Approximate string matching in the next-generation 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 201X

Datum des Disputation: ***XX.XX.201X***

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

Abstract

Bla bla bla.

CONTENTS

1. Introduction . . . 1
1.1 Motivation . . . 1
1.2 Organization of this manuscript . . . 1

Part I Approximate String Matching 3

2. Stringology preliminaries . . . 5
2.1 Definitions . . . 5
2.2 Alignments . . . 5
2.3 Distance functions . . . 6
2.4 Optimal alignments . . . 7
2.5 String matching . . . 8
2.5.1 Online methods . . . 9
2.5.2 Indexed methods . . . 10
2.5.3 Filtering methods . . . 14

3. Online Methods . . . 17
3.1 Banded Myers' bit-vector algorithm . . . 17
3.2 Increased bit-parallelism using SIMD instructions . . . 17

4. Indexed Methods . . . 19
4.1 Classic Full-Text Indices . . . 20
4.1.1 Suffix array . . . 20
4.1.2 q-Gram index . . . 21
4.2 Succint Full-Text Indices . . . 22
4.2.1 Burrows-Wheeler transform . . . 22
4.2.2 Rank dictionaries . . . 24
4.2.3 FM-index . . . 26
4.3 Multiple backtracking . . . 26
4.3.1 k-Mismatches . . . 26
4.3.2 k-Differences . . . 26

5. Filtering Methods . . . 27
5.1 Gapped q-grams . . . 27
5.2 Approximate seeds . . . 27
5.3 Suffix filters . . . 27

Part II Read Mapping 29

6. Sequencing preliminaries 31
6.1 Sequencing technologies 31
6.1.1 Illumina 31
6.1.2 Ion Torrent 31
6.1.3 454 Life Sciences 31
6.1.4 SOLiD 31
6.2 Sequencing applications 32
6.2.1 DNA-seq 32
6.2.2 RNA-seq 32
6.2.3 ChIP-seq 32
6.3 Sequencing quality 32

7. Mappability 33
7.1 Genome mappability 33
7.1.1 Uniqueome 34
7.1.2 Paired-end mappability 35
7.1.3 Pileup mappability 35
7.2 Mapping quality score 35
7.3 Genome mappability score 36

8. Read mappers engineering 39
8.1 Masai 39
8.1.1 Approximate seeds 39
8.1.2 Multiple backtracking 39
8.1.3 Single-end mapping 39
8.1.4 Paired-end mapping 39
8.2 Yara 39
8.2.1 Single-end mapping 39
8.2.2 Paired-end mapping 39
8.2.3 Parallelization 39
8.2.4 Hardware acceleration 39

9. Read mappers evaluation 41
9.1 Popular read mappers 41
9.1.1 Bowtie 41
9.1.2 BWA 42
9.1.3 Soap 43
9.1.4 SHRiMP 43
9.1.5 RazerS 43
9.1.6 mr(s)Fast 44
9.1.7 GEM 44
9.1.8 Masai 44
9.1.9 Yara 44
9.2 Rabema results 46
9.3 Variant detection results 46
9.4 Runtime results 46

10. Discussion . . . 47

A. Related problems . . . 49
A.1 Dictionary search . . . 49
A.1.1 Online methods . . . 49
A.1.2 Indexed methods . . . 49
A.1.3 Filtering methods . . . 49
A.2 Local similarity search . . . 50
A.2.1 Online methods . . . 50
A.2.2 Indexed methods . . . 50
A.2.3 Filtering methods . . . 50
A.3 Overlaps computation . . . 50
A.3.1 Online methods . . . 51
A.3.2 Indexed methods . . . 51

B. Declaration . . . 53

Bibliography . . . 54

Chapter 1 Introduction

1.1 Motivation

This work has been motivated by recent advances of molecular genetics. The human genome has been sequenced in 2001. Also mouse, drosophila, etc. Nowadays # reference model genomes are available in genbank.

Next-generation sequencing has been the second revolution. NGS produces billions of reads for 1000$ dollars. Why should one re-sequence a known genome? Resequencing applications include variant calling, etc. So NGS impacts biomedicine.

Given a set of reads, two approaches are possible: assembly and mapping.

Assembly methods are based on overlaps, de brujin graphs, or...

Read mapping methods work on a previously assembled reference genome.

The typical SNPs analysis pipeline 1.1 consists of...

In this work we focus on read mapping, although many core algorithms considered are also applicable to assembly, as well as to later pipeline stages.

