Approximate Nearest Neighbor Problem in High Dimensions

by

Alexandr Andoni

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by . . . .
Piotr Indyk
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Approximate Nearest Neighbor Problem in High Dimensions

by

Alexandr Andoni

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We investigate the problem of finding the approximate nearest neighbor when the data set points are the substrings of a given text T. The exact version of this problem is defined as follows. Given a text T of length n, we want to build a data structure that supports the following operation: given a pattern P, find the substring of T that is the closest to P. Since the exact version of this problem is surprisingly difficult, we address the approximate version, in which we are allowed to return a substring of T that is at most c times further than the actual closest substring of T. This problem occurs, for example, in computational biology [4, 5].

In particular, we study the case where the length of the pattern P, denoted by m, is *not* known in advance, which is the most natural scenario. We present a data structure that uses $O(n^{1+1/c})$ space and has $\tilde{O}\left(n^{1/c} + mn^{o(1)}\right)$ query time[1] when the distance between two strings is the Hamming distance. These bounds essentially match the earlier bounds of [12], which assumed that the pattern length m is *fixed in advance*. Furthermore, our data structure can be constructed in $\tilde{O}\left(n^{1+1/c} + n^{1+o(1)}M^{1/3}\right)$ time, where M is an upper bound for m. This time essentially matches the preprocessing time of [12] as long as the term $\tilde{O}(n^{1+1/c})$ dominates the running time, which is the case when, for example, $c < 3$.

We also extend our results to the case where the distances are measured according to the l_1 distance. The query time and the space bound are essentially the same, while the preprocessing time becomes $\tilde{O}\left(n^{1+1/c} + n^{1+o(1)}M^{2/3}\right)$.

Thesis Supervisor: Piotr Indyk
Title: Associate Professor

[1]We use notation $f(n) = \tilde{O}(g(n))$ to denote $f(n) = g(n)\log^{O(1)} g(n)$.

Acknowledgments

For all my interest and research achievements in the thrilling field of theoretical computer science, I am most and deeply grateful to Professor Piotr Indyk. Not only was he the best possible advisor for the last one year and a half, but he was also the first professor to offer me the immediate opportunity to do research in this field, back in June 2003. Since then, his insight, his drive, and, in no way less important, his amicability cultivated in me deeper and deeper passion for theoretical computer science.

I also thank Professor Piotr Indyk for being always eager to meet with me, for discussing with me all my good and (usually) bad ideas, and for suggesting new directions and ideas in the most needed moments. He taught me to genuinely enjoy the field.

For my progress on this thesis, the help of my peers was very important, especially, that of Mihai Pătraşcu and Ilya Baran. They were foremost my friends to whom I could relate my problems, as well as computer scientists who could at least agree with me that my problems are interesting. I thank Mihai particularly for asking me many of the right questions, questions that pushed me further. I thank Ilya particularly for his comments on earlier drafts of this work.

Furthermore, the support of my generous friends proved essential to my morale and health. I thank all of them for their genuine support, for relaxing conversations and laughs, and for not letting me enjoy my meals in solitude. My special thanks I give to my friend and roommate Tudor Leu for always being there to break my solitude, for sharing many of the good times sitting in the cluster late at night, and, most importantly, for his admirable courage to live in the same apartment with me. In addition, my exceptional thanks are for Dumitru Daniliuc for being my best friend for the past several years, the friend on whom I could rely for anything.

Yet another low bow I give to the people and agencies on whose financial support I most essentially depended in all my years at MIT, as well as to my high school professors and mentors, whose enormous effort not only brought me the opportunity of better higher education, but who foremost taught me the love for science and computer science in particular.

Still, my absolutely deepest gratitude I owe to Ana Sîrbu and especially to my family for all their unconditional support, care, and understanding. My reverence for them cannot be expressed in words.

Contents

Chapter 1

Introduction

The nearest neighbor problem is defined as follows: given a set $\mathcal{S}$ of n points in $\mathbb{R}^m$, construct a data structure that, given any $q \in \mathbb{R}^m$, quickly finds the point $p \in \mathcal{S}$ that has the smallest distance to q. This problem and its decision version (the R-*near* neighbor) are the central problems in computational geometry. Since the exact problem is surprisingly difficult (for example, it is an open problem to design an algorithm for $m = 3$ which uses sub-quadratic space and has $\log^{O(1)} n$ query time), recent research has focused on designing efficient *approximation* algorithms. Furthermore, the approximate nearest neighbor is reducible to the approximate R-near neighbor [15], and, therefore, we will be primarily concentrating our attention on the latter problem. In the approximate R-near neighbor problem[1], the data structure needs to report a point within distance cR from q for some constant $c > 1$, but only if there exists a point at distance R from q. We will refer to this problem as an (R, c)-*near neighbor (NN)* problem.

The approximate near and nearest neighbor problems have been studied for a long time. The approximate nearest neighbor algorithms were first discovered for the "low-dimensional" version of the problem, where m is constant (see, e.g., [2] and the references therein). Later, a few results were obtained for the "high-dimensional" case, where m is a parameter (see, e.g., [18, 15, 19, 8]). In particular, the Locality-Sensitive Hashing (LSH) algorithm of [15]

[1] The approximate nearest neighbor problem is defined in an analogous way.

solves the (R,c)-near neighbor problem using[2] $O(mn^{1+1/c})$ preprocessing time, $O(mn + n^{1+1/c})$ space and $O(mn^{1/c})$ query time. By using the dimensionality reduction of [19], the query time can be further reduced to $\tilde{O}(m + n^{1/c})$, while the preprocessing time can be reduced to $\tilde{O}(mn+n^{1+1/c})$. The LSH algorithm has been successfully used in several applied scenarios, including computational biology (cf. [6, 5] and the references therein, or [16], p. 414).

The bounds of the LSH algorithm can sometimes be even further reduced if the points in the set $\mathcal{S}$ are not arbitrary, but instead are implicitly defined by a (smaller) data set. This is the case for many of the applications of the approximate nearest neighbor problem.

Particularly interesting is the case in which $\mathcal{S}$ is defined as the set of m-substrings of a sequence of numbers $T[0 \dots n-1]$; we call the resulting problem an (R,c)-*substring near neighbor (SNN)* problem. (R,c)-SNN problem occurs, for example, in computational biology [4, 5]. Its exact version (i.e., when $c = 1$) has been a focus of several papers in the combinatorial pattern matching area (cf. [7] and the references therein).

Obviously, one can solve (R,c)-SNN by reducing it to (R,c)-NN. Specifically, we can enumerate all m-length substrings of T and use them as an input to the (R,c)-NN problem. Then, if one uses the LSH algorithm to solve the near neighbor problem, then the space usage can be reduced from $O(nm + n^{1+1/c})$ to $O(n^{1+1/c})$ (since one can represent the substrings implicitly). Moreover, the preprocessing time can be reduced from $O(mn^{1+1/c})$ to $O(\log m \cdot n^{1+1/c})$ by using FFT [12].

A deficiency of this approach lies in the fact that the query pattern size m must be *fixed in advance*. This assumption is somewhat restrictive in the context of searching in sequence data. A straight-forward solution would be to build a data structure for each possible $m \in \{0 \dots M-1\}$, where M is the maximum query size. However, the space and the preprocessing time would increase to $\tilde{O}(n^{1+1/c}M)$.

In this work, we give improved algorithms for the approximate substring near neighbor problem for *unknown* string length m. Our algorithms achieve query time of $\tilde{O}(n^{1/c}+mn^{o(1)})$,

[2]The bounds refer to the time needed to solve the problem in the m-dimensional Hamming space $\{0,1\}^m$; slightly worse bounds are known for more general spaces.

while keeping space of $\tilde{O}(n^{1+1/c})$. Note that this essentially matches the query and the space bounds for the case where m is fixed in advance. If the distances are measured according to the Hamming metric, the preprocessing time is $\tilde{O}\left(n^{1+1/c} + n^{1+o(1)}M^{1/3}\right)$. Thus, our preprocessing essentially matches the bound for the case of fixed m, as long as $c < 3$.

If the distances are measured according to the l_1 norm, we achieve the same query and space bounds, as well as preprocessing time of $\tilde{O}\left(n^{1+1/c} + n^{1+o(1)}M^{2/3}\right)$. For this algorithm, we need to assume that the alphabet Σ of the text is discrete; that is, $\Sigma = \{0 \ldots \Delta\}$. Although such an assumption is not very common in computational geometry, it is typically satisfied in practice when the bounded precision arithmetic is used.

1.1 Our Techniques

Our algorithms are based on the Locality-Sensitive Hashing (LSH) algorithm. The basic LSH algorithm proceeds by constructing $L = O(n^{1/c})$ hash tables. Each point $p \in \mathcal{S}$ is then hashed into each table; the i^{th} table uses a hash function g_i. The query point is hashed L times as well; the points colliding with the query are reported. For a more detailed description of LSH, see the next section.

In order to eliminate the need to know the value of m in advance, we replace each hash table by a *trie*[3]. Specifically, for each g_i, we build a trie on the strings $g_1(p) \ldots g_L(p)$, where p is a suffix of T. Searching in a trie does not require advance knowledge of the search depth. At the same time, we show that, for the case of the Hamming distance, the LSH analysis of [15] works just as well even if we stop the search at an arbitrary moment.

Unfortunately, constructing the trie of strings $g_1(p) \ldots g_L(p)$ cannot be accomplished using the approach of [12]. In a naive algorithm, which explicitly constructs the tries, constructing one trie would take $O(Mn)$ time instead of the optimal $\tilde{O}(n)$. We show how to reduce this time considerably, to $\tilde{O}(M^{1/3}n)$.

In order to reduce the query and preprocessing bounds even further, we redesign the

[3]An implementation of LSH using a trie has been investigated earlier in [21]. However, the authors used that approach to get a simpler algorithm for the near neighbor, not for the string near neighbor problem.

LSH scheme. In the new scheme, the functions g_i are not totally independent. Instead, they are obtained by concatenating tuples of a smaller number of independent hash functions. The smaller number of the "base" hash functions enables faster query time and preprocessing computation. Using this approach, we achieve $\tilde{O}\left(n^{1/c} + mn^{o(1)}\right)$ query time and $\tilde{O}\left(n^{1+1/c} + n^{1+o(1)} M^{1/3}\right)$ preprocessing time. This part is the most involved part of the algorithm.

For the more general l_1 norm, we assume that the numbers are integers in the range $\Sigma = \{0 \ldots \Delta\}$. One approach to solve the l_1 case is to reduce it to the Hamming metric case. Then, we replace each character from Σ by its unary representation: a character a is replaced by a ones followed by $\Delta - a$ zeros. Unfortunately, this reduction multiplies the running time by a factor of Δ.

