

From Claim 3 and Lemma 1 we conclude that stage 1 of the algorithm correctly finds a witness for every offset (a, b) that has simple witnesses. The time complexity of this stage is $O(|P_2| \cdot \log^{4+o(1)} |P_1|) = O(\frac{m^3}{l} \cdot \log^{4+o(1)} m)$.

6.2 Stage 2

In the following stages, we will describe only how to find witnesses of type 1, as handling the witnesses of type 2 is symmetrical. The second stage is composed of four sub-stages. We will not give detailed proofs for the correctness of these steps, as the proofs are similar to the proof of Lemma 1.

Stage 2a. In this stage we find witnesses w such that the two locations of w are in A_1. For each location (x, y) in P, we define rectangles as follows:

$$H^2_{i,(x,y)} = \{(x+i, y') \in \mathbb{Z}^2 : y' \leq y\}$$
$$V^2_{i,(x,y)} = \{(x', y+i) \in \mathbb{Z}^2 : x' \leq x\}.$$

Using these rectangles, we build strings P_1^2 and P_2^2 by replacing each character $P[x, y]$ in P by $4l$ characters that correspond to the rectangles $H^2_{i,(x,y)}$ and $V^2_{i,(x,y)}$ for $i = -l+1, \ldots, l-1$. Each of the $4l$ characters is chosen by traversing the appropriate rectangle similarly to the construction of P_1 and P_2 in stage 1. Then, we solve the matching problem for P_1^2 and P_2^2 and find witnesses for the mismatches. The correctness of this sub-stage follows from the fact that for two locations in A_1, one of the locations is inside the rectangles of the other location.

Stage 2b. In this stage we find all the witnesses w such that one location of w is in A_1, and the other location is in $B_1 \cup B_2 \cup C \cup A_3 \cup B_4$.

As in the previous stages, we create new strings by replacing each character in P with pointers to other occurrences of the character. However, instead of creating one matching problem instance, we will create $O(m^2/l)$ instances. The main idea is to group the different offsets into set and build a matching problem instance for each set, and then use the instance to find witnesses for the offsets in the set. In each set, all the offsets are close to each other, and we use this fact in our construction.

Consider a set of offsets $(a, b), (a, b+1), \ldots, (a, b+l/2-1)$ (we assume that l is even), where b is a multiple of l. Define the rectangles

$$H^{3,a,b}_{(x,y)} = \{(x', y') \in \mathbb{Z}^2 : y \leq y' \leq y+m-b-l \text{ and } 0 \leq x' \leq m-a\}$$
$$H^{4,a,b}_{(x,y)} = \{(x', y') \in \mathbb{Z}^2 : y' \leq y \text{ and } x \leq x' \leq m-a\}$$
$$H^{5,a,b}_{(x,y)} = \{(x'+a, y') : (x', y') \in H^{3,a,b}_{(x,y)}\}$$

and

$$H^{6,a,b}_{(x,y)} = \{(x'+a, y') : (x', y') \in H^{4,a,b}_{(x,y)}\}.$$

Next, build the strings $P_1^{3,a,b}$ and $P_2^{3,a,b}$ as follows: For every location (x, y) with $1 \leq x \leq l$ and $1 \leq y \leq l/2$, the string $P_1^{3,a,b}$ contains two characters for $P[x, y]$ corresponding to the rectangles $H^{3,a,b}_{(x,y)}$ and $H^{4,a,b}_{(x,y)}$. For every location (x, y) with $a+1 \leq x \leq a+l$ and $b+1 \leq y \leq b+3l/2$, the string $P_2^{3,a,b}$ contains two characters for $P[x, y]$ corresponding to the rectangles $H^{5,a,b}_{(x,y)}$ and $H^{6,a,b}_{(x,y)}$. Moreover, the string $P_2^{3,a,b}$ is padded with don't care symbols.