Figure 1.1: *NGS pipeline.*

- Plan A: de-novo assembly
- Plan B: reference mapping
- Plan C: reference guided de-novo assembly

1.2 Organization of this manuscript

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. The reader familiar with basic stringology can skip this section and proceed to section **??**.

2.1 Definitions

Let us start by defining primitive objects of stringology: alphabets and strings. An alphabet is a finite 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 ϵ the empty string s.t. $|\epsilon| = 0$. Given an alphabet Σ, we define $\Sigma^0 = \{\epsilon\}$ as the set containing the empty string, Σ^n as the set of all strings over Σ of length n, and $\Sigma^* = \cup_{n=0}^{\infty} \Sigma^n$ as the set of all strings over Σ. Finally, we call any subset of Σ^* a language over Σ.

We now define concatenation, the most fundamental operation on strings. The concatenation operator of two strings is denoted with $\cdot$ and defined as $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$. Given two strings, $x \in \Sigma^m$ with $x = x_1 x_2 \dots x_m$, and $y \in \Sigma^n$ with $y = y_1 y_2 \dots y_n$, their concatenation $x \cdot y$ (or simply denoted xy) is the string $z \in \Sigma^{m+n}$ consisting of the symbols $x_1 x_2 \dots x_m y_1 y_2 \dots y_n$.

From concatenation we can derive the notion of prefix, suffix, and substring. 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. $y = w \cdot x \cdot z$, and then we say that x occurs within y at position $|w|$.

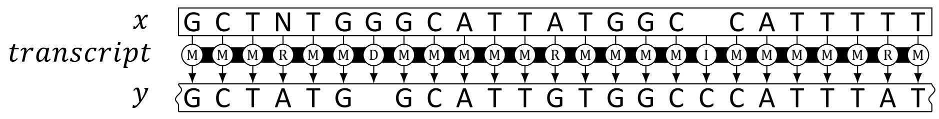
Example 2.1. These definitions allow us to model basic biological sequences. Let us consider the alphabet consisting of DNA bases: $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. Examples of strings over Σ are $x =$A, $y =$AGGTAC, $z =$TA. For instance, $y \in \Sigma^6$ and $|y| = 6$. Moreover, the concatenation $x \cdot z$ produces ATA. The string x is a prefix of y, and the string z is a substring of y occurring at position 4 in y.

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.*



x
G C T N T G G G C A T T A T G G C C A T T T T T
transcript
M M M R M M D M M M M M R M M M M I M M M M M R M
y
G C T A T G G C A T T G T G G C C C A T T T A T

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 Σ. Subsequently, we can define the alignment of two strings of length m, n over Σ to be a string of length between $\min\{m, n\}$ and $m + n$ over the pair alphabet $(\Sigma \cup \{-\}) \times (\Sigma \cup \{-\})$.

Example 2.2. An alignment of the strings $x = AAAA$ and $y = CCCC$ is given by the string $z = \binom{A}{A}\binom{A}{-}\binom{A}{C}\binom{G}{G}$

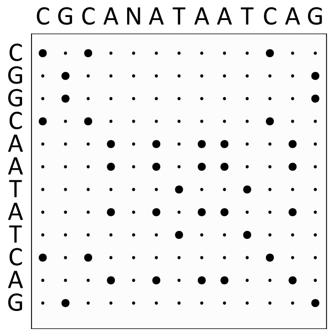
A dotplot is a way to visualize any alignment between two strings and highlight their similarities. Given two string x, y of length m, n, a dotplot is a $m \times n$ matrix containing a dot at position (i, j) iff the symbol x_i matches symbol y_j. We define a dotplot trace to be a monotonical path in 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 **??**.

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

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



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 ∞ in order to disallow indels.

Example 2.3. 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:

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

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

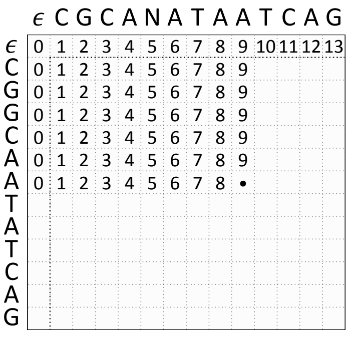
$$d(\epsilon, y_{1..j}) = \sum_{l=1}^{j} \omega(-, y_l) \text{ for all } 1 \leq j \leq n \tag{2.5}$$

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 $O(nm)$ 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 $O(\min\{n, m\})$, 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 $O(m + n)$ 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 $D[m, n]$.

