View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Explore Bristol Research

Clifford, R., Fontaine, A., Starikovskaia, T., & Wedel Vildhoj, H. (2016). Dynamic and approximate pattern matching in 2D. In S. Inenaga, K. Sadakane, & T. Sakai (Eds.), String Processing and Information Retrieval: 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings. (pp. 133-144). (Lecture Notes in Computer Science; Vol. 9954). Springer Verlag. DOI: 10.1007/978-3-319-46049-9_13

Peer reviewed version

Link to published version (if available):
10.1007/978-3-319-46049-9_13

Link to publication record in Explore Bristol Research
PDF-document

This is the accepted author manuscript (AAM). The final published version (version of record) is available online via Springer at http://link.springer.com/chapter/10.1007%2F978-3-319-46049-9_13. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/about/ebr-terms.html

Dynamic and approximate pattern matching in 2D

Raphaël Clifford[1], Allyx Fontaine[1], Tatiana Starikovskaya[1], and Hjalte Wedel Vildhøj[2]

[1] Dept. of Computer Science, University of Bristol, UK
[2] Technical University of Denmark, DTU Compute, Denmark

Abstract. We consider dynamic and online variants of 2D pattern matching between an $m \times m$ pattern and an $n \times n$ text. All the algorithms we give are randomised and give correct outputs with at least constant probability.

- For dynamic 2D exact matching where updates change individual symbols in the text, we show updates can be performed in $\mathcal{O}(\log^2 n)$ time and queries in $\mathcal{O}(\log^2 m)$ time.
- We then consider a model where an update is a new 2D pattern and a query is a location in the text. For this setting we show that Hamming distance queries can be answered in $\mathcal{O}(\log m + H)$ time, where H is the relevant Hamming distance.
- Extending this work to allow approximation, we give an efficient algorithm which returns a $(1+\varepsilon)$ approximation of the Hamming distance at a given location in $\mathcal{O}(\varepsilon^{-2} \log^2 m \log\log n)$ time.

Finally, we consider a different setting inspired by previous work on locality sensitive hashing (LSH). Given a threshold k and after building the 2D text index and receiving a 2D query pattern, we must output a location where the Hamming distance is at most $(1+\varepsilon)k$ as long as there exists a location where the Hamming distance is at most k.

- For our LSH inspired 2D indexing problem, the text can be preprocessed in $\mathcal{O}(n^{2(4/3+1/(1+\varepsilon))} \log^3 n)$ time into a data structure of size $\mathcal{O}(n^{2(1+1/(1+\varepsilon))})$ with query time $\mathcal{O}(n^{2(1/(1+\varepsilon))} m^2)$.

1 Introduction

Two dimensional pattern matching has been a topic of study and great interest for many years. The original motivation comes from image processing and recognition where one is attempting to find possibly approximate occurrences of a 2D-pattern inside a larger 2D-text. For exact matching offline, linear time solutions are known [12, 11, 15] and the indexing problem is solved efficiently with the help of 2D-suffix trees [16]. A number of other variants have also been studied including 2D-compressed pattern matching, matching with rotations, pattern matching with non-rectangular patterns as well as others [2, 4, 9, 3, 5, 14, 6, 7].

We will consider a number of variants of 2D-pattern matching which have to date received little attention. These can broadly be described under the headings of online and dynamic pattern matching. Our focus will be both on exact matching as well as exact and approximate Hamming distance computation. We will also tackle a problem formulation inspired by the locality sensitive hashing work of Andoni and Indyk [10]. Here we are given a pattern as a query and we must report a location in the text where the Hamming distance is not too large as long as one exists. We will now formalise the problems we tackle. All the algorithms we develop will be randomised giving correct answers with at least constant probability. For each problem our input text will be a square matrix $\mathbf{T}$ (the text) of size $n \times n$ and the pattern $\mathbf{P}$ will be of size $m \times m$.

To start we consider a dynamic version of the classic 2D-pattern matching problem. The problem can be seen as a generalisation of the 1D problem considered in [8], where updates are only allowed in the text and the pattern remains static. Our solution relies heavily on Karp-Rabin fingerprinting [18]. The main technical hurdle we overcome is the difficulty in combining fingerprints of adjacent rectangular matrices. We circumvent this problem by only ever combining the fingerprints of two matrices if they are placed horizontally next to each other.

Problem 1 (Dynamic text static pattern matching in 2D). Given a text $\mathbf{T}$ and a pattern $\mathbf{P}$, build a dynamic index that supports an update $(\sigma, (i, j))$ which sets $T[i, j] \leftarrow \sigma$ and query (i, j) which returns True if there is an exact match at location (i, j) in the text and False otherwise.

Our solution to Problem 1 will in fact support the arrival of entire new patterns efficiently as well. For our next two problems we consider online pattern matching problems where the only update is the arrival of a new pattern and a query will return the exact or approximate Hamming distance at some position in the text. Our aim is to perform all three steps, preprocessing, updates and queries as quickly as possible. We denote by $Ham(\mathbf{P}, \mathbf{T})(i, j)$ the Hamming distance between the 2D-pattern $\mathbf{P}$ and the $m \times m$ submatrix of $\mathbf{T}$ with top left corner (i, j).

