# Approximate string matching in the high-throughput sequencing era

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
vorgelegt von

Enrico Siragusa


am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

Berlin 2014

Datum des Disputation: ***XX.XX.2014***

Gutachter:
***Prof. Dr. Knut Reinert****, Freie Universität Berlin, Deutschland*
***Prof. Dr. XXX XXX****, XXX, XXX*

# Abstract

Abstract...

# CONTENTS


**1. Introduction** . . . . . 1
1.1 Motivation . . . . . 1
1.2 Outline . . . . . 1
1.3 Contributions . . . . . 1

**Part I Approximate String Matching** 3

**2. Stringology preliminaries** . . . . . 5
2.1 Definitions . . . . . 5
2.2 Alignments . . . . . 6
2.3 Distance functions . . . . . 7
2.4 Optimal alignments . . . . . 8
2.5 String matching . . . . . 8
2.5.1 Online methods . . . . . 10
2.5.2 Indexed methods . . . . . 10
2.5.3 Filtering methods . . . . . 14

**3. Online Methods** . . . . . 19
3.1 Banded Myers' bit-vector algorithm . . . . . 19
3.2 Increased bit-parallelism using SIMD instructions . . . . . 19

**4. Indexed Methods** . . . . . 21
4.1 Classic Full-Text Indices . . . . . 22
4.1.1 Suffix array . . . . . 22
4.1.2 $q$-Gram index . . . . . 25
4.2 FM-indices . . . . . 27
4.2.1 Burrows-Wheeler transform . . . . . 27
4.2.2 Rank dictionaries . . . . . 30
4.2.3 Top-down traversal . . . . . 33
4.3 Algorithms . . . . . 34
4.3.1 Bounded top-down traversal . . . . . 34
4.3.2 Exact pattern matching . . . . . 34
4.3.3 Backtracking $k$-mismatches . . . . . 34
4.3.4 Backtracking $k$-differences . . . . . 35
4.3.5 Multiple backtracking . . . . . 37

**5. Filtering Methods** . . . 41
5.1 Gapped $q$-grams . . . 42
5.1.1 Characteristic functions . . . 43
5.1.2 Full-sensitivity . . . 44
5.1.3 Optimal threshold . . . 45
5.1.4 Maximum error . . . 46
5.1.5 Specificity . . . 47
5.1.6 Optimal gapped $q$-grams . . . 47
5.2 Approximate seeds . . . 48
5.2.1 Parameterization . . . 48
5.2.2 Redundancy . . . 49

**Part II Read Mapping** 51

**6. Background** . . . 53
6.1 Sequencing technologies . . . 53
6.1.1 Illumina . . . 53
6.1.2 Ion Torrent . . . 53
6.1.3 454 Life Sciences . . . 53
6.1.4 SOLiD . . . 53
6.2 Sequencing applications . . . 54
6.2.1 DNA-seq . . . 54
6.2.2 RNA-seq . . . 54
6.2.3 ChIP-seq . . . 54
6.3 Sequencing quality . . . 54
6.4 Mappability . . . 54
6.4.1 Genome mappability . . . 55
6.4.2 Mapping quality score . . . 57
6.4.3 Genome mappability score . . . 58

**7. Read mappers engineering** . . . 61
7.1 Masai . . . 61
7.1.1 Approximate seeds . . . 61
7.1.2 Multiple backtracking . . . 61
7.1.3 Single-end mapping . . . 61
7.1.4 Paired-end mapping . . . 61
7.2 Yara . . . 61
7.2.1 Single-end mapping . . . 61
7.2.2 Paired-end mapping . . . 61
7.2.3 Parallelization . . . 61
7.2.4 Hardware acceleration . . . 61

**8. Read mappers evaluation** . . . 63
8.1 Popular read mappers . . . 63
8.1.1 Bowtie . . . 63
8.1.2 BWA . . . 64
8.1.3 Soap . . . 65

8.1.4 SHRiMP . . . 65
8.1.5 RazerS . . . 65
8.1.6 mr(s)Fast . . . 66
8.1.7 GEM . . . 66
8.1.8 Masai . . . 66
8.1.9 Yara . . . 66
8.2 Rabema results . . . 68
8.3 Variant detection results . . . 68
8.4 Runtime results . . . 68

**9. Discussion** . . . 69

**A. Related problems** . . . 71
A.1 Dictionary search . . . 71
A.1.1 Online methods . . . 71
A.1.2 Indexed methods . . . 71
A.1.3 Filtering methods . . . 71
A.2 Local similarity search . . . 72
A.2.1 Online methods . . . 72
A.2.2 Indexed methods . . . 72
A.2.3 Filtering methods . . . 72
A.3 Overlaps computation . . . 72
A.3.1 Online methods . . . 72
A.3.2 Indexed methods . . . 73

**B. Declaration** . . . 75

**Bibliography** . . . 76

CHAPTER 1

# Introduction

## 1.1 Motivation

## 1.2 Outline

## 1.3 Contributions

# Part I

# APPROXIMATE STRING MATCHING

CHAPTER

# 2 Stringology preliminaries

We now introduce fundamental definitions and problems of stringology, in order to keep the manuscript self-contained. We provide a summary of notations at the end of this manuscript. The reader familiar with basic stringology can skip this chapter and proceed to chapter 3.

## 2.1 Definitions

Let us start by introducing primitive objects of stringology: alphabets and strings. An *alphabet* is a finite ordered set of symbols (or characters); a *string* (or word) over an alphabet is a finite sequence of symbols from that alphabet. We denote the length of a string $s$ by $|s|$, and by $\epsilon$ the empty string s.t. $|\epsilon| = 0$.

**Definition 2.1.** Given an alphabet $\Sigma$, we define $\Sigma^0 = \{\epsilon\}$ as the set containing the empty string, $\Sigma^n$ as the set of all strings over $\Sigma$ of length $n$, and $\Sigma^* = \cup_{n=0}^{\infty}\Sigma^n$ as the set of all strings over $\Sigma$. Finally, we call any subset of $\Sigma^*$ a language over $\Sigma$.

**Definition 2.2.** We define the concatenation operator of two strings as $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$. Given two strings $x \in \Sigma^m$ with $x = x_1x_2 \dots x_m$ and $y \in \Sigma^n$ with $y = y_1y_2 \dots y_n$, their concatenation $x \cdot y$ (simply denoted $xy$) is the string $z \in \Sigma^{m+n}$ consisting of the symbols $x_1x_2 \dots x_my_1y_2 \dots y_n$.

**Definition 2.3.** A string $x$ is a *prefix* of $y$ iff there is some string $z$ s.t. $y = x \cdot z$; analogously, $x$ is a *suffix* of $y$ iff there is some string $z$ s.t. $y = z \cdot x$; moreover, $x$ is a *substring* of $y$ iff there is some string $w, z$ s.t. $y = w \cdot x \cdot z$. We denote by $y_{1\dots i}$ the prefix of $y$ ending at position $i \in \mathbb{N}$, by $y_{i\dots n}$ (or simply $y_{i\dots}$) the suffix of $y$ beginning at position $i \in \mathbb{N}$, and by $y_{i\dots j}$ the substring of $y$ within positions $i \leq j \in \mathbb{N}$.

**Definition 2.4.** Given an alphabet $\Sigma$ of size $\sigma$, we define the function $\rho : \Sigma \to [1 \dots \sigma]$ denoting the *lexicographic rank* of any alphabet symbol, s.t. $\rho(a) < \rho(b) \iff a < b$ for any distinct $a, b \in \Sigma$.

**Definition 2.5.** We define the *lexicographical order* $<_{lex}$ between two non-empty strings $x, y$ as $x <_{lex} y \iff x_1 < y_1$, or $x_1 = y_1$ and $x_{2\dots} <_{lex} y_{2\dots}$

**Definition 2.6.** We define a *string collection* as an ordered multiset $\mathbb{S} = \{s^1, s^2, \dots, s^c\}$ of non necessarily distinct strings over a common alphabet $\Sigma$. We denote by $\|\mathbb{S}\| = \sum_{i=1}^{c} |s^i|$

the total length of the string collection. We also extend the notation of prefix, suffix and substring to multisets, e.g. $\mathbb{S}_{(d,i)\dots(d,j)}$ denotes the substring $s^d_{i\dots j}$.

**Definition 2.7.** We call *terminator symbol* a symbol $\$ \notin \Sigma$ s.t. $\rho(\$) < \rho(a)$ for any $a \in \Sigma$.

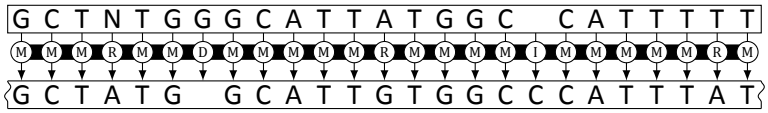
**Definition 2.8.** Given a string $s$ over $\Sigma$, we call *padded string* the concatenation of $s$ with a terminator symbol \$.

**Definition 2.9.** Given a string collection $\mathbb{S}$ over $\Sigma$, we call *padded string collection* the collection consisting of strings $s^i \in \mathbb{S}$ padded with terminator symbols $\$^i$ s.t. $\rho(\$^i) < \rho(\$^j) \iff i < j$.

## 2.2 Alignments

The next step is to define the minimal set of edit operations to transform one string into another: substitutions, insertions and deletions. Given two strings $x, y$ of equal length $n$, the string $x$ can be transformed into the string $y$ by substituting (or replacing) all symbols $x_i$ s.t. $x_i \neq y_i$ into $y_i$, for $1 \leq i \leq n$. If the given strings have different lengths, insertion and deletion of symbols from $x$ become necessary to transform it into $y$. Therefore, given any two strings $x, y$, we define as edit transcript for $x, y$ any finite sequence of substitutions, insertions and deletions transforming $x$ into $y$. See Figure 2.1 for an example.

***Figure 2.1:*** *Example of edit transcript transforming the string $x = AAAA$ into $y = CCCC$. The transcript character M indicates a match, R a replacement, I an insertion, and D a deletion.*



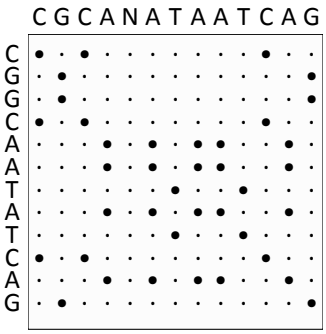

An alignment is an alternative yet equivalent way of visualizing a transformation between strings. While an edit transcript provides an explicit sequence of edit operations transforming one string into another, an alignment relates pairs of corresponding symbols between two strings. Because some symbols in one string are not related to any symbol in the other string, i.e. some symbols are inserted or removed, we first need to introduce a gap symbol $-$, which is not part of the string alphabet $\Sigma$. Subsequently, we can define the alignment of two strings of length $m, n$ over $\Sigma$ to be a string of length between $\min\{m, n\}$ and $m + n$ over the pair alphabet $(\Sigma \cup \{-\}) \times (\Sigma \cup \{-\})$.

**Example 2.1.** An alignment of the strings $x = AAAA$ and $y = CCCC$ is given by the string $z = \binom{A}{A}\binom{A}{-}\binom{A}{C}\binom{G}{G}$

A dotplot is a way to visualize any alignment between two strings and highlight their similarities. Given two string $x, y$ of length $m, n$, a dotplot is a $m \times n$ matrix containing a

dot at position $(i, j)$ iff the symbol $x_i$ matches symbol $y_j$. We define a dotplot trace to be a monotonical path in the matrix connecting non-decreasing positions of the matrix. A dotplot trace corresponds to an alignment and vice versa. In a trace, match and mismatch columns of the corresponding alignment appear as diagonal stretches, while insertions and deletions are horizontal or vertical stretches. See Figure 2.2.

***Figure 2.2:*** *Example of dotplot of the strings $x = AAAA$ and $y = CCCC$. The highlighted trace corresponds to the alignment of example 2.1.*




## 2.3 Distance functions

We can assign a cost to any alignment and to its associated edit transformation by defining a weight function $\omega : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}_0^+$, where:

- $\omega(\alpha, \beta)$ for all $(\alpha, \beta) \in \Sigma \times \Sigma$ defines the cost of substituting $\alpha$ with $\beta$,
- $\omega(\alpha, -)$ for all $\alpha \in \Sigma$ defines the cost of deleting the symbol $\alpha$,
- $\omega(-, \beta)$ for all $\beta \in \Sigma$ defines the cost of inserting the symbol $\beta$,

and by defining the total cost $C(z)$ of an alignment $z$ between two strings as the sum of the weights of all its alignment symbols:

$$C(z) = \sum_{i=0}^{|z|} \omega(z_i) \tag{2.1}$$

Consequently, we can define the distance function $d : \Sigma^* \times \Sigma^* \to \mathbb{R}_0^+$ by taking the minimum cost over all possible alignments of $x, y$:

$$d(x, y) = \sum_{z \in \mathbb{A}(x,y)} C(z) \tag{2.2}$$

In particular, the edit or *Levenshtein distance* between two strings $x, y \in \Sigma^*$ is defined as the function $d_E : \Sigma^* \times \Sigma^* \to \mathbb{N}_0$ counting the *minimum* number of edit operation necessary to transform $x$ into $y$. It is obtained by defining for all $(\alpha, \beta) \in \Sigma \times \Sigma$, $\omega(\alpha, \beta) = 1$ iff

$\alpha \neq \beta$ and 0 otherwise, and $\omega(\alpha, -)$ and $\omega(-, \beta)$ as 1. The *Hamming distance* between two strings $x, y \in \Sigma^n$ is defined as the function $d_H : \Sigma^n \times \Sigma^n \to \mathbb{N}_0$ counting the number of substitutions necessary to transform $x$ into $y$. We obtain it by defining $\omega(\alpha, \beta)$ as in the edit distance, and by setting all $\omega(\alpha, -)$ and $\omega(-, \beta)$ to be $\infty$ in order to disallow indels.

**Example 2.2.** TODO: example of edit and hamming distance.

## 2.4 Optimal alignments

The problem of finding an optimal alignment between two strings is equivalent to the problem of finding their minimum distance [Gusfield, 1997]. A solution to this optimization problem can be efficiently computed via dynamic programming (DP). Below we describe the three essential components of the DP approach: the recurrence relation, the DP table, and the traceback.

Given two strings $x, y$ of length $m, n$, for all $1 \leq i \leq m$ and $1 \leq j \leq n$ we define with $d(x_{1..i}, y_{1..j})$ the distance between their prefixes $x_{1..i}$ and $y_{1..j}$. The base conditions of the recurrence relation are:

$$d(\epsilon, \epsilon) = 0 \tag{2.3}$$

$$d(x_{1..i}, \epsilon) = \sum_{l=1}^{i} \omega(x_l, -) \text{ for all } 1 \leq i \leq m \tag{2.4}$$

$$d(\epsilon, y_{1..j}) = \sum_{l=1}^{j} \omega(-, y_l) \text{ for all } 1 \leq j \leq n \tag{2.5}$$

and the recursive case is defined as follows:

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} d(x_{1..i-1}, y_{1..j}) & + \quad \omega(x_i, -) \\ d(x_{1..i}, y_{1..j-1}) & + \quad \omega(-, y_j) \\ d(x_{1..i-1}, y_{1..j-1}) & + \quad \omega(x_i, y_j) \end{cases} \tag{2.6}$$

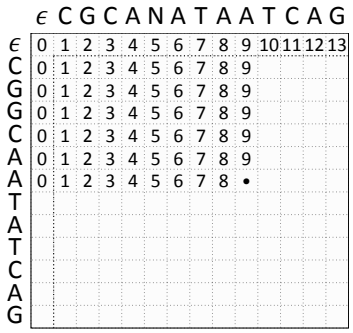
We can compute the above recurrence relation in time $\mathcal{O}(nm)$ using a dynamic programming table $D$ of $(m+1)\times(n+1)$ cells, where cell $D[i, j]$ stores the value of $d(x_{1..i}, y_{1..j})$. The sole distance without any alignment can be computed in space $\mathcal{O}(\min\{n, m\})$, as we only need column $D[: j-1]$ to compute column $D[: j]$ (or row $D[i-1 :]$ to compute $D[i :]$) and we can fill the table $D$ either column-wise or row-wise[1]. An optimal alignment can be computed in time $\mathcal{O}(m+n)$ via *traceback* on the table $D$: We start in the cell $D[m, n]$ and go backwards (either left, up-left, or up) to the previous cell by deciding which condition of equation 2.6 yielded the value of $D[m, n]$.

## 2.5 String matching

We can now define *exact string matching*, perhaps the most fundamental problem in stringology.

[1] Note that $D$ can be filled also diagonal-wise or antidiagonal-wise.

***Figure 2.3:*** *DP table representing the computation of the edit distance* $d_E(x_{1..5}, y_{1..4})$.




**Definition 2.10.** [Gusfield, 1997] Given a string $p$ of length $m$, called the *pattern*, and a string $t$ of length $n$, called the *text*, the exact string matching problem is to find all occurrences of pattern $p$ into text $t$.

This problem has been extensively studied from the theoretical standpoint and is well solved in practice. Nonetheless, the definition of distance functions between strings let us generalize exact string matching into a more challenging problem: *approximate string matching*.

**Definition 2.11.** Given a text $t$, a pattern $p$, and a *distance threshold* $k \in \mathbb{N}$, the approximate string matching problem is to find all occurrences of $p$ into $t$ within distance $k$.

The approximate string matching problem under the Hamming distance is commonly referred as the *$k$-mismatches* problem and under the edit distance as the *$k$-differences* problem. For $k$-mismatches and $k$-differences, it must hold $k > 0$ as the case $k = 0$ corresponds to exact string matching, and $k < m$ as a pattern occurs at any position in the text if we substitute all its $m$ characters.

**Definition 2.12.** Under the edit or Hamming distance, we define the *error rate* as $\epsilon = \frac{k}{m}$, with $0 < \epsilon < 1$ given the above conditions.

We can classify string matching problems in two categories, *online* and *offline*, depending on which string, the pattern or the text, is given first. Algorithms for online string matching work by preprocessing the pattern and scanning the text from left to right (or right to left); their worst-case runtime complexity ranges from $\mathcal{O}(nm)$ to $\mathcal{O}(n)$ while their worst-case memory complexity ranges from $\mathcal{O}(\sigma^k m^k)$ to $\mathcal{O}(m)$. Algorithms for offline string matching are instead allowed to preprocess the text; their worst-case runtime complexity ranges from $\mathcal{O}(m)$ to $\mathcal{O}(\sigma^k m^k)$ while their worst-case memory complexity is usually $\mathcal{O}(n)$. In practice, if the text is long, static and searched frequently, offline methods largely outperform online methods in terms of runtime, provided the necessary amount of memory. Therefore, we concentrate on offline algorithms.

We can subdivide algorithms for offline string matching in two categories: *fully-indexed* and *filtering*. Fully-indexed algorithms work solely on the index of the text, while filtering

methods first use the index to discard uninteresting portions of the text and subsequently use an online method to verify narrow areas of the text. Filtering methods outperform fully-indexed methods for a vast range of inputs (when the error rate is low) and are thus very interesting from a practical standpoint. Nonetheless, filtering methods are just opportunistic combinations of online and fully-indexed methods.

In the following of this section we thus give a brief and non-exhaustive overview of the fundamental techniques for online and (both fully-indexed and filtering) offline string matching. This overview serves as an introduction to the more involved algorithms presented in the next chapters and directly used in applications of part II. For an extensive treatment of the subject, we refer the reader to complete surveys on exact [Faro and Lecroq, 2013] and approximate [Navarro, 2001] online string matching methods, as well as to a succinct survey on indexed methods [Navarro *et al.*, 2001].