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



	ϵ	C	G	C	A	N	A	T	A	A	T	C	A	G
ϵ	0	1	2	3	4	5	6	7	8	9	10	11	12	13
C	0	1	2	3	4	5	6	7	8	9				
G	0	1	2	3	4	5	6	7	8	9				
G	0	1	2	3	4	5	6	7	8	9				
C	0	1	2	3	4	5	6	7	8	9				
A	0	1	2	3	4	5	6	7	8	9				
A	0	1	2	3	4	5	6	7	8	•				
T														
A														
T														
C														
A														
G														

2.5 String matching

We can now define *exact string matching*, perhaps the most fundamental problem in stringology. Given a string p (with $|p| = m$) called the *pattern* and a longer string t (with

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

$|t| = n$) called the *text*, the exact string matching problem is to find all occurrences, if any, of pattern p into text t [Gusfield, 1997]. This problem has been extensively studied from the theoretical standpoint and is well solved in practice.

The definition of distance functions between strings let us generalize exact string matching into a more challenging problem: *approximate string matching*. Given a text t, a pattern p, and a *distance threshold* $k \in \mathbb{N}$, the approximate string matching (a.s.m.) problem is to find all occurrences of p into t within distance k. The a.s.m. 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. Under these distances, we define the *error rate* as $\epsilon = \frac{k}{m}$, with $0 < \epsilon < 1$ given the above conditions, and we alternatively refer to these a.s.m. problems as ϵ-mismatches and ϵ-differences.

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[2] 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 chapter **??** and directly used in applications of part **??**. 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 succint survey on indexed methods [Navarro *et al.*, 2001].

2.5.1 Online methods

We consider two classes of algorithms for online string matching, those based on automata and those based on dynamic programming.

[2] When the error rate is low.

Automata

Exact search of one pattern. Knuth-Morris-Pratt automaton.

Exact search of multiple patterns. Aho-corasick automaton.

Approximate search of one pattern. Ukkonen automaton.

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, a.s.m. consists of computing the edit distance between the pattern and all substrings 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.

We pose $x = p$ and $y = t$ and consider Equation 2.6. 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 $t = ...$*

	ϵ	C	G	C	A	N	A	T	A	A	T	C	A	G	A	A	A
ϵ	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

No matter how efficient online methods can be, these approaches quickly become impractical when the text is long and searched frequently. If the text is static and given in advance, we are allowed to preprocess it. Therefore we build an index of the text beforehand and use it to speed up subsequent searches. To this intent we introduce the *suffix tree*, an optimal data structure to index all substrings of a text. We take the suffix tree as an idealized data structure to elegantly expose our indexed algorithms solving string matching problems. In chapter **??** we will consider other substring indices to replace the suffix tree in practice.

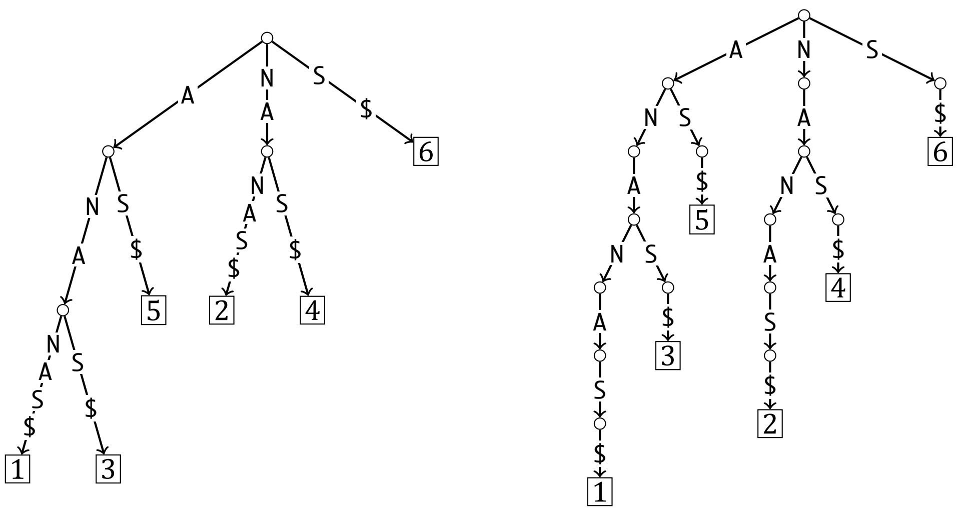
Trie

Suffix tree and suffix trie

The suffix tree [Morrison, 1968] is a lexicographically ordered tree data structure representing all suffixes of a string. Assume w.l.o.g. a string s of length n, padded with a *terminator symbol* \$ not being part of the string alphabet Σ[3]. The suffix tree $\mathcal{S}$ of the string s has one node designated as the root and n leaves, each one pointing to a distinct suffix of s, denoted as $l_1 \dots l_n$. Each internal node has more than one child, and each edge is labeled with a non-empty substring of s. Each path from the root to a leaf l_i spells the suffix $s_{i..n}$. Figure 2.5 illustrates.