Problem 2 (Online Exact Hamming Distance in 2D). Given a text $\mathbf{T}$, build a dynamic index that supports updates with a pattern $\mathbf{P}$ and queries which return the value $Ham(\mathbf{P}, \mathbf{T})(i, j)$.

Our solution uses as a preliminary step linearisation of the input by encoding carefully selected substrings of the 2D-text with their Karp-Rabin fingerprints. This will allow us to search efficiently first for mismatches within columns and then rows using dynamic lower common ancestor queries in suitably constructed suffix trees.

To provide faster solutions we then extend this online Hamming distance problem to allow a $(1 + \varepsilon)$ approximation. We show that we can find the approximate value considerably faster than the exact value. To achieve this we use the technique known as sketching [1]. This technique was originally developed for 1D strings but can be transferred to our case by storing sketches of selected substrings of the text $\mathbf{T}$.

Problem 3 (Online Approximate Hamming distance in 2D). Given a binary text $\mathbf{T}$, construct a dynamic index that supports updates with a binary pattern $\mathbf{P}$ and queries which return a $(1+\varepsilon)$ approximation of the Hamming distance $Ham(\mathbf{P},\mathbf{T})(i,j)$.

Finally we turn to a closely related indexing problem. Here we may preprocess the 2D-text and we receive a 2D-pattern as a query along with a threshold k and a constant ε. We must output a location in the text where the Hamming distance is no more than $(1+\varepsilon)k$ as long as there exists a location where the Hamming distance is no more than k.

Problem 4 (Submatrix Near Neighbour Problem). We are initially given a text $\mathbf{T}$, an integer k and a constant $\varepsilon > 0$. Construct an index that supports the following query. Given a pattern $\mathbf{P}$, output a position (i,j) such that $Ham(\mathbf{T},\mathbf{P})(i,j) \leq (1+\varepsilon)\cdot k$ if there exists a submatrix of $\mathbf{T}$ with Hamming distance at most k from $\mathbf{P}$. Otherwise if there is not, the query may either report a location with true Hamming distance up to $(1+\varepsilon)k$ or no location at all.

In the 1D case Andoni and Indyk [10] solved the same problem we study by developing an index on suffixes of a 1D string. To construct their index Andoni and Indyk [10] heavily relied on relationships between suffixes of a 1D string. These relationships do not exist in the 2D case and so we have introduced new techniques and ideas to construct the index. These are our main contribution for Problem 4.

Definitions and notation

We will use two kinds of partitioning of the text and pattern which we term *belts* and *canonical submatrices*. Let S be an $s \times t$ matrix. A *belt* of height $h \leq s$ for the matrix S is a submatrix of S with size $h \times t$. A *canonical submatrix* of S is a submatrix of S with size $2^i \times 2^j$ where $i \leq \log s$ and $j \leq \log t$ are both integers. We will also write $\mathbf{T}[i, i+x-1; j, j+y-1]$ to denote the $x \times y$ submatrix of $\mathbf{T}$ with top left corner at some position (i,j) in the text. We assume throughout that all logarithms are taken base two and for convenience of presentation that both m and n are an exact power of two.

2 Dynamic Text Static Pattern Matching in 2D

As our first contribution we describe a dynamic randomised index that supports efficient exact pattern matching queries as well as updates to $\mathbf{T}$ and hence solves Problem 1.

Theorem 1. *The text $\mathbf{T}$ can be preprocessed in $\mathcal{O}(n^2 \log n)$ time into a data structure of size $\mathcal{O}(n^2)$ so that after processing the pattern $\mathbf{P}$ in $\mathcal{O}(m^2 \log m)$ time, we can support single character updates in $\mathcal{O}(\log^2 n)$ time and query if $\mathbf{P}$ occurs at a position (i,j) of $\mathbf{T}$ w.h.p. in $\mathcal{O}(\log^2 m)$ time.*

The main idea of our dynamic index is to compute the Karp-Rabin fingerprints of submatrices of $\mathbf{T}$ of power of two size in order to be able to compute the fingerprint of the $m \times m$ submatrix with the top left corner at the position (i, j) of $\mathbf{T}$ efficiently. A straightforward partitioning will not suffice however due to the difficulty in computing fingerprints of the concatenation of rectangular matrices.

We start by giving the definition of Karp-Rabin fingerprints for matrices.

Definition 1. *Let S be an $s \times t$ matrix for some $s, t \leq n$. Let $p \geq n^4$ be a prime and r be a random integer in $\mathbb{F}_p$. We define the Karp-Rabin fingerprint φ for S as:*

$$\varphi(S) = \sum_{i=1}^{s} \sum_{j=1}^{t} S[i, j] r^{i+(j-1)s} \pmod{p}$$

Lemma 1. *The Karp-Rabin fingerprints of any two $s \times t$ matrices S, S', where $s, t \leq n$, have the following properties:*

1. *If $S = S'$, then $\varphi(S) = \varphi(S')$;*
2. *If $S \neq S'$, then the probability $\varphi(S) = \varphi(S')$ is at most $1/n^2$.*