To avoid this deficiency, we proceed by using locality-sensitive hash functions designed[4] specifically for the l_1 norm. In particular, we compute the value of the hash function on a point in the m-dimensional space by imposing a regular grid in $\mathbb{R}^m$, and shifting it at random. Then each point is hashed to the grid cell containing it. We show that such a hash function is locality-sensitive. Moreover, we show that, by using pattern-matching techniques (notably, algorithms for the *less-than-matching* problem [1]), we can perform the preprocessing in less than $O(Mn)$ time per function g_i. We mention that less-than matching has been earlier used for a geometric problem in [9].

Finally, to achieve the stated bounds for l_1, we apply the technique of reusable g_i functions, as in the case of the Hamming distance.

1.2 Preliminaries

1.2.1 Notation

For a string $A \in \Sigma^*$ of length $|A|$ and a string $X \in \{0,1\}^*$, we define:

[4] One can observe that such functions can be alternatively obtained by performing the unary mapping into the Hamming space, and then using the bit sampling hash functions of [15], where the sampled positions form arithmetic progression. However, this view is not useful for the purpose of our algorithm.

- A_i^m is the substring of A of length m starting at position i (if the substring runs out of bounds of A, we pad it with 0s at the end);

- $A \odot X = (A[0] \odot X[0], A[1] \odot X[1], \dots A[n-1] \odot X[n-1])$, where $n = \min\{|A|, |X|\}$, and $\odot$ is a product operation such that for any $c \in \Sigma$, $c \odot 1 = c$ and $c \odot 0 = 0$.

Further, let $I \subseteq \{0, \dots M-1\}$ be a set of size $k \leq M$; we call I a *projection set*. For a string A, $|A| = M$, we define:

- $A|_I$ is the string $(A_{i_1} A_{i_2} \dots A_{i_k})$ of length k, where $\{i_1, i_2, \dots i_k\} = I$, and $i_1 < i_2 < \dots < i_k$;

- X_I is a string of length M with $X_I[i] = 1$ if $i \in I$ and $X_I[i] = 0$ if $i \notin I$.

1.2.2 Problem definition

We assume that the text $T[0 \dots n-1]$ and the query pattern $P[0 \dots m-1]$ are in some alphabet space Σ. Furthermore, for two strings $A, B \in \Sigma^m$, we define $D(A, B)$ to be the distance between the strings A and B (examples of the distance D are the Hamming distance and the l_1 distance). Finally, we assume that $\Sigma \subset \mathbb{N}$ and that $|\Sigma| \leq O(n)$ since we can reduce the size of the alphabet to the number of encountered characters.

In our study, we focus on the following problem.

Definition 1.2.1. *The* (R, c)-Substring Near Neighbor (SNN) *is defined as follows. Given:*

- *Text* $T[0 \dots n-1]$, $T[i] \in \Sigma$;

- *Maximum query size* M;

construct a data structure $\mathcal{D}$ *that supports* (R, c)*-near substring query. An* (R, c)*-near substring query on* $\mathcal{D}$ *is of the form:*

- **Input** *is a pattern* $P[0 \dots m-1]$, $P[i] \in \Sigma$, $1 \leq m \leq M$;

- **Output** *is a position* i *such that* $D(T_i^m, P) \leq cR$ *if there exists* i^* *such that* $D(T_{i^*}^m, P) \leq R$.

In the following section, we present the LSH scheme for solving the (R, c)-near neighbor problem. LSH is our main tool for solving the (R, c)-SNN problem.

1.2.3 Locality-Sensitive Hashing

In this section we briefly describe the LSH scheme (Locality-Sensitive Hashing) from [15, 11]. The LSH scheme solves the (R, c)-near neighbor problem, which is defined below.

Definition 1.2.2. *The* (R, c)-near neighbor problem *is defined as follows. Given a set $\mathcal{S}$ of n points in the metric space (Σ^d, D), construct a data structure that, for a query point $q \in \Sigma^d$, outputs a point v such that $D(v, q) \leq cR$ if there exists a point v^* such that $D(v^*, q) \leq R$.*

We call a *ball* of radius r centered at v, the set $B(v, r) = \{q \mid D(v, q) \leq r\}$.

Generic locality-sensitive hashing scheme

The generic LSH scheme is based on an LSH family of hash functions that can be defined as follows.

Definition 1.2.3. *A family $\mathcal{H} = \{h : \Sigma^d \to U\}$ is called* (r_1, r_2, p_1, p_2)-sensitive*, if for any $q \in \mathcal{S}$:*

- *If $v \in B(q, r_1)$, then $Pr[h(q) = h(v)] \geq p_1$;*
- *If $v \notin B(q, r_2)$, then $Pr[h(q) = h(v)] \leq p_2$.*

Naturally, we would like $r_1 < r_2$ and $p_1 > p_2$; that is, if the query point q is close to v, then q and v should likely fall in the same bucket. Similarly, if q is far from v, then q and v should be less likely to fall in the same bucket. In particular, we choose $r_1 = R$ and $r_2 = cR$.

Since the gap between probabilities p_1 and p_2 might not be sufficient, we need to amplify this gap. For this purpose, we concatenate several functions $h \in \mathcal{H}$. In particular, for some value k, define a function family $\mathcal{G} = \{g : \Sigma^d \to U^k\}$ of functions $g(v) = (h_1(v), \ldots, h_k(v))$, where $h_i \in \mathcal{H}$. Next, for some value L, choose L functions $g_1, \ldots, g_L$ from $\mathcal{G}$ independently at random. During preprocessing, the algorithm stores each $v \in \mathcal{S}$ in buckets $g_i(v)$, for all

$i = 1, \ldots, L$. Since the total number of buckets may be large, the algorithm retains only the non-empty buckets by resorting to hashing.

To process a query q, the algorithm searches buckets $g_1(q), \ldots, g_L(q)$. For each point v found in one of these buckets, the algorithm computes the distance from q to v and reports the point v iff $D(v, q) \leq cR$. If the buckets $g_1(q), \ldots, g_L(q)$ contain too many points (more than $3L$), the algorithm stops after checking $3L$ points and reports that no point was found. Query time is $O\left(L(k+d)\right)$, assuming that computing one function g takes $O(k+d)$ time, which is the case for the LSH family we consider.

If we choose $k = \log_{1/p_2} n$ and $L = n^\rho$, $\rho = \frac{\log 1/p_1}{\log 1/p_2}$, then, with constant probability, the algorithm will report a point $v \in B(q, cR)$ if there exists a point $v^* \in B(q, R)$.

LSH family for the Hamming metric

Next, we present the LSH family $\mathcal{H}$ used for the Hamming metric.

Define an (r_1, r_2, p_1, p_2)-sensitive function $h : \Sigma^d \rightarrow \Sigma$ as $h(v) = v_i = v|_{\{i\}}$, where i is drawn uniformly at random from $\{0 \ldots d-1\}$. In other words, h is a projection along a coordinate i. Thus, a function $g = (h_1, \ldots h_k)$ is equal to $g(v) = v|_{I_i}$ where I_i is a set of size k, with each element being chosen from $\{0, \ldots d-1\}$ at random with replacement.

In our study, we will use a slight modification of the functions g. In particular, we define a function g as $g(v) = v \odot X_{I_i}$, where I_i is chosen in the same way. This modification does not affect the algorithm and its guarantees.

Note that if we set $r_1 = R$ and $r_2 = cR$, then $p_1 = 1 - R/d$ and $p_2 = 1 - cR/d$. With these settings, we obtain parameters $k = \frac{\log n}{-\log(1-cR/d)}$ and $L = O(n^{1/c})$ [15].

Chapter 2

Achieving $O(n^{1+1/c})$ space for the Hamming distance

In this Chapter, we describe in detail our basic approach for solving the (R, c)-SNN problem.

As mentioned previously, if we know the pattern size m in advance, we can construct an LSH data structure on the data set $\mathcal{P} = \{T_i^m \mid i = 0 \ldots n - m\}$ (note that the "dimension" of the points is $d = m$). If we do not know m in advance, a straight-forward approach would be to construct the above data structure for all possible $m \in \{0, \ldots M - 1\}$. However, this approach takes $\tilde{O}(n^{1+1/c} \cdot M)$ space.

To reduce the space to $O(n^{1+1/c})$, we employ the same technique, however, with a small modification. For a particular $i \in \{1 \ldots L\}$, instead of hashing strings $g_i(T_j^m)$, we store the strings $g_i(T_j^m)$ in a compressed trie. Specifically, we construct a data structure $\mathcal{D}_M$ that represents the LSH data structure on the points $\mathcal{P} = \{T_j^M, j = 0 \ldots n - 1\}$. For each $i = 1 \ldots L$, we construct a trie S_i on the strings $g_i(T_j^M)$, $j = 0, \ldots n - 1$.

Observe that now we can easily perform queries for patterns of length M as follows. First, for a given pattern $P[0 \ldots M - 1]$, and for a given $i \in \{1 \ldots L\}$, compute $g_i(P) = P \odot X_{I_i}$. Using the ordinary pattern matching in a compressed trie, search for the pattern $g_i(P)$ in the trie S_i. The search returns the set J_i of indices j corresponding to strings $g_i(T_j^M)$, such that $g_i(T_j^M) = g_i(P)$. Next, we process the strings T_j^M as we would in the standard LSH scheme: examine consecutively the strings T_j^M, $j \in J_i$, and compute the distances $D(T_j^M, P)$. If $D(T_j^M, P) \leq cR$, return j and stop. Otherwise, after we examine more than $3L$ strings T_j^M (over all $i = 1, \ldots, L$), return `NO`. The correctness of this algorithm follows directly from the correctness of the standard LSH scheme for the Hamming distance.

Next, we describe how to perform a query for a pattern P of length m, where $m \leq M$. For this case, it will be essential that we have constructed tries on the strings $g_i(T_j^M)$ (instead of hashing). Thus, for a query pattern $P[0 \ldots m-1]$, and an $i \in \{1 \ldots L\}$, perform a search of $g_i(P)$ in the trie S_i. This search will return a set J_i of positions j such that $g_i(P)$ is a prefix of $g_i(T_j^M)$. Next, we consider substrings T_j^m, that is, the substrings of T that start at the same positions $j \in J_i$, but are of length only m. We process the strings T_j^m exactly as in the standard LSH: examine all the strings T_j^m, $j \in J_i$, and compute the distances $D(T_j^m, P)$. If $D(T_j^m, P) \leq cR$, return j and stop. Otherwise, after we examine more than $3L$ strings T_j^m (over all $i = 1, \ldots, L$), return `NO`.