In the following of this manuscript we consider w.l.o.g. *suffix tries* instead of suffix trees. On suffix tries, internal nodes can have only one child and each edge is labeled by one single character (see Figure **??**). This fact simplifies the exposition of all given algorithms without affecting their runtime complexity nor their result. However, we remark that all given algorithms can be generalized to work on trees.

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



Therefore, from now on we assume the text t to be indexed using a suffix trie $\mathcal{T}$. Given a node x of $\mathcal{T}$, we denote with $label(x)$ the label of the edge entering into x, with $\mathbb{C}(x)$ the set of children of x being internal nodes, with $\mathbb{E}(x)$ the set of children of x being leaves, and with $\mathbb{L}(x)$ the set of all leaves of the subtree rooted in x. We remark that entering edges of internal nodes in $\mathbb{C}(x)$ are always labeled with symbols in Σ, while entering edges of leaves in $\mathbb{L}(x)$ and $\mathbb{E}(x)$ are always labeled with terminator symbols.

Exact string matching

Using the suffix trie $\mathcal{T}$ of the text t, we can find all occurrences of a pattern p into t in optimal time $\mathcal{O}(m)$, thus independently of n. Algorithm 2.1 searches a pattern p by starting in the root node of $\mathcal{T}$ and following the path spelling the pattern. If the search ends

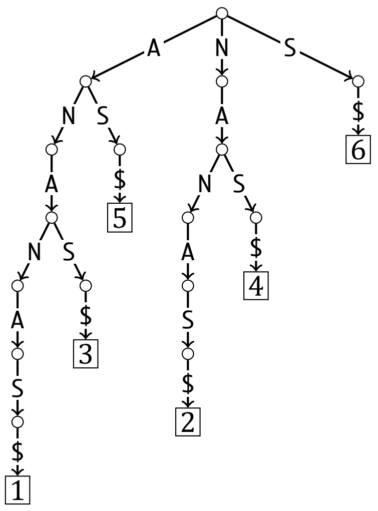
[3] The terminator symbol is necessary to ensure that no suffix $s_{i..n}$ is a prefix of another suffix $s_{j..n}$.

up in a node x, each leaf $l_i \in \mathbb{L}(x)$ points to a distinct suffix $t_{i..n}$ such that $t_{i..i+m} = p$. Algorithm 2.1 is correct since each path from the root to any internal node of the suffix trie $\mathcal{T}$ spells a different unique substring of t; consequently all equal substrings of t are represented by a single common path.

Algorithm 2.1 Exact string matching on a suffix trie.

```
1: procedure ExactSearch($x, p$)
2:     if $p = \epsilon$ then
3:         report $\mathbb{L}(x)$
4:     else if $\exists\, c_x \in \mathbb{C}(x)$ : $label(c_x) = p_1$ then
5:         ExactSearch($c_x, p_{2..|p|}$)
6:     end if
7: end procedure
```

Figure 2.6: Exact string matching on a suffix tree. The pattern NA is searched exactly in the text ANANAS.



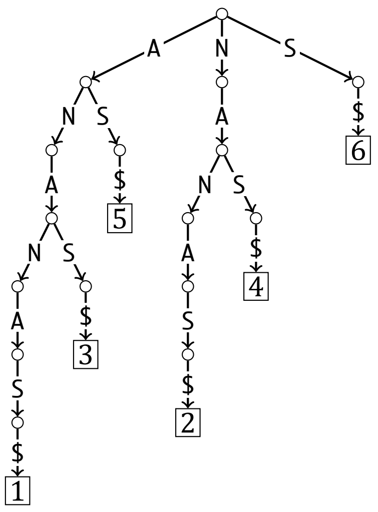
Backtracking k-mismatches

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 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 2.2.