### 2.5.1 Online methods

#### Dynamic programming

The dynamic programming algorithm **??** to compute the distance of two strings can be easily turned into a string matching algorithm. Since an occurrence of the pattern can start and end anywhere in the text, $k$-differences consists of computing the edit distance between the pattern and any substring of the text. The problem can be thus solved by computing the edit distance between the text and the pattern without penalizing leading and trailing deletions in the text.

Consider equation 2.6 and pose $x = p$ and $y = t$. Since an occurrence of the pattern can start anywhere in the text, we change the initialization of the top row $D[0\ :]$ of the DP matrix according to the condition:

$$d(\epsilon, y_{1..j}) = 0 \text{ for all } 1 \leq j \leq n \tag{2.7}$$

and since an occurrence of the pattern can end anywhere in the text, we check all cells $D[m, j]$ for all $1 \leq j \leq n$ in the bottom row of $D$ for the condition $D[m, j] \leq k$.

***Figure 2.4:*** *DP table representing the match of $p = ...$ in $t = ....$*

| | $\epsilon$ | C | G | C | A | N | A | T | A | A | T | C | A | G | A | A | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\epsilon$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |
| G | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |
| G | 3 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |
| C | 4 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |
| A | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |
| A | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | • | | | | | | | |

### 2.5.2 Indexed methods

Indexed methods are more attractive than online methods when several patterns are to be searched in the same static text. These methods work by building an index of the text

beforehand to speed up subsequent searches. To this intent, we now introduce *suffix tries*, idealized data structures to index texts. Moreover, we define a set of generic operations to traverse suffix tries in a top-down fashion. In chapter 4, we will consider various data structures replacing suffix tries in practice. The generic traversal operations here introduced will let us formulate generic algorithms solving string matching problems on any of these data structures.

### Trie

Consider a padded string collection $\mathbb{S}$ (definition 2.9) consisting of $c$ strings. Note that padding is necessary to ensure that no string $s^i \in \mathbb{S}$ is a prefix of another string $s^j \in \mathbb{S}$.

**Definition 2.13.** The trie $\mathcal{S}$ of $\mathbb{S}$ is a lexicographically ordered tree data structure having one node designated as the root and $c$ leaves, denoted as $l_1 \dots l_c$, where leaf $l_i$ points to string $s^i$. The entering edge of any node of $\mathcal{S}$ is labeled with a symbol in $\Sigma$, while the entering edge of any leaf of $\mathcal{S}$ is labeled with a terminator symbol. Any path from the root to a leaf $l_i$ spells the string $s^i$, including its terminator symbol $\$^i$.

### Suffix trie

We define the suffix trie as the trie of all suffixes of a string.

**Definition 2.14.** Given a padded string $s$ (definition 2.8) of length $n$, the suffix trie $\mathcal{S}$ of $s$ has $n$ leaves where leaf $l_i$ points to suffix $s_{i..n}$.

We generalize the suffix trie to index string collections.

**Definition 2.15.** Given a padded string collection $\mathbb{S}$ (definition 2.9), any leaf of the *generalized suffix trie* is labeled with a pair $(i, j)$ where $i$ points to the string $s^i \in \mathbb{S}$ and $1 \leq j \leq n_i$ points to one of the $n_i$ suffixes of $s^i$. Thus each path from the root to a leaf $(i, j)$ spells the suffix $\mathbb{S}_{(i,j)\dots}$.

Note that the suffix trie so defined contains $\mathcal{O}(n^2)$ nodes, yet we only consider the suffix trie as an idealized data structure to expose our algorithms. The optimal data structure to represent in linear space all suffixes of a given string is the *suffix tree* [Morrison, 1968]. However, the suffix tree comes with the restriction that internal nodes must have more than one child and with the property that edges can labeled by strings of arbitrary length (see Figure 2.5). This latter property slightly complicates the exposition of our string matching algorithms but it does not affect their runtime complexity nor their result. For this reason, in the following of this manuscript we always consider w.l.o.g. suffix tries instead of suffix trees.

### Top-down traversal

We now give a set of generic operations to traverse tries in a top-down fashion. Given a trie $\mathcal{T}$, we define the following operations inspecting any pointed node $x$ of $\mathcal{T}$:

***Figure 2.5:*** *Suffix trie and suffix tree for the string ANANAS$.*

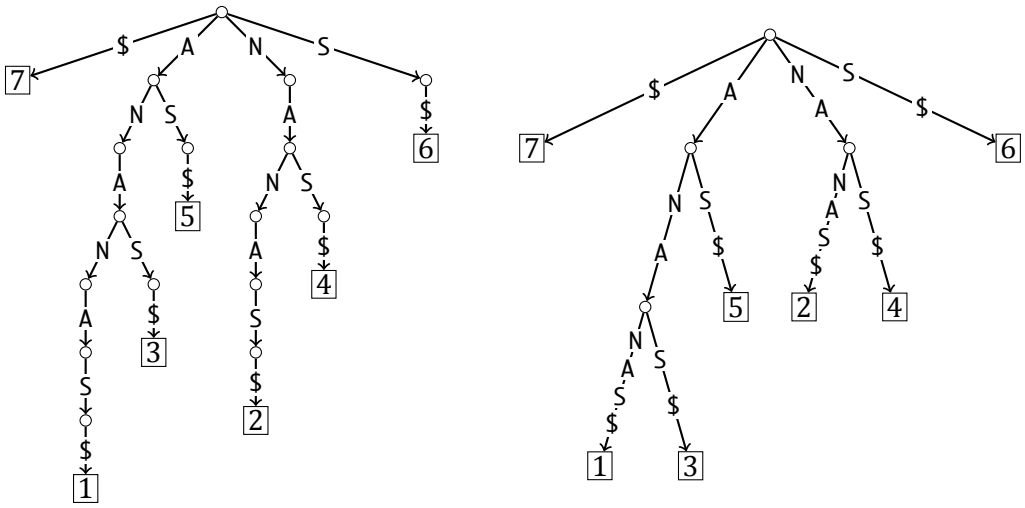
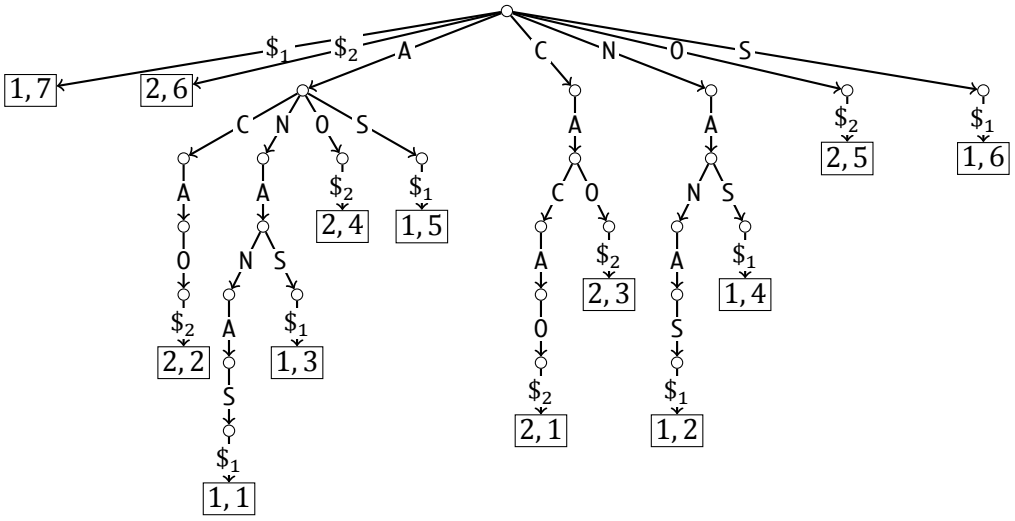


***Figure 2.6:*** *Generalized suffix trie for the string collection* $\mathbb{S} = \{$ ANANAS$\$_1$, CACAO$\$_2$ $\}$.



- ISROOT($x$) returns true iff the pointed node is the root;
- ISLEAF($x$) returns true iff all outgoing edges are labeled by terminator symbols;
- LABEL($x$) returns the symbol labeling the edge entering $x$;
- OCCURRENCES($x$) returns the list of positions pointed by leaves below $x$;

and the following operations moving from pointed node $x$, and returning true on success and false otherwise:

- GODOWN($x$) moves to the lexicographically smallest child of $x$;
- GODOWN($x, c$) moves to the child of $x$ whose entering edge is labeled by $c$;
- GORIGHT($x$) moves to the lexicographically next child of $x$;

**Algorithm 2.1** GODOWN($x$)

**Input** $x$ : pointer to a suffix trie node
**Output** boolean indicating success

```
1: if ISLEAF(x) then
2:     return false
3: if GODOWN(x, min_lex Σ) then
4:     return true
5: else
6:     return GORIGHT(x)
```

**Algorithm 2.2** GORIGHT($x$)

**Input** $x$ : pointer to a suffix trie node
**Output** boolean indicating success

```
1: if not ISROOT(x) then
2:     while c ← next_lex Σ, LABEL(x) do
3:         GOUP(x)
4:         if GODOWN(x, c) then
5:             return true
6: return false
```

- GOUP($x$) moves to the parent node of $x$.

Time complexities of the above operations depend on the data structure implementing the trie. Usually LABEL is $\mathcal{O}(1)$, both variants of GODOWN and GORIGHT can be $\mathcal{O}(1)$ or logarithmic in $n$, OCCURRENCES can be linear in the number of occurrences, GOUP is $\mathcal{O}(1)$ but with an additional $\mathcal{O}(n)$ space complexity to stack all parent nodes. Note that is always possible to implement GODOWN and GORIGHT by relying on GODOWN a symbol and GOUP, but their complexity becomes $\mathcal{O}(\sigma)$. In chapter 4, we consider various data structures to implement suffix tries, we show how to implement these operations and give their complexities.

#### String matching algorithms

Using the above generic top-down traversal operations we are able to formulate generic string matching algorithms. Here we formulate the simple algorithm for exact string matching on a suffix trie. In chapter 4, we give analogous algorithms solving $k$-mismatches and $k$-differences by means of a backtracking top-down traversal [Ukkonen, 1993; Baeza-Yates and Gonnet, 1999].

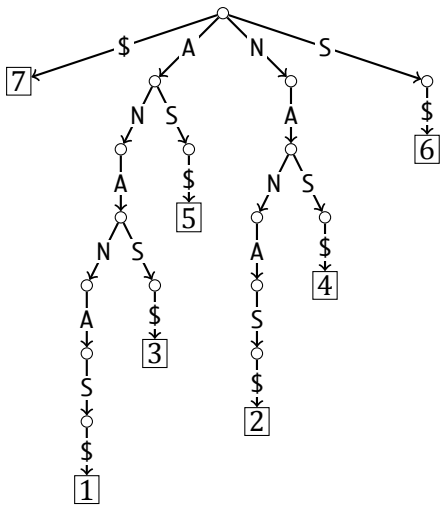
We assume the text $t$ to be indexed by its suffix trie $\mathcal{T}$. Algorithm **??** searches a pattern $p$ by starting in the root node of $\mathcal{T}$ and following the path spelling the pattern. If the search ends up in a node $x$, each leaf $l_i$ below $x$ points to a distinct suffix $t_{i..n}$ such that $t_{i..i+m}$ equals $p$. If we implement GODOWN in constant time and OCCURRENCES in linear time, we find all occurrences of $p$ into $t$ in optimal time $\mathcal{O}(m + o)$, where $m$ is the length of $p$ and $o$ its number of occurrences in $t$.

**Algorithm 2.3** ExactSearch$(t, p)$

**Input** $t$ : pointer to a suffix trie node
$p$ : pointer to a pattern
**Output** list of all occurrences of the pattern in the text

```
1: if atEnd(p) then
2:     report occurrences(t)
3: else if goDown(t, value(p)) then
4:     goNext(p)
5:     ExactSearch(t, p)
```

***Figure 2.7:*** *Exact string matching on a suffix tree. The pattern NA is searched exactly in the text ANANAS$.*




### 2.5.3 Filtering methods

The goal of filtering methods is to obtain algorithms with favorable average running times. The principle under which they work is that large and uninteresting portions of the text can be quickly discarded, while only narrow and highly similar portions have to be verified with a conventional online method. We remark that any filtering method can either work in a online fashion or take advantage of an index of the text to speed up the filtration phase. Here we always consider the filtration phase to be indexed.

Filtering methods work under the assumption that given patterns occur in the text with a *low average probability*. Such probability is a function of the error rate $\epsilon$, in addition to the alphabet size $\sigma$, and can be computed or estimated under the assumption of the text being generated by a specific random source. Under the uniform Bernoulli model, where each symbol of $\Sigma$ occurs with probability $\frac{1}{\sigma}$, Navarro and Baeza-Yates [2000] estimates that $\epsilon < 1 - \frac{1}{\sigma}$ is a conservative bound on the error rate which ensures few matches,

and for which filtering algorithms are effective. For higher error rates, non-filtering online and indexed methods work better.

We call a filter *lossless* or *full-sensitive* if it guarantees not to discard any occurrence of the pattern, otherwise we call it *lossy*. Lossy filters can be designed to solve approximately $\epsilon$-differences. We focus our attention on lossless filters.

We now consider two classes of filtering methods: those based on *seeds* and those based on *q-grams*. Filters based on seeds partition the pattern into *non-overlapping* factors called seeds. We easily derive full-sensitive partitioning strategies by applying the pigeonhole principle. Instead, filters based on $q$-grams consider all *overlapping* substrings of the pattern having length $q$, the so-called $q$-grams. A simple lemma gives us a lower bound on the number of $q$-grams that must be present in a narrow window of the text as necessary condition for an approximate occurrence of the pattern.

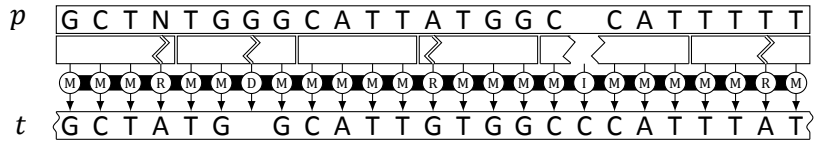
### Seed filters

We start by considering the case of two arbitrary strings $x, y$ within edit distance $k$.

**Lemma 2.1.** *[Baeza-Yates and Navarro, 1999] Let $x, y$ be two strings s.t. $d_E(x, y) = k$. If we partition w.l.o.g. $y$ into $k + 1$ non-overlapping seeds, then at least one seed will occur as a factor of $x$.*

It is immediate to see that any edit distance error can cover at most one seed, therefore at least one seed of $y$ will not be covered by any seed and occur as a factor of $x$. Figure 2.8 shows an example.

***Figure 2.8:*** *Filtration with exact seeds.*




Thus, we solve $k$-differences by partitioning the pattern into $k+1$ seeds and searching all seeds into the text, e.g. with the help of a substring index. As Lemma 2.1 gives us a necessary but not sufficient condition, we must verify whether any candidate location induced by an occurrence of some seed corresponds to an approximate occurrence of the pattern in the text. Thus, we verify any substring $s$ of the text of length $|s| \in [m-k, m+k]$ containing one seed of $p$. Note how we are reducing one approximate search into many smaller exact searches.

How many verifications a seed produces? Let us assume the text to be generated according to the uniform Bernoulli model. The emission probability of any symbol in $\Sigma$ is $p = \frac{1}{\sigma}$ and under i.i.d. assumptions the emission (and occurrence) probability of any word of length $q$ is simply

$$\Pr(H > 0) = \frac{1}{\sigma^q} \tag{2.8}$$

thus the expected number of occurrences of a seed of length $q$ in a text of length $n$ is

$$E[H] = \sum_{i=1}^{n-q+1} \Pr(H > 0) = \frac{n-q+1}{\sigma^q} \leq \frac{n}{\sigma^q}. \tag{2.9}$$

Note how lemma 2.1 requires us to partition the pattern into $k+1$ seeds but leaves us free to choose their length. This leads to the question of which is an optimal pattern partitioning, i.e. a partitioning that minimizes the expected number of verifications. We fix[2] the length of all seeds to be

$$q = \left\lfloor \frac{m}{k+1} \right\rfloor \tag{2.10}$$

to minimize the expected number of occurrences of any seed. Under these conditions, the expected number of verifications produced by our seed filter is

$$E[V] = E[H] \cdot (k+1) < \frac{n(k+1)}{\sigma^q} \tag{2.11}$$

We now turn to the effect of the error rate on the runtime of the resulting $k$-differences algorithm. For which error rate we expect the resulting algorithm to have sublinear runtime? If we use the classic $\mathcal{O}(m^2)$ DP algorithm of section 2.5.1 to verify candidate locations, the expected runtime must be

$$E[V] \cdot m^2 < cn \tag{2.12}$$

for some constant $c$. By substituting $E[V]$ and solving for $q$, we obtain

$$q > \log_\sigma \frac{m^3}{c} \tag{2.13}$$

and since we chose $q$ as a function of $m$ and $k$ in equation 2.10, it follows that

$$\epsilon = \frac{k}{m} < \frac{m}{\log_\sigma m} \tag{2.14}$$

is the error rate for which this $k$-differences algorithm has expected sublinear runtime.

Nonetheless, texts of practical interest like genomes and natural texts do not fit well the uniform Bernoulli model. On those texts uniform seed length can lead to suboptimal filtration. In section 5.2 we analyze the statistical properties of these texts and the effects of pattern partitioning. Later, in order to obtain more robust filters, we generalize filtration with seeds to consider approximate seeds.

[2] We ignore for simplicity that some seed could have length $\left\lceil \frac{m}{k+1} \right\rceil$.

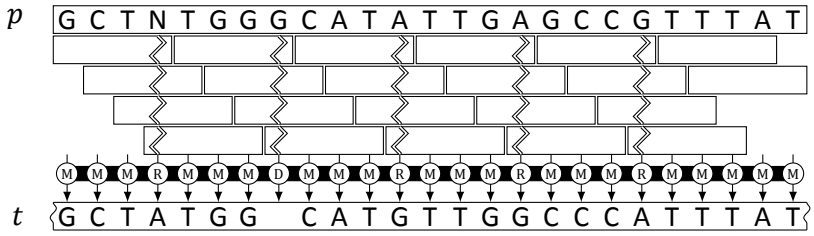
### $q$-Gram filters

$q$-Gram filters are based on the so-called $q$-gram similarity measure $\tau_q : \Sigma^* \times \Sigma^* \to \mathbb{N}_0$, defined as the number of substrings of length $q$ common to two given strings. The following lemma relates $q$-gram similarity to edit and Hamming distance. It gives a lower bound on the $q$-gram similarity $\tau_q(x, y)$ of any two strings $x, y$ within edit distance $k$. This means that $\tau_q(x, y) \geq k$ is a necessary but not sufficient condition for $d_E(x, y) \leq k$.

**Lemma 2.2** (The $q$-gram lemma)**.** *[Jokinen and Ukkonen, 1991] Let $x, y$ be two strings s.t. $d_E(x, y) = k$, and assume w.l.o.g. $|x| \leq |y|$ and $|x| = m$. Then $x$ and $y$ have $q$-gram similarity $\tau_q(m, k) \geq m - q + 1 - kq$.*

The first part of the threshold function $\tau$ counts the number of $q$-grams of $x$ (i.e. $m - q + 1$), while the second part counts how many $q$-grams can be covered by $k$ errors ($kq$, i.e. at most $q$ per error). Figure 2.9 illustrates. Note how the threshold function $\tau$ depends only on the parameters $q, m, k$ and not on any specific $q$-gram.