The correctness of this algorithm follows from the correctness of the standard LSH algorithm. The argument is simple, but somewhat delicate: we argue the correctness by showing an equivalence of our instance $\mathcal{O}$ to another problem instance. Specifically, we define a new instance $\mathcal{O}'$ of LSH obtained through the following steps:

1. Construct an LSH data structure on the strings $T_j^m \circ 0^{M-m}$ of length M, for $j = 0 \ldots n-1$;

2. Let L and k be the LSH parameters for the Hamming distance for distance R and dimension M (note that these are equal to the values L and k in the original instance $\mathcal{O}$);

3. For each $i = 1 \ldots L$, compute the strings $g_i(T_j^m \circ 0^{M-m})$, $j = 0 \ldots n-1$;

4. Perform a search query on the pattern $P \circ 0^{M-m}$;

5. For each $i = 1 \ldots L$, let J_i' be the set of all indices j such that $g_i(P \circ 0^{M-m}) = g_i(T_j^m \circ 0^{M-m})$.

In the above instance $\mathcal{O}'$, LSH guarantees to return an i such that $D(T_i^m \circ 0^{M-m}, P \circ 0^{M-m}) \leq cR$ if there exists i^* such that $D(T_{i^*}^m \circ 0^{M-m}, P \circ 0^{M-m}) \leq R$. Furthermore, if we observe that $D(T_i^m \circ 0^{M-m}, P \circ 0^{M-m}) = D(T_i^m, P)$, we can restate the above guarantee as follows: the query P in instance $\mathcal{I}'$ will return i such that $D(T_i^m, P) \leq cR$ if there exists i^* such that $D(T_{i^*}^m, P) \leq R$.

Finally, we note that $J_i' = J_i$ since the assertion that $g_i(P \circ 0^{M-m}) = g_i(T_j^m \circ 0^{M-m})$ is equivalent to the assertion that $g_i(P)$ is the prefix of $g_i(T_j^M)$. Thus, our instance $\mathcal{O}$ returns precisely the same answer as the instance $\mathcal{I}$, that is, the position i such that $D(T_i^m, P) \leq cR$ if there exists i^* such that $D(T_{i^*}^m, P) \leq R$. This is the desired answer.

A small technicality is that while searching the buckets $g_i(P)$, $i = 1, \ldots L$, we can encounter positions j where $j > n - m$ (corresponding to the the substrings T_j^m that run out of T). We eliminate these false matches using standard trie techniques: the substrings that run out of T continue with symbols that are outside of the alphabet Σ; in such case, a query will match a string T_j^M iff the $j + |P| \leq n$. Note that we can do such an update of the trie after the trie is constructed. This update will take only $O(n \log n)$ time per trie, thus not affecting the preprocessing times.

Concluding, we can use the data structure $\mathcal{D}_M$ to answer all queries of size m where $m \leq M$. The query time is $O(n^{1/c} m)$, whereas the space requirement is $O(n \cdot L) = O(n^{1+1/c})$ since a compressed trie on n strings takes $O(n)$ space. We further improve the query time in section 3.2.

2.1 Preprocessing for the Hamming distance

In this section, we analyze the preprocessing time necessary to construct the data structure $\mathcal{D}_M$. We first give a general technique that will be applied to both Hamming and l_1 metrics. Then, based on this technique, we show how to achieve the preprocessing time of $O(n^{1+1/c} M^{1/3} \log^{4/3} n)$ for the Hamming metric. Further improvements to preprocessing are presented in section 3.3.

In the preprocessing stage, we need to construct the tries S_i, for $i = 1, \ldots L$. Each trie S_i

is a compressed trie on n strings of length M. In general, constructing one trie S_i would take $O(nM)$ time, yielding a preprocessing time of $O(n^{1+1/c}M)$. We reduce this time as follows. Consider a compressed trie S_i on n strings $g_i(T_j^M)$, $j = 0, \ldots n-1$. To simplify the notation we use T_j for T_j^M. We reduce constructing the trie S_i to basically sorting the strings $g_i(T_j)$. In particular, suppose we have an oracle that can *compare* two strings $g_i(T_{j_1})$ and $g_i(T_{j_2})$ in time τ. Then, we can sort the strings $g_i(T_j)$ in time $O(\tau n \log n)$. To construct the trie, we need our comparison operation to also return the first position l at which the two strings differ. In this way, we can augment the list of sorted strings with extra information: for every two adjacent strings, we store their longest common prefix. With this information, we obtain a suffix array on strings $g_i(T_j)$, from which we can easily compute the trie S_i [20].

In conclusion, we need a comparison operation that, given two positions j_1 and j_2, will produce the first position at which the strings $g_i(T_{j_1})$ and $g_i(T_{j_2})$ differ.

In the following section we describe how to implement this operation in $\tau = O(M^{1/3} \log^{1/3} n)$ time for the Hamming metric. This directly implies a $O(n^{1+1/c} M^{1/3} \log^{4/3} n)$-time preprocessing.

2.2 $O(M^{1/3} \log^{1/3} n)$ string comparison for the Hamming distance

Consider some function g_i. Remember that $g_i(T_j) = T_j \odot X_{I_i}$, where I_i is a set with k elements, each element being chosen from the set $\{0 \ldots M-1\}$ at random with repetition. Let the number of different elements in I be k' ($k' \leq k$). Furthermore, to simplify the notation, we drop the subscripts from X_{I_i} and I_i.

We need to implement a comparison operation, that, given two positions j_1 and j_2, returns the first position at which the strings $T_{j_1} \odot X$ and $T_{j_2} \odot X$ differ. $X = X_I$ where $I = \{i_1, i_2, \ldots i_{k'}\}$, where k elements are chosen at random from $\{0 \ldots M-1\}$ with repetitions (k' being the number of different chosen elements).

To solve this problem, we give two comparison algorithms: *Comparison A* runs in time

$O(\sqrt{k'}) = O(\sqrt{k})$; and *Comparison B* runs in time $O(M/k \cdot \log n)$. We use Comparison A if $k \leq M^{2/3} \log^{2/3} n$ and Comparison B if $k > M^{2/3} \log^{2/3} n$ to obtain a maximum running time of $O(M^{1/3} \log^{1/3} n)$.

2.2.1 Comparison A

We need to compare the strings $T_{j_1} \odot X$ and $T_{j_2} \odot X$ according to positions $\{i_1, i_2, \ldots i_{k'}\}$. Assume that $i_1 < i_2 < \ldots i_{k'}$. Then, we need to find the smallest p such that $T_{j_1}[i_p] \neq T_{j_2}[i_p]$.

Partition the ordered set $I = \{i_1, \ldots i_{k'}\}$ into $\sqrt{k'}$ blocks, each of size $\sqrt{k'}$: $I = I_1 \cup I_1 \cup \ldots I_{\sqrt{k'}}$, where a block is $I_b = \{i_{b,1}, i_{b,2}, \ldots i_{b,\sqrt{k'}}\} = \{i_{\sqrt{k'}(b-1)+w} \mid w = 1 \ldots \sqrt{k'}\}$.

In this algorithm, we first find the block b at which the two strings differ, and then we find the position within the block b at which the strings differ.

Comparison A algorithm is:

1. Step 1. Find the smallest $b \in \{1 \ldots \sqrt{k'}\}$, for which strings $T_{j_1} \odot X_{I_b} \neq T_{j_2} \odot X_{I_b}$ (we elaborate on this step below);

2. Step 2. Once we find such b, iterate over positions $i_{b,1}, i_{b,2}, \ldots, i_{b,\sqrt{k'}}$ to find the smallest index w such that $T_{j_1}[i_{b,w}] \neq T_{j_2}[i_{b,w}]$. The position $i_p = i_{b,w}$ will be the smallest position where the strings $T_{j_1} \odot X$ and $T_{j_2} \odot X$ differ.

If we are able to check whether $T_{j_1} \odot X_{I_b} = T_{j_2} \odot X_{I_b}$ for any $b \in \{1 \ldots \sqrt{k'}\}$ in $O(1)$ time, then the algorithm above runs in $O(\sqrt{k'})$ time. Step one takes $O(\sqrt{k'})$ because there are at most $\sqrt{k'}$ pairs of blocks to compare. Step two takes $O(\sqrt{k'})$ because the length of any block b is $\sqrt{k'}$.

Next, we show the only remaining part: how to check $T_{j_1} \odot X_{I_b} = T_{j_2} \odot X_{I_b}$ for any $b \in \{1 \ldots \sqrt{k'}\}$ in $O(1)$ time. For this, we compute Rabin-Karp fingerprints [17] for each of the strings $T_j \odot X_{I_b}$, $b = 1 \ldots \sqrt{k'}$, $j = 0 \ldots n-1$. In particular define the fingerprint of $T_j \odot X_{I_b}$ as

$$F_b[j] = \left(\sum_{l=0}^{M-1} T[j+l] \cdot X_{I_b}[j+l] \cdot |\Sigma|^l \right) \bmod R$$

If we choose R to be a random prime of value at most $n^{O(1)}$, then $F_b[j_1] = F_b[j_2] \Leftrightarrow T_{j_1} \odot X_{I_b} = T_{j_2} \odot X_{I_b}$ for all b and all j_1, j_2 with high probability.

Thus, we want to compute the fingerprints $F_b[j]$ for all $b = 1 \dots \sqrt{k'}$ and all $j = 0 \dots n-1$. To accomplish this, we use the Fast Fourier Transform in the field $\mathbb{Z}_R$, which yields an additional time of $O(n \log M \sqrt{k'})$ for the entire fingerprinting. For a particular $b \in \{1, \dots \sqrt{k'}\}$, we compute $F_b[j]$, $j = 0 \dots n-1$, by computing the convolution of T and U, where $U[0 \dots M-1]$ is defined as $U[l] = (X_{I_b}[M-1-l] \cdot |\Sigma|^{M-1-l}) \bmod R$. If we denote with C the convolution $T * U$, then $C[j] = \left(\sum_{l=1}^{M} T[j-M+l] \cdot U[M-l]\right) \bmod R = \left(\sum_{l=0}^{M-1} T[j-(M-1)+l] \cdot X_{I_b}[l] \cdot |\Sigma|^{l}\right) \bmod R = F_b[j-(M-1)]$. Computing $T * U$ takes $O(n \log M)$ time.

Consequently, we need $O(n \log M)$ time to compute the fingerprints $F_b[j]$ for a particular b, and $O\left(n \log M \sqrt{k'}\right)$ time for all the fingerprints. This adds $O\left(n \log M \sqrt{k'}\right)$ time to the time needed for constructing a compressed trie, which is $O\left(\sqrt{k'}\right)$ time per a comparison operation. Thus, computing the fingerprints does not increase the time of constructing the desired trie.

2.2.2 Comparison B

For comparison B we will achieve $O(M/k \cdot \log n)$ time with high probability.