Algorithm 2.2 k-mismatches on a suffix trie.

```
1: procedure KMismatches($x, p, e$)
2:     if $e = 0$ then
3:         ExactSearch($x, p$)
4:     else
5:         for all $c_x \in \mathbb{C}(x)$ do
6:             if $label(c_x) = p_1$ then
7:                 KMismatches($c_x, p_{2..|p|}, e$)
8:             else
9:                 KMismatches($c_x, p_{2..|p|}, e - 1$)
10:            end if
11:    end if
12: end procedure
```

Figure 2.7: *Approximate string matching on a suffix tree.*



Backtracking k-differences

We can compute k-differences on a suffix tree in two different ways. Algorithm 2.3 explicitly enumerates errors by recursing on the suffix trie. Algorithm 2.4 computes the edit distance on the suffix trie.

In algorithm 2.4, 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 $O(1)$ 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 2.3 reports more occurrences than algorithm 2.4. Discuss neighborhood, condensed neighborhood, and super-condensed neighborhood.

Algorithm 2.3 k-differences on a suffix trie.

```
procedure KDifferences($x, p, e$)
    if $e = 0$ then
        ExactSearch($x, p$)
    else
        KDifferences($x, p_{2..|p|}, e - 1$)
        for all $c_x \in \mathbb{C}(x)$ do
            KDifferences($c_x, p, e - 1$)
            if $label(c_x) = p_1$ then
                KDifferences($c_x, p_{2..|p|}, e$)
            else
                KDifferences($c_x, p_{2..|p|}, e - 1$)
            end if
    end if
end procedure
```

Algorithm 2.4 k-difference on a suffix trie.

```
procedure KDifferences($x, p, e$)
    for all $c_x \in \mathbb{C}(x)$ do
        KDifferences($c_x, p_{2..|p|}, e - d_E(repr(x), p)$)
end procedure
```

2.5.3 Filtering methods

The goal of filtering methods is to obtain algorithms that have favorable average running times. The principle under which they work is that large and uninteresting portions of the text can be quickly discarded, while narrow and highly similar portions can 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 ϵ, in addition to the alphabet size σ, 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 Σ 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 ϵ-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 can derive full-sensitive partitioning strategies by application of the pigeonhole principle. Instead, filters based on q-grams consider all *overlapping* substrings of the pattern having length q, the so-called q-grams. Eventually, simple lemmas gives us lower bounds on the number of q-grams that must be present in a narrow window of the text as necessary condition for an occurrence of the pattern.

Seed filters

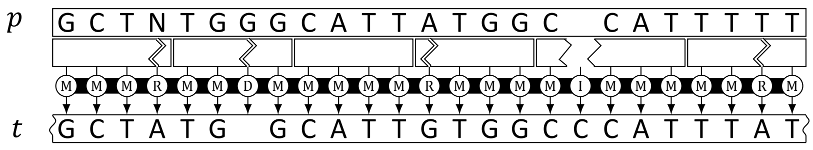
We start by considering the case of two arbitrary strings x, y s.t. $d_E(x, y) \leq k$. If we partition w.l.o.g. y into $k+1$ non-overlapping seeds, then at least one seed will occur as a factor of x.

Lemma 2.1. *[Baeza-Yates and Navarro, 1999] Let x, y be two strings s.t. $d_E(x, y) = k$ for some $k \in \mathbb{N}_0$. If $y = y^1 y^2 \dots y^{k+1}$ then $x = a y^i b$ for some a, b.*

Proof. We proceed by induction on k. For $k = 0$, the string y is partitioned into only one factor, y itself. The condition $d_E(x, y) = 0$ implies $x = y$, which is true for $a = \epsilon$ and $b = \epsilon$. We suppose the case $k = j - 1$ to be true, thus since $d_E(x, y) = j - 1$ and $y = y^1 y^2 \dots y^j$ then $x = a y^i b$ for some a, b. We consider the case $k = j$. The j-th error can be in (i) $y^1 \dots y^{i-1}$, (ii) y^i, or (iii) $y^{i+1} \dots y^j$. In case i or iii, $x = a y^i b$ clearly holds. In case ii, if we partition y^i in two factors $y^{i'}$ and $y^{i''}$, then either $x = a y^{i'} b'$ or $x = a' y^{i''} b$. □

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. Figure 2.8 shows an example. Note that we are reducing one approximate search into many smaller exact searches. 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 $m - k \leq |s| \leq m + k$ containing one seed of p.