Proof. The first claim of the lemma is trivial. To prove the second claim notice that since $\varphi(S) - \varphi(S')$ is a non-trivial polynomial of degree $s \cdot t$, the number of its roots $\in \mathbb{F}_p$ is at most $s \cdot t$. The probability we choose a root randomly from $\mathbb{F}_p$ is at most $\mathcal{O}(s \cdot t/n^4)$. The result holds since $s \cdot t \leq n^2$. □

Moreover, from the definition of Karp-Rabin fingerprints we immediately obtain the following observation. We say that two submatrices are adjacent on the vertical side if they are placed horizontally next to each other. That is $S = \mathbf{T}[i : i+s-1, j : j+t-1]$ and $S' = \mathbf{T}[i : i+s-1, j+t : j+t+t'-1]$

Lemma 2. *Let S, S' be two submatrices of $\mathbf{T}$ adjacent on the vertical side. We can compute the Karp-Rabin fingerprint of $S'' = \mathbf{T}[i : i+s-1, j : j+t+t'-1]$ as*

$$\varphi(S'') = \varphi(S) + r^{st} \cdot \varphi(S') \pmod{p}$$

Proof. The proof follows immediately from the definition. □

We now present our dynamic index. For each $i = 0, 1, \ldots, \log n$ we divide $\mathbf{T}$ into $n/2^i$ non-overlapping belts of height 2^i. For each $j = 0, 1, \ldots, \log n$ we then partition each belt into $n/2^j$ canonical submatrices of width 2^j. For each of the canonical submatrices we store its Karp-Rabin fingerprint in a lookup table. It follows from the fingerprint definition that an individual fingerprint can be updated in constant time if a letter at a particular position in the text is changed. When we change one letter in $\mathbf{T}$, we need to update only $\mathcal{O}(\log^2 n)$ fingerprints, which can be therefore be done in $\mathcal{O}(\log^2 n)$ time in total. The partitioning into belts and canonical submatrices is illustrated in Figure 1.

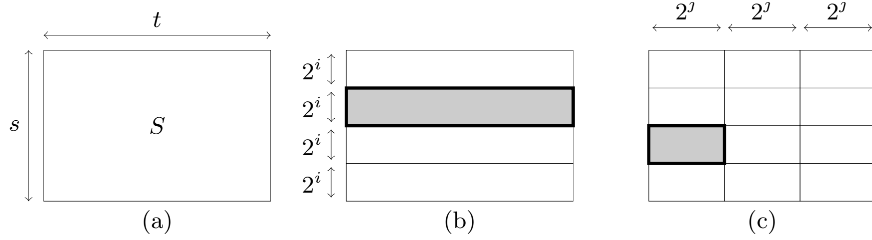

Fig. 1. (a) A matrix S of size $s \times t$. (b) Partition of S into non-overlapping belts of height 2^i. In gray is represented one such belt. (c) Partition of S into canonical submatrices of height 2^i and width 2^j. In gray is represented one such canonical submatrix.

When a pattern arrives, we process it in the following way. For each $i = 0, 1, \ldots, \log m$ we compute and store the Karp-Rabin fingerprints of all $m-2^i+1$ belts of height 2^i. For a fixed value of i we compute the 2D-fingerprints of all $m - 2^i + 1$ belts of height 2^i in two steps. The first step is computing the 2D-fingerprints of all submatrices of size $2^i \times 1$, which we do column by column. For each column j, we first compute the fingerprint φ_1 of the string $\mathbf{P}[1 : 2^i, j]$, and then for each $\ell \geq 1$ we compute the fingerprint $\varphi_{\ell+1}$ of $\mathbf{P}[\ell + 1 : \ell + 2^i, j]$ from the fingerprint $\varphi_{\ell-1}$ of $\mathbf{P}[\ell : \ell + 2^i - 1, j]$ in constant time. As there are m columns of length m each, this step requires $\mathcal{O}(m^2)$ time. The second step consists in computing for each belt its 2D-fingerprint from its columns' fingerprints as described in Lemma 2 in time $\mathcal{O}(m)$.

Suppose now that we are asked if $\mathbf{T}[i, i+m-1; j, j+m-1]$ matches pattern $\mathbf{P}$. We can divide $\mathbf{T}[i : i+m-1, j : j+m-1]$ into $\mathcal{O}(\log m)$ non-overlapping belts with heights that are powers of two. Each belt can then be divided vertically into $\mathcal{O}(\log m)$ canonical submatrices for which we already know their Karp-Rabin fingerprints. With the help of Lemma 2 we compute Karp-Rabin fingerprints of the belts in $\mathcal{O}(\log^2 m)$ time and compare them to those of the pattern.

Construction of the index. We now explain how we construct the text index. We iteratively compute Karp-Rabin fingerprints of canonical matrices of height 2^i, $i = 0, 1, \ldots, \log n$. When the height is fixed, we iteratively compute Karp-Rabin fingerprints of canonical matrices of width 2^j, for $j = 0, 1, \ldots, \log n$.