We rely on the fact that the positions from I are chosen at random from $\{0, \dots M-1\}$. In particular, if we find the positions $p_1 < p_2 < \dots < p_M$ at which the strings T_{j_1} and T_{j_2} differ, then, in expectation, one of the first $O(M/k)$ positions is element of the set I.

Next, we describe the algorithm more formally, and then we prove that it runs in $O(M/k \log n)$ time, w.h.p. For this algorithm, we assume that we have a suffix tree on the text T (which can be easily constructed in $O(n \log n)$ time; see, for example [10]).

Comparison B algorithm is:

1. Set $p = 0$;

2. Find the first position at which the strings T_{j_1+p} and T_{j_2+p} differ (we can find such

position in $O(1)$ time using the suffix tree on T [3]); set p equal to this position (indexed in the original T_{j_1});

3. If $p \notin I$, then loop to the step 2;

4. If $p \in I$, then return p and stop.

Now, we will show that this algorithm runs in $O(M/k \log n)$ time w.h.p. Let $p_1 < p_2 < \cdots < p_M$ be the positions at which the strings T_{j_1} and T_{j_2} differ. Let $l = 3M/k \log n$. Then, $Pr[p_1, \ldots p_l \notin I] = (1 - l/M)^k = (1 - 3 \log n/k)^k \leq \exp[-3k \log n/k] = n^{-3}$. The probability that this happens for any of the $\binom{n}{2}$ pairs T_{j_1} and T_{j_2} is at most n^{-1} by the union bound. Therefore, with probability at least $1 - n^{-1}$, comparison B will make $O(M/k \cdot \log n)$ loops, yielding the same running time, for any pair T_{j_1}, T_{j_2}.

Chapter 3

Improved query and preprocessing times

In the previous Chapter, we obtained the following bounds for our data structure: space of $O(n^{1+1/c})$; query time of $O(n^{1/c}m)$; and preprocessing time of $O(n^{1+1/c}M^{1/3}\log^{4/3} n)$. Note that, while the space bound matches the bound for the case when m is known in advance, the query and preprocessing bounds are off by, respectively, $\tilde{O}(m)$ and $\tilde{O}(M^{1/3})$ factors. In this Chapter we improve significantly these two factors.

To improve the dependence on m and M for, respectively, query and preprocessing, we redesign the LSH scheme. We show that we can use g_i functions that are not completely independent and, in fact, we reuse some of the "base" hash functions. By doing so, we are able to compute all the values $g_i(p)$ for a point p in parallel, reducing the time below the original bound of $O(n^{1/c}m)$ needed to evaluate the original functions $g_i(p)$. Using the new scheme, we achieve $\tilde{O}(n^{1/c} + mn^{o(1)})$ query time and $\tilde{O}(n^{1+1/c} + n^{1+o(1)}M^{1/3})$ preprocessing time for the Hamming distance.

We describe first how we redesign the LSH scheme. Next, we explain how we use the new scheme to achieve better query and preprocessing times.

3.1 Reusable LSH functions

As defined in section 1.2.3, the basic LSH scheme consists of L functions g_i, $i = 1 \dots L$, where $g_i = (h_{i,1}, h_{i,2}, \dots, h_{i,k})$. Each function $h_{i,j}$ is drawn randomly from the family $\mathcal{H}$, where $\mathcal{H} = \{h : \Sigma^d \to \{0,1\} \mid h(v) = v|_r, r \in \{0, \dots d-1\}\}$; in other words, $h_{i,j}$ is a projection along a randomly-chosen coordinate. The best performance is achieved for parameters $k = \frac{\log n}{\log 1/p_2}$ and $L = n^\rho = O(n^{1/c})$. Recall that $p_1 = 1 - \frac{R}{d}$, $p_2 = 1 - \frac{cR}{d}$, and $\rho = \frac{\log 1/p_1}{\log 1/p_2}$.

We redesign this LSH scheme as follows. Let t be an integer (specified later), and let $w = n^{\rho/t}$. Define functions u_j, for $j = 1 \dots w$, as $u_j \in \mathcal{H}^{k/t}$; each u_j is drawn uniformly at random from $\mathcal{H}^{k/t}$. Furthermore, redefine the functions g_i as being t-tuples of distinct functions u_j; namely, $g_i = (u_{j_1}, u_{j_2}, \dots u_{j_t}) \in \mathcal{H}^k$ where $1 \le j_1 < j_2 < \dots < j_t \le w$. Note that there are in total $L = \binom{w}{t}$ functions g_i. The rest of the scheme is exactly as before.

Now, we need to verify that the query time of the redesigned LSH is close to the original bound. To this end, we have to show that there are not too many collisions with points at distance $\ge cR$. We need also to show correctness, i.e., that the algorithm has a constant probability of reporting a point within distance cR, if such a point exists.

We can bound the number of collisions with points at distance $\ge cR$ by ensuring that the number of false positives is $O(L)$, which is achieved by choosing k as before $k = \frac{\log n}{\log 1/p_2}$. Since each particular g_i is indistinguishable from uniformly random on $\mathcal{H}^k$, the original analysis applies here as well.

A harder task is estimating the probability of the failure of the new scheme. A query fails when there is a point p as distance R from the query point q, but $g_i(p) \neq g_i(q)$ for all i. The probability of this event is exactly the probability that p and q collide on no more than $t-1$ functions u_j. Thus,

$$\begin{aligned}
Pr[\text{fail}] &= \textstyle\sum_{i=0}^{t-1} \binom{w}{i} p_1^{i \cdot k/t} (1 - p_1^{k/t})^{w-i} = \sum_{i=0}^{t-1} \binom{w}{i} w^{-i} (1 - w^{-1})^{w-i} \le e^{-w^{-1}w} \sum_{i=0}^{t-1} \binom{w}{i} w^{-i} (1 - w^{-1})^{-i} \\
&= \tfrac{1}{e} \left(1 + ww^{-1}(1 - w^{-1})^{-1} + \tfrac{w(w-1)}{2} w^{-2} (1 - w^{-1})^{-2} + \textstyle\sum_{i=3}^{t-1} \binom{w}{i} w^{-i} (1 - w^{-1})^{-i}\right) \\
&\le \tfrac{1}{e} \left(1 + (1 + \tfrac{1}{w-1}) + \tfrac{1}{2}(1 + \tfrac{1}{w-1}) + \textstyle\sum_{i=3}^{t-1} \frac{w^i}{i!} w^{-i}\right) = \tfrac{1}{e} \left(1 + 1 + 1/2 + \tfrac{3/2}{w-1} + \textstyle\sum_{i=3}^{t-1} 1/i!\right) \\
&= \tfrac{1}{e} \left(\textstyle\sum_{i=0}^{t-1} 1/i! + \Theta(n^{-\rho/t})\right) \le \tfrac{1}{e} \left(e - 1/t! + \Theta(n^{-\rho/t})\right) \\
&\le 1 + \Theta(n^{-\rho/t}) - \Theta(1/t!)
\end{aligned}$$

If we set $t = \sqrt{\frac{\rho \log n}{\ln \log n}}$, we obtain that

$$Pr[\text{fail}] \;\leq\; 1 + \Theta(e^{-\sqrt{\rho \log n \ln \log n}}) - \Theta(e^{-t \ln t + t}/\sqrt{t}) = 1 - \Theta(e^{-t \ln t + t}/\sqrt{t}) = 1 - \Theta(1/t!)$$

To reduce this probability to a constant less than 1, it is enough to repeat the entire structure $U = \Theta(t!) = O(e^{\sqrt{\rho \log n \ln \log n}})$ times, using independent random bits. Thus, while one data structure has $L = \binom{w}{t} \leq n^{\rho}/t!$ functions g_i, all U structures have $U \cdot L = O\left(t! n^{\rho}/t!\right) = O(n^{\rho})$ functions g_i that are encoded by only $w \cdot U = O\left(\exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right) = n^{o(1)}$ independently chosen functions $u \in \mathcal{H}^{k/t}$.

The query time is still $O(n^{1/c}m)$ since we have $O(n^{1/c})$ functions g_i, as well as an expected of $O(LU) = O(n^{\rho}) = O(n^{1/c})$ collisions with the non-matching points in the LSH buckets.

This redesigned LSH scheme can be employed to achieve better query and preprocessing times as will be shown in the following sections. As will become clear later, the core of the improvement consists in the fact that there are only $O\left(\exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$ independently chosen functions $u \in \mathcal{H}^{k/t}$, and the main functions g_i are merely t-tuples of the functions u.

3.2 Query time of $\tilde{O}\left(n^{1/c} + mn^{o(1)}\right)$ for the Hamming distance

To improve the query time, we identify and improve the bottlenecks existing in the current approach to performing a query (specifically, the query algorithm from Chapter 2). In the current algorithm, we have a trie S_i per each function g_i. Then, for a query P, we search the string $g_i(P)$ in S_i (again, as mentioned in section 1.2.3, $g_i(\cdot)$ represents in fact a bit-mask, and the bits outside the boundaries of P are discarded).

We examine the bottlenecks in this approach and how we can eliminate them using the new LSH scheme. Consider a query on string P. We have in total $LU = O(n^{1/c})$ functions g_i (over all U data structures), each sampling at most $k = O(M)$ bits. The query time can be decomposed into two terms:

$\mathcal{T}_s$: Time spent on computing functions $g_i(P)$ and searching $g_i(P)$ in the trie S_i, for each

i;

$\mathcal{T}_c$: Time spent on examining the points found in the bucket $g_i(P)$ (computing the distances to substrings colliding with P to decide when one is a cR-NN).

Both $\mathcal{T}_s$ and $\mathcal{T}_c$ are potentially $O(n^{1/c}m)$. We show how to reduce $\mathcal{T}_s$ to $\tilde{O}\left(n^{1/c} + m \cdot \exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$ and $\mathcal{T}_c$ to $\tilde{O}(n^{1/c} + m)$. We analyze $\mathcal{T}_c$ first.

Lemma 3.2.1. *It is possible to preprocess the text T such that, for a query string P of length m, after $O(m \log n/\epsilon^2)$ processing of P, one can test whether $|P - T_j^m|_H \leq R$ or $|P - T_j^m|_H \geq cR$ for any j in $O(\log^2 n/\epsilon^2)$ time (assuming that one of these cases holds). Using this test, there is an algorithm achieving $\mathcal{T}_c = O(n^{1/c} \log^2 n/\epsilon^2 + m \log n/\epsilon^2)$. The preprocessing of T can be accomplished in $O(n \log^3 n/\epsilon^2)$ time.*

Proof. We can approximate the distance $|P - T_j^m|_H$ by using the sketching technique of [19] (after a corresponding preprocessing). Assume for the beginning that the query P has length $m = M$. In the initial preprocessing of T, we compute the sketches $sk(T_j^M)$ for all j. Next, for a query P, we compute the sketch of P, $sk(P)$; and, from the sketches $sk(P)$ and $sk(T_j^M)$, we can approximate the distance $|T_j^M - P|_H$ with error c with probability $1 - n^{-O(1)}$ in $O(\log n/\epsilon^2)$ time (for details, see [19], especially lemma 2 and section 4). If the test reports that a point T_j^M is at a distance $\leq cR$ (i.e., the test does not classify the point as being at a distance $> cR$), then we stop LSH and return this point as the result (note that there is only a small probability, $n^{-O(1)}$, that the test reports that a point T_j^M, with $|P - T_j i^M|_H \leq R$, is at a distance $> cR$).