Figure 2.8: *Filtration with exact seeds.*


p
G C T N T G G G C A T T A T G G C C A T T T T T
M M M R M M D M M M M M R M M M M I M M M M M R M
t
G C T A T G G C A T T G T G G C C C A T T T A T

How many verifications we expect to have?

$$p_\alpha = \frac{1}{\sigma} \text{ for all } \alpha \in \Sigma \tag{2.8}$$

$$\Pr(H > 0) = \frac{1}{\sigma^q} \tag{2.9}$$

$$E(H) = \sum_{i=1}^{n-q+1} \Pr(H > 0) = \frac{n - q + 1}{\sigma^q} \leq \frac{n}{\sigma^q} \tag{2.10}$$

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

Which is the runtime of the algorithm?

How to choose the partitioning? Which length of q makes filtration lossless? $q = \lfloor \frac{m}{k+1} \rfloor$.

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 distance[4]. It gives a lower bound on the q-gram similarity $\tau_q(x, y)$ for any two strings x, y for which $d_E(x, y) = k$. 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. *[?] Let x, y be two strings with edit distance k and min$\{|x|, |y|\} = m$, then x and y have q-gram similarity $\tau_q(m, k) \geq m - q + 1 - kq$.*

Proof. By induction on k. □

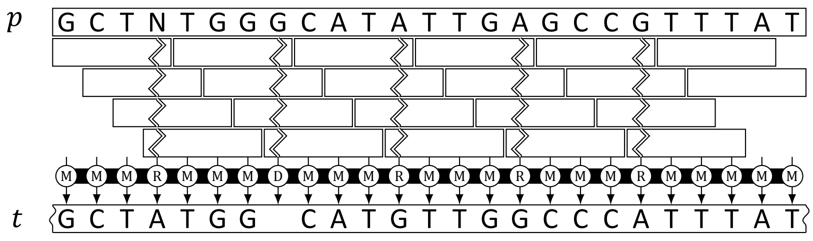
How can we use this result to solve approximate string matching? The lemma itself does not give us the direct solution, indeed it considers the edit distance between two arbitrary strings, while in a.s.m. the pattern can match any substring of the text. In the case of Hamming distance, if the pattern matches any substring s of t, then s must have length m. In the case of edit distance, it must hold $m - k \leq |s| \leq m + k$. The dot-plot representation helps us to visualize this concept. Hamming distance occurrences cover one single diagonal of the dot-plot, while edit distance occurrences are enclosed inside a parallelogram of side $2k + 1$.

Overlapping parallelograms?

We can design a filtration algorithm that scans the text and counts how many q-grams of the pattern falls 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. To speed up the filtration phase, the q-grams can be counted with the help of a substring index.

Which length of q makes filtration lossless? $q = \lfloor \frac{m}{k+1} \rfloor$.

Figure 2.9: *Filtration with q-grams.*


p
G C T N T G G G C A T A T T G A G C C G T T T A T
M M M R M M M D M M M R M M M R M M M R M M M M M
t
G C T A T G G C A T G T T G G C C C A T T T A T

[4] Thus it relates q-gram similarity also to Hamming distance.

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 text collections. A practical study on suffix trees [Kurtz, 1999] considers efficient implementations achieving 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 bigger memories. Unfortunately, memory latency improvements have been more modest, leading to the so called memory wall effect [**?**]: 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 deliberately 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 succint or even compressed data-structure 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 2.X GB of memory and at the same time improve query time by a factor of X over classic indices!

In this chapter, we introduce some classic full-text indices (suffix arrays and q-gram indices) and subsequently succint full-text indices (our FM-index implementations) replacing suffix trees. Afterwards we give 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 tree is superfluous for pattern matching. We can omit suffix tree's internal nodes and outgoing edges. Indeed, leaves pointing to the sorted suffixes are sufficient to perform exact pattern matching or even trie traversals. We can compute on the fly paths from the root to any internal node, via binary searches over the leaves. We are thus willing to pay an additional logarithmic time complexity to reduce space by a linear factor.

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.*

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 [**?**]. 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. For instance, we construct the suffix array of the human genome in about one hour on a modern computer and the suffix array itself fits in 12 GB of memory.

Suffix trie traversal

We now concentrate on replacing suffix tree functionalities. We replace algorithm 2.1 by algorithm 4.1. The worst case runtime of algorithm 4.1 is $\mathcal{O}(m \log n)$, as the binary search consists of $\mathcal{O}(\log n)$ steps, and each step is performed in $\mathcal{O}(m)$ time, as it requires in the worst case a full lexicographical comparison between the pattern and any suffix of the text.