For each i we start by computing Karp-Rabin fingerprints of all $2^i \times 1$ submatrices in $\mathcal{O}(n^2)$ time in a straightforward manner. When Karp-Rabin fingerprints of $2^i \times 2^j$ submatrices are computed, we can compute Karp-Rabin fingerprints of $2^i \times 2^{j+1}$ submatrices in $\mathcal{O}(n^2/2^j)$ time using Lemma 2. In total, to compute the fingerprints of all submatrices of height 2^i, we need $\mathcal{O}(n^2)$ time. In total, we will need $\mathcal{O}(n^2 \log n)$ time for all submatrices.

3 Online exact 2D Hamming distance

In this section we consider Problem 2. We are given an $n \times n$ text $\mathbf{T}$ which we process first. Updates come in the form of new $m \times m$ patterns $\mathbf{P}$ and a query asks us to return the Hamming distance between $\mathbf{P}$ and the text at location (i, j).

Theorem 2. *The text $\mathbf{T}$ can be preprocessed in $\mathcal{O}(n^2 \log n)$ time into a data structure of size $\mathcal{O}(n^2 \log n)$ so that we can support updates with a new pattern $\mathbf{P}$ in $\mathcal{O}(m^2)$ time and process Hamming distance queries to return up to H mismatches between $\mathbf{P}$ and $\mathbf{T}$ at a position (i, j) in $\mathcal{O}(\log m + H)$ time.*

The index for online exact Hamming distance in 2D

For each $i = 1, 2, \dots, \log n$ we consider $n - 2^i + 1$ belts of height 2^i. We define a linearisation of a belt as a string of length n, where the j-th supercharacter is the Karp-Rabin fingerprint of the j-th column of the belt.

Lemma 3. *The linearisations for all belts of height 2^i for a fixed i can be computed in $\mathcal{O}(n^2)$ time.*

Proof. It suffices to note that for a fixed j the Karp-Rabin fingerprints of j-th columns of all $n - 2^i$ belts can be computed in $\mathcal{O}(n)$ time [18]. □

The main idea will be to first find columns within the pattern that mismatch and then to look within those columns to find individual mismatches. In order to do this efficiently, we compute all linearisations for all belts in $\mathcal{O}(n^2 \log n)$ time and then build a suffix tree for them. We also augment the suffix tree with an efficient dynamic lower common ancestor (LCA) data structure [13]. The suffix tree and the data structure can be built in $\mathcal{O}(n^2 \log n)$ time. We then build a suffix tree for all columns of $\mathbf{T}$ and augment it with the dynamic LCA data structure as well.

When the pattern arrives, we partition it into $\mathcal{O}(\log m)$ non-overlapping belts of power of two heights. We linearise the belts in the way described above and add the linearisations to the generalised suffix tree for the text belts. We also add columns of the pattern to the generalised suffix tree for the columns. This takes time $\mathcal{O}(m^2)$, see [17]. Finally, we update the LCA data structures. In total, this takes $\mathcal{O}(m^2)$ time.

We then work with each of the pattern belts independently. We will use the technique known as *kangaroo jumping* [17, Chapter 9.4]. To find the first H mismatches between the pattern belt of height 2^i and the text, we find the leaf in the suffix tree for the text belt of height 2^i containing the pattern belt and the leaf for the pattern belt and use an LCA query to find the first column of the pattern belt that does not match the corresponding column of the text belt. We then use the generalised suffix tree for the columns and kangaroo jump using LCA queries to report all mismatches in the column in constant time per mismatch. We then go back to the suffix tree for the belts and proceed. When a new pattern update arrives we need first to delete the previous pattern which was added to the two trees.

4 Online approximate Hamming distance in 2D

In this section we consider Problem 3. Assume that we are given an $n \times n$ matrix $\mathbf{T}$ and a constant $\varepsilon > 0$. We assume that we are also given an $m \times m$ pattern matrix $\mathbf{P}$ and that we can process it before answering queries. We will give a text index for $\mathbf{T}$ that will support the following queries: Given a position (i, j) return a $(1 + \varepsilon)$-approximation of the Hamming distance between $\mathbf{P}$ and $\mathbf{T}[i : i + m - 1, j : j + m - 1]$.

Theorem 3. *The text* $\mathbf{T}$ *can be preprocessed in* $\mathcal{O}(\varepsilon^{-2} n^2 \log^3 n \log\log n)$ *time into a data structure of size* $\mathcal{O}(\varepsilon^{-2} n^2 \log^2 n \log\log n)$*. After processing a new pattern* $\mathbf{P}$ *in* $\mathcal{O}(\varepsilon^{-2} m^2 \log\log n)$ *time, we can compute a* $(1 + \varepsilon)$*-approximation of the Hamming distance for any position* (i, j) *in* $\mathbf{T}$ *in* $\mathcal{O}(\varepsilon^{-2} \log^2 m \log\log n)$ *time. The answer is correct with constant probability.*

The index for online approximate Hamming distance in 2D

Consider all $\mathcal{O}(n^2 \log^2 n)$ canonical submatrices of $\mathbf{T}$ of sizes $2^i \times 2^j$ for $i = 1, 2, \ldots, \log n$ and $j = 1, 2, \ldots, \log n$. Let C be a constant to be defined later. For each canonical submatrix we create and store $\gamma = C \log\log n$ vectors (sketches) of length $1/\varepsilon^2$ as follows.