If P is of length $m < M$, then we compute the sketches $sk(P)$ and $sk(T_j^m)$ from $O(\log n)$ sketches of smaller lengths. Specifically, we divide the string P into $O(\log m)$ diadic intervals, compute the sketches for each diadic interval, and finally add sketches (modulo 2) to obtain the sketch for the entire P. Similarly, to obtain $sk(T_j^m)$, we precompute the sketches of all substrings of T of length a power of two (in the T-preprocessing stage); and, for a query P, we add the $O(\log m)$ sketches for the diadic intervals of T_j^m to obtain the sketch of T_j^m. Thus, computation of the sketch of T_j^m takes $O(\log^2 n/\epsilon^2)$ time. Precomputing the sketch of P takes $O(m \log n/\epsilon^2)$ time. With these two times, we conclude that $T_c = O(n^{1/c} \log^2 n/\epsilon^2 + m \log n/\epsilon^2)$. To precompute all the sketches of all the substrings of T of length a power of two, we need $O(n \log^2 M \log n/\epsilon^2)$ time by applying FFT along the lines of [14] or section 2.2.1 (since a sketch is just a tuple of dot products of the vector with a random vector). □

Next, we show how to improve $\mathcal{T}_s$, the time for searching $g_i(P)$ in the corresponding tries.

Lemma 3.2.2. *Using the new LSH scheme, it is possible to match $g_i(P)$ in the tries S_i, for all i's, in $\mathcal{T}_s = O\left(n^{1/c} \log^{3/2} n + m \cdot \exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$ time.*

Proof. To achieve the stated time, we augment each trie S_i with some additional information that enables a faster traversal of the trie using specific fingerprints of the searched string (i.e., $g_i(P)$); the new LSH helps us in computing these fingerprints for $g_i(P)$ for all tries S_i in parallel. For ease of notation, we drop the subscript i from g_i and S_i. Recall that S is a trie on strings $g(T_j^M)$, $j = 1 \dots n-1$; we will drop the superscript M for T_j^M as well.

We augment the trie S with additional $\log(M)$ tries $S^{(l)}$, $l = 0 \dots \log M - 1$. For each l, let $f_l : \Sigma^{2^l} \to \{0, \dots n^{O(1)}\}$ be a fingerprint function on strings of length 2^l. The trie $S^{(l)}$ is a trie on the following n strings: for $j \in \{0 \dots n-1\}$, take the string $g(T_j)$, break up $g(T_j)$ into $M/2^l$ blocks each of length 2^l, and apply to each block the fingerprint function f_l; thus, the resulting string is

$$F_j^{(l)} = \left\langle f_l\left(g(T_j)[0 : 2^l - 1]\right), f_l\left(g(T_j)[2^l : 2 \cdot 2^l - 1]\right) \dots f_l\left(g(T_j)[M - 2^l : M - 1]\right)\right\rangle$$

Note that, in particular, $S = S^{(0)}$.

Further, for each trie $S^{(l+1)}$ and each node $N_{l+1} \in S^{(l+1)}$, we add a *refinement link* pointing to a node N_l in $S^{(l)}$, the node which we call the *equivalent* node of N_{l+1}. By definition, the equivalent node of N_{l+1} is the node N_l of $S^{(l)}$ that contains in its subtree exactly the same leaves as the subtree of N_{l+1} (a leaf in a trie is a substring T_j). Note that such node N_l always exists. Furthermore, if $str(N_{l+1})$ denotes the substring of T corresponding to the path from the root to N_{l+1}, then $str(N_{l+1}) = str(N_l)$ or $str(N_{l+1})$ is a prefix of $str(N_l)$.

Using l tries $S^{(0)}, \ldots S^{(\log M-1)}$ and the refinement links, we can speed up searching of a string in the trie S by using initially "rougher" tries (with higher l) for rough matching of $g(P)$, and gradually switching to "finer" tries (with smaller l) for finer matching. Specifically, for a string $G = g(P)$, we break up G into diadic substrings $G = G_{l_1} G_{l_2} \ldots G_{l_r}$, where G_{l_i} has length 2^{l_i}, and l_i's are strictly decreasing ($l_i > l_{i-1}$). Then, we match G in S as follows. In the trie $S^{(l_1)}$, follow the edge corresponding to the symbol $f_{l_1}(G_{l_1})$. Next, follow sequentially the refinement links into the tries $S^{(l_1-1)} \ldots S^{(l_2)}$. In $S^{(l_2)}$, follow the edge corresponding to $f_{l_2}(G_{l_2})$ (unless we already jumped this block while following the refinement links). Continue this procedure until we finish it in the trie G_{l_r}, where the final node gives all the matching substrings $g(T_j)$. If, at any moment, one of the traversed trie edges is longer than one fingerprint symbol or a refinement link increased $str(N_l)$ of current node to $str(N_l) \geq |G|$, then we stop as well (since, at this moment, we matched all $|G|$ positions of G, and the current node yields all the matches). Note that, if we know all the fingerprints $f_l(G_l)$, then we can match $g_i(P)$ in the trie S_i in time $O(\log n)$.

The remaining question is how to compute the fingerprints $f_{l_1}(G_{l_1}), f_{l_2}(G_{l_2}), \ldots f_{l_r}(G_{l_r})$ (for each of the UL functions g_i). We will show that we can compute the fingerprints for one of the U independent sets of g_i's in $\tilde{O}\left(L + m \cdot \exp\left[\sqrt{\rho \log n \ln \log n}\right]\right)$; this gives a total time of $\tilde{O}\left(UL + U \cdot m \cdot \exp\left[\sqrt{\rho \log n \ln \log n}\right]\right) = \tilde{O}\left(n^{1/c} + m \cdot \exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$. To this purpose, consider one of the U independent data structures, and some diadic substring $G_{l_j} = G[a:b]$, with a corresponding fingerprinting function $f = f_{l_j}$, for which we want to compute the fingerprints $f_{l_j}(G_{l_j}) = f_{l_j}(g_i(P)[a:b]) = f(g_i(P)[a:b])$ for all L functions g_i in the considered independent data structure. Remember that each of the L functions g_i is defined as a t-tuple of functions $u_{h_1}, \ldots u_{h_t}$, $1 \leq h_1 < h_2 < \cdots < h_t \leq w$.

For computing the fingerprints $f(g_i(P)[a:b])$, for all g_i, we rely on the following idea: we first compute similar fingerprints for all the functions u_h, $1 \leq h \leq w$, and then combine them to obtain the fingerprints for the functions g_i. To be able to combine easily the fingerprints of the functions u_h, we use the fingerprinting function of Rabin-Karp [17], which was already used in section 2.2.1. With this fingerprinting function, the fingerprint for g_i is just the sum modulo R of the fingerprints for the functions $u_{h_1}, u_{h_2}, \ldots u_{h_t}$ (remember that R is a random prime for the fingerprinting function). Specifically,

$$f\left(g_i(P)[a:b]\right) = \left(\sum_{x=1}^{t} f\left(u_{h_x}(P)[a:b]\right)\right) \pmod{R} \tag{3.1}$$

A technicality is that a particular position in P can be sampled in several u_h's, thus contributing multiple times the same term to the above sum. However, this technicality is easily dealt with if we use $t|\Sigma| < O(\log n \cdot |\Sigma|)$ as the base in the fingerprinting function (instead of $|\Sigma|$ as was used in section 2.2.1). With this new base, the "oversampled" position will contribute in exactly the same way to the fingerprint of $f(g_i(P)[a:b])$ as well as to the fingerprints of the strings in the trie.

Finally, we can conclude that $\mathcal{T}_s = O\left(n^{1/c}\log^{3/2} n + m \cdot \exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$. First, for computing the fingerprints for all substrings $G_{l_1}, \ldots, G_{l_r}$, for all functions u_h, $h = 1 \ldots w$, we need only $O\left(w \sum_{j=1}^{r} l_j\right) = O(mw) = O\left(m \cdot \exp\left[\sqrt{\rho \log n \ln \log n}\right]\right)$ time. For all U independent data structures, this takes $O(mwU) = O\left(m \cdot \exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$ time. Once we have the fingerprints for the functions u_h, we can combine them to get all the fingerprints for all the functions g_i; this takes a total of $O(\log^{1/2} n \cdot LU \cdot \log n) = O(n^{1/c}\log^{3/2} n)$ time (because we need only $O(t) < O(\log^{1/2} n)$ time for computing a fingerprint for one function g_i once we have the fingerprints of the corresponding t functions u_h). $\square$

3.3 Preprocessing time of $\tilde{O}(n^{1+1/c} + n^{1+o(1)}M^{1/3})$ for the Hamming distance

We will show that to carry out the necessary preprocessing, we need $\tilde{O}\left(n^{1+1/c} + nM^{1/3}e^{2\sqrt{\rho \log n \ln \log n}}\right)$ time. As in section 2.1, the bottleneck of the preprocessing stage is constructing the tries S_i. Furthermore, the trie augmentation from the previous section requires constructing the tries $S_i^1, \ldots S_i^{(\log M - 1)}$. We will first explain how to construct the tries $S_i = S_i^0$ in time $\tilde{O}\left(n^{1+1/c} + nM^{1/3}\exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$. Then, we present how we construct the tries

$S_i^1, \ldots S_i^{(\log M-1)}$ from the trie $S_i^{(0)}$, for all i, in $\tilde{O}(n^{1+1/c})$ time.

Lemma 3.3.1. *We can construct the tries $S_i = S_i^{(0)}$ for all $U \cdot L$ functions g_i in time* $O\left(n^{1+1/c}\log^{3/2} n + nM^{1/3}\exp\left[2\sqrt{\rho \log n \ln\log n}\right]\log^{4/3} n\right)$.

Proof. We again use the inter-dependence of the LSH functions in the redesigned scheme. Consider one of the U independent data structures. In the considered independent data structure, we have w functions u_h; the functions g_i are defined as t-tuples of the functions u_h. Thus, we can first construct w tries corresponding to the functions u_h, $1 \le h \le w$ (i.e., a trie for the function u_h is the trie on the strings $u_h(T_j^M)$); and, from these, we can construct the tries for the functions g_i. For constructing the tries for the functions u_h, we can use the algorithm from section 2.1, which will take $O(w \cdot nM^{1/3}\log^{4/3} n)$ total time since there are w functions u_h.