As in the previous stages, we solve the matching problem for $P_1^{3,a,b}$ and $P_2^{3,a,b}$ and find witnesses for the mismatches. If $\{(x,y),(x',y')\}$ is a witness for (a,b') where $b \leq b' \leq b+l/2$, $(x,y) \in \{1,\dots,l\} \times \{1,\dots,l/2\}$, and $(x',y') \in B_1 \cup B_2 \cup C \cup A_3 \cup B_4$, then one of the rectangles $H^{3,a,b'}_{(x,y)}$ and $H^{4,a,b'}_{(x,y)}$ contains the location (x',y'), and it follows that $P_1^{3,a,b'}$ does not match to $P_2^{3,a,b'}$. Moreover, from every witness to the mismatch of $P_1^{3,a,b'}$ and $P_2^{3,a,b'}$, we can obtain a witness for (a,b').

Finding witnesses with one location of the witness in $\{1,\dots,l\} \times \{l/2+1,\dots,l\}$ and the other in $B_1 \cup B_2 \cup C \cup A_3 \cup B_4$ is done in a similar way.

Stage 2c. This stage finds all the witnesses w such that one location of w is in A_1, and the other location is in $A_2 \cup B_3$. This stage is analogous to stage 2b, and we omit the details.

Stage 2d. In this final sub-stage, we find all the witnesses w such that one location of w is in A_1, and the other location is in A_4. Recall that

$$H^2_{i,(x,y)} = \{(x+i,y') \in \mathbb{Z}^2 : y' \leq y\}.$$

For every set of offsets $\{(a+i,b+j) : i=0,\dots,l-1 \text{ and } j=0,\dots,l-1\}$ where a and b are multiples of l, build strings $P_1^{5,a,b}$ and $P_2^{5,a,b}$: For every location (x,y) with $m-a-2l+2 \leq x \leq m$ and $m-b-2l+2 \leq y \leq m$, $P_1^{5,a,b}$ contains $3l-3$ characters for $P[x,y]$, corresponding to the rectangles $H^2_{i,(x,y)}$ for $i = m-a-3l+2,\dots,m-a-1$. Similarly, for every (x,y) with $1 \leq x \leq 2l-1$ and $1 \leq y \leq 2l-1$, $P_2^{5,a,b}$ contains $3l-3$ characters for $P[x,y]$, corresponding to the rectangles $H^2_{i,(x,y)}$ for $i = m-a-3l+2,\dots,m-a-1$.

The total time complexity of stage 2 is $O(m^2 l \cdot \log^{4+o(1)} m)$.

6.3 Stage 3

Let (a,b) be some offset for which no witness was found during the previous stages. We first look for witnesses with one location in $B_3 \cup A_4$ and the other location in $A_3 \cup B_4 \cup A_4$. Define $D' = D \setminus (A_1 \cup B_1 \cup A_2)$. For a rectangle $D'' \subseteq D'$ define $D'' + (a,b) = \{(x+a, x+b) : (x,y) \in D''\}$. Since there are no simple witnesses for (a,b), and no witnesses that satisfy condition 3 in the definition of a simple witness, we conclude that there are no type 2 witnesses with both locations inside D'. Therefore, we make the following observation:

Claim 4 *For every rectangle $D'' \subseteq D'$, the number of distinct characters inside the region D'' of P is less than or equal to the number of distinct characters inside the region D'' of P, with equality if and only if there is no witness w for (a,b) whose both locations are in D''.*

From Claim 4, we devise the following algorithm for finding a witness in D': Check the number of distinct characters inside the regions D' and $D' + (a, b)$ of P. If these numbers are equal then stop (no witness was found). Otherwise, find a minimal rectangle D'' in D' for which the number of distinct characters in D'' is strictly less than the number of distinct characters in $D'' + (a, b)$. Then, one of the pairs of opposite corners of the rectangle D'' is a witness for (a, b). The problem with this algorithm is that we do not know how to efficiently find such a rectangle D''. Instead, we will find a rectangle D^* which will be approximately equal to D''.