For each pair (i, j) and for each $k = 1, 2, \ldots, \gamma$ we create and store $1/\varepsilon^2$ sign matrices $S_\ell^{i,j,k}$ of size $2^i \times 2^j$. Each entry of a sign matrix is an i.u.d. ± 1 random variable. We now define the k-th sketch of a $2^i \times 2^j$ matrix M as:

$$(\langle M, S_1^{i,j,k} \rangle, \langle M, S_2^{i,j,k} \rangle, \ldots, \langle M, S_{1/\varepsilon^2}^{i,j,k} \rangle)$$

where $\langle M, S_\ell^{i,j,k} \rangle = tr(M^T, S_\ell^{i,j,k})$ is also known as the Hilbert-Schmidt inner product of matrices M and $S_\ell^{i,j,k}$. This sketching technique is a simple variant of the second moment sketches of Alon et al. [1].

Suppose we have two $2^i \times 2^j$ matrices A and B. For each k we approximate the Hamming distance between A and B using the sketches obtained with the help of the sign matrices $S_1^{i,j,k}, S_2^{i,j,k}, \ldots, S_{1/\varepsilon^2}^{i,j,k}$. In particular, the Hamming distance approximation we derive from the k-th sketches is $h_k = \varepsilon^2 \| \langle S_1^{i,j,k}, (A - B) \rangle, \ldots, \langle S_{1/\varepsilon^2}^{i,j,k}, (A - B) \rangle \|_2^2$. It follows from standard techniques that:

Lemma 4. *We can choose a constant* C *so that the median of the Hamming distance approximations over all* $\gamma = C \log\log n$ *sketches for the matrices* A *and* B *will belong to the interval* $[H, (1 + \varepsilon)H]$ *with probability at least* $1 - \frac{1}{2 \log^2 n}$*, where* H *is the Hamming distance between* A *and* B*.*

We process the queries in the following way. For each arriving pattern, we partition $\mathbf{P}$ into $\mathcal{O}(\log^2 m)$ non-overlapping submatrices of sizes $2^i \times 2^j$. Next, we compute sketches of all submatrices in the partition with the help of sign matrices, which takes $\mathcal{O}(\varepsilon^{-2} m^2 \log\log n)$ time, but we only need to do this once. When a query arrives, that is when we receive a position (i, j), we consider the

same partitioning of $\mathbf{T}[i : i+m-1, j : j+m-1]$. For each corresponding pair of submatrices in the partitioning of $\mathbf{P}$ and $\mathbf{T}[i : i+m-1, j : j+m-1]$ we compute the $(1+\varepsilon)$-approximations of Hamming distances with the help of the sketches. By Lemma 4 and the union bound, the sum of these values will be a $(1+\varepsilon)$-approximation between $\mathbf{P}$ and $\mathbf{T}[i : i+m-1, j : j+m-1]$ with constant probability. Processing a query takes $\mathcal{O}(\varepsilon^{-2} \log^2 m \log\log n)$ time.

4.1 Construction of the index

We finally explain how to compute the sketches of the canonical matrices. To compute the sketches for one canonical matrix of size $2^i \times 2^j$ we need only perform a sequence of 2D convolutions. In total, computing the sketches of all canonical submatrices of size $2^i \times 2^j$ takes $\mathcal{O}(\varepsilon^{-2} n^2 \log n \log\log n)$ time. Therefore, computing all sketches of all canonical submatrices over all sizes takes $\mathcal{O}(\varepsilon^{-2} n^2 \log^3 n \log\log n)$ time.

5 Submatrix near neighbour problem

In this section we consider Problem 4. Assume that we are given an $n \times n$ matrix $\mathbf{T}$, an integer k, and a constant $\varepsilon > 0$. We will give a text index for $\mathbf{T}$ which will support the following queries: Given an $m \times m$ pattern matrix $\mathbf{P}$ such that there is a k-mismatch occurrence of $\mathbf{P}$ in $\mathbf{T}$, return an occurrence where the Hamming distance is at most $(1+\varepsilon) \cdot k$. Let $N = n^2$ and $M = m^2$. We will show that

Theorem 4. $\mathbf{T}$ *can be preprocessed in* $\mathcal{O}(N^{4/3+1/(1+\varepsilon)} \log^3 N)$ *time into a data structure of size* $\mathcal{O}(N^{1+1/(1+\varepsilon)})$ *with query time* $\mathcal{O}(N^{1/(1+\varepsilon)} M)$. *If* $\mathbf{T}$ *contains a* k*-mismatch occurrence of* $\mathbf{P}$*, then the data structure w.h.p. retrieves a* $(1+\varepsilon) \cdot k$*-mismatch occurrence of* $\mathbf{P}$ *in* $\mathbf{T}$.

The index for submatrix nearest neighbour search

We will start by recalling the notion of the L-encoding of a matrix.