Once we have the w tries corresponding to the functions u_h, $1 \le h \le w$, we can construct the tries for the functions g_i in $O(Ln\log^{3/2} n)$ time. Recall from section 2.1 that if, for two substrings $T_{j_1}^M$ and $T_{j_2}^M$, in time τ, we can find the first position where $g_i(T_{j_1}^M)$ and $g_i(T_{j_2}^M)$ differ, then we can sort and ultimately construct the trie on the strings $g_i(T_j^M)$, for each i, in $O(\tau Ln\log n)$ time. Indeed, at this moment, it is straight-forward to find the first position where $g_i(T_{j_1}^M)$ and $g_i(T_{j_2}^M)$ differ: this position is the first position where $u_h(T_{j_1}^M)$ and $u_h(T_{j_2}^M)$ differ, for one of the t function u_h that define the function g_i. Thus, for two substrings $T_{j_1}^M$ and $T_{j_2}^M$, and some function g_i, we can find the first position where $u_h(T_{j_1}^M)$ and $u_h(T_{j_2}^M)$ differ for all t functions defining g_i (using the tries for the functions u_h); the smallest of these positions is the position of the first difference of $g_i(T_{j_1}^M)$ and $g_i(T_{j_2}^M)$. Now, we can conclude that one "comparison" operation takes $\tau = O(t) = O(\sqrt{\log n})$ time (again, finding the first difference of two strings in a u_h's trie can be done in $O(1)$ time [3]). Since there is a total of L functions g_i (in one of the U independent data structures), the total time for constructing the tries for g_i's is $O(\tau Ln\log n) = O(Ln\log^{3/2} n)$.

Summing up the times for constructing the u_h tries and then the g_i tries, we get $O(wnM^{1/3}\log^{4/3} n + Ln\log^{3/2} n) = O\left(Ln\log^{3/2} + nM^{1/3}\exp\left[\sqrt{\rho\log n\ln\log n}\right]\log^{4/3} n\right)$.

For all U independent data structures, the total time for constructing all $S_i^{(0)}$ is

$U \cdot O\left(wnM^{1/3}\log^{4/3} n + Ln\log^{3/2} n\right) = O\left(ULn\log^{3/2} n + nM^{1/3}wU\log^{4/3} n\right) =$
$O\left(n^{1+1/c}\log^{3/2} n + nM^{1/3}\exp\left[2\sqrt{\rho\log n\ln\log n}\right]\log^{4/3} n\right)$. This time dominates the preprocessing time. $\square$

Next, we show how to construct the tries $S_i^1, \ldots S_i^{(\log M-1)}$ from a trie $S_i = S_i^{(0)}$. We will construct the trie $S_i^{(l)}$ for some given i and l as follows. Recall that the trie $S_i^{(l)}$ contains the strings $F_j^{(l)} = \left\langle f_l\left(g_i(T_j^M)[0 : 2^l - 1]\right), f_l\left(g_i(T_j^M)[2^l : 2 \cdot 2^l - 1]\right) \ldots f_l\left(g_i(T_j^M)[M - 2^l : M - 1]\right)\right\rangle$ (i.e., the string obtained by fingerprinting 2^l-length blocks of $g_i(T_j^M)$). As for the tries S_i, we need to find the sorted list of leaves $F_j^{(l)}$, as well as the position of the first difference of each two consecutive $F_j^{(l)}$ in the sorted list. With this information, it is easy to construct the trie $S_i^{(l)}$ [20].

Finding the sorted order of leaves $F_j^{(l)}$ is straight-forward: the order is exactly the same as the order of the leaves $g_i(T_j^M)$ in the trie S_i. Similarly, in constant time, we can find the position where two consecutive $F_{j_1}^{(l)}$ and $F_{j_2}^{(l)}$ differ for the first time. If p is the position where $g_i(T_{j_1}^M)$ and $g_i(T_{j_2}^M)$ differ for the first time, then $F_{j_1}^{(l)}$ and $F_{j_2}^{(l)}$ differ for the first time at the position $\lfloor p/2^l \rfloor$. Thus, constructing one trie $S_i^{(l)}$ takes $O(n)$ time. For all $\log n$ fingerprint sizes l and for all LU functions g_i, this takes time $O(\log n \cdot ULn) = O(n^{1+1/c}\log n)$.

Chapter 4

Extension to the l_1 distance

In this Chapter, we extend our results to the l_1 distance, for which we assume that the distance between two strings $A, B \in \Sigma^m$ is $D(A, B) = \sum_{i=0}^{m-1} |A[i] - B[i]|$.

For l_1 metric, we achieve space of $\tilde{O}(n^{1+1/c})$ and query time of $\tilde{O}(n^{1/c} + mn^{o(1)})$, as in the case for the Hamming metric (up to logarithmic factors). Preprocessing time is $\tilde{O}(n^{1+1/c} + n^{1+o(1)} M^{2/3})$, which, for $c < 3/2$, matches the preprocessing time of $\tilde{O}(n^{1+1/c})$ for the case then query length m is fixed in advance (up to logarithmic factors).

To achieve the stated bounds, we exhibit first an LSH family of hash functions for l_1. Then, using this LSH family, we show how we can obtain $\tilde{O}(n^{1+1/c})$ space and $\tilde{O}(n^{1/c} m)$ query time. In section 4.3, we discuss the basic approach to preprocessing. The final query and preprocessing algorithms are presented in section 4.4, where we improve query time to $\tilde{O}(n^{1/c} + mn^{o(1)})$ and preprocessing time to $\tilde{O}(n^{1+1/c} + n^{1+o(1)} M^{2/3})$, by using the redesigned LSH scheme from section 3.1.

4.1 An LSH family for l_1

In this section, we describe an LSH family for l_1 distance. Let $t = \alpha R$, where $\alpha > 1$ will be chosen later. We define an LSH function h_u for a pattern P of length M as follows:

$$h_u(P) = \left(\left\lfloor \frac{P[0] + d_{u,0}}{t} \right\rfloor, \left\lfloor \frac{P[1] + d_{u,1}}{t} \right\rfloor, \ldots \left\lfloor \frac{P[M-1] + d_{u,M-1}}{t} \right\rfloor \right) \tag{4.1}$$

where $d_{u,j}$ is drawn uniformly at random from $\{0, \ldots t-1\}$ for $0 \leq j < M$.

The following lemma proves that the resulting function is locality-sensitive and yields an efficient LSH family.

Lemma 4.1.1. *The function h_u is locality sensitive with parameters $r_1 = R$, $r_2 = cR$, $p_1 = 1 - \frac{1}{\alpha}$, and $p_2 = 1 - \frac{c/\alpha}{1+c/\alpha}$. Moreover, if we set $\alpha = \log n$, we obtain an LSH scheme with parameters $k = O(\log^2 n)$, and $L = n^\rho = O(n^{1/c})$.*

Proof. First, we prove that for two strings A and B with $D(A, B) \leq R$, we have that $Pr[h_u(A) = h_u(B)] \geq 1 - 1/\alpha = p_1$. Let $\delta_i = |A[i] - B[i]|$ for $0 \leq i < M$; then $\sum \delta_i \leq R$. Thus,

$$Pr[h_u(A) = h_u(B)] = (1 - \delta_0/t) \cdot (1 - \delta_1/t) \cdot \ldots (1 - \delta_{M-1}/t) \geq 1 - R/t = 1 - 1/\alpha$$

Next, we prove that for two strings A and B with $D(A, B) \geq cR$, we have that $Pr[h_u(A) = h_u(B)] \leq 1 - \frac{c/\alpha}{1+c/\alpha} = p_2$. Let $\delta_i = |A[i] - B[i]|$ for $0 \leq i < M$; then $\sum \delta_i \geq cR$. Thus,

$$\begin{aligned} Pr[h_u(A) = h_u(B)] &= (1 - \delta_0/t) \cdot (1 - \delta_1/t) \cdot \ldots (1 - \delta_{M-1}/t) \leq (1 - cR/tM)^M \\ &\leq e^{-cM/\alpha M} = e^{-c/\alpha} \leq \frac{1}{1+c/\alpha} = 1 - \frac{c/\alpha}{1+c/\alpha} = 1 - \frac{c}{1+c/\alpha} \cdot \frac{1}{\alpha} \end{aligned}$$

We choose the standard $k = \log_{1/p_2} n$ and $L = n^\rho$, $\rho = \frac{\log 1/p_1}{\log 1/p_2}$. Note that we obtain $L = n^\rho \leq O(n^{1/c'})$, where $c' = c\frac{1}{1+c/\alpha}$, with the same analysis as in [15]. Thus, for $\alpha = \log n$, we have $n^\rho \leq O\left(n^{\frac{1}{c}(1+c/\alpha)}\right) \leq O\left(n^{1/c + 1/\log n}\right) = O(n^{1/c})$. Finally, we obtain $k = O(\log^2 n)$ and $L = O(n^{1/c})$. □

4.2 Achieving $\tilde{O}(n^{1+1/c})$ space for l_1

To achieve $\tilde{O}(n^{1+1/c})$ space for l_1, we deploy the same technique as we did for the Hamming metric in Chapter 2. The only real difference is how, for a particular function g_i, $1 \leq i \leq L$, we represent a compressed trie similar to the one in the Hamming metric case. To accomplish this, we define g_i as a tuple of k functions h_u, $u = 0 \dots k-1$, much as in LSH but with the k functions h_u intertwined. In particular, for a pattern P of length $|P| = M$, define $g_i(P)$ as a kM-tuple, such that $g_i(P)[lk+u] = \left\lfloor \frac{P[l]+d_{u,l}}{t} \right\rfloor$ where we choose $d_{u,l}$ uniformly at random from $\{0, \dots t-1\}$ for all $0 \le u < k$ and $0 \le l < M$. As we did previously, if the pattern P is shorter than M, we cut off $g_i(P)$ at the corresponding place.

We construct a trie S_i on the strings $g_i(T_j^M)$, $0 \le j < n$, as in the case of the Hamming distance. A query is performed exactly as described in the Chapter 2 with $g_i(P)$ being trimmed after km elements ($m = |P|$). Query time is $O(Lkm) = O(n^{1/c} m \log^2 n)$; this time can be further improved to $\tilde{O}(n^{1/c} + mn^{o(1)})$ using the redesigned LSH scheme from section 3.1 as is described in section 4.4.1. Space requirement is $O(n^{1+1/c})$ since, as before, a compressed trie on n strings takes only $O(n)$ space.