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 $s_{A[i]}$, $s_{A[i+1]}$ 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 4.1 Exact string matching on a suffix array.

```
1: procedure ExactSearch($x, p$)
2:     if $p = \epsilon$ then
3:         report $\mathbb{L}(x)$
4:     end if
5: end procedure
```

Figure 4.2: *Exact string matching on a suffix array. The pattern NA is searched exactly in the text ANANAS.*

4.1.2 q-Gram index

If we prune our idealized suffix tree to a fixed height q, we can improve again the query time over the suffix array. The idea is to supplement the suffix array A with an additional q-gram directory D storing the suffix array ranges computed by algorithm 4.1 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 $h : \Sigma^q \to [1 \ldots \sigma^q]$ defined as in [Knuth, 1973]:

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

where $p \in \Sigma^q$ is any q-gram and the function $\rho : \Sigma \to [0 \ldots \sigma - 1]$ denotes the lexicographic rank of any symbol in the alphabet Σ. 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)] = LowerSearch(p). Note that the right interval returned by algorithm **??** is equivalent to the left interval of the lexicographically following q-gram and therefore available in $D[h(p) + 1]$.

Figure 4.3: *q-Gram index for the string ANANAS.*

Suffix trie traversal

At this point, we are able to replace algorithm 4.1 with algorithm 4.2. Algorithm 4.2 runs in $\mathcal{O}(q)$ time, but in practice the time to compute the function h can be neglected and the lookup requires fetching only two memory locations from D. The downside is that in practice this approach is applicable only for small alphabet and pattern sizes. For instance, $|\Sigma| = 4$ and $q = 14$ require a directory consisting of 268 M entries that, using a 32 bits encoding, consume 1 GB of memory.

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. 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 sorted by their relative text positions. The latter property allows to compress the suffix array bucket-wise e.g. using Elias δ-coding [**?**] or to devise cache-oblivious strategies to process the occurrences [**?**].

If the patterns are longer than q, the q-gram index is still useful. We can devise an hybrid algorithm using the directory D to conduct the search up to depth q and later continue with binary searches. This hybrid index cuts the most expensive binary searches and increases memory locality. Furthermore, this hybrid index can be useful if the suffix array has to reside in external memory.

Algorithm 4.2 Exact string matching on a q-gram index.

```
1: procedure ExactSearch($A, D, p$)
2:     report $A[D[h(p)], D[h(p) + 1]]$
3: end procedure
```

4.2 Succint Full-Text 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 [**?**] 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 widely re-implemented and employed by many popular Bioinformatics tools e.g. Bowtie [Langmead *et al.*, 2009], BWA [Li and Durbin, 2009], Soap2 [Li *et al.*, 2009b], 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 succint FM-index implementations covering texts and text collections.

4.2.1 Burrows-Wheeler transform

Let t be a string of length n over an alphabet Σ, terminated by a symbol $\$ \notin \Sigma$ such that $\$ <_{lex} c$ for all $c \in \Sigma$. Consider the square matrix M consisting of all cyclic shifts of the text t (the i-th cyclic shift has the form $t_{i \dots n} t_{1 \dots i-1}$) sorted in lexicographical order. Note how the matrix M is related to the suffix array A of t: the cyclic shift in the i-th row is $M[i :] = t_{A[i] \dots n} t_{1 \dots A[i-1]}$ (except when $A[i] = 1$).

Definition 4.2. The BWT of t is the string l obtained concatenating the symbols in the last column of the cyclic shifts matrix M, i.e. $l = M[: n]$.

The matrix M is conceptual. 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 $t_{A[i]-1}$ (and when $A[i] = 1$ the character t_n). Various direct BWT construction algorithms 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.

Multiple sequences

TODO.

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

We recover t as follows:

$$t_i = f_{\Psi^{i-1}(j)} \tag{4.2}$$

where

$$\Psi^0(j) = j \tag{4.3}$$

$$\Psi^{i+1}(j) = \Psi(\Psi^i(j)) \tag{4.4}$$

Example 4.1. Recover t.
$l =$
$\Psi = (\dots)$

We recover $\bar{t}$ as follows:

$$\bar{t}_i = t_{LF^{i-1}(j)} \tag{4.5}$$

where

$$LF^0(j) = j \tag{4.6}$$

$$LF^{i+1}(j) = LF(LF^i(j)) \tag{4.7}$$

Example 4.2. Recover $\bar{t}$.
$l =$
$LF = (\ldots)$

LF-mapping

Again, the permutation LF is conceptual. We do not have to explicitly store it 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 of the matrix M [Burrows and Wheeler, 1994]:

- For all $i \in [1, n]$ I, the character l_i precedes the character f_i in the original text t;
- For all characters $c \in \Sigma$ the i-th occurrence of c in f corresponds to the i-th occurrence of c in l.