***Figure 2.9:*** *Filtration with q-grams.*




Lemma 2.2 itself does not give us a direct solution to approximate string matching. Indeed, it considers the edit distance between two arbitrary strings, while in approximate string matching the pattern can match any substring of the text. More precisely, if the pattern approximately matches any substring $s$ of $t$, then in the case of $k$-mismatches $s$ must have length $m$; in the case of $k$-difference, the length of $s$ must be within $m - k$ and $m + k$. The dot plot representation helps us to visualize this concept: $k$-mismatches occurrences cover one single diagonal of the dot plot, while $k$-difference occurrences are enclosed inside a parallelogram of side $2k + 1$.

Therefore, we can design an online filtration algorithm scanning the text and counting how many $q$-grams of the pattern fall into each parallelograms. Only the parallelograms exceeding the threshold $\tau_q(m, k)$ have to be verified with an online method, e.g. standard DP. Alternatively, to speed up the filtration phase, we can count the $q$-grams with the help of a substring index.

Which is the biggest $q$-gram length yielding lossless filtration given $m$ and $k$? In order to satisfy lemma 2.2, the $q$-gram threshold must be greater than zero, i.e. it must hold $\tau_q(m, k) \geq 1$. Thus, by substituting $\tau$, we obtain $q \leq \left\lfloor \frac{m}{k+1} \right\rfloor$, analogously to equation 2.10 of seed filters.

CHAPTER 3

# Online Methods

## 3.1 Banded Myers' bit-vector algorithm

## 3.2 Increased bit-parallelism using SIMD instructions

CHAPTER

# 4 Indexed Methods

Suffix trees are elegant data structures but they are rarely used in practice. Although suffix trees provides optimal construction and query time, their high space consumption prohibits practical applicability to large string collections. A practical study on suffix trees [Kurtz, 1999] reports that efficient implementations achieve sizes between 12 $n$ and 20 $n$ bytes per character. For instance, two years before completing the sequencing of the human genome, Kurtz conjectured the resources required for computing the suffix tree for the complete human genome (consisting of about $3 \cdot 10^9$ bp) in 45.31 GB of memory and nine hours of CPU time, and concluded that "it seems feasible to compute the suffix tree for the entire human genome on some computers".

We might be tempted to think that such memory requirements are not anymore a limiting factor as, at the time of writing, standard personal computers come with 32 GB of main memory. Indeed, over the last decades, the semiconductors industry followed the exponential trends dictated by Moores' law and yielded not only exponentially faster microprocessors but also larger memories. Unfortunately, memory latency improvements have been more modest, leading to the so called memory wall effect [Wilkes, 1995]: data access times are taking an increasingly fraction of total computation times. Thus, if in 1973 Knuth wrote that "space optimization is closely related to time optimization in a disk memory", forty years later we can simply say that space optimization is closely related to time optimization.

Over the last years, a significant effort has been devoted to the engineering of more space-efficient data structures to replace the suffix tree in practical applications. In particular, much research has been done into designing succinct (or even compressed) data structures providing efficient query times using space proportional to that of the uncompressed (or compressed) input. Thanks to this research, we are able to index the human genome in as little as 3.5 GB of memory and at the same time improve query time over classic indices.

In this chapter, we introduce some classic full-text indices (suffix arrays and $q$-gram indices) and subsequently a succinct full-text index (an uncompressed variant of the original FM-index). Afterwards we give generic approximate string matching algorithms working on any of these data structures.

## 4.1 Classic Full-Text Indices

### 4.1.1 Suffix array

The key idea of the suffix array [Manber and Myers, 1990] is that most information explicitly encoded in a suffix trie is superfluous for pattern matching. We can omit the explicit representation of suffix trie's internal nodes and outgoing edges. Leaves pointing to the sorted suffixes are sufficient to perform exact pattern matching or even trie traversals. Indeed, we can compute on the fly any path from the root to an internal node via binary search over the leaves. In this way, we are willing to pay an additional logarithmic time complexity to reduce space by a linear factor.

We define the suffix array and later show how to emulate suffix trie traversal.

**Definition 4.1.** The suffix array of a string $s$ of length $n$ is defined as an array $A$ containing a permutation of the interval $[1, n]$, such that $s_{A[i]\ldots n} <_{lex} s_{A[i+1]\ldots n}$ for all $1 \leq i < n$.

***Figure 4.1:*** *Suffix array for the string* ANANAS$.

| $i$ | $A[i]$ | $s_{A[i]\ldots n}$ |
|---|---|---|
| 1 | 7 | $ |
| 2 | 1 | ANANAS$ |
| 3 | 3 | ANAS$ |
| 4 | 5 | AS$ |
| 5 | 2 | NANAS$ |
| 6 | 4 | NAS$ |
| 7 | 6 | S$ |

We define the generalized suffix array to index string collections, analogously to the generalized suffix trie introduced in section 2.5.2.

**Definition 4.2.** The generalized suffix array of a padded string collection $\mathbb{S}$ (definition 2.9), consisting of $c$ strings of total length $n$, is defined as an array $A$ of length $n$ containing a permutation of all pairs $(i, j)$ where $i$ points to a string $s^i \in \mathbb{S}$ and $1 \leq j \leq n_i$ points to one of the $n_i$ suffixes of $s^i$. Pairs are ordered such that $\mathbb{S}_{A[i]\ldots} <_{lex} \mathbb{S}_{A[i+1]\ldots}$ for all $1 \leq i < n$.

We can construct the suffix array in $\mathcal{O}(n)$ time, for instance using the [Kärkkäinen and Sanders, 2003] algorithm, or using non-optimal but practically faster algorithms, e.g. [Schürmann and Stoye, 2007]. The space consumption of the suffix array is $n \log n$ bits. When $n < 2^{32}$, a 32 bit integer is sufficient to encode any value in the range $[1, n]$. Consequently, the space consumption of suffix arrays for texts shorter than 4 GB is $4n$ bytes.

[Weese, 2013] gives a generalization of Kärkkäinen and Sanders algorithm to construct the generalized suffix array in $\mathcal{O}(n)$ time. The space consumption of the generalized suffix array is $n \log cn^*$ bits, where $n^* = \max n_i$. For instance, the human genome GRCh38 is a collection of 24 sequences, among whose the largest one consists of 248 Mbp.

***Figure 4.2:*** *Generalized suffix array for the string collection* $\mathbb{S} = \{$ ANANAS$\$_1$, CACAO$\$_2$ $\}$.

| $i$ | $A[i]$ | $\mathbb{S}_{A[i]...}$ |
|---|---|---|
| 1 | (1,7) | $\$_1$ |
| 2 | (2,6) | $\$_2$ |
| 3 | (2,2) | ACAO$\$_2$ |
| 4 | (1,1) | ANANAS$\$_1$ |
| 5 | (1,3) | ANAS$\$_1$ |
| 6 | (2,4) | AO$\$_2$ |
| 7 | (1,5) | AS$\$_1$ |
| 8 | (2,1) | CACAO$\$_2$ |
| 9 | (2,3) | CAO$\$_2$ |
| 10 | (1,2) | NANAS$\$_1$ |
| 11 | (1,4) | NAS$\$_1$ |
| 12 | (2,5) | O$\$_2$ |
| 13 | (1,6) | S$\$_1$ |

Hence, to represent the generalized suffix array of GRCh38 we use pairs of $1 + 4$ bytes. Our suffix array of GRCh38 fits in 15 GB of memory and we are able to construct it in about one hour on a modern computer [Weese, 2013].

**Top-down traversal**

We now concentrate on implementing suffix trie functionalities. Any suffix trie node is univocally identified by an interval of the suffix array $A$. Thus, while traversing the trie, we maintain the interval $[l, r]$ associated to the pointed node. In addition, we also remember the depth $d$ of the pointed node. Recall from section 2.5.2 that any leaf node is defined by an interval containing only terminator symbols. The edge label entering any internal node at depth $d$ is defined by the $d$-th symbol in any suffix $\mathbb{S}_{A[i]}$ with $i \in [l, r]$. The occurrences below any node correspond by definition to the interval $[l, r]$ of $A$. Summing up, we represent the pointed node $x$ by the integers $\{l, r, d\}$ and define the following operations on it:

- ISLEAF$(x)$ returns true iff $A[r] + d = n_{A[r]}$;
- LABEL$(x)$ returns $\mathbb{S}_{A[l]+d}$;
- OCCURRENCES$(x)$ returns $A[l \dots r]$.

Binary search is the key to implement function GODOWN a symbol. Functions L (algorithm **??**) and R (algorithm **??**) compute in $\mathcal{O}(\log n)$ binary search steps the position in $A$ of the left and right interval corresponding to the child node following the edge labeled by a given symbol $c$. Note that line 6 of algorithms **??** and **??** may involve a comparison beyond the end of strings in $\mathbb{S}$, however we defined $t_i$ as the empty word $\epsilon$ if $i > |t|$ and $\epsilon <_{lex} c$ for all $c \in \Sigma$.

**Algorithm 4.1** L$(x, c)$

**Input** $x$ : pointer to a suffix array node
$c$ : character to query
**Output** integer denoting the left interval

1: $l_1 \leftarrow x.l$
2: $l_2 \leftarrow x.r$
3: **while** $l_1 < l_2$ **do**
4: $\quad i \leftarrow \lfloor \frac{l_1+l_2}{2} \rfloor$
5: $\quad$ **if** $\mathbb{S}_{A[i]+x.d} <_{lex} c$ **then**
6: $\quad\quad l_1 \leftarrow i + 1$
7: $\quad$ **else**
8: $\quad\quad l_2 \leftarrow i$
9: **return** $l_1$

**Algorithm 4.2** R$(x, c)$

**Input** $x$ : pointer to a suffix array node
$c$ : character to query
**Output** integer denoting the right interval

1: $r_1 \leftarrow x.l$
2: $r_2 \leftarrow x.r$
3: **while** $r_1 < r_2$ **do**
4: $\quad i \leftarrow \lfloor \frac{r_1+r_2}{2} \rfloor$
5: $\quad$ **if** $\mathbb{S}_{A[i]+x.d} \leq_{lex} c$ **then**
6: $\quad\quad r_1 \leftarrow i + 1$
7: $\quad$ **else**
8: $\quad\quad r_2 \leftarrow i$
9: **return** $r_1$

**Algorithm 4.3** goRoot$(x)$

**Input** $x$ : pointer to a suffix array node

1: $x \leftarrow \{0, 1, n\}$

**Algorithm 4.4** goDown$(x, c)$

**Input** $x$ : pointer to a suffix array node
$c$ : character to query
**Output** boolean indicating success

1: **if** isLeaf$(x)$ **then**
2: $\quad$ **return false**
3: $x.d \leftarrow x.d + 1$
4: $x.l \leftarrow$ L$(x, c)$
5: $x.r \leftarrow$ R$(x, c)$
6: **return** $x.l < x.r$

We now implement goDown and goRight to have time complexity independent of the alphabet size $\sigma$. Indeed, the time complexity of goDown and goRight is $\mathcal{O}(\log n)$, as they rely on a single call of using only R. In this way, the time complexity of exact string matching algorithm **??** is $\mathcal{O}(m \log n)$.

Note that we can iterate down the trie spelling a full pattern within a single call of L and R: in line 6, instead of comparing a single character, it suffices to compare the full pattern to the current suffix. However, the worst case runtime of algorithm **??** remains $\mathcal{O}(m \log n)$, as each step of the binary search now requires a full lexicographical comparison between the pattern and any suffix of the text, which is performed in $\mathcal{O}(m)$ time in the worst case. As shown in [Manber and Myers, 1990], we can decrease the worst case runtime to $\mathcal{O}(m + \log n)$ at the expense of additional $n \log n$ bits, by storing the precomputed longest common prefixes (LCP) between any two consecutive suffixes $\mathbb{S}_{A[i]\ldots}$, $\mathbb{S}_{A[i+1]\ldots}$ for all $1 \leq i < n$. Alternatively, we can reduce the *average case* runtime to $\mathcal{O}(m + \log n)$ without storing any additional information, by using the MLR heuristic [Manber and Myers, 1990]. In practice, the MLR heuristic outperforms the SA + LCP algorithm, due to the higher cost of fetching additional data from the LCP table.

**Algorithm 4.5** GODOWN($x$)

**Input** $x$ : pointer to a suffix array node
**Output** boolean indicating success

1: **if** ISLEAF($x$) **then**
2: **return false**
3: $x.d \leftarrow x.d + 1$
4: $x.l \leftarrow \mathrm{R}(x, \epsilon)$
5: $c_l \leftarrow \mathbb{S}_{A[x.l]+x.d}$
6: $c_r \leftarrow \mathbb{S}_{A[x.r]+x.d}$
7: **if** $c_l \neq c_r$ **then**
8: $x.r \leftarrow \mathrm{R}(x, c_l)$
9: **return** $x.l < x.r$

**Algorithm 4.6** GORIGHT($x$)

**Input** $x$ : pointer to a suffix array node
**Output** boolean indicating success

1: **if** ISROOT($x$) **then**
2: **return false**
3: $c_l \leftarrow \mathbb{S}_{A[x.l]+x.d}$
4: $c_r \leftarrow \mathbb{S}_{A[x.r]+x.d}$
5: **if** $c_l \neq c_r$ **then**
6: $x.l \leftarrow x.r$
7: $x.r \leftarrow \mathrm{R}(x, c_l)$
8: **return** $x.l < x.r$

### 4.1.2 $q$-Gram index

If we restrict the traversal of our idealized suffix trie to a fixed depth $q$, we can remove the logarithmic factor introduced by the suffix array. The idea is to supplement the suffix array $A$ with a so-called $q$-gram directory: an additional array $D$ of length $\Sigma^q + 1$, storing the suffix array ranges computed by algorithm **??** for any possible word of length $q$.

With the aim of addressing $q$-grams in the directory $D$, we impose a canonical code on $q$-grams through a bijective function $h : \Sigma^q \to [1 \dots \sigma^q]$ defined as in [Knuth, 1973]:

$$h(p) = 1 + \sum_{i=1}^{q} \rho_0(p_i) \cdot \sigma^{q-i} \tag{4.1}$$

where $p \in \Sigma^q$ is any $q$-gram and $\rho_0$ is the zero-based lexicographic rank defined on $\Sigma$ (recall the lexicographic rank function $\rho$ from definition 2.4 and pose $\rho_0(x) = \rho(x) - 1$). This allows us to store in and retrieve from $D[h(p)]$, for each $q$-gram $p \in \Sigma^q$, the left suffix array interval returned by algorithm **??**, i.e. $D[h(p)] = \mathrm{L}(1, n, p)$. Note that the right interval returned by algorithm **??** is equivalent to the left interval of the lexicographical successor $q$-gram and therefore available in $D[h(p) + 1]$.

The downside of the $q$-gram index is that in practice it is only applicable to small alphabet and pattern sizes. For instance, parameters $|\Sigma| = 4$ and $q = 14$ require a $q$-gram directory consisting of 268 M entries that, using a 32 bits encoding, consume 1 GB of memory.

#### Top-down traversal

We now extend suffix array traversal operations to use the $q$-gram directory $D$. Again, we maintain the current range $[l, r]$ and the current depth $d$. In addition, while traversing the trie, we maintain the interval $[l_h, r_h]$ in $D$ and, in order to answer LABEL($x$), the label $e$ of the edge entering the current node. Summing up, we represent the pointed node $x$ by the elements $\{l, r, d, l_h, r_h, e\}$. We define the basic node operations as follows:

***Figure 4.3:*** *q-Gram index for the string ANANAS over the alphabet* $\Sigma = \{A, N, S\}$.

| $p$ | $h$ | $D[h]$ | $i$ | $A[i]$ | $s_{A[i]\ldots n}$ |
|---|---|---|---|---|---|
| AA | 1 | 2 | 1 | 7 | $ |
| AN | 2 | 2 | 2 | 1 | ANANAS$ |
| AS | 3 | 4 | 3 | 3 | ANAS$ |
| NA | 4 | 5 | 4 | 5 | AS$ |
| NN | 5 | 6 | 5 | 2 | NANAS$ |
| NS | 6 | 6 | 6 | 4 | NAS$ |
| SA | 7 | 6 | 7 | 6 | S$ |
| SN | 8 | 6 | | | |
| SS | 9 | 6 | | | |
| | 10 | 6 | | | |

- ISLEAF$(x)$ returns true iff $d = q$;
- LABEL$(x)$ returns $e$;
- OCCURRENCES$(x)$ returns $A[l \ldots r]$.

We define functions L and R to use the directory $D$ instead of $A$ and obtain GODOWN in $\mathcal{O}(1)$. Note how the canonical code assigned by $h$ preserves the lexicographical ordering for all words not longer than $q$, i.e. $v <_{lex} w$ iff $h(v) < h(w)$ for all $v, w \in \Sigma^{\leq q}$.

```
1: procedure L(x → {l_h, d}, c)
2:     l_h ← l_h + ρ_0(c) · σ^d
3:     return D[l_h]
```

```
1: procedure R(x → {r_h, d}, c)
2:     r_h ← r_h − ρ_0(c) · σ^d
3:     return D[r_h]
```

```
1: procedure GOROOT(x → {l, r, d, l_h, r_h})
2:     d ← 0
3:     l ← 1
4:     r ← n
5:     l_h ← 1
6:     r_h ← σ^q
```

```
1: procedure GODOWN(x → {l, r, d, e}, c)
2:     if ISLEAF(x) then
3:         return false
4:     d ← d + 1
5:     e ← c
6:     l ← L(x, e)
7:     r ← R(x, e)
8:     return l < r
```

Exact string matching algorithm **??** runs in time $\mathcal{O}(q)$, nonetheless we can improve it to perform $\mathcal{O}(1)$ lookups in $D$. It suffices to compute the canonical code of the pattern in one step, as shown in algorithm 4.7.

If the patterns are shorter or equal to the fixed length $q$, we access the suffix array only to locate the occurrences, as the directory $D$ alone is sufficient to count the occurrences. In this case, the total ordering of the text suffixes in the suffix array can be relaxed to prefixes of length $q$. This gives us a twofold advantage, as we can: (i) construct the suffix array more efficiently using bucket sorting and (ii) maintain leaves in each bucket

**Algorithm 4.7** Exact string matching on a $q$-gram index.

```
1: procedure ExactSearch(t, p)
2:     l ← D[h(p)]
3:     r ← D[h(p) + 1]
4:     report A[l ... r]
```

sorted by their relative text positions. The latter property allows to compress the suffix array bucket-wise e.g. using Elias $\delta$ encoding [Elias, 1975] or to devise cache-oblivious strategies to process the occurrences [Hach *et al.*, 2010].

If the patterns are longer than $q$, the $q$-gram index is still usable. We can devise an hybrid algorithm that uses the directory $D$ to conduct the search up to depth $q$ and later continue with binary searches on the suffix array. This hybrid index cuts the most expensive binary searches and increases memory locality. Furthermore, this hybrid index becomes useful whenever the suffix array is too big to fit in main memory and has to reside in external memory.

## 4.2 FM-indices

The Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994] is a transformation defining a permutation of an input string. The transformed string exposes two important properties: reversibility and compressibility. The former property allows us to reconstruct the original string from its BWT, the latter property makes the transformed string more amenable to compression. Thanks to these two properties, the BWT has been recognized as a fundamental method for text compression and practically used in the bzip2 [Seward, 1996] tool.