Definition 2 ([16]). *The L-encoding of an* $n \times n$ *matrix* $\mathbf{T}$ *is a string* $s_1 s_2 \ldots s_n$ *of length* n^2*, where* $s_i = T[i : i, 1 : i-1]T[1 : i, i : i]$. *(See Fig. 2)*

Note that if $\mathbf{P}$ occurs in the top left corner of $\mathbf{T}$ with k mismatches, then the L-encoding of $\mathbf{T}$ starts with a k-mismatch occurrence of the L-encoding of $\mathbf{P}$. A suffix of $\mathbf{T}$ is the L-encoding of a square submatrix with bottom right hand corner in the last row or in the last column of $\mathbf{T}$. Let $S_1, S_2, \ldots, S_N$ be the suffixes of $\mathbf{T}$. A k-mismatch occurrence of $\mathbf{P}$ in $\mathbf{T}$ guarantees that at least one of the L-encodings $S_1, S_2, \ldots, S_N$ starts with a k-mismatch occurrence of the L-encoding of $\mathbf{P}$, and vice versa. We will make use of data structure by Andoni and Indyk which we call *sketch forest*. The following corollary follows directly from the work of [10, Section 2].

$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,4}$	$s_{1,5}$
$s_{2,1}$	$s_{2,2}$	$s_{2,3}$	$s_{2,4}$	$s_{2,5}$
$s_{3,1}$	$s_{3,2}$	$s_{3,3}$	$s_{3,4}$	$s_{3,5}$
$s_{4,1}$	$s_{4,2}$	$s_{4,3}$	$s_{4,4}$	$s_{4,5}$
$s_{5,1}$	$s_{5,2}$	$s_{5,3}$	$s_{5,4}$	$s_{5,5}$

(a)

$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,4}$	$s_{1,5}$
$s_{2,1}$	$s_{2,2}$	$s_{2,3}$	$s_{2,4}$	$s_{2,5}$
$s_{3,1}$	$s_{3,2}$	$s_{3,3}$	$s_{3,4}$	$s_{3,5}$
$s_{4,1}$	$s_{4,2}$	$s_{4,3}$	$s_{4,4}$	$s_{4,5}$
$s_{5,1}$	$s_{5,2}$	$s_{5,3}$	$s_{5,4}$	$s_{5,5}$

(b)

Fig. 2. A submatrix S of the text matrix $\mathbf{T}$. (a) The L-encoding of the submatrix S is $s_{1,1}s_{2,1}s_{1,2}s_{2,2}s_{3,1}s_{3,2}s_{1,3}s_{2,3}s_{3,3}\ldots s_{5,5}$. $L_{3:4}$ is the L-shape formed by the 3-rd and the 4-th rows and the 3-rd and the 4-th columns (shown in bold). (b) Let g be a projection onto a set of $\ell = 9$ positions $\{1, 2, 6, 8, 10, 17, 18, 19, 20\}$ (highlighted in gray), i.e. $g(S) = s_{1,1}s_{2,1}s_{3,2}s_{2,3}s_{4,1}s_{5,1}s_{5,2}s_{5,3}s_{5,4}$. The blocks will be $\{1, 2\}$, $\{6, 8, 10\}$, $\{17, 18, 19\}$, $\{20\}$. The corresponding partitioning of S into L-shapes and rectangles is shown on the figure by bold lines.

Corollary 1. *A sketch forest on a set of strings $S = \{S_1, S_2, \ldots, S_N\}$ occupies $\mathcal{O}(N^{1+1/(1+\varepsilon)})$ space. If at least one of the strings starts with a k-mismatch of the L-encoding of $\mathbf{P}$, then the data structure will identify in $\mathcal{O}(N^{1/(1+\varepsilon)}M)$ time a subset of $\mathcal{O}(N^{1/(1+\varepsilon)})$ suffixes of $\mathbf{T}$ that w.h.p. contains at least one suffix starting with a $(1+\varepsilon)\cdot k$-mismatch occurrence of the L-encoding of $\mathbf{P}$.*

After having identified the subset of $\mathcal{O}(N^{1/(1+\varepsilon)})$ suffixes of $\mathbf{T}$, we check for each of them if it starts with a $(1+\varepsilon)\cdot k$-mismatch occurrence of the L-encoding of $\mathbf{P}$ in a straightforward manner, comparing the letters of the suffix and the L-encoding of $\mathbf{P}$ one by one. In total, this takes $\mathcal{O}(N^{1/(1+\varepsilon)}M)$ more time.

The work of Andoni and Indyk heavily relied for its efficiency on the fact that different suffixes of a single string are suffixes of each other. However, in our linearisation of the text $\mathbf{T}$ this is no longer true. This requires us to devise a new method to construct the sketch forest efficiently which we now describe.

5.1 Construction

In this section we explain how we build the sketch forest. We start by describing its main elements.

Let $p_1 = 1 - k/N$, and $p_2 = 1 - (1+\varepsilon)\cdot k/N$. The intuition behind these values is as follows: If S_1, S_2 are two strings of length N, then p_1 is a lower bound for the probability of two letters $S_1[i], S_2[i]$ to be equal if the Hamming distance between S_1 and S_2 is at most k. On the other hand, p_2 is an upper bound for the probability of two letters $S_1[i], S_2[i]$ to be equal if the Hamming distance between S_1 and S_2 is at least $(1+\varepsilon)\cdot k$.