In the following section, we describe how we construct the tries S_i on the strings $g_i(T_j^M)$, $0 \le j < n$.

4.3 Preprocessing time of $\tilde{O}(n^{1+1/c} M^{2/3})$ for l_1

In this section, we show the basic approach to preprocessing for l_1, achieving $\tilde{O}\left(n^{1+1/c} M^{2/3}\right)$ time. The final preprocessing algorithm is described in section 4.4.2, where we achieve $\tilde{O}\left(n^{1+1/c} + nMn^{o(1)}\right)$ time.

We show how to construct a trie S_i in time $\tilde{O}\left(nM^{2/3}\right)$, and, since there $L = O(n^{1/c})$ tries to construct, we obtain the stated bound. As in section 2.1, we construct a trie S_i by sorting the strings $g_i(T_j^M)$. Once again, we implement the following comparison operation: given two positions j_1 and j_2, produce the first position at which the strings $g_i(T_{j_1}^M)$ and $g_i(T_{j_2}^M)$ differ. To simplify the exposition, we duplicate k times each character of T to obtain T'; thus, the length of T' is $nk = O(n \log^2 n)$. Now, $g_i(T'_j)[l]$ is defined simply as $\left\lfloor \frac{T'_j[l]+d_l}{t} \right\rfloor$, where we choose d_l uniformly at random from $\{0, \dots t-1\}$ for $0 \le l < kM$. We can do similar transformation on the pattern P when we search $g_i(P)$ in the trie. Let $n' = nk$ and $M' = Mk$. As before, we will refer to $T_j'^{M'}$ as T'_j.

The algorithm for the comparison is practically the same with the Comparison A from section 2.2.1. We divide each pattern T'_j into $B = M'^{1/3}$ blocks each of size $W = M'^{2/3}$ (the asymmetry stems from the fact that it will be harder to compute the fingerprints).

To check the blocks for equality fast, we use fingerprints. Call $F_b[j]$, $0 \le b < B$, $0 \le j < n'$, a fingerprint of the block $g_i(T'_j)[Wb \dots Wb+W-1]$. Formally, our fingerprints satisfy the following property: for two positions j_1, j_2, $g_i(T'_{j_1})[Wb \dots Wb+W-1] = g_i(T'_{j_2})[Wb \dots Wb+W-1]$ iff $F_b[j_1] = F_b[j_2]$ (w.h.p.).

If we can compute such fingerprints $F_b[j]$ in time $\tilde{O}(n'M'^{2/3})$, then we have a $\tilde{O}(n'M'^{2/3})$-time algorithm for constructing the compressed trie. We can implement the desired comparison operation in $\tilde{O}(M^{2/3})$ time by first finding the block b for which blocks $g_i(T'_{j_1})[Wb \dots Wb+W-1]$ and $g_i(T'_{j_2})[Wb \dots Wb+W-1]$ differ (using the fingerprints $F_b[j_1]$ and $F_b[j_2]$), and then finding the position of the first difference within the block b by scanning the block. Fingerprinting takes $\tilde{O}(n'M'^{2/3})$ and sorting takes $O(n \log n M'^{2/3})$ time. Total time would be, therefore, $\tilde{O}(n'M'^{2/3})$.

Next, we show how we can compute fingerprints $F_b[j]$ for a fixed block b and all $0 \le j < n'$, in time $\tilde{O}(n'M'^{1/3})$. Since there are $B = M'^{1/3}$ blocks in total, this immediately yields a $\tilde{O}(n'M'^{2/3})$-time algorithm for computing all the fingerprints $F_b[j]$, $0 \le b < B$, $0 \le j < n'$.

Lemma 4.3.1. *We can compute fingerprints $F_b[j]$ for a fixed block b and all $0 \le j < n'$, in time $\tilde{O}(n'M'^{1/3})$. Additionally, each fingerprint $F_b[j]$ has a description length of only $\kappa = O(\log n)$.*

Proof. Define $F_b[j]$ as a tuple of $\kappa = O(\log n)$ fingerprints $F_b^{(f)}[j]$, $1 \le f \le \kappa$, where $F_b^{(f)}[j]$ is defined as:

$$F_b^{(f)}[j] = \sum_{l=0}^{W-1} \beta_l^{(f)} \cdot g_i(T'_j)[Wb + l]$$

where, $0 \le l < W$, $1 \le f \le \kappa$, and $\beta_l^{(f)}$ is drawn uniformly at random from $\{0, 1\}$.

We prove that these fingerprints are correct and then we show how we compute them in $\tilde{O}(n'M'^{1/3})$ time.

First, note that, for two positions $j_1 \neq j_2$, if $g_i(T'_{j_1})[Wb \ldots Wb+W-1] \neq g_i(T'_{j_2})[Wb \ldots Wb+W-1]$, then $F_b^{(f)}[j_1] = F_b^{(f)}[j_2]$ with probability at most $1/2$. Therefore, if $g_i(T'_{j_1})[Wb \ldots Wb+W-1] \neq g_i(T'_{j_2})[Wb \ldots Wb+W-1]$, then $F_b[j_1] \neq F_b[j_2]$ with high probability.

Next, we show how to compute these fingerprints in $\tilde{O}(n'M'^{1/3})$ time. Specifically, we need to show that we can compute fingerprints $F_b^{(f)}$ for a fixed f in $\tilde{O}(n'M'^{1/3})$ time.

Fix some value f. We will seek to simplify the expression for $g_i(T'_j)[Wb+l] = \left\lfloor \frac{T'_j[Wb+l]+d_{Wb+l}}{t} \right\rfloor$. Define $x_j = \left\lfloor \frac{T'[j]}{t} \right\rfloor$, and $y_j = T'_j - x_j t$. Then $g_i(T'_j)[Wb+l] = x_{j+Wb+l} + \left\lfloor \frac{y_{j+Wb+l}+d_{Wb+l}}{t} \right\rfloor = x_{j+Wb+l} + \left\lfloor \frac{t+y_{j+Wb+l}-\theta_{Wb+l}}{t} \right\rfloor$, where $\theta_l = t - d_l$, $1 \leq \theta_l \leq t$. Let $\chi_{j,l}$ be 1 if $y_{j+Wb+l} \geq \theta_{Wb+l}$ and 0 if $y_{j+Wb+l} < \theta_{Wb+l}$. With all these notations, we can write $g_i(T'_j)[Wb+l]$ as $g_i(T'_j)[Wb+l] = x_{j+Wb+l} + \chi_{j,l}$.

We can compute $F_b^{(f)}$ as

$$F_b^{(f)}[j] = \sum_{l=0}^{W-1} \beta_l^{(f)} \cdot g_i(T'_j)[Wb+l] = \sum_{l=0}^{W-1} \beta_l^{(f)} \cdot (x_{j+Wb+l} + \chi_{j,l}) = \sigma_j^{(1)} + \sigma_j^{(2)}$$

where $\sigma_j^{(1)} = \sum_{l=0}^{W-1} \beta_l^{(f)} x_{j+Wb+l}$ and $\sigma_j^{(2)} = \sum_{l=0}^{W-1} \beta_l^{(f)} \chi_{j,l}$.

We can compute $\sigma_j^{(1)}$ for all $0 \leq j < n'$ via FFT as in section 2.2.1 in time $O(n' \log M')$.

To compute $\sigma_j^{(2)}$, we use the *less-than matching problem* [1]. Call $LT[j]$ the number of positions l in which $y_{j+Wb+l} \geq \theta_{Wb+l}^{(f)}$, where we define $\theta_l^{(f)}$ as: $\theta_{Wb+l}^{(f)} = \theta_{Wb+l}$ if $\beta_l^{(f)} = 1$; and $\theta_{Wb+l}^{(f)} = t$ if $\beta_l^{(f)} = 0$. Then $\sigma_j^{(2)} = LT[j]$. But $LT[j]$ is precisely the array computed by the less-than matching problem [1] (shifted by Wb positions). Thus, we can compute the array $\sigma_j^{(2)} = L[j]$ in $O(n'W^{1/2} \log^{1/2} W)$ time. □

To conclude, we can compute F_b for a fixed b, in $O(n'W^{1/2} \log^{1/2} W \cdot \kappa) = O(nM^{1/3} \log^{4+1/6} n)$ time. To compute F_b for all b, we need $O(nM^{2/3} \log^{4+5/6} n)$. This time overweights the time for sorting. Total preprocessing time is $L \cdot O(nM^{2/3} \log^{4+5/6} n) = O(n^{1+1/c} M^{2/3} \log^{4+5/6} n)$.

4.4 Improved query and preprocessing times for l_1

Using the redesigned LSH described in section 3.1, we can reduce query time to $\tilde{O}\left(n^{1/c} + me^{2\sqrt{\rho \log n \ln \log n}}\right)$ and preprocessing time to $\tilde{O}\left(n^{1+1/c} + nM^{2/3}e^{2\sqrt{\rho \log n \ln \log n}}\right)$. The ideas are similar to those from Chapter 3.

4.4.1 Query time of $\tilde{O}(n^{1/c} + mn^{o(1)})$

As in section 3.2, query time can be decomposed into two terms:

$\mathcal{T}_s$: Time spent on computing functions $g_i(P)$ and searching $g_i(P)$ in the trie S_i, for each i;

$\mathcal{T}_c$: Time spent on examining the points found in the bucket $g_i(P)$ (computing the distances to substrings colliding with P to decide when one is a cR-NN).

Using the redesigned LSH scheme, we can reduce $\mathcal{T}_s$ to $\tilde{O}\left(n^{1/c} + m \cdot \exp\left[2\sqrt{\rho \log n \ln \log n}\right]\right)$ and $\mathcal{T}_c$ to $\tilde{O}(n^{1/c})$ as demonstrated by the lemmas below. The lemmas are l_1 equivalents of the lemmas from section 3.2. We analyze $\mathcal{T}_c$ first.

Lemma 4.4.1. *We can preprocess the text T such that, for a query string P of length m, after $\tilde{O}(m)$ processing of P, we can detect whether $|P - T_j^m|_1 \leq R$ or $|P - T_j^m|_1 > cR$ for any j in $O(\log n/\epsilon^2)$ time (assuming that one of this cases holds). This implies that $\mathcal{T}_c = \tilde{O}(n^{1/c} + m)$. The preprocessing of T can be accomplished in $\tilde{O}(n)$.*

Proof. We approximate the distance $|P - T_j^m|_1$ by using the sketching technique of [13] (after a corresponding preprocessing). Assume for now that the query P has length $m = M$. In the initial preprocessing of T, we compute sketches $sk(T_j^M)$ for all i. Next, for a query P, we compute the sketch $sk(P)$ of P; and, from the sketches $sk(P)$ and $sk(T_j^M)$, we can approximate the distance $|T_j^M - P|_1$ with error c with probability $1 - n^{-O(1)}$ in $O(\log n/\epsilon^2)$ time (for details, see [13], especially section 3). If the test reports that a point T_j^M is at distance $\leq cR$ (i.e., the test does not classify the point T_j^M as being at distance $> cR$), then we stop LSH and report this point as the result (note that there is only a small probability, $n^{-O(1)}$, that the test reports that a point T_j^M, with $|P - T_j^M|_1 \leq R$, is at distance $> cR$).