More recently, Ferragina and Manzini proposed the BWT as a tool for full-text indexing. They showed in [Ferragina and Manzini, 2000] that the BWT alone allows to perform exact pattern matching and engineered in [Ferragina and Manzini, 2001] a compressed full-text index called FM-index. Over the last years, the FM-index has been widely employed under different re-implementations by many popular Bioinformatics tools e.g. Bowtie [Langmead *et al.*, 2009] and BWA [Li and Durbin, 2009], and is now considered a fundamental method for the indexing of genomic sequences.

In the next subsections we give the fundamental ideas behind the BWT and the FM-index. Subsequently, we discuss our succinct FM-index implementations covering strings and string collections.

### 4.2.1 Burrows-Wheeler transform

Let $s$ be a padded string (definition 2.8) of length $n$ over an alphabet $\Sigma$. In the following we consider the string $s$ to be cyclic and its subscript $s_i$ to be *modular*, e.g. $s_0 = s_n$ and $s_{n+i} = s_i$ for any $i \in \mathbb{N}$. Consider the square matrix consisting of all cyclic shifts of the string $s$ (the $i$-th cyclic shift has the form $s_{i \dots n} s_{1 \dots i-1}$) sorted in lexicographical order (see figure 4.4). For convenience, we denote by $f$ the first column $s_{A[i]}$, and by $l$ the last column

$s_{A[i]-1}$. Note how the cyclic shifts matrix is related to the suffix array $A$ of $s$: the $i$-th cyclic shift is $s_{A[i]\ldots n}s_{1\ldots A[i]-1}$.

**Definition 4.3.** The BWT of $s$ is the string obtained concatenating the symbols in the last column of the cyclic shifts matrix of $s$, i.e. it is the string $l = s_{A[i]-1}$.

***Figure 4.4:*** *Burrows-Wheeler transform for the string* ANANAS$.

| $i$ | $A[i]$ | $s_{A[i]}$ | | $s_{A[i]-1}$ |
|---|---|---|---|---|
| 1 | 7 | $ | ANANA | S |
| 2 | 1 | A | NANAS | $ |
| 3 | 3 | A | NAS$A | N |
| 4 | 5 | A | S$ANA | N |
| 5 | 2 | N | ANAS$ | A |
| 6 | 4 | N | AS$AN | A |
| 7 | 6 | S | $ANAN | A |

We easily generalize the BWT to string collections. Indeed, definition 4.3 still holds if we consider a padded string collection $\mathbb{S}$ (definition 2.9) and its cyclic shifts matrix sorted in lexicographical order.

The cyclic shifts matrix is conceptual and we do not have to construct it explicitly to derive the BWT of a text. We can obtain the BWT in linear time by scanning the suffix array $A$ and assigning to the $i$-th BWT symbol the text character $s_{A[i]-1}$. Various direct BWT construction algorithms working within $o(n)$ bits plus constant space have been recently proposed [Bauer *et al.*, 2013; Crochemore *et al.*, 2013], as constructing the suffix array is not desirable due to its space consumption of $n \log n$ bits.

### Inversion

We now describe how to invert the BWT to reconstruct the original text. Inverting the BWT means being able to know where any BWT character occurs in the original text. To this extent, we introduce two permutations $LF : [1,n] \rightarrow [1,n]$ and $\Psi : [1,n] \rightarrow [1,n]$, with $LF = \Psi^{-1}$, where the value of $LF(i)$ gives the position $j$ in $f$ where character $l_i$ occurs and the value $\Psi(j)$ gives back the position $i$ in $l$ where $f_j$ occurs. Figure 4.5 illustrates.

We recover $s_i$ as $f_{\Psi^{i-1}(j)}$, where the iterated $\Psi$ is defined as

$$\begin{array}{rcl} \Psi^0(j) & = & j \\ \Psi^{i+1}(j) & = & \Psi(\Psi^i(j)) \end{array} \tag{4.2}$$

Conversely, we recover $\bar{s}_i$ as $s_{LF^{i-1}(j)}$, where the iterated $LF$ is defined as

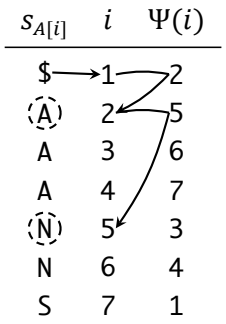
$$\begin{array}{rcl} LF^0(j) & = & j \\ LF^{i+1}(j) & = & LF(LF^i(j)) \end{array} \tag{4.3}$$

***Figure 4.5:*** *Functions LF and Ψ for the string* ANANAS$. *Note how* $LF(\Psi(i)) = \Psi(LF(i)) = i$ *for any* $i$. *Also note how the* $i$*-th occurrence of any character* $c$ *in* $l$ *is the* $i$*-th occurrence of* $c$ *in* $f$.

| $i$ | $\Psi(i)$ | $LF(i)$ | $s_{A[i]}$ | | $s_{A[i]-1}$ |
|---|---|---|---|---|---|
| 1 | 2 | 7 | $ | ANANA | S |
| 2 | 5 | 1 | A | NANAS | $ |
| 3 | 6 | (5) | A | NAS$A | (N) |
| 4 | 7 | 6 | A | S$ANA | N |
| 5 | (3) | 2 | (N) | ANAS$ | A |
| 6 | 4 | 3 | N | AS$AN | A |
| 7 | 1 | 4 | S | $ANAN | A |

We recover the string $s$ by starting in $f$ at the position of \$ and following the cycle defined by the permutation Ψ. Or we recover the reverse string $\bar{s}$ by starting in $l$ at the position of \$ and following the cycle defined by the permutation $LF$.

***Figure 4.6:*** *Recovering the string* ANANAS$ *from the permutation* Ψ. *Here we depict only the first two steps of the inversion recovering* AN.




Inverting the generalized BWT works in the same way. Indeed, permutations Ψ and $LF$ are composed of $c$ cycles, where each cycle corresponds to a distinct string in the collection. We recover $s^i$ by starting at the position of $\$^i$ and following the cycle of Ψ (or $LF$) associated to $s^i$.

### Permutation LF

The permutation $LF$ is conceptual: we do not have to represent it explicitly but we can deduce it from the BWT $l$ with the help of some additional character counts. This is possible due to two simple observations on the cyclic shifts matrix [Burrows and Wheeler, 1994]:

- for all $i \in [1, n]$, the character $l_i$ precedes the character $f_i$ in the original string $s$;

- for all characters $c \in \Sigma$ the $i$-th occurrence of $c$ in $f$ corresponds to the $i$-th occurrence of $c$ in $l$.

These observation are obvious since $f = s_{A[i]}$ and $l = s_{A[i]-1}$ (see figure 4.4). Given the above observations, we define the permutation $LF$ as [Burrows and Wheeler, 1994; Ferragina and Manzini, 2000]:

$$LF(i) = C(l_i) + Occ(l_i, i) \tag{4.4}$$

where we denote with $C : \Sigma \to [1, n]$ the total number of occurrences in $s$ of all characters alphabetically smaller than $c$, and with $Occ : \Sigma \times [1, n] \to [1, n]$ the number of occurrences of character $c$ in the prefix $l_{1...i}$.

The key problem of representing the permutation $LF$ is how to represent function $Occ$, as function $C$ can be easily tabulated by a small array of size $\sigma \log n$ bits. In the next subsection we address the problem of representing function $Occ$ efficiently. Subsequently, in subsection 4.2.3 we see how to build a full-text index out of function $LF$.

### 4.2.2 Rank dictionaries

We want to answer efficiently the question "how many times a given character $c$ occurs in the prefix $l_{1...i}$?", ideally in constant time and linear space. The general problem on arbitrary strings has been tackled by several studies on the succinct representation of data structures [Jacobson, 1989]. Our specific question takes the name of *rank query* and a data structure answering rank queries is called *rank dictionary*.

**Definition 4.4.** Given a string $s$ over an alphabet $\Sigma$ and a character $c \in \Sigma$, $\text{rank}_c(s, i)$ returns the number of occurrences of $c$ in the prefix $s_{1...i}$.

The key idea of rank dictionaries is to maintain a succinct (or even compressed) representation of the input string and attach a dictionary to it. By doing so, Jacobson showed how it is possible to answer rank queries in constant time (on the RAM model) using $n + o(n)$ bits for an input binary string of $n$ bits [Jacobson, 1989]. First we consider the binary case $\Sigma_B = \{0, 1\}$ and later we extend it to arbitrary alphabets. In addition, we discuss implementation details that are crucial to obtain practical efficiency.

#### Binary alphabet

We start by describing a simple rank dictionary answering rank queries in constant time but consuming $2n$ bits and later we extend it to consume only $n + o(n)$ bits. Given the binary string $s \in \Sigma_B$, we partition it in blocks of size $b$. In the following we assume w.l.o.g. the string length $n$ to be a multiple of $b$, conversely values in all subsequent formulas must be rounded accordingly.

We attach to the binary string $s$ an array $R$ of length $\frac{n}{b}$, where the $i$-th entry gives a summary of the number of occurrences of the bit 1 in $s_{1...ib}$, i.e. $R[\frac{i}{b}] = \text{rank}_1(s, \frac{i}{b})$. Note

that $\text{rank}_0(s, i) = i - \text{rank}_1(s, i)$ so we consider only $\text{rank}_1(s, i)$. Therefore we are able to rewrite our rank query as:

$$\text{rank}_1(s, i) = R[\frac{i}{b}] + \text{rank}_1(s_{\frac{i}{b} \cdots \frac{i}{b}+b}, i \mod b) \tag{4.5}$$

and answer it in time $\mathcal{O}(b)$ as (i) we fetch in constant time the rank summary from $R$ and (ii) we compute the number of occurrences of the bit 1 within a block of length $b$. If we pose $b = \log n$, we can compute step ii in $\mathcal{O}(1)$ by means of the four-Russians tabulation technique [**?**]. Moreover, the array $R$ stores $\frac{n}{\log n}$ positions and each position in $s$ requires $\log n$ bits, so $R$ consumes $n$ bits. Thus, this rank dictionary consumes $2n$ bits.

***Figure 4.7:*** *Example of rank dictionaries for the string $s =$ 010101100100. On the one-level rank dictionary, $b = 4$ and $rank_1(s, 6) = R[2] + rank_1(s_{5\ldots8}, 2) = 3$. On the two-levels rank dictionary, $b = 2$ and $rank_1(s, 6) = R^2[2] + R[3] + rank_1(s_{6\ldots6}, 1) = 3$.*

| $i$ | $s_i$ | $R[\frac{i}{b}]$ |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 1 | |
| 3 | 0 | |
| 4 | 1 | |
| 5 | 0 | 2 |
| 6 | 1 | |
| 7 | 1 | |
| 8 | 0 | |
| 9 | 0 | 4 |
| 10 | 1 | |
| 11 | 0 | |
| 12 | 0 | |

| $i$ | $s_i$ | $R[\frac{i}{b}]$ | $R^2[\frac{i}{b^2}]$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 1 | | |
| 3 | 0 | 1 | |
| 4 | 1 | | |
| 5 | 0 | 0 | 2 |
| 6 | 1 | | |
| 7 | 1 | 1 | |
| 8 | 0 | | |
| 9 | 0 | 0 | 4 |
| 10 | 1 | | |
| 11 | 0 | 1 | |
| 12 | 0 | | |

To squeeze our rank dictionary to consume only $n + o(n)$ bits of space, we add another array $R^2$ summarizing the ranks on $b^2$ bits boundaries and let our initial array $R$ store only local positions within the corresponding blocks defined by $R^2$. We rewrite $\text{rank}_1(s, i)$ accordingly:

$$\text{rank}_1(s, i) = R^2[\frac{i}{b^2}] + R[\frac{i}{b}] + \text{rank}_1(s_{\frac{i}{b} \cdots \frac{i}{b}+b}, i \mod b) \tag{4.6}$$

Each entry of $R$ now has to represent only $b^2$ possible values and thus consumes only $2 \log b$ bits. Summing up, this two-levels rank dictionary consumes $n$ bits for the input string, $\mathcal{O}(\frac{n \log n}{b^2})$ bits for $R^2$ and $\mathcal{O}(\frac{n \log b}{b})$ bits for $R$. By posing $b = \log n$ we have $\mathcal{O}(\frac{n}{\log n})$ bits for $R^2$ and $\mathcal{O}(\frac{n \log \log n}{\log n})$ bits for $R$. Hence, the two-levels rank dictionary consumes $n + o(n)$ bits.

We implemented generic multi-levels rank dictionaries where the block size $b$ is a template parameter adjustable at compile time. Whenever the input string has length

inferior to 4 GB, we employ one-level rank dictionaries with $b = 32$ bits or two-levels rank dictionaries with $b = 16$ bits and $b^2 = 32$ bits; otherwise, we employ two-levels rank dictionaries with $b = 32$ bits and $b^2 = 64$ bits, or three-levels rank dictionaries with $b = 16$ bits, $b^2 = 32$ bits and $b^3 = 64$ bits. In order to reduce the number of cache misses, we interleaved the succinct representation of the input string with the lowest level summaries array $R$. Moreover, we compute step ii in $\mathcal{O}(1)$ using the SSE 4.2 popcnt instruction [Intel, 2011].

### Small alphabets

The extension to small alphabets e.g. $\Sigma_{\text{DNA}}$ is easy. Here we show how to extend the one-level rank dictionary. Given an input string $s$ over $\Sigma_{\text{DNA}}$ of size $n$ bits (and thus of length $\frac{n}{\log \sigma}$ symbols), we partition it in blocks of $b$ symbols (which do not correspond to $b$ bits as in the $\Sigma_B$ case). We supplement each block of $s$ with an occurrences summary for all symbols in $\Sigma$, thus we use a matrix $R_\sigma$ of size $\frac{n}{b} \times \sigma$ entries. We rewrite $\text{rank}_c(s, i)$ as:

$$\text{rank}_c(s, i) = R_\sigma[\frac{i}{b}][\rho(c)] + \text{rank}_c(s_{\frac{i}{b} \cdots \frac{i}{b}+b}, i \mod b) \tag{4.7}$$

where $i$ now is the $i$-th symbol in $s$, not the $i$-th bit as in the $\Sigma_B$ case.

In order to answer rank queries in constant time, we have to count the number of occurrences of the character $c$ inside a block of length $b$. We pose $b = \frac{\log n}{\log \sigma}$ symbols, thus $b = \log n$ bits as in the $\Sigma_B$ case. The matrix $R_\sigma$ has $\frac{n}{\log n}$ entries, each one consuming $\sigma \log n$ bits. Thus $R_\sigma$ consumes $n\sigma$ bits, and the whole rank dictionary $n + n\sigma$ bits.

***Figure 4.8:*** *Example of rank dictionaries for the string* $s =$ `CCCGCA` *under the binary encoding* $\{A, C, G, T\} = \{00, 01, 10, 11\}$ *of the alphabet* $\Sigma_{DNA}$.

| $i$ | $s_{\frac{i}{2}}$ | $s_i$ | $R_\sigma[\frac{i}{b}]$ | $\sigma$ |
|---|---|---|---|---|
| 1 | C | 0 | 0 | A |
| 2 | | 1 | 0 | C |
| 3 | C | 0 | 0 | G |
| 4 | | 1 | 0 | T |
| 5 | C | 0 | 0 | A |
| 6 | | 1 | 2 | C |
| 7 | G | 1 | 0 | G |
| 8 | | 0 | 0 | T |
| 9 | C | 0 | 0 | A |
| 10 | | 1 | 3 | C |
| 11 | A | 0 | 1 | G |
| 12 | | 0 | 0 | T |

### 4.2.3 Top-down traversal

We now turn to the problem of implementing a full-text index relying on the permutation $LF$. This is possible because of the relationship between the cyclic shifts matrix and the suffix array $A$. First we see how to emulate a top-down traversal of the suffix trie, which is sufficient to count the number of occurrences of any substring in the original text. Later we focus on how to represent the leaves, which are necessary to locate occurrences in the original text.

We represent the pointed node $x$ by the elements $\{l, r, e\}$, where $[l, r]$ represents the current suffix array interval and $e$ is the label of the edge entering the current node.

- ISLEAF$(x)$ returns true iff $l_r = \$$;
- LABEL$(x)$ returns $e$.

Given a padded string collection $\mathbb{S}$, its BWT plus its associated permutation $LF$ let us recover any substring of $\bar{\mathbb{S}}$. Nonetheless, we want to recover substrings of $\mathbb{S}$ in forward direction, as the suffix trie $\mathcal{S}$ spells all forward substrings of $\mathbb{S}$. Therefore, given a string $\mathbb{S}$ we consider the BWT of $\bar{\mathbb{S}}$, such that $LF$ let us recover any substring of $\mathbb{S}$. Now we can use the permutation $LF$ to decode the intervals computed during a suffix trie traversal.

From the root node, we easily obtain the interval of its child node labeled by $c$ as $[C(c), C(c+1)]$. Now suppose an arbitrary node $v$ of known interval $[b_v, e_v]$ s.t. the path from the root to $v$ spells the substring $s_v$; we want to reach the node $w$ of unknown interval $[b_w, e_w]$ s.t. the path from the root to $w$ spells $c \cdot s_v$ for some $c \in \Sigma$. Thus, we know all prefixes of $\bar{\mathbb{S}}$ ending with $s_v$ (hence all suffixes of $\mathbb{S}$ starting with $s_v$) and we are looking for all the prefixes of $\bar{\mathbb{S}}$ ending with $c \cdot s_v$ (hence all suffixes of $\mathbb{S}$ starting with $s_v \cdot c$). All these characters $c$ are in $l_{b_v \ldots e_v}$, since $l_i$ is the character $\mathbb{S}_{A[i]-1}$ preceding the suffix pointed by $A[i]$. Moreover, we know that these characters $c$ are (i) contiguous and (ii) in relative order in $f$ [Ferragina and Manzini, 2000]. If $b$ and $e$ are the first and last position in $l$ within $[b_v, e_v]$ such that $l_b = c$ and $l_e = c$, then $b_w = LF(b)$ and $e_w = LF(e)$. We can rewrite $LF(b)$ as:

$$\begin{aligned} LF(b) &= C(l_b) + Occ(l_b, b) \\ &= C(c) + Occ(c, b) \\ &= C(c) + Occ(c, b_v - 1) + 1 \end{aligned} \tag{4.8}$$

Analogously, we can rewrite $LF(e)$ as $C(c) + Occ(c, e_v)$. Therefore, GODOWN a symbol computes two values of permutation $LF$ and runs in time $\mathcal{O}(1)$. Conversely, GODOWN and GORIGHT run in time $\mathcal{O}(\sigma)$ (see algorithm **??**).

In order to locate occurrences we need the suffix array $A$, yet we do not want to maintain the whole array $A$. As proposed by Ferragina and Manzini, we maintain a *sampled* suffix array $A^\epsilon$ containing positions sampled at regular intervals in the input string. In order to determine if and where we sampled any $A[i]$ in $A^\epsilon$, we maintain a binary rank dictionary $S$ of length $n$: if $S[i] = 1$, then we sampled $A[i]$ in $A^\epsilon[rank_1(S, i)]$. We obtain any $A[i]$ by finding the smallest $j \geq 0$ such that $LF^j(i)$ is in $A^\epsilon$, and then $A[i] = A[LF^j(i)] + j$.