Let $\mathcal{H}$ be a set of projections of a string along a fixed coordinate, i.e. the j-th projection maps a string onto its j-th letter. A sketch forest is defined by

a family of $N^\rho = \mathcal{O}(N^{1/(1+\varepsilon)})$ random functions $g_i \in \mathcal{H}^\ell$, where $\rho = \frac{\log p_1}{\log p_2}$ and $\ell = \frac{\log N}{\log 1/p_2}$. The choice of ρ and ℓ guarantees low error probability and space complexity. Each of the functions g_i can be considered as a projection along a randomly chosen set of coordinates of size $\ell \leq N$. The sketch forest contains exactly one trie for each projection function in the family. A trie $\mathcal{T}_{g_i}$ contains sketches $g_i(S_1), g_i(S_2), \ldots, g_i(S_N)$ of all strings in the set.

Fix a projection function $g \in \{g_1, g_2, \ldots, g_{N^\rho}\}$. We will show that the trie $\mathcal{T}_g$ can be built in $\mathcal{O}(N^{4/3}\log^2 N)$ time. As an immediate corollary, all tries in the sketch forest can be built in $\mathcal{O}(N^{4/3+1/(1+\varepsilon)}\log^2 N)$ time.

We start building the trie $\mathcal{T}_g$ by sorting the strings $g(S_1), g(S_2), \ldots, g(S_N)$ lexicographically and computing the longest common prefixes of all adjacent strings in that order. Below we show that this can be done in $\mathcal{O}(N^{4/3}\log^2 N)$ time. After having sorted the strings we build $\mathcal{T}_g$ in $\mathcal{O}(N)$ time by using this longest common prefix information.

We now explain how we sort $g(S_1), g(S_2), \ldots, g(S_N)$. Our algorithm will follow the lines of that of [10], but because $S_1, S_2, \ldots, S_N$ are suffixes of a 2D string and not a 1D string as in [10], we will have to introduce some new techniques.

String sorting in $\mathcal{O}(N^{4/3}\log^2 N)$ time. We will give two methods for sorting strings $g(S_1), g(S_2), \ldots, g(S_N)$. *Sort A* will run in $\mathcal{O}(N\sqrt{\ell}\log^2 N)$ time and *Sort B* will run in $\mathcal{O}(N\log^2 N/\ell)$ time. We will use Sort A if $\ell \leq N^{2/3}$ and Sort B if $\ell > N^{2/3}$.

Both Sort A and Sort B need to make at most $N\log N$ string comparisons. Note that in fact all we need to compare two strings is to find the first mismatch between them. For Sort A, we will show that after $\mathcal{O}(N\sqrt{\ell}\log^2 N)$-time preprocessing it is possible to find the first mismatch between any two strings in $\mathcal{O}(\sqrt{\ell})$ time. As a result, the total running time of sort A is $\mathcal{O}(N\sqrt{\ell}\log^2 N)$. For Sort B, we will show that the first mismatch between any two strings can be found in $\mathcal{O}(N\log N/\ell)$ time, which will give $\mathcal{O}(N^2\log^2 N/\ell)$ time in total.

Sort A. Let g be a projection function onto positions $p_1 < p_2 < \ldots < p_\ell$. We will divide this set into $\mathcal{O}(\sqrt{\ell})$ blocks of consecutive positions of length at most $\sqrt{\ell}$ each. The method will consist of two steps. We will start by finding the first block containing a mismatch. After having found the block, we will iterate over all positions in it to find the desired mismatch. The second step can be implemented in a straightforward manner and requires $\mathcal{O}(\sqrt{\ell})$ time.

We will now explain how we implement the first step. Let us start by explaining how we divide the sequence $p_1 < p_2 < \ldots < p_\ell$ into blocks. Remember that these are positions in the L-encoding of an $n \times n$ matrix. Let $L_{i:j}$ be the L-shape formed by the i-th to j-th rows and the i-th to j-th columns (see Fig. 2 for an example).

We start by greedily dividing the matrix into L-shapes, where each L-shape either contains at most $\sqrt{\ell}$ sampled positions (type I L-shapes) or is of form $L_{i:i}$ (type II L-shapes). We first find the largest i_1 such that $L_{1:i_1}$ contains at most $\sqrt{\ell}$ sampled positions. We then try to find the largest i_2 such that $L_{i_1+1:i_2}$ contains

at most $\sqrt{\ell}$ sampled positions. If such i_2 does not exist, we let $i_2 = i_1 + 1$, and continue in the same fashion. We further divide each type-II L-shape into the smallest number of horizontal and vertical rectangles containing at most $\sqrt{\ell}$ sampled positions each. The corner element forms a separate 1×1 rectangle.

This partitioning of the matrix into L-shapes and rectangles defines a partitioning of $p_1 < p_2 < \ldots < p_\ell$ into $\mathcal{O}(\sqrt{\ell})$ blocks, containing at most $\sqrt{\ell}$ of the sampled positions each. Note that positions in each block are consecutive, that is they form a single range of the sequence $p_1 < p_2 < \ldots < p_\ell$. Each block defines a projection of a matrix onto at most $\sqrt{\ell}$ positions, and we will now define and compute a hash function of these projections.

For a rectangular block, we define the hash function to be the Karp-Rabin fingerprint of the projection. We can compute the values of this hash function for all suffixes $S_1, S_2, \ldots, S_N$ in $\mathcal{O}(N \log N)$ time as a convolution of rows or columns of $\mathbf{T}$ with a suitable vector.