The remaining details are exactly the same as in the lemma 3.2.1: if P is of length $m < M$, then we compute the $sk(P)$ and $sk(T_j^m)$ from $O(\log n)$ sketches of smaller length. Specifically, we divide the string P into $O(\log m)$ diadic intervals, compute the sketches for each diadic interval, and finally add sketches to obtain the sketch for the entire P. Similarly, to obtain $sk(T_j^m)$, we precompute the sketches of all substrings of T of length a power of two (in the T-preprocessing stage); and, for a query P, we add the $O(\log m)$ sketches for the diadic intervals of T_j^m to obtain the sketch of T_j^m. Thus, computation of the sketch of T_j^m takes $O(\log^2 n/\epsilon^2)$ time. Precomputing the sketch of P takes $O(m \log n/\epsilon^2)$ time. With these two times, we conclude that $T_c = \tilde{O}(n^{1/c} + m)$. To precompute all the sketches of all the substrings of T of length a power of two, we need $\tilde{O}(n)$ time by applying FFT along the lines of [14] or section 2.2.1 (since a sketch is just a tuple of dot products of the vector with a random vector). $\square$

Lemma 4.4.2. *Using the new LSH scheme, it is possible to match $g_i(P)$ in the tries S_i, for all i's, in $\mathcal{T}_s = \tilde{O}\left(n^{1/c} + m \cdot \exp[2\sqrt{\rho \log n \ln \log n}]\right)$ time.*

Proof. The algorithm is very similar to the one in the lemma 3.2.2: compute $O(\log M)$ fingerprints of $g_i(P)$ by combining the same fingerprints for functions u_h; then search each $g_i(P)$ using these fingerprints and additional tries $S^{(l)}$. The only nuance is that we treat u_h as having k/t basic LSH functions intertwined as described in section 4.1. Moreover, for functions u_h, we consider we have k positions per symbol in the string (even though only k/t of these are filled by u_h). This view allows an easy combination of fingerprints of severals u_h's into the fingerprint of a function g_i. For ease of notation, we drop the subscript i from g_i and S_i. Recall that S is a trie on strings $g(T_j^M)$, $j = 1 \ldots n-1$; we will drop the superscript M for T_j^M as well.

Formally, we have the following three modifications to the algorithm from 3.2.2.

1. For the fingerprinting function f_l, we use the same fingerprinting function as in the lemma 3.2.2, but we consider f_l as acting on $\Sigma^{2^l k}$, where $\Sigma = \{0, \ldots, \Delta - 1\}$, with Δ

being the maximum value of a coordinate[1]. Furthermore, we define the fingerprint $F_j^{(l)}$ as

$$F_j^{(l)} = \left\langle f_l\left(g(T_j)[0 : 2^l k - 1]\right), f_l\left(g(T_j)[2^l k : 2 \cdot 2^l k - 1]\right) \dots f_l\left(g(T_j)[M - 2^l k : M - 1]\right)\right\rangle$$

2. We need to introduce the following modification to the layout of the functions u_h. Suppose $u_h = (s_1, s_2, \dots s_{k/t})$, where each s_w is a function from the LSH family for l_1 (as in equation 4.1). Then, we write $u_h(P)$ as the vector

$$\begin{array}{lllll} \langle 0^{k-k/t}, & s_1(P[1]), & s_2(P[1]), & \dots & s_{k/t}(P[1]), \\ 0^{k-k/t}, & s_1(P[2]), & s_2(P[2]), & \dots & s_{k/t}(P[2]), \\ \vdots & & & & \\ 0^{k-k/t}, & s_1(P[m]), & s_2(P[m]), & \dots & s_{k/t}(P[m])\rangle \end{array}$$

3. Finally, we compute the fingerprints $f_l(g_i(P))$ from the fingerprints $f_l(u_h(P))$ in a way that is different from that in equation 3.1. Specifically, for a fingerprinting function $f = f_l$, a function $g_i = (u_{h_1}, u_{h_2}, \dots, u_{h_t})$, a diadic interval $G_l = G[a : b]$, we have that

$$f(g_i(P)[a : b]) = \left(\sum_{x=1}^{t} |\Sigma|^{k - x \cdot k/t} \cdot f\left(u_{h_x}(P)[a : b]\right)\right) \pmod R$$

All the other details are precisely the same as in the lemma 3.2.2. We obtain the same running time as in the lemma 3.2.2, multiplied by $k = O(\log^2 n)$. □

4.4.2 Preprocessing time of $\tilde{O}(n^{1+1/c} + n^{1+o(1)} M^{2/3})$

Final preprocessing algorithm for l_1 is very similar to the preprocessing algorithm for Hamming distance described in section 3.3.

Again, there are two steps: constructing all tries $S_i = S_i^0$, and then, constructing all tries $S_i^{(l)}$, for $l > 0$. For constructing the tries $S_i = S_i^0$, we first construct the tries corresponding

[1]As mentioned in the introduction, l_1^m is discretized to $[0, \Delta]^m$.

to the functions u_h; this takes time $\tilde{O}\left(U \cdot w \cdot nM^{2/3}\right) = \tilde{O}\left(nM^{2/3} \exp[2\sqrt{\rho \log n \ln \log n}]\right)$ time (by using the algorithm from section 4.3). The tries for functions u_h enable us to find in $O(1)$ time the first difference of $u_h(T_{j_1})$ and $u_h(T_{j_2})$ for any j_1, j_2. Thus, for a function $g_i = (u_{h_1}, \ldots u_{h_t})$, if p is the smallest index of the first differences of $u_{h_x}(T_{j_1})$ and $u_{h_x}(T_{j_2})$, $x = 1 \ldots t$, then p is also the position where $g_i(T_{j_1})$ and $g_i(T_{j_2})$ differ for the first time. With this comparison operation, we can construct the tries $S_i = S_i^0$ in time $\tilde{O}(n^{1+1/c})$.

The construction of the tries $S_i^{(l)}$, for $l > 0$, is the same as in section 3.3.

The resulting preprocessing time is $\tilde{O}\left(n^{1+1/c} + nM^{2/3} \exp[2\sqrt{\rho \log n \ln \log n}]\right)$.

Chapter 5

Conclusions and Future Work

5.1 Our Contributions

In this thesis, we have investigated the problem of finding the approximate nearest neighbor when the data set points are the substrings of a given text T. In particular, we focused on the case when m, the length of the query pattern P, is not known in advance; this scenario is the most natural in the context of pattern matching.

We have shown a data structure that achieves space and query bounds that essentially match the bounds of [12], which assumed that the pattern length m is given in advance. For preprocessing, when the string distance is measured according to the Hamming distance, the data structure needs $\tilde{O}\left(n^{1+1/c} + nM^{1/3}n^{o(1)}\right)$ time. This preprocessing time essentially matches the preprocessing in the case of m given in advance, as long as $c < 3$.

Similar data structure exists for l_1. While the space and query bounds are essentially the same, preprocessing is only slightly worse, $\tilde{O}\left(n^{1+1/c} + nM^{2/3}n^{o(1)}\right)$ time. Again, this preprocessing time essentially matches that in the case of given m, for $c < 3/2$.

For achieving the stated bounds, we deploy two core ideas[1]. In the first step, we replace the hash tables containing LSH buckets by tries; the tries have the advantage over the hash tables in that the tries support variable-length queries. Using the tries, we achieve the stated

[1]As mentioned previously, using tries for LSH was also proposed by [21], but they used the tries for a different problem and not for (R, c)-SNN.

space bound. Further, in the second step, we redesign the LSH scheme such that it uses a smaller number of independent functions, thus, allowing us to compute some of the LSH functions in parallel. With this redesign, we improve the query and the preprocessing times to the stated bounds.

5.2 Future Work

The most immediate open question is to improve preprocessing times of our data structures. Currently, preprocessing times are $\tilde{O}\left(n^{1+1/c} + nM^{1/3}n^{o(1)}\right)$ for Hamming and $\tilde{O}\left(n^{1+1/c} + nM^{2/3}n^{o(1)}\right)$ for l_1. When these times will reach a bound of $\tilde{O}\left(n^{1+1/c}\right)$, our data structures will essentially match all the bounds obtained for the case when m is given in advance.

A more general open question would ask whether we could achieve bounds that are even better than the bounds of [12] for the case when m is given in advance. Although such an improvement might be possible, it would also imply a better algorithm for the approximate nearest neighbor problem, since we can likely reduce (R,c)-NN to (R,c)-SNN. A better algorithm for (R,c)-NN would certainly be of great interest, as this problem is one of the core problems in high-dimensional computational geometry.

Bibliography

[1] A Amir and M Farach. Eficient 2-dimensional approximate matching of non-rectangular figures. *Information and Computation*, 118:1–11, 1995.

[2] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

[3] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer-Verlag, 2000.

[4] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[5] J. Buhler. Provably sensitive indexing strategies for biosequence similarity search. *Proceedings of the Annual International Conference on Computational Molecular Biology (RECOMB02)*, 2002.

[6] J. Buhler and M. Tompa. Finding motifs using random projections. *Proceedings of the Annual International Conference on Computational Molecular Biology (RECOMB01)*, 2001.

[7] R. Cole, L.A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. *Annual ACM Symposium on Theory of Computing*, pages 91–100, 2004.

[8] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the ACM Symposium on Computational Geometry*, 2004.

[9] A. Efrat, P. Indyk, and S. Venkatasubramanian. Pattern matching for sets of segments. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2001.

[10] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, page 137. IEEE Computer Society, 1997.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.

[12] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. *Annual Symposium on Foundations of Computer Science*, 1998.

[13] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. *Annual Symposium on Foundations of Computer Science*, 2000.

[14] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series datasets using sketches. *Proceedings of the 26th International Conference on Very Large Databases (VLDB)*, 2000.

[15] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the Symposium on Theory of Computing*, 1998.

[16] N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms*. The MIT Press Cambridge, MA, 2004.

[17] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 1987.

[18] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, 1997.

[19] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proceedings of the Thirtieth ACM Symposium on Theory of Computing*, pages 614–623, 1998.

[20] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 1993.

[21] S. Muthukrishnan and S. C. Sahinalp. Simple and practical sequence nearest neighbors with block operations. *CPM*, 2002.