If we sample one text position out of $\log^{1+\epsilon} n$, for some $\epsilon > 0$, then $A^\epsilon$ consumes $\mathcal{O}(\frac{n}{\log^\epsilon n})$ space and OCCURRENCES$(x)$ returns all occurrences in $\mathcal{O}(o \cdot \log^{1+\epsilon} n)$ time [Fer-

**Algorithm 4.8** GOROOT($x$)

**Input** $x$ : pointer to a FM-index node

1: $x.l \leftarrow 1$
2: $x.r \leftarrow n$

**Algorithm 4.9** GODOWN($x, c$)

**Input** $x$ : pointer to a FM-index node
$c$ : char to query
**Output** boolean indicating success

1: **if** ISLEAF($x$) **then**
2: **return false**
3: $x.l \leftarrow \mathrm{LF}(x.l, c)$
4: $x.r \leftarrow \mathrm{LF}(x.r, c)$
5: $x.e \leftarrow c$
6: **return** $x.l < x.r$

ragina and Manzini, 2000]. In practice, we usually sample text positions at rates between $2^{-3}$ and $2^{-5}$. The rank dictionary $S$ consumes always $n + o(n)$ extra space.

## 4.3 Algorithms

### 4.3.1 Bounded top-down traversal

**Algorithm 4.10** DFS($x, d$)

**Input** $x$ : pointer to the root node of the suffix trie
$d$ : integer bounding the traversal depth

1: **if** $d > 0$ **then**
2: **if** GODOWN($x$) **then**
3: **repeat**
4: DFS($x, d - 1$)
5: **until** GORIGHT($x$)

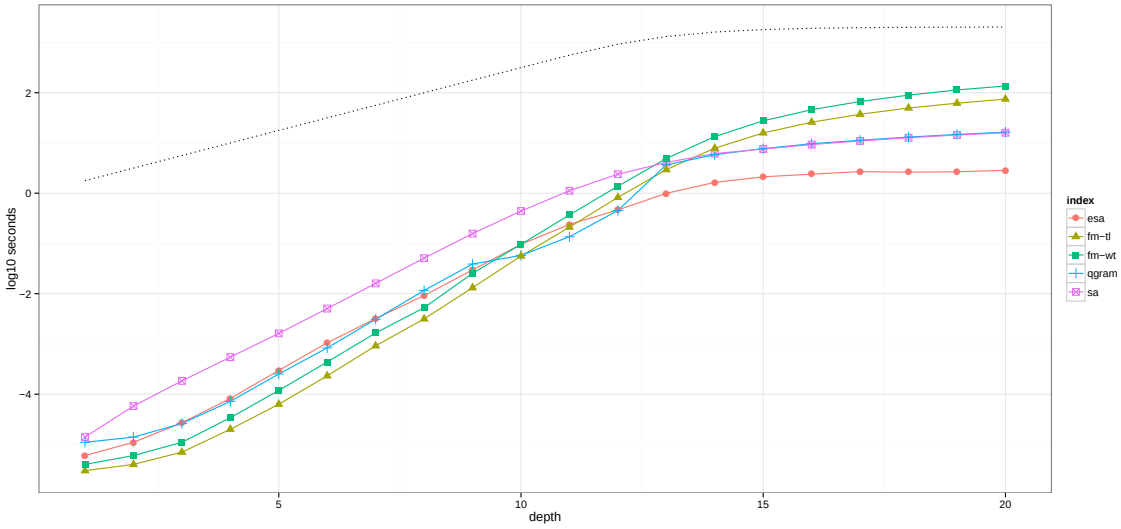
### 4.3.2 Exact pattern matching

We now benchmark algorithm **??**.

### 4.3.3 Backtracking $k$-mismatches

We can solve $k$-mismatches by backtracking [Ukkonen, 1993; Baeza-Yates and Gonnet, 1999] on the suffix trie $\mathcal{T}$, in average time sublinear in $n$ [Navarro and Baeza-Yates, 2000]. A top-down traversal on $\mathcal{T}$ spells incrementally all distinct substrings of $t$. While traversing each branch of the trie, we incrementally compute the distance between the query and the spelled string. If the computed distance exceeds $k$, we stop the traversal and proceed on the next branch. Conversely, if we completely spelled the pattern $p$ and we ended

***Figure 4.9:*** *Runtime of bounded top-down traversal of various suffix trie implementations.*




up in a node $x$, each leaf $l_i \in \mathbb{L}(x)$ points to a distinct suffix $t_{i..n}$ such that $d_H(t_{i..i+m}, p) \leq k$. See algorithm **??**.

**Algorithm 4.11** KMISMATCHES$(t, p, e)$

**Input** $t$ : pointer to the root node of the suffix trie of the text
$p$ : pointer to the root node of the trie of the patterns
$e$ : integer bounding the number of mismatches
**Output** list of all occurrences of the pattern in the text

```
1:  if e = k then
2:      ExactSearch(t, p)
3:  else if e < k then
4:      if atEnd(p) then
5:          report occurrences(t)
6:      else if goDown(t) then
7:          repeat
8:              d ← ω(label(t), value(p))
9:              goNext(p)
10:             KMismatches(t, p, e + d)
11:             goPrevious(p)
12:         until goRight(t)
```

### 4.3.4 Backtracking $k$-differences

Algorithm for $k$-differences.

**Algorithm 4.12** KDIFFERENCES$(t, p, D)$

**Input** $t$ : pointer to the root node of the suffix trie of the text
$p$ : pointer to the root node of the trie of the patterns
$D$ : vector of integers
**Output** list of all occurrences of the pattern in the text

```
1: if D[m] ≤ k then
2:     report OCCURRENCES(t)
3: else if minD ≤ k then
4:     if GODOWN(t) then
5:         repeat
6:             D' ← DP(D, LABEL(t), p)
7:             KDIFFERENCES(t, p, D')
8:         until GORIGHT(t)
```

We can compute $k$-differences on a suffix tree in two different ways. Algorithm **??** explicitly enumerates errors by recursing on the suffix trie. Algorithm **??** computes the edit distance on the suffix trie.

In algorithm **??**, any online method can be used to compute the edit distance. However, for theoretical considerations, it is important to consider an algorithm which computes in $\mathcal{O}(1)$ per node. Note that it is sufficient to have an algorithm capable of checking whether the current edit distance is within the imposed threshold $k$.

Algorithm **??** reports more occurrences than algorithm **??**. Discuss neighborhood, condensed neighborhood, and super-condensed neighborhood.

***Figure 4.10:*** *Runtime of exact string matching on various suffix trie implementations.*

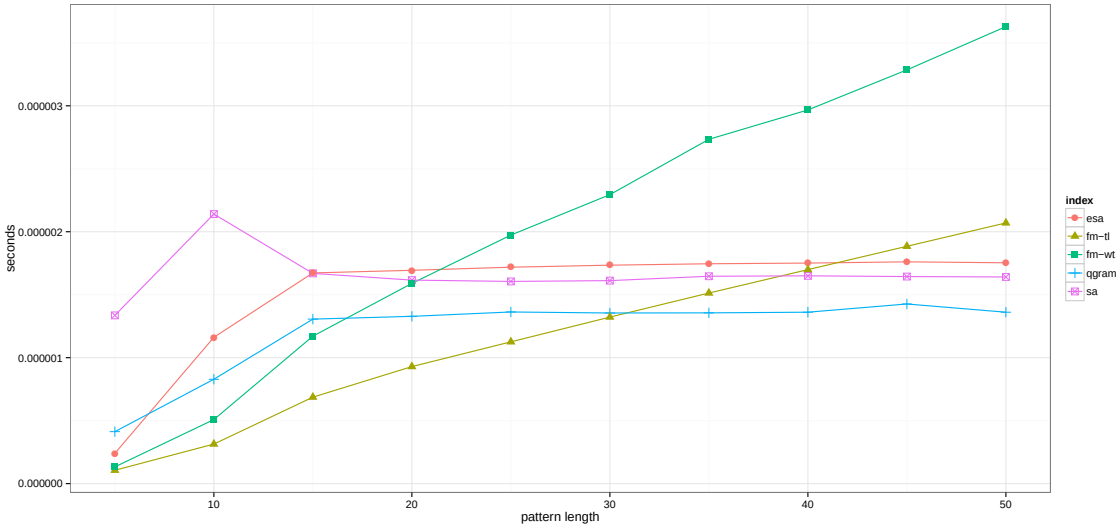


***Figure 4.11:*** *Approximate string matching on a suffix trie.*

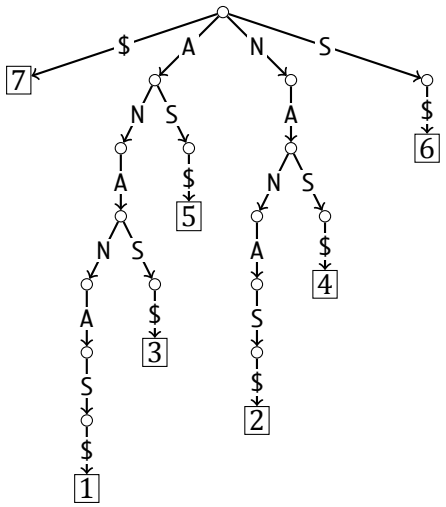
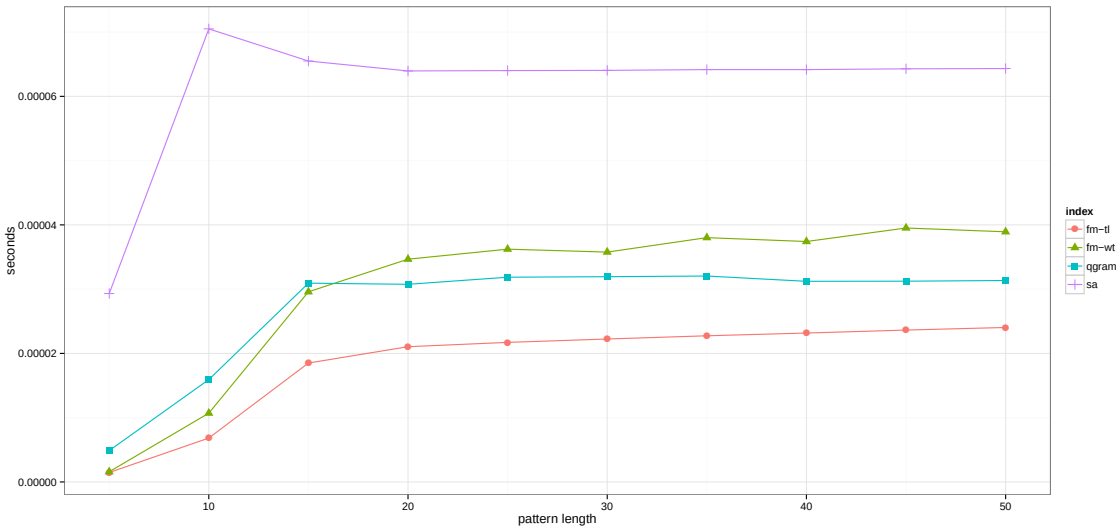



***Figure 4.12:*** *Runtime of k-mismatches on various suffix trie implementations.*




### 4.3.5 Multiple backtracking

We start by giving the simple exact string matching algorithm using suffix trie iterators.

**Algorithm 4.13** MULTIPLEEXACTSEARCH$(t, p)$

**Input** $t$ : pointer to the root node of the suffix trie of the text
$p$ : pointer to the root node of the trie of the patterns
**Output** list of all occurrences of any pattern in the text

```
1: if isLeaf(p) then
2:     report occurrences(t) × occurrences(p)
3: else
4:     goDown(p)
5:     repeat
6:         if goDown(t, label(p)) then
7:             MultipleExactSearch(t, p)
8:             goUp(t)
9:     until goRight(p)
```

**Algorithm 4.14** MULTIPLEKMISMATCHES$(t, p, e)$

**Input** $t$ : pointer to the root node of the suffix trie of the text
$p$ : pointer to the root node of the trie of the patterns
$e$ : integer bounding the number of mismatches
**Output** list of all occurrences of any pattern in the text

```
1: if e = k then
2:     MultipleExactSearch(t, p)
3: else if e < k then
4:     if isLeaf(p) then
5:         report occurrences(t) × occurrences(p)
6:     else if goDown(t) then
7:         repeat
8:             goDown(p)
9:             repeat
10:                d ← ω(label(t), label(p))
11:                MultipleKMismatches(t, p, e + d)
12:            until goRight(p)
13:            goUp(p)
14:        until goRight(t)
```

***Figure 4.13:*** *Runtime of multiple exact string matching on various suffix trie implementations.*

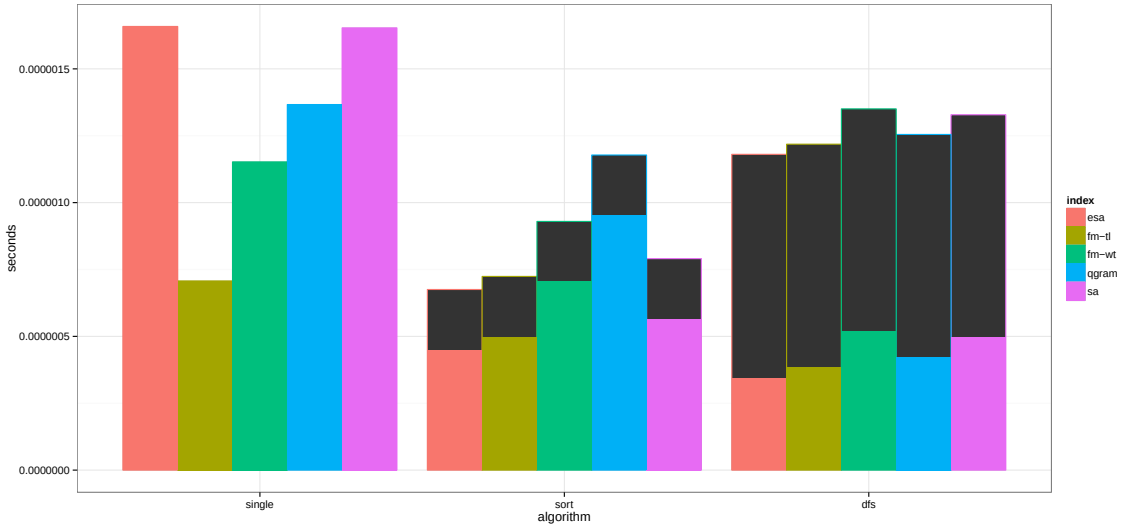
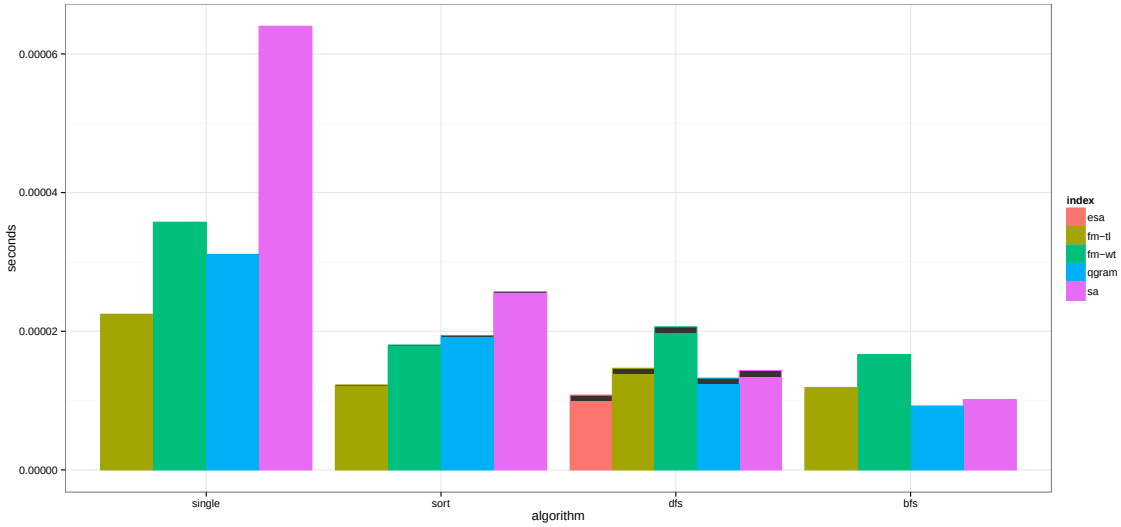



***Figure 4.14:*** *Runtime of multiple k-mismatches on various suffix trie implementations.*



CHAPTER

# 5 Filtering Methods

We now concentrate on designing stronger filters for approximate string matching. Whenever the total time of our filtering algorithm is dominated by verification time, we can think of adopting a stronger filter yielding a lower number of candidate locations to verify. The idea is to improve the total time by means of a significant reduction of verification time, at the expense of a moderate increase in filtration time.

We have already seen in section 2.5.3 an example of such trade-off between filtration speed and strength: non-overlapping seeds versus $q$-gram filters. The former filters minimize filtration time but yield weak filtration, the latter provide a more involved but stronger filtration. In practice, we know that seeds filters work best at low error rates, while $q$-gram filters become competitive at higher error rates. Thus we would adopt one or the other filtration method depending on the requested error rate.

The simple analysis of section 2.5.3 show us that filtration specificity is strongly correlated to the parameter $q$ being the seed or $q$-gram length. Therefore the crux of designing stronger filters lies into increasing the seed or $q$-gram length while guaranteeing full-sensitivity. To this intent, the following techniques have been proposed in literature: (i) gapped $q$-grams generalizing contiguous $q$-grams, (ii) multiple gapped $q$-grams further generalizing gapped $q$-grams; (iii) approximate seeds generalizing exact seeds,

The idea of gapped $q$-grams is to lower the correlation between consecutive $q$-grams. We have seen in the $q$-gram lemma 2.2 that one error covers $q$ consecutive $q$-grams. Thus, the occurrence of a $q$-gram is strongly correlated to the occurrence of its preceding and following $q$-grams. Therefore, we introduce *don't care positions* to skip characters at fixed positions inside all $q$-grams. By doing so, a mismatch does not cover anymore all consecutive $q$-grams. As less $q$-grams are now covered by mismatches, we can (i) increase the filtering threshold or (ii) increase the number $q$ of considered characters, in either cases preserving full-sensitivity. Note that however, this concept poorly extends to edit distance, and the design of gapped $q$-grams is hard.

TODO families.

The idea of approximate seeds is that, to increase seed length, we can factorize the pattern in less than $k + 1$ non-overlapping seeds. The pigeonhole lemma 2.1 however tells us that, by doing so, no seed will be guaranteed to be error-free. Therefore, we are obliged to search the seeds approximately, but within a distance threshold smaller than the original threshold $k$. We can search approximate seeds with backtracking techniques of section **??**. In a nutshell, instead of reducing an approximate search into smaller exact searches, we reduce it into smaller approximate searches.

Problems of exact and approximate seeds filters are that (i) it is not evident which factorization yields optimal filtration, and (ii) they yield duplicate occurrences whenever errors are not distributed in the worst-case combination. Suffix filters are based on tighter pigeonhole lemmas yielding stronger, less redundant, and more adaptive filtration using more efficiently the index. The drawback is that the effort to implement them is slightly higher.

In the following of this chapter we first present (multiple) gapped $q$-grams and problems associated with their design. Then we move to more practical approximate seeds, which we will adopt later in chapter **??**. Overall, through this chapter:

- we present a framework for the design of (multiple) gapped $q$-grams consisting of efficient exact and approximate solutions;
- we provide generic parallel implementations of filters based on exact and approximate seeds;
- we evaluate these filtration schemes in practice.

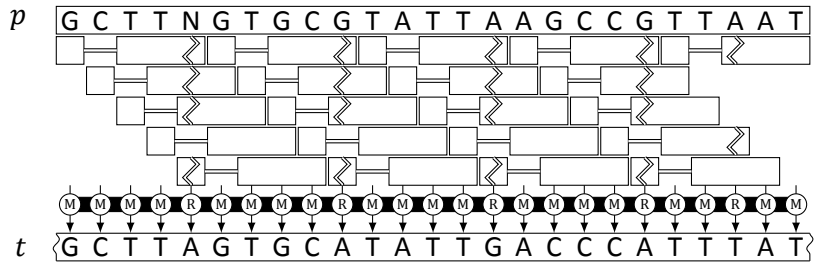
## 5.1 Gapped $q$-grams

In section 2.5.3 we introduced the $q$-gram similarity measure as the number of substrings of length $q$ common to two given strings. We now generalize this similarity measure to consider the number of common subsequences[1] of length $q$ whose symbols are taken from a fixed set $Q$ of positions.