In order to find D^*, consider the following intersection counting problem: Given a set S of points in the plane where each point has a color, preprocess S in order to answer efficiently queries of the form "what is the number of distinct color in the points inside the rectangle $(-\infty, b] \times [c, d]$?". Denote by n the number of points in S. Gupta et al. [14] showed a data-structure for this problem with preprocessing time $O(n \log^2 n)$ which answers queries in $O(\log^2 n)$ time. Using this result, we obtain the following lemma:

Lemma 2. *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 $P[a\,..\,b, c\,..\,d]$?", where at least one of a, b, c, d is either 1, m, or a multiple of l.*

We now return to the problem of finding the rectangle D^*. The algorithm is as follows: First, build the data-structure of Lemma 2 on P. Then, compute the number of distinct characters inside the regions D' and $D' + (a, b)$ of P. If these numbers are equal, stop. Otherwise, find a minimal rectangle $D^* = \{x, \ldots, m-a\} \times \{y, \ldots, z\} \subseteq D$ such that the number of distinct characters inside the regions D^* and $D^* + (a, b)$ of P are not equal, and x is a multiple of l. Finding D^* is is done using binary search and queries to the data-structure of Lemma 2 (note that the queries we make satisfy the conditions of Lemma 2).

Let X be the set of all locations (c, d) such that $c \in \{x, \ldots, x+l-1\} \cup \{m-a-l+1, \ldots, m-a\}$ and $d \in \{y, z\}$. From Claim 4 and the minimality of X, we obtain that there is a witness w for (a, b) of type 1 whose both locations are in X. We find such a witness as follows:

1. Initialize tables $V[1\,..\,|\Sigma|]$ and $L[1\,..\,|\Sigma|]$ to zeros.
2. Go over the locations in X in some order. For every location (c, d), if $V[P[c, d]] \notin \{0, P[c+a, d+b]\}$ output the witness $\{L[P[c, d]], (c, d)\}$. Otherwise, set $V[P[c, d]] \leftarrow P[c+a, d+b]$ and $L[P[c, d]] \leftarrow (c, d)$.

By Lemma 2, the time complexity of this stage is $O(\frac{m^3}{l} \log^2 m + m^2 \log^3 m + m^2 l)$ (note that the initialization of the tables V and L above takes $O(1)$ time). Therefore, the total time complexity for preprocessing the pattern is $O((\frac{m}{l} + l)m^2 \cdot \log^{4+o(1)} m)$. The last expression is $O(m^{5/2} \cdot \log^{4+o(1)} m)$ when $l = \Theta(\sqrt{m})$.

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, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications and a lower bound. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 929–942, 2003.
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. Separable attributes: a technique for solving the submatrices character count problem. In *Proc. 13th Symposium on Discrete Algorithms (SODA)*, pages 400–401, 2002.
5. A. Amir and R. Cole. Personal communications, 2004.
6. A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49:111–115, 1994.
7. A. Apostolico, P. Erdős, and M. Lewenstein. Parameterized matching with mismatches. *manuscript.*
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. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th ACM Symposium on the Theory of Computation (STOC)*, pages 71–80, 1993.
10. B. S. Baker. Parameterized string pattern matching. *J. Comput. Systems Sci.*, 52(1):28–42, 1996.
11. B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. on Computing*, 26(5):1343–1362, 1997.
12. R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. In *Proc. 32nd ACM Symposium on the Theory of Computation (STOC)*, pages 407–415, 2000.
13. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing (STOC)*, 67:135–143, 1984.
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. C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. In *Proc. 12th European Symposium on Algorithms (ESA)*, pages 414–425, 2004.
16. S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. *Proc. 36th Symposium on Foundation of Computer Science (FOCS)*, pages 631–637, 1995.
17. M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.
18. U. Vishkin. Optimal parallel pattern matching in strings. In *Proc. 12th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 91–113, 1985.
19. U. Vishkin. Deterministic sampling — a new technique for fast pattern matching. *SIAM J. on Computing*, 20:303–314, 1991.