Example 1. Consider Fig. 2. The hash function for the block $\{17, 18, 19\}$ is the Karp-Rabin fingerprint of $s_{5,1}s_{5,2}s_{5,3}$.

For an L-shaped block we define the hash function differently. First, we divide the L-shape into two halves, a horizontal one and a vertical one. The hash function will be defined as a pair of fingerprints. The first fingerprint will be defined to be the Karp-Rabin fingerprint of a permutation of the projection on the sampled positions in the horizontal half obtained by reading the positions by columns, and the second fingerprint as the Karp-Rabin fingerprint of a permutation of the projection on the sampled positions in the vertical half obtained by reading the positions by columns.

Example 2. Consider Fig. 2. The L-shape $L_{3:4}$ is divided into two halves by a dashed line. The hash function of the horizontal half is the Karp-Rabin fingerprint of $s_{4,1}s_{3,2}$. The hash function of the vertical half is the Karp-Rabin fingerprint of $s_{2,3}$.

The Karp-Rabin fingerprints of the horizontal and vertical parts for a fixed L-shape and all suffixes $S_1, S_2, \ldots, S_N$ can be computed in $\mathcal{O}(N \log N)$ time as a sequence of 2D convolutions. In total, computing the hash functions for all L-shaped blocks takes $\mathcal{O}(N\sqrt{\ell} \log N)$ time.

Sort B. Similarly to Section 3, we consider $n - 2^i$ belts of $\mathbf{T}$ of height 2^i for each $i = 1, 2, \ldots, \log n$. We then linearise them, build a suffix tree and augment it with the LCA data structure. The tree can be constructed in $\mathcal{O}(N \log N)$ time and occupies $\mathcal{O}(N \log N)$ space. With the help of the suffix tree and kangaroo jumps we can report up to t mismatches between any two $2^i \times j$ submatrices S_1, S_2 of $\mathbf{T}$ in $\mathcal{O}(t)$ time.

We also build a generalised suffix tree for all columns and rows of $\mathbf{T}$, which occupies $\mathcal{O}(N)$ space and augment it with the LCA data structure as well.

As it was shown in [10], w.h.p. the first mismatch between $g(S_i)$ and $g(S_j)$ is contained in the first $3N \log N/\ell$ mismatches between S_i and S_j. We will use

binary search and the suffix trees for the belts to extract these mismatches. When a mismatch is extracted, we check if it belongs to $\{p_1, p_2, \ldots, p_\ell\}$ in constant time and stop if it does.

We start by finding the smallest t such that there are at least $3N \log N/\ell$ mismatches between the $t \times t$ top left submatrices of S_i and S_j. We do so by binary search on t. For each value of t we divide the $t \times t$ top left submatrices into a logarithmic number of even smaller submatrices of size power of two by t. For any pair of such submatrices of S_i and S_j we can use the suffix trees for the belts and for the columns to list the mismatches between them in constant time per mismatch using the kangaroo method. We stop when we have found $3N \log N/\ell$ mismatches, so we never spend more than $\mathcal{O}(3N \log N/\ell)$ time.

We guarantee that there are at least $3N \log N/\ell$ mismatches between the $t \times t$ submatrices of S_i and S_j. Unfortunately, there can be much more mismatches if the L-shapes $L_{t:t}$ of these submatrices contain many mismatches. However, using the suffix trees for columns and for rows, we can list the mismatches between these two L-shapes in order in constant time per mismatch.

References

1. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC '96*, pages 20–29. ACM, 1996.
2. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Data Compression Conference, 1992. DCC'92.*, pages 279–288. IEEE, 1992.
3. A. Amir and G. Benson. Two-dimensional periodicity in rectangular arrays. *SIAM J. on Comp.*, 27(1):90–106, 1998.
4. A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Journal of Algorithms*, 24(2):354 – 379, 1997.
5. A. Amir, A. Butman, M. Crochemore, G. M. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. *Theoretical Computer Science*, 314(1):173–187, 2004.
6. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *SODA*, pages 212–223, 1991.
7. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Inf. and Computation*, 118(1):1–11, 1995.
8. A. Amir, G. M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms (TALG)*, 3(2), 2007.
9. A. Amir, G. M. Landau, and D. Sokol. Inplace run-length 2d compressed search. *Theoretical Computer Science*, 290(3):1361–1383, 2003.
10. A. Andoni and P. Indyk. Efficient algorithms for substring near neighbor problem. In *SODA '06*, pages 1203–1212, 2006.
11. T. P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. on Comp.*, 7(4):533–541, 1978.
12. R. S. Bird. Two dimensional pattern matching. *IPL*, 6(5):168–170, 1977.
13. R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.
14. K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In *CPM '02*, pages 235–248. Springer, 2002.

15. Z. Galil and K. Park. Truly alphabet-independent two-dimensional pattern matching. In *FOCS '92*, pages 247–256, 1992.
16. R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. on Comp.*, 24(3):520–562, 1995.
17. D. Gusfield. *Algorithms on strings, trees and sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.
18. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res Dev*, 31(2):249 –260, 1987.