**Definition 5.1.** We call $Q$-gram a finite sequence $Q$ of natural numbers starting with the unit element, i.e. $Q \subset \mathbb{N}$ and $1 \in Q$. We denote by $w(Q)$ the *weight* of $Q$ as the cardinality of the set $|Q|$, and by $s(Q)$ the *span* of $Q$ as the maximum element of the set $Q$.

In literature, $Q$-grams are visualized as words over the alphabet $\{1, *\}$ or $\{\#, -\}$. We adopt the former notation and represent the $Q$-gram by the word $w \in \{1, *\}^{s(Q)}$ such that $w_j = 1$ iff $j \in Q$. Figure **??** shows an example of $Q$-gram.

***Figure 5.1:*** *Filtration with gapped q-grams.*




The first thing we remark is that, compared to contiguous $q$-grams, we obtain a higher threshold, as mismatches do not cover don't care positions. The $q$-gram lemma (**??**) does

[1] A subsequence is a non-contiguous sequence of symbols of a given string.

not provide us anymore with a tight threshold, but only with a lower bound. Note that, as in the $q$-gram lemma, the threshold depends only on $Q$ and parameters $m, k$. Indeed, the pattern of covered $q$-grams does not depend on the text or pattern sequences but only on their transcript, i.e. on the position of the mismatches in the pattern.

Gapped $q$-grams raise hard combinatorial questions. Given a gapped $q$-gram, (i) does it yield a full-sensitive filter for $k$-mismatches? If so, either (ii) which is the maximum $q$-gram threshold $t$ that guarantees full-sensitivity? or (iii) which is the maximum error $k$ for which full-sensitivity is guaranteed? If the answer to question **??** is negative and the filter is lossy, (iv) how many false negatives our filter discards? Considering filtration efficiency, (v) how many false positives our filter produces? If we take the weight as a simplified criterion predicting filtration efficiency, (vi) which is the maximum weight lossless shape?

Question ii has been first considered in [Burkhardt and Karkkainen, 2001; Kucherov *et al.*, 2005], the more general questions i and vi have been introduced in [Nicolas and Rivals, 2005], while we consider here for the first time questions iii-v. With the aim of elucidating these questions, we first introduce simple characteristic functions to formally define transcripts detected by gapped $q$-grams. Afterwards, we recapitulate known results for questions i-vi and give new exact and approximate solutions.

### 5.1.1 Characteristic functions

Consider an arbitrary transcript $\sigma$ as a $m$-dimensional vector over $\mathbb{B}$, where $|\sigma|_0$ indicates the Hamming distance of the transcript. Let $\mathbb{B}_k^m \subset \mathbb{B}^m$ be the set containing all transcripts $\sigma$ such that $|\sigma|_0 = k$.

**Definition 5.2.** A $Q$-gram *occurs* at position $i$ in a similarity $\sigma$ iff $\forall j \in Q\ \sigma_{i+j} = 1$. Fixed a $Q$-gram threshold $t$, the $Q$-gram detects $\sigma$ iff it occurs at least $t$ times in $\sigma$.

#### Boolean functions

Let $T_Q^m : \mathbb{B}^m \to \mathbb{B}$ denote a *boolean function* such that $T_Q^m(\sigma)$ is true iff the $Q$-gram occurs at least one time in a similarity $\sigma$ of length $m$. We define such boolean function as the disjunction

$$T_Q^m(\sigma) = \bigvee_{i=1}^{m-s(Q)+1} \bigwedge_{j \in Q} \sigma_{i+j} \tag{5.1}$$

where each *clause* of $T_Q^m$ represents a single possible occurrence of $Q$ in $\sigma$. We define an analogous boolean function for a $Q$-gram family $F$ as the disjunction

$$T_F^m(\sigma) = \bigvee_{Q_i \in F} T_{Q_i}^m(\sigma) \tag{5.2}$$

By definition, $T_Q^m$ and $T_F^m$ are *monotone nondecreasing* boolean functions in *disjunctive normal form* (*DNF*). Since all monotone boolean functions in DNF are minimal, $T_Q^m$ and $T_F^m$ are *minimal*.

In general, $(Q, t)$ detects $\sigma$ iff $\sigma$ satisfies at least $t$ clauses of $T_Q^m$.

#### Pseudo-boolean functions

Let the function $t_Q^m : \mathbb{B}^m \to \mathbb{N}_0$ be the boolean function $T_Q^m$ acting on $\mathbb{N}_0$. We defined such *pseudo-boolean function* as

$$t_Q^m(\sigma) = \sum_{i=1}^{m-s(Q)+1} \prod_{j \in Q} \sigma_{i+j} \tag{5.3}$$

Here $t_Q^m(\sigma)$ counts how many times a $Q$-gram occurs in a similarity $\sigma$ of length $m$. It is useful to define the complementary function $\bar{t}_Q^m$, counting how many times a $Q$-gram does not occur in a similarity $\sigma$, as

$$\bar{t}_Q^m(\sigma) = m - s(Q) + 1 - t_Q^m(\sigma) \tag{5.4}$$

Analogously, we define a pseudo-boolean function for a $Q$-gram family $F$

$$t_F^m(\sigma) = \sum_{Q_i \in F} t_{Q_i}^m(\sigma) \tag{5.5}$$

along with its complementary function

$$\bar{t}_F^m(\sigma) = \sum_{Q_i \in F} (m - s(Q_i) + 1) - t_F^m(\sigma) \tag{5.6}$$

The above functions expose important properties which will let us devise approximate solutions. *Nondecreasing monotonicity* of functions $t_Q^m$ and $t_F^m$ follow from nondecreasing monotonicity of their boolean counterparts $T_Q^m$ and $T_F^m$. Consequently $\bar{t}_Q^m$ and $\bar{t}_F^m$ are *monotone nonincreasing*. From definition **??**, function $t_Q^m$ is *supermodular*, thus it follows that $\bar{t}_Q^m$ is *submodular*. Since super and submodular functions are closed under non-negative linear combination, functions $t_F^m$ and $\bar{t}_F^m$ are respectively super and submodular.

### 5.1.2 Full-sensitivity

Non Detection [Nicolas and Rivals, 2005]. Does a given gapped $q$-gram yield a full-sensitive filter for $k$-mismatches?

#### Problem definition

**Instance** A $Q$-gram, two integers $m, k$ with $0 < k < m$.
**Question** Does it exist a similarity $\sigma \in \mathbb{B}_k^m$ such that $T_Q^m(\sigma)$ is false?

#### Hardness results

Non Detection is *strongly* NP-complete [Nicolas and Rivals, 2005]. Nicolas and Rivals introduce an intermediate problem, called Soapy Set Cover. They reduce Exact Cover by 3-Sets to Soapy Set Cover and Soapy Set Cover to Non Detection. Strong NP-completeness implies that no *FPTAS* nor any *pseudo-polynomial* algorithm for it exist, under the assumption that $P \neq NP$.

#### Our FPRAS solution

Describe FPRAS.

### 5.1.3 Optimal threshold

Which is the highest $Q$-gram threshold $t$ that guarantees full-sensitivity? This problem has been introduced in [Burkhardt and Karkkainen, 2001].

#### Problem definition

**Instance** A $Q$-gram, two integers $m, k$ such that $0 < k < m$.
**Solution** The largest integer $t^*$ such that NON DETECTION for $(Q, t^*), m, k$ answers *no*.

Recalling $Q$-gram pseudo-boolean functions 5.3, we can define the optimal threshold problem as the minimization of a supermodular function subject to linear constraints

$$\begin{array}{ll} \min & t_Q^m(\sigma) \\ w.r.t. & \\ & \sigma \in \mathbb{B}_k^m \end{array} \tag{5.7}$$

#### Exact DP solution

Optimal threshold is fixed-parameter tractable (FPT) in the span of the $q$-gram shape. Burkhardt and Karkkainen give a DP algorithm computing the optimal threshold in time $O(m \cdot k \cdot 2^{s(Q)})$ [Burkhardt and Karkkainen, 2001]. Kucherov *et al.* give an extension for $Q$-gram families in [Kucherov *et al.*, 2005].

All possible assignments, i.e. $\mathbb{B}^s(Q)$, for the first clause $\bigwedge_{j \in Q} \sigma_j$ of the boolean function $T_Q^m$ are considered and all satisfying assignments for it, i.e. $\phi(Q)$, are computed. Clause $Q_2$ is considered next...

#### Our exact ILP solution

We reduce this problem to *maximum coverage* [**?**] and solve it with the following ILP

$$\begin{array}{ll} \max & |c|_1 \\ w.r.t. & \\ & \sigma \in \mathbb{B}_k^m \\ & c \in \mathbb{B}^{m-s(Q)+1} \\ & \sigma_i \geq c_j \end{array} \tag{5.8}$$

where variable $c_j$ indicates the truthfulness of the $j$-th clause in $T_Q^m$. Given the ILP solution $c^*$, we obtain the optimal threshold $t^* = |c^*|_1$.

**Our APX solution**

We reduce the complementary optimal threshold problem to the maximization of a submodular function subject to linear constraints

$$\begin{array}{ll} \max & \bar{t}_Q^m(\sigma) \\ w.r.t. & \\ & \sigma \in \bar{\mathbb{B}}_k^m \end{array} \tag{5.9}$$

where the complementary optimal threshold is $\bar{t}^* = m - s(Q) + 1 - t^*$.

We compute an approximate solution via deepest descent. Our greedy algorithm for complementary Optimal Threshold has an APX-ratio of $1 + 1/e$ [**?**]. The same *absolute error* applies to Optimal Threshold.

### 5.1.4 Maximum error

Which is the maximum error $k^*$ for which full-sensitivity is guaranteed?

**Problem definition**

**Instance** A $Q$-gram, an integer $m > 0$.
**Solution** The largest integer $k^*$ such that Non Detection for $Q, m, k^*$ answers *no*.

Recalling pseudo-boolean functions 5.3, we define our problem as the minimization of a linear function subject to submodular constraints

$$\begin{array}{ll} \min & |\sigma|_1 \\ w.r.t. & \\ & \sigma \in \mathbb{B}^m \\ & \bar{t}_Q^m(\sigma) \leq 0 \end{array} \tag{5.10}$$

**Our ILP solution**

We reduce our problem to Minimum Set Cover [**?**], solve it with the following ILP

$$\begin{array}{ll} \min & |\sigma|_1 \\ w.r.t. & \\ & \sigma \in \mathbb{B}^m \\ & b \in \mathbb{B}^{m-s(Q)+1} \\ & A\sigma \geq b \end{array} \tag{5.11}$$

where the value $A_{ij}$ of the coefficient matrix $A$ is defined as

$$A_{ij} = \begin{cases} 1 & \text{if } i - j + 1 \in Q \\ 0 & \text{if } i - j + 1 \notin Q \end{cases} \tag{5.12}$$

and find the maximum error for which full-sensitivity is guaranteed as $k^* = |\bar{\sigma}^*|_1$ given the solution $\sigma^*$ to the ILP.

Note that contiguous $q$-grams provide an interesting special case of this ILP. If $A$ has the *consecutive ones property*, it is *totally unimodular*. The *polytope* defined by a totally unimodular coefficient matrix is *integral*. Hence the optimal solution of the relaxed LP is also the optimal solution of the original ILP.

#### Our APX solution

Again, we compute an approximate solution via deepest descent. APX-ratio of $H_{w(Q)}$ [**?**].

### 5.1.5 Specificity

How many false positives our filter produces?

#### Problem definition

**Instance** A $Q$-gram, two integers $m, k$ such that $0 < k < m$.
**Solution** The number of false positives produced by the $Q$-gram.

False positives are true points of the boolean function 5.1 which have weight inferior to $m-k$ and satisfy more than $t$ clauses of $T^m_{Q^t}$. Hence we define the function $\mathrm{FP}^m_k$ counting the number of false positives of filter $(Q, t)$ in instance $(m, k)$ as

$$\mathrm{FP}^m_k(Q, t) = \sum_{\sigma \in \mathbb{B}^m_k} T^m_{Q,t}(\sigma) \tag{5.13}$$

#### Our FPRAS solution

We reduce counting false positives to counting the number of true assignments to a boolean function in DNF. The latter problem can be approximated via importance sampling. **?** introduces a *RPTAS* approximating the number of true points of a DNF.

### 5.1.6 Optimal gapped $q$-grams

Introduced in [Nicolas and Rivals, 2005].

#### Problem definition

**Instance** A finite set $S$ of similarities.
**Solution** A $Q$-gram that detects all similarities of $S$.
**Measure** The weight $w(Q)$ of the $Q$-gram.

Given a set of similarities, find the $Q$-gram of maximum weight detecting all similarities. This applies to the DNA Homology Search Framework as well.

**Hardness and inapproximability results**

Nicolas et al. [Nicolas and Rivals, 2005] perform an approximation preserving reduction from MAXIMUM INDEPENDENT SET. MWLS is NP-hard and APX-hard within $(|S|)^{0.25-\epsilon}$ unless $P = NP$. If $S = \mathbb{B}_k^m$, $|S| = \binom{m}{k}$.

**Branch-and-bound search**

Burkhardt et al. [Burkhardt and Karkkainen, 2001] bounding criterion. If $Q_1 \subseteq Q_2$, then $t_{Q_2}(m,k) \leq t_{Q_1}(m,k)$. Such bounding criterion allows to discard parts of the search space which do not solve the considered $(m,k)$ instance.

## 5.2 Approximate seeds

Approximate seeds have been proposed in [Myers, 1994; Navarro and Baeza-Yates, 2000] as a practical and effective generalization of exact seeds, yielding stronger filters both for $k$-mismatches and $k$-difference. The key idea of approximate seeds is to reduce an approximate search into *smaller* approximate searches, as opposed to exact seeds reducing an approximate search into smaller exact searches.

Again, we start by considering two arbitrary strings $x, y$ within edit distance $k$.

**Lemma 5.1.** *[Myers, 1994; Navarro and Baeza-Yates, 2000] Let $x, y$ be two strings s.t. $d_E(x,y) = k$. If we partition w.l.o.g. $y$ into $s$ non-overlapping seeds s.t. $1 \leq s \leq k+1$, then at least one seed will occur as a factor of $x$ within distance $\lfloor k/s \rfloor$.*

To prove full-sensitivity it suffices to see that, if none of the seeds occurs within its assigned distance, the total distance must be greater than $s \cdot \lfloor k/s \rfloor = k$. Figure 5.2 illustrates.

In order to solve $k$-mismatches, we partition the pattern $p$ in non-overlapping seeds $p^1, p^2, \ldots, p^s$. According to lemma 5.1, full-sensitivity is guaranteed if we search all seeds $p^i$ within a distance threshold of $\lfloor k/s \rfloor$. Note that we are not obliged to assign the same distance threshold to all seeds. Indeed, we can assign any arbitrary distance threshold $k_i$ to each seed $p^i$, as long as we satisfy the following inequality

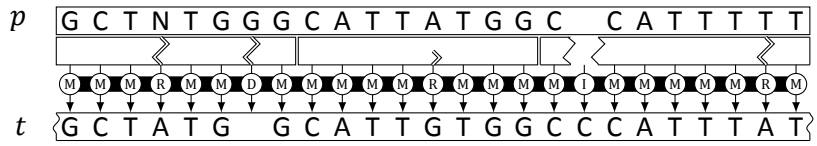
$$s + \sum_{i=1}^{s} k_i > k. \tag{5.14}$$

We usually tend to distribute evenly distance thresholds because seeds with the highest threshold dominate the overall filtration time. Therefore, we assign to $(k \bmod s) + 1$ seeds distance $\lfloor k/s \rfloor$ and to the remaining seeds distance $\lfloor k/s \rfloor - 1$ [Siragusa *et al.*, 2013]. Also note how we obtain lemma 2.1 for exact seeds by posing $s = k+1$ and all $k_i = 0$.

### 5.2.1 Parameterization

With approximate seeds we are free to choose the number of seeds $s$ which gives us $q = \lfloor m/s \rfloor$, or vice versa, if we fix the minimum seed length $q$ then it must hold $s = \lfloor m/q \rfloor$.

***Figure 5.2:*** *Filtration with approximate seeds.*




The resulting filter is flexible, indeed by increasing $q$ filtration becomes more specific at the expense of a higher filtration time. For $s = 1$ we obtain the most specific filtration i.e. we perform only backtracking and we do not even need to verify the reported locations, while for $s = k + 1$ we obtain the fastest but weakest filtration.

In practice, it is not easy to find which filter parameterization yields an optimal filtration. Both Myers, and Navarro and Baeza-Yates carried out involved analysis to estimate the optimal parameters. Navarro and Baeza-Yates find out that the optimal number of seeds $\Theta(\frac{m}{\log_\sigma n})$ yields an overall time complexity sublinear for an error rate $\epsilon < 1 - \frac{e}{\sqrt{\sigma}}$. Myers reports an analogous sublinear time when $q = \Theta(\log_\sigma n)$ is the seed length. However, these results did not help us in our experiments. The optimal partitioning depends on (i) the full-text index, (ii) the verification algorithm, (iii) the statistical properties of the text. A difference of one from the optimal number of seeds can result in a runtime penalty of an order of magnitude.

### 5.2.2 Redundancy

Another practical problem of seeds is redundant filtration. Whenever errors are not distributed according to a worst-case combination for lemma 2.1 or 5.1, more than one seed reports the same candidate location. Consider the case of exact seeds: If two errors fall in some seed, then at least two seeds will occur exactly.

In practice, we can choose to avoid redundancy before or after verifying candidate locations. After collecting all candidate locations for a pattern, we have to consider only the induced diagonal positions in the implicit DP matrix. This requires locating and sorting all candidate locations for a pattern. Alternatively, we can prefer to verify directly all candidate locations, e.g. if the verification algorithm is fast and the number of redundant candidate locations is low. Nonetheless, to avoid reporting duplicate occurrences, we have to consider redundant all occurrences either beginning or ending at the same position in the text. This requires memorizing and sorting all occurrences of a pattern.

How many redundant error configurations[2] are produced by seeds filters? For simplicity we consider exact seeds and combinations of exactly $k$ errors. We have to cover all possible ways of distributing $k$ errors among $k + 1$ seeds, that is $\binom{2k}{k}$ error combinations. However, for a fixed seed, we cover all combinations where the seed itself contains no errors and all $k$ errors are distributed among the remaining $k$ seeds, i.e. $\binom{2k-1}{k}$ error combinations. Thus the fraction of error combinations covered by filtration with exact

[2] Here we consider the fraction of redundant error combinations and not the fraction of redundant candidate locations reported by the filter.

seeds over the minimal ones is

$$\frac{(k+1)\binom{2k-1}{k}}{\binom{2k}{k}} = \frac{k+1}{2} \tag{5.15}$$

For instance, filtration with exact seeds covers three times more combinations than required when $k = 5$. With approximate seeds we partition the pattern in less seeds and thus we indirectly reduce filter redundancy.

# Part II

# READ MAPPING

CHAPTER 6

# Background

Next generation sequencing is a terrific technology. A wealth of applications have been developed on top of it. Data analysis pipelines for variant calling and structural variation discovery from DNA-seq, mRNA transcripts abundance estimation and novel non-coding RNA discovery from RNA-seq, transcription factor binding-sites prediction from ChIP-seq. All these applications rely on a common prerequisite step: mapping NGS reads to a known reference genome.

Read mapping is a critical step in all NGS data analysis pipelines. NGS reads produced by all current technologies contain sequencing errors, in form of single miscalled bases or stretches of oligonucleotides. Moreover, the donor genome from which reads have been sequenced contains small genomic variations (SNVs, Indels) in addition to CNV, inversions and translocations. After all, spotting genomic variation is one reason for which we resequence genomes. Thus, when mapping a read to a reference genome, it is not sufficient to consider the loci where the reads map exactly; it is necessary to consider any loci of relevant sequence similarity, being possible origins of the sequenced reads.

## 6.1 Sequencing technologies

### 6.1.1 Illumina

Illumina / Solexa.

### 6.1.2 Ion Torrent

Life Technologies / Ion Torrent.

### 6.1.3 454 Life Sciences

Roche / 454 Life Sciences.

### 6.1.4 SOLiD

ABI / SOLiD.

## 6.2 Sequencing applications

### 6.2.1 DNA-seq

### 6.2.2 RNA-seq

### 6.2.3 ChIP-seq

## 6.3 Sequencing quality

Phred base quality values have been introduced in [Ewing *et al.*, 1998; Ewing and Green, 1998] to assess the quality of sequencing single bases in capillary reads. Instead of directly discarding low-quality regions present in capillary reads, Phred calls each base and annotates it with a quality score encoding the probability that it has been wrongly called. As this method has been widely accepted, base callers annotate reads issue of all sequencing technologies with Phred base quality scores.

To formally define Phred base quality values, let us fix the alphabet $\Sigma = \{ \text{A, C, G, T} \}$, and consider a known donor genome $g$ over $\Sigma$ and a read $r$ sequenced at location $l$ from the template $g_{l \dots l+|r|-1}$. We define the base calling error $\epsilon_i$ at position $i$ in the read $r$, as the probability $\epsilon_i$ of miscalling a base $r_i$ instead of calling its corresponding base $g_{l+i-1}$ in the donor genome. Therefore, we define the Phred base quality $Q_i$ at position $i$ as:

$$Q_i = -10 \log_{10} \epsilon_i. \tag{6.1}$$

Given the above, the probability $p(r_i | g_{l+i-1})$ of calling the base $r_i$ in the read $r$, given the donor genome base $g_{l+i-1}$, is:

$$p(r_i | g_{l+i-1}) = \begin{cases} 1 - \epsilon_i & \text{if } g_{l+i-1} = r_i \\ \frac{\epsilon_i}{|\Sigma|-1} & \text{if } g_{l+i-1} \in \Sigma \setminus \{r_i\} \end{cases} \tag{6.2}$$

and assuming i.i.d. base calling errors, it follows that the probability $p(r | g, l)$ of observing the read $r$, given the donor genome template $g_{l \dots l+|r|-1}$, is:

$$p(r | g, l) = \prod_{i=1}^{|r|} p(r_i | g_{l+i-1}) \tag{6.3}$$

## 6.4 Mappability

Genome resequencing is a non-trivial task.

The difficulty of unambiguously finding the correct mapping location of next-generation sequencing reads comes from the non-random nature of genomes. Genomes evolved through multiple types of duplication events, including (i) whole-genome duplications [**?**] or large-scale segmental duplications in chromosomes [**?**], (ii) transposition of repetitive elements as short tandem repeats (microsatellites) and interspersed nuclear elements (LINE, SINE) [**?**], (iii) proliferation of repetitive structural elements as telomeres

and centromeres [**?**]. As a result of these events, about 50 % of the human genome is composed of repeats.

An analysis of the $k$-mer spectra of the genomes of some model organisms shows how genomes are statistically different from texts randomly generated according to uniform bernoulli models.

Repeats present in general technical challenges for all *de novo* assembly and sequence alignment programs [Lee and Schatz, 2012]. In the case of short reads mapping, the first evident effect of the heavy tail in the $k$-mer distribution of reference genomes is the dramatic loss of specificity in certain regions, which increases the computational cost of programs based on filtration methods. But the most subtle challenge lies in the interpretation of these results: it is not evident how to consider reads mapping to multiple locations.

Common strategies to deal with multi-reads are (i) to discard them all, (ii) to randomly pick one best mapping location, (iii) to consider all or up to $k$ best mapping locations within a given distance threshold [Treangen and Salzberg, 2011].

A practical challenge is represented by reporting and handling the resulting datasets of mapping locations, which can have a size up to two orders of magnitude bigger compared to the corresponding input read sets.

Genome mappability can bias NGS analysis more than we might think at a first glance. Two recent studies [Derrien *et al.*, 2012; Lee and Schatz, 2012] show which is the bias of mappability.

### 6.4.1 Genome mappability

We now give a definition of genome mappability analogous to [Derrien *et al.*, 2012]. Let fix a $q$-gram length, a distance measure as the Hamming or edit distance, and a distance threshold $k$. Given a genomic sequence $g$, we define the $(q,k)$-frequency $F_k^q(l)$ of the $q$-gram $g_{l\ldots l+q-1}$ at location $l$ in $g$ as the number of occurrences of the $q$-gram in $g$ and its reverse complement $\bar{g}$. We define the $(q,k)$-mappability $M_k^q(l)$ as the inverse $(q,k)$-frequency, i.e. $M_k^q(l) = F_k^q(l)^{-1}$ with $M_k^q : \mathbb{N} \to ]0,1]$. Note that $M_k^q(l)$ can be seen as the prior probability that any read of length $q$ originating at location $l$ will be mapped correctly. The values of $(q,k)$-frequency and mappability obviously vary with the distance threshold $k$. Nonetheless, under any distance measure, it hold that the $q$-gram at location $l$ is unique up to distance $k$ iff $M_k^q(l) = 1$ and repeated otherwise.

Which is the minimum $q$-gram length from which we expect $(q,k)$-mappability to be 1 for the genomes of model organisms? Let consider the simple case of exact $(q,0)$-mappability. By assuming a genomic sequence of length $n$ as being randomly generated under the uniform bernoulli model, the emission probability of any nucleotide is $p = \frac{1}{4}$ and, under i.i.d. assumptions, the emission probability of any $q$-gram is $p_q = \frac{1}{4^q}$. It follows that the expected value of $(q,0)$-frequency is $E[F_0^q] \simeq \frac{2n}{4^q}$. Thus, for $E[F_0^q] \leq 1$ it must

hold:

$$2n \leq 4^q \tag{6.4}$$

$$\log 2n \leq \log 4^q \tag{6.5}$$

$$q \geq log_4 2n \tag{6.6}$$

Thus, we would expect any $q$-gram of length $\log_4 2n$ to occur about once in a genomic sequence of length $n$. Sticking to these assumption, in the human genome ($n \approx 3 \cdot 10^9$) almost all 17-mers would be unique, on fly ($n \approx 1.2 \cdot 10^8$) all 15-mers, on worm ($n \approx 4.2 \cdot 10^7$) all 13-mers. However, the $q$-gram distribution of model genomes does not fit the uniform bernoulli distribution. In [**?**] the $k$-mers distribution can be approximated by a double Pareto log-normal distribution, i.e. a distribution with a heavy tail. This is a result of the evolution of genomes being driven by gene duplications, retrotransposons [**?**].

Consequently, we would expect 36 bp reads produced by early Illumina sequencers to induce an almost perfect mappability. However, reads have to be mapped approximately to the reference genome. The expected number of approximate occurrences of a $k$-mer is higher than the exact one. Thus the above estimate is a lower bound.

**Uniqueome**

Derrien *et al.* quantified the whole genome unique mappability for human, mouse, fly, and worm. At a (36, 2) mapping, about 30 % of the human genome is not uniquely mappable. Unique mappability rises to 83 % by increasing the read length to 75 bp; however to map a significant fraction of the reads, we should consider 3–4 edit distance errors.

| | H.sapiens (hg19) | M.musculus (mm9) | D.mel (dm3) |
|---|---|---|---|
| Repeats content [%] | 45.25 | 42.33 | 26.50 |
| Uniqueome (36 bp, 2 msm) [%] | 69.99 | 72.07 | 68.09 |
| Uniqueome (50 bp, 2 msm) [%] | 76.59 | 77.06 | 69.44 |
| Uniqueome (75 bp, 2 msm) [%] | 83.09 | 81.65 | 71.00 |

The uniqueome plays an important role in ChIP-seq experiments. It is common practice [**?**] to rely on short (36 bp) reads and discard the non-unique ones. Not only a significant fraction of the sequencing data is thrown out. Worse than that, we end up with holes in 30 % of the genome. A ChIP-seq peak caller considering multi-reads calls up to 30 % more peaks.

Cite regions of clinical relevance, e.g. HLA-A. Cite regions of biological relevance, e.g. 5S rRNA.

**Paired-end mappability**

**Pileup mappability**

If we focus our attention to the resequencing accuracy at a single locus, we have to consider the mappability of all the possible reads spanning that given locus. Pileup mappability [Derrien *et al.*, 2012] at position $i$ is the average mappability of all reads spanning position $i$.

$M_p(i) = 1/q \sum_{j=i}^{i+1} M(j)$

### 6.4.2 Mapping quality score

Mapping quality has been introduced in [Li *et al.*, 2008]. The study considers short reads of length ranging from 30 bp to 40 bp, produced by early Illumina/Solexa and ABI/SOLiD sequencing technologies, whose sequencing error rates were quite high. Given the short lengths and high error rates, a significant fraction of such reads can be aligned to multiple mapping locations, even considering only co-optimal Hamming distance locations.

The key point is that the Hamming distance is not an adequate scoring scheme to guess the correct mapping location of many reads. The authors claim[1] that:

> It is possible to act conservatively by discarding reads that map ambiguously at some level, but this leaves no information in the repetitive regions and it also discards data, reducing coverage in an uneven fashion, which may complicate the calculation of coverage.

Since base callers output base call probabilities in Phred-scale along with the reads, Li *et al.* propose a novel probabilistic scoring scheme called mapping quality, giving the probability that a given read has been aligned correctly at a given mapping location in the reference genome.

By applying Bayes' theorem, we can derive the posterior probability $p(l|g,r)$, that location $l$ in the reference genome $g$ is the correct mapping location of read $r$. Assuming uniform coverage, each location $l \in [1, |g| - |r| + 1]$ has equal probability of being the origin of a read in the donor genome, thus the prior probability $p(l)$ is simply:

$$p(l) = \frac{1}{|g| - |r| + 1} \tag{6.7}$$

Therefore, recalling $p(r|g,l)$ from equation 6.3, the posterior probability $p(l|g,r)$ equals the probability of the read $r$ originating at location $l$ normalized over all possible locations in the reference genome:

$$p(l|g,r) = \frac{p(r|g,l)}{\sum_{i=1}^{|g|-|r|+1} p(r|g,i)} \tag{6.8}$$

which in Phred-scale becomes:

$$Q(l|g,r) = -10 \log_{10}[1 - p(l|g,r)] \tag{6.9}$$

[1] Li *et al.* do not show in their study what is the effect of relying on mapping quality rather than on mapping uniqueness.

Computing the exact mapping quality as in equation 6.9 requires aligning each read to all positions in the reference genome. On one hand, this computation would not be practical, indeed the vast majority of a reference genome is discarded when mapping reads by means of filtering and fully-indexed methods. On the other hand, the contribution of discarded locations to the sum in equation 6.8 can be neglected. Therefore, equation 6.8 is approximated using only relevant mapping locations found by the read mapper.

Mapping quality has been initially used in [Li *et al.*, 2008] and [Li *et al.*, 2009a] to maximize variant calling confidence by discarding reads whose best mapping location is below a given mapping quality threshold. This measure has been widely accepted: nowadays it is computed by most popular read mappers and used by almost all variant calling pipelines e.g. the Genome Analysis ToolKit (GATK) [DePristo *et al.*, 2011].

Nonetheless, some important objections can be moved against mapping quality. First, the mapping quality score is derived under the unlikely assumption of the reference genome being equal to the donor genome. In other words, mapping quality considers only errors due to base miscalls and disregards genetic variation; thus the risk is to prefer mapping locations supported by known low base qualities rather than by true but unknown SNVs. Second, mapping quality is nonetheless strongly correlated to mapping uniqueness, as discussed in section 6.4; it is easy to see that the mapping probability in equation 6.8 is diluted in presence of a large number of co-optimal mapping locations. Third, mapping quality tends to become less relevant as base calls improve, due to advances of sequencing technologies, and thus degenerates in a shallow measure of uniqueness.

### 6.4.3 Genome mappability score

Genome mappability score (GMS) [Lee and Schatz, 2012] is analogous to pileup mappability. Instead of considering the inverse mapping frequency $(q, k)$-mappability, we can interpret mapping quality (see subsection 6.4.2) as the probability that a read originating at a given position can be mapped correctly. Therefore, we consider the average mapping probability of any read spanning a location $l$ of a reference genome $g$[2]:

$$p(l|g) = \sum_{r \in \mathcal{R}(l)} \frac{p(l|g,r)}{|\mathcal{R}(l)|} \tag{6.10}$$

which in Phread-scale becomes:

$$Q(l|g) = \sum_{r \in \mathcal{R}(l)} \frac{1 - 10^{-\frac{Q(l|g,r)}{10}}}{|\mathcal{R}(l)|} \tag{6.11}$$

and thus, fixed a genomic sequence $g$, we define the genome mappability score $\mathrm{GMS}(l)$ as its percentual value:

$$\mathrm{GMS}(l) = 100Q(l|g) \tag{6.12}$$

[2] Equation 3 in [Lee and Schatz, 2012] is not precise, please refer to our equation 6.10.

Lee and Schatz simulate reads having length and error profiles similar to those issue by actual sequencing technologies, define low GMS regions as those locations for which $\mathrm{GMS}(l) \leq 10$, and measure the percentage of such locations in the human genome.

| Sequencing technology | Read length [bp] | Error rate [%] (msm, ins, del) | Low GMS [%] | High GMS [%] |
|---|---|---|---|---|
| SOLiD-like | 75 | (0.10, 0.00, 0.00) | 11.14 | 88.86 |
| Illumina-like | 100 | (0.10, 0.00, 0.00) | 10.51 | 89.49 |
| Ion Torrent-like | 200 | (0.04, 0.01, 0.95) | 9.35 | 90.65 |
| Roche/454-like | 800 | (0.18, 0.54, 0.36) | 8.91 | 91.09 |
| PacBio-like | 2000 | (1.40, 11.47, 3.43) | 100.0 | 0.00 |
| PacBio EC-like | 2000 | (0.33, 0.33, 0.33) | 8.61 | 91.39 |

CHAPTER 7

# Read mappers engineering

## 7.1 Masai

### 7.1.1 Approximate seeds

### 7.1.2 Multiple backtracking

### 7.1.3 Single-end mapping

#### All-mapping

#### Mapping by strata

### 7.1.4 Paired-end mapping

## 7.2 Yara

### 7.2.1 Single-end mapping

#### All-mapping

#### Mapping by strata

### 7.2.2 Paired-end mapping

#### All-mapping

#### Mapping by strata

### 7.2.3 Parallelization

### 7.2.4 Hardware acceleration

CHAPTER

# 8 Read mappers evaluation

## 8.1 Popular read mappers

Critic the surveys classifying hundreds of mappers [Li and Homer, 2010], [Fonseca *et al.*, 2012].

Critic the benchmarks [Hatem *et al.*, 2013] [Holtgrewe *et al.*, 2011].

The task of a read mapper is to guess where a read originates. Fixed a similarity scoring scheme that confidently models this problem, the optimal alignment under this scoring scheme correspond to the most likely explanation and induces a locus being the origin of the read. The simplest scoring scheme is the edit distance; more involved scoring schemes take into account base quality values, score gaps using affine cost functions, or allow to trim for free a prefix or a suffix of the read.

The above definition does not consider two problems: what if there are many co-optimal candidates, and what if the correct solution corresponds to a sub-optimal candidate. The former problem is exacerbated by genome mappability. One would expect such situations to arise very rarely, but instead it is a relevant problem. The latter problem arises whenever our model is not adequate to explain the difference between a read and its genomic origin. For instance, an evolutionary event producing an indel of length $l$ might be considered as a unit, whether edit distance would consider it as $l$ independent events. Under the edit distance, an alignment with less than $l$ independent point mutations would be considered more likely than an alignment containing only one indel of length $l$.

From the former problem, we conclude that considering only one optimal mapping location is not sufficient, no matter how good our scoring scheme can be. The latter problem tells us to be careful about relying on strict optimality. Therefore, in general a read mapper should return a comprehensive set of relevant mapping locations along with the likelihood that they correspond to the original location.

### 8.1.1 Bowtie

Bowtie [Langmead *et al.*, 2009] is a mapper designed to have a small memory footprint and quickly report a few good mapping locations for early generation Illumina/Solexa and ABI/SOLiD short reads of length up to 50 bp. It achieves the former goal by indexing the reference genome with an FM-index and the latter goal by performing a greedy depth-first traversal on it.

The greedy depth-first traversal visits first the subtree yielding the least number of mismatches and stops after having found a candidate (not guaranteed to be optimal when $k > 1$). In addition, Bowtie speeds up backtracking by applying case pruning, a simple application of the pigeonhole principle. However this technique is mostly suited for $k = 1$ and requires the index of the forward and reverse text.

Bowtie can be configured to search by strata, however the search time increases significantly while the traversal still misses a large fraction of the search space due to seeding heuristics. Main practical drawbacks of the tool are too many cryptic options.

Bowtie 2 [Langmead and Salzberg, 2012] has been designed to quickly report a couple of mapping locations for recent Illumina/Solexa, Ion Torrent and Roche/454 reads, usually having lengths in the range from 100 bp to 400 bp.

This tool uses an heuristic seed-and-extend approach, collecting seeds of fixed length, partially overlapping, and searching them exactly in the reference genome using an FM-index. Candidate locations to verify are chosen randomly, to avoid uncompressing large CSA intervals and executing many DP instances. Each mapping location is verified using a striped vectorial dynamic programming algorithm, implemented using SIMD instructions, previously introduced by [Farrar, 2007] and extended to compute end-to-end alignments.

Bowtie 2 can be configured to report end-to-end or local alignments, scored using a tunable affine scoring scheme. For this reason, it is believed to be good at reporting alignments containing indels. However, its completely heuristic filtration strategy, independent of the scoring scheme, makes it hard to believe what it promises.

### 8.1.2 BWA

BWA-backtrack [Li and Durbin, 2009] is designed to map Illumina/Solexa reads up to 100 bp and report a few best end-to-end alignments. The program performs a greedy breadth-first search on an FM-index of the reference genome. Nodes to be visited are ranked by edit distance score: the best node is popped from a priority queue and visited, its children are then inserted again in the queue. The traversal considers indels using a more involved 9-fold recursion. Backtracking is sped up by adopting a more stringent pruning strategy that nonetheless takes some preprocessing time and requires the index of the reverse reference genome.

BWA performs paired-end alignments by trying to anchor both paired-end reads and verifying the corresponding mate, within an estimated insert size, using the classic DP-based Smith-Waterman algorithm. Consequently, the program in paired-end mode aligns reads at a slower rate than in single-end mode. The program is not fully multi-threaded, therefore BWA scales poorly on modern multi-core machines.

BWA-SW [Li and Durbin, 2010] is designed to map Roche/454 reads, which have an average length of 400 bp. It is an heuristic version of BWT-SW, designed to report a few good local alignments.

This version of BWA adopts a double indexing strategy: it indexes all substrings of one read in a DAWG. It performs Smith-Waterman of all read substrings directly on the FM-index, by backtracking as soon as no viable alignment can be obtained. As in BWA-

backtrack, the traversal proceeds in a greedy fashion. In addition, BWA-SW implements some seeding heuristics to limit backtracking and jump in the reference genome to verify candidate locations whenever this becomes favorable.

This version of BWA does not support paired-end reads, presumably because it was meant for Roche/454 reads.

### 8.1.3 Soap

Soap 2 [Li *et al.*, 2009b] is very similar to Bowtie: it has been designed to produce a very quick but shallow mapping of Illumina/Solexa reads up to 75 bp with no more than 2 mismatches and no indels. However, its underlying algorithm is based on the so-called bi-directional (or 2-way) BWT. The tool support paired-end mapping but at a slower alignment rate. Practical drawbacks are the lack of native output in the de-facto standard SAM format and is closed source. Soap 3 [Liu *et al.*, 2012] is algorithmically similar to Soap 2 but targets only NVIDIA CUDA accelerators.

### 8.1.4 SHRiMP

SHRiMP 2 TODO.

### 8.1.5 RazerS

RazerS [Weese *et al.*, 2009] has been designed to report all mapping locations within a fixed hamming or edit distance error rate. It is based on a full-sensitive $q$-gram filtration method (SWIFT semi-global) combined with the Myers edit distance verification algorithm. On demand, the SWIFT filter can be configured to become lossy within a fixed loss rate. The lossy filter becomes more stringent and produces a lower number of candidates to verify, thus improving the overall speed of the program. All in all, the SWIFT filter is very slow while not highly specific.

RazerS 3 [Weese *et al.*, 2012] is a faster version featuring shared-memory parallelism, a faster banded-Myers verification algorithm, and a faster filtration scheme based on exact seeds that however turns out to be very weak on mammal genomes. Because of this, RazerS 3 is one-two orders of magnitude slower than Bowtie 2 and BWA-backtrack on mammal genomes.

All RazerS versions index the reads and scan the reference genome. One positive aspect of this strategy is that no preprocessing of the reference genome is required. However, other mapping strategies beyond all-mapping, e.g. mapping by strata, cannot be efficiently implemented. Moreover, the program exhibit an high memory footprint as it must remember the mapping locations of all input reads until the whole reference genome has been scanned.

### 8.1.6 mr(s)Fast

The tools mrFast [Ahmadi *et al.*, 2012] and mrsFast [Hach *et al.*, 2010] are designed to report all mapping locations within a fixed absolute number errors, respectively under the hamming and edit distance, given Illumina/Solexa reads of length ranging from 50 bp to 125 bp. Similarly to RazerS 3, they are based on a full-sensitive filtration strategy using exact seeds, which turns out to be very weak on mammal genomes.

The peculiarity of their underlying method is a cache-oblivious strategy to mitigate the high cost of verifying clusters of candidate locations. In addition, mrsFast computes the edit distance between one read and one mapping location in the reference genome with an antidiagonal-wise vectorial dynamic programming algorithm, implemented using SIMD instructions.

These tools are as slow as RazerS 3 and appealing for nothing more than all-mapping. They lack multi-threading support and exhibit various bugs. Furthermore, they only accept reads of fixed length and produce files of impractical size.

### 8.1.7 GEM

The GEM mapper [Marco-Sola *et al.*, 2012] is a flexible read aligner for Illumina/Solexa, ABI/SOLiD, and Ion Torrent reads. It is full-sensitive and can be configured either as an all-mapper, as a best/unique-mapper, or to search by strata.

GEM uses a combination of state of the art approximate string matching methods, e.g. approximate seeds and suffix filters. The program indexes the reference genome with an FM-index, tries to find an optimal filtration strategy per read, and verifies candidate locations using Myers algorithm. Paired-reads are either mapped independently and then combined, or left/right are mapped and their mates verified using an online strategy.

Unfortunately the tool lacks direct SAM output, it is not open source, and provides many obscure parameters.

### 8.1.8 Masai

### 8.1.9 Yara

| | platform | | | | strategy | | | method | | | index | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Illumina | Ion | 454 | SOLiD | best | strata | all | alignment | optimal | algorithm | type | reference | reads |
| Bowtie | $\leq 50$ | ✗ | ✗ | ✓ | ✓ | ✓ | • | mismatches | ✗ | backtracking | FM-index | ✓ | ✗ |
| Bowtie 2 | $\geq 75$ | ✓ | ✓ | ✗ | ✓ | • | • | local | ✗ | exact seeds | FM-index | ✓ | ✗ |
| BWA | $\leq 100$ | ✗ | ✗ | ✗ | ✓ | ✗ | • | indels | ✗ | backtracking | FM-index | ✓ | ✗ |
| BWA-SW | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | local | ✗ | backtracking | FM-index | ✓ | ✓ |
| Soap 2 | $\leq 75$ | ✗ | ✗ | ✗ | ✓ | ✓ | • | mismatches | ✗ | backtracking | FM-index | ✓ | ✗ |
| RazerS | $\geq 50$ | ✗ | ✗ | ✗ | • | • | ✓ | indels | ✓ | $q$-grams | $q$-gram index | ✗ | ✓ |
| RazerS 3 | $\geq 50$ | ✓ | ✗ | ✗ | • | • | ✓ | indels | ✓ | exact seeds | $q$-gram index | ✗ | ✓ |
| SHRiMP 2 | ✓ | ✓ | ✓ | ✓ | ✓ | • | • | local | ✗ | $q$-grams | $q$-gram index | ✓ | ✗ |
| mrsFast | $\leq 75$ | ✗ | ✗ | ✗ | • | ✗ | ✓ | mismatches | ✓ | exact seeds | $q$-gram index | ✓ | ✓ |
| mrFast | $\leq 125$ | ✗ | ✗ | ✗ | • | ✗ | ✓ | indels | ✓ | exact seeds | $q$-gram index | ✓ | ✓ |
| GEM | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | indels | ✓ | apx seeds | FM-index | ✓ | ✗ |
| Masai | ✓ | ✗ | ✗ | ✗ | ✓ | • | ✓ | indels | ✓ | apx seeds | generic | ✓ | ✓ |
| Yara | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | indels | ✓ | apx seeds | generic | ✓ | ✗ |

## 8.2 Rabema results

## 8.3 Variant detection results

## 8.4 Runtime results

CHAPTER 9

# Discussion

APPENDIX

# A Related problems

## A.1 Dictionary search

Dictionary search is a restriction of string matching. Given a set of database strings $\mathbb{D}$ and a query string $q$, the approximate dictionary search problem is to find all strings in $\mathbb{D}$ within distance $k$ from $q$. Note that usually the query string $q$ has length similar to strings in $\mathbb{D}$, as $||d| - |q|| \leq k$ is a necessary condition for $d_E(d, q) \leq k$.

### A.1.1 Online methods

The problem can be solved by checking whether $d_E(d, q) \leq k$ for all $d \in \mathbb{D}$. Answering the question whether the distance $d_E(d, q) \leq k$ is an easier problem than computing the edit distance $d_E(d, q)$: a band of size $k + 1$ is sufficient.

**Lemma A.1.** *The $k$-differences global alignment problem can be solved by computing only a diagonal band of the DP matrix of width $k+1$, where the leftmost band diagonal is $\lfloor \frac{m-n+k}{2} \rfloor$ cells left of the main diagonal (see Figure* **??***).*

*Proof.* Indirect. Assume that a cell outside the band is part of a global alignment with at most $k$ errors. If the cell is left of the band, the traceback that starts in the top left corner would go down at least $c = \lfloor \frac{m-n+k}{2} \rfloor + 1$ cells. Then it needs to go right at least $n - m + c$ cells to end in the bottom right corner. Hence it contains at least $n - m + 2c > n - m + 2\frac{m-n+k}{2} = k$ errors. The assumption that the cell is right of the band can be falsified analogously. □

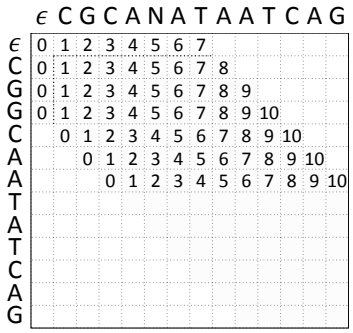
### A.1.2 Indexed methods

Using a radix tree $\mathcal{D}$ we can find all strings in $\mathbb{D}$ equal to a query string $q$, in optimal time $\mathcal{O}(|q|)$ and independently of $||\mathbb{D}||$.

### A.1.3 Filtering methods

Filtering methods of section 2.5.3 can be directly applied to solve the dictionary search problem. Database strings satisfying the filtering condition can be verified with algorithm **??**.

***Figure A.1:*** *DP table representing the match of $p = ...$ in $t = ....$*




**Algorithm A.1** Exact dictionary search on a radix trie.

```
1: procedure ExactSearch(x, p)
2:     if p = ϵ then
3:         report 𝔼(x)
4:     else if ∃ c ∈ ℂ(x) : label(c) = p_1 then
5:         ExactSearch(c, p_{2..|p|})
```

## A.2 Local similarity search

Define score and scoring scheme.
Define local similarity.

### A.2.1 Online methods

Give dynamic programming solution.

### A.2.2 Indexed methods

Backtracking over substring index. BWT-SW.

### A.2.3 Filtering methods

SWIFT/Stellar is based on the $q$-gram lemma.

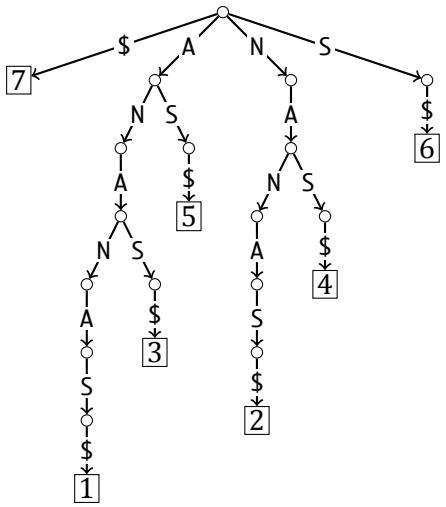
## A.3 Overlaps computation

Define problem.

### A.3.1 Online methods

DP solution.

***Figure A.2:*** *Exact dictionary search on a suffix trie.*




### A.3.2 Indexed methods

Indexed solution, exact and approximate.

Appendix

# B Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Enrico Siragusa
July 1, 2014

# BIBLIOGRAPHY


Ahmadi, A., Behm, A., Honnalli, N., Li, C., Weng, L., and Xie, X. (2012). Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic Acids Res.*, **40**(6), page e41.

Baeza-Yates, R. A. and Gonnet, G. H. (1999). A fast algorithm on average for all-against-all sequence matching. In *SPIRE/CRIWG*, pages 16–23. IEEE.

Baeza-Yates, R. A. and Navarro, G. (1999). Faster approximate string matching. *Algorithmica*, **23**(2), pages 127–158.

Bauer, M. J., Cox, A. J., and Rosone, G. (2013). Lightweight algorithms for constructing and inverting the bwt of string collections. *Theoretical Computer Science*, **483**, pages 134–148.

Burkhardt, S. and Karkkainen, J. (2001). Better filtering with gapped q-grams. In *CPM*, volume 1, pages 73–85. Springer.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm.

Crochemore, M., Grossi, R., Kärkkäinen, J., and Landau, G. M. (2013). A constant-space comparison-based algorithm for computing the burrows–wheeler transform. In *Combinatorial Pattern Matching*, pages 74–82. Springer.

DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., del Angel, G., Rivas, M. A., Hanna, M., *et al.* (2011). A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature genetics*, **43**(5), pages 491–498.

Derrien, T., Estellé, J., Marco Sola, S., Knowles, D. G., Raineri, E., Guigó, R., and Ribeca, P. (2012). Fast computation and applications of genome mappability. *PLoS ONE*, **7**(1), page e30377.

Elias, P. (1975). Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, **21**(2), pages 194–203.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using phred. ii. error probabilities. *Genome research*, **8**(3), pages 186–194.

Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using phred. i. accuracy assessment. *Genome research*, **8**(3), pages 175–185.

Faro, S. and Lecroq, T. (2013). The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys (CSUR)*, **45**(2), page 13.

Farrar, M. (2007). Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, **23**(2), pages 156–161.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE.

Ferragina, P. and Manzini, G. (2001). An experimental study of an opportunistic index. In *SODA*, pages 269–278.

Fonseca, N. A., Rung, J., Brazma, A., and Marioni, J. C. (2012). Tools for mapping high-throughput sequencing data. *Bioinformatics*, **28**(24), pages 3169–3177.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: Computer science and computational biology*. Cambridge University Press, New York, NY, USA.

Hach, F., Hormozdiari, F., Alkan, C., Hormozdiari, F., Birol, I., Eichler, E. E., and Sahinalp, S. C. (2010). mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nat. Methods*, **7**(8), pages 576–577.

Hatem, A., Bozda, D., Toland, A. E., and Çatalyürek, Ü. V. (2013). Benchmarking short sequence mapping tools. *BMC bioinformatics*, **14**(1), page 184.

Holtgrewe, M., Emde, A.-K., Weese, D., and Reinert, K. (2011). A novel and well-defined benchmarking method for second generation read mapping. *BMC Bioinformatics*, **12**, page 210.

Intel (2011). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE.

Jokinen, P. and Ukkonen, E. (1991). Two algorithms for approxmate string matching in static texts. In *Mathematical Foundations of Computer Science 1991*, pages 240–248. Springer.

Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. *ICALP*, pages 943–955.

Knuth, D. (1973). *The Art of Computer Programming. Volume 3, Addision-Wesley*.

Kucherov, G., Noé, L., and Roytberg, M. (2005). Multiseed lossless filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, **2**(1), pages 51–61.

Kurtz, S. (1999). Reducing the space requirement of suffix trees. *Software-Practice and Experience*, **29**(13), pages 1149–71.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nat. Methods*, **9**(4), pages 357–359.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**(3), page R25.

Lee, H. and Schatz, M. C. (2012). Genomic dark matter: the reliability of short read mapping illustrated by the genome mappability score. *Bioinformatics*, **28**(16), pages 2097–2105.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, **25**(14), pages 1754–1760.

Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, **26**(5), pages 589–595.

Li, H. and Homer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Brief Bioinform.*, **11**(5), pages 473–483.

Li, H., Ruan, J., and Durbin, R. (2008). Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, **18**(11), pages 1851–1858.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., and 1000 Genome Project Data Processing Subgroup (2009a). The sequence alignment/map format and SAMtools. *Bioinformatics*, **25**(16), pages 2078–2079.

Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. (2009b). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**(15), pages 1966–1967.

Liu, C.-M., Wong, T., Wu, E., Luo, R., Yiu, S.-M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., *et al.* (2012). Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, **28**(6), pages 878–879.

Manber, U. and Myers, G. (1990). Suffix arrays: a new method for on-line string searches. In *SODA*, pages 319–327.

Marco-Sola, S., Sammeth, M., Guigó, R., and Ribeca, P. (2012). The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, **9**(12), pages 1185–1188.

Morrison, D. R. (1968). Patricia-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, **15**(4), pages 514–534.

Myers, E. W. (1994). A sublinear algorithm for approximate keyword searching. *Algorithmica*, **12**(4-5), pages 345–374.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Comput. Surv.*, **33**(1), pages 31–88.

Navarro, G. and Baeza-Yates, R. A. (2000). A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, **1**(1), pages 205–239.

Navarro, G., Baeza-Yates, R. A., Sutinen, E., and Tarhio, J. (2001). Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, **24**(4), pages 19–27.

Nicolas, F. and Rivals, E. (2005). Hardness of optimal spaced seed design. In *Combinatorial Pattern Matching*, pages 144–155. Springer.

Schürmann, K.-B. and Stoye, J. (2007). An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, **37**(3), pages 309–329.

Seward, J. (1996). bzip2 and libbzip2. *avaliable at http://www. bzip. org*.

Siragusa, E., Weese, D., and Reinert, K. (2013). Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Res.*

Treangen, T. J. and Salzberg, S. L. (2011). Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, **13**(1), pages 36–46.

Ukkonen, E. (1993). Approximate string-matching over suffix trees. In *CPM*, pages 228–242.

Weese, D. (2013). *Indices and Applications in High-Throughput Sequencing*. Ph.D. thesis, Freie Universität Berlin.

Weese, D., Emde, A.-K., Rausch, T., Döring, A., and Reinert, K. (2009). RazerS–fast read mapping with sensitivity control. *Genome Res.*, **19**(9), pages 1646–1654.

Weese, D., Holtgrewe, M., and Reinert, K. (2012). RazerS 3: Faster, fully sensitive read mapping. *Bioinformatics*. 10.1093/bioinformatics/bts505.

Wilkes, M. V. (1995). The memory wall and the cmos end-point. *ACM SIGARCH Computer Architecture News*, **23**(4), pages 4–6.

# LIST OF FIGURES


2.1 Example of edit transcript . . . . . 6
2.2 Example of dotplot . . . . . 7
2.3 Example of DP table . . . . . 9
2.4 Example of approximate string matching via DP . . . . . 10
2.5 Example of suffix trie and suffix tree . . . . . 12
2.6 Example of generalized suffix trie . . . . . 12
2.7 Exact string matching on a suffix tree . . . . . 14
2.8 Filtration with exact seeds. . . . . 15
2.9 Filtration with $q$-grams. . . . . 17

4.1 Example of suffix array . . . . . 22
4.2 Example of generalized suffix array . . . . . 23
4.3 Example of $q$-gram index . . . . . 26
4.4 Example of Burrows-Wheeler transform . . . . . 28
4.5 Example of functions $LF$ and $\Psi$ . . . . . 29
4.6 Example of BWT inversion . . . . . 29
4.7 Example of rank dictionaries . . . . . 31
4.8 Example of rank dictionaries . . . . . 32
4.9 Top-down traversal runtime . . . . . 35
4.10 Exact string matching runtime . . . . . 36
4.11 Approximate string matching on a suffix trie. . . . . 37
4.12 $k$-mismatches runtime . . . . . 37
4.13 Multiple exact string matching runtime . . . . . 39
4.14 Multiple $k$-mismatches runtime . . . . . 39

5.1 Filtration with gapped $q$-grams . . . . . 42
5.2 Filtration with approximate seeds . . . . . 49

A.1 Example of $k$-differences global aligment via DP. . . . . 72
A.2 Exact dictionary search on a suffix trie. . . . . 73

# LIST OF TABLES

# LIST OF AlgorithmS


2.1 GODOWN($x$) . . . . . 13
2.2 GORIGHT($x$) . . . . . 13
2.3 EXACTSEARCH($t, p$) . . . . . 14
4.1 L($x, c$) . . . . . 24
4.2 R($x, c$) . . . . . 24
4.3 GOROOT($x$) . . . . . 24
4.4 GODOWN($x, c$) . . . . . 24
4.5 GODOWN($x$) . . . . . 25
4.6 GORIGHT($x$) . . . . . 25
4.7 Exact string matching on a $q$-gram index. . . . . 27
4.8 GOROOT($x$) . . . . . 34
4.9 GODOWN($x, c$) . . . . . 34
4.10 DFS($x, d$) . . . . . 34
4.11 KMISMATCHES($t, p, e$) . . . . . 35
4.12 KDIFFERENCES($t, p, D$) . . . . . 36
4.13 MULTIPLEEXACTSEARCH($t, p$) . . . . . 38
4.14 MULTIPLEKMISMATCHES($t, p, e$) . . . . . 38
A.1 Exact dictionary search on a radix trie. . . . . 72