Given the above observations, we define the permutation LF as [Burrows and Wheeler, 1994; Ferragina and Manzini, 2000]:

$$LF(i) = C(l_i) + Occ(l_i, i) \tag{4.8}$$

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

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 represent the function Occ in succint space and at the same time answer efficiently the question: how many times a given character c occurs in the prefix $l_{1 \ldots i}$? The general problem on arbitrary sequences has been tackled by several studies on the succint 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 4.3. Given a sequence s over an alphabet Σ and a character $c \in \Sigma$, $rank_c(s, i)$ returns the number of occurrences of c in the prefix $s_{1 \ldots i}$.

Rank dictionaries maintain a succint (or compressed) representation of the input sequence and attach a dictionary to it. By doing so, it is possible to answer rank queries in constant time on the RAM model, using $n + o(n)$ bits for an input binary sequence of n bits [Jacobson, 1989]. First we consider the binary case $\Sigma_B = \{0, 1\}$ and later we extend it to arbitrary alphabets.

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 sequence $s \in \Sigma_B$, we partition it in blocks of size $b = \log n$ bits. We attach to the binary sequence 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 $s_{1 \dots ib}$, i.e. $R[\frac{i}{b}] = rank_1(s, \frac{i}{b})$. Note that $rank_0(s,i) = i - rank_1(s,i)$ so we consider only $rank_1(s,i)$. Therefore we are able to rewrite our rank query as:

$$rank_1(s,i) = R[\frac{i}{b}] + rank_1(s_{\frac{i}{b} \dots \frac{i}{b}+b}, i \mod b) \tag{4.9}$$

and answer it in constant time as (i) we fetch in constant time the rank summary from R and (ii) we compute in constant time[1] the number of occurrences of the bit 1 in the subsequence of length $\mathcal{O}(\log n)$. 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.

To squeeze our rank dictionary to consume only $n+o(n)$ bits of space, we add another array R' summarizing the ranks on $\log^2 n$ bits boundaries and let our initial array R store only local positions within the corresponding block defined by R'. We rewrite $rank_1(s,i)$ accordingly:

$$rank_1(s,i) = R'[\frac{i}{b^2}] + R[\frac{i}{b}] + rank_1(s_{\frac{i}{b} \dots \frac{i}{b}+b}, i \mod b) \tag{4.10}$$

Each entry of R now has to represent only values in the range $[1, \log^2 n]$ and thus consumes $\log \log^2 n$ only bits. This two-levels rank dictionary consumes n bits for the input sequence, $\mathcal{O}(\frac{n}{\log n})$ bits for R' and $\mathcal{O}(\frac{n \log \log^2 n}{\log n})$ bits for R. Overall, this two-levels rank dictionary consumes $n + o(n)$ bits.

Small alphabets

The extension to small alphabets, e.g. Σ_{DNA} is easy. Here we show how to extend the one-level rank dictionary. Given an input sequence $s \in \Sigma_{\text{DNA}}$ of size n bits (and length $\frac{n}{\log \sigma}$ symbols), we partition it in blocks of $b_\sigma = \frac{\log n}{\log \sigma}$ symbols (as before, $b = \log n$ bits). We supplement each block with an occurrences summary for all symbols in Σ, thus we use a matrix R_σ of size $\frac{n}{b_\sigma} \times \sigma$ entries. We rewrite $rank_c(s,i)$[2] as:

$$rank_c(s,i) = R_\sigma[\frac{i}{b_\sigma}][\rho(c)] + rank_c(s_{\frac{i}{b_\sigma} \dots \frac{i}{b_\sigma}+b_\sigma}, i \mod b_\sigma) \tag{4.11}$$

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 $\log n$ bits. The matrix R_σ has $\frac{n}{\log n}$ entries, each one consuming $\sigma \log n$ bits. Thus R_σ consumes $n\sigma$ bits, and the whole rank dictionary $n + n\sigma$ bits.

[1] On modern processors using the SSE 4.2 popcnt instruction [?], otherwise by means of the four-Russians tabulation technique [?].

[2] Note that i is the i-th symbol in s, not the i-th bit in s.

Wavelet tree

For large alphabets.

4.2.3 FM-index

We now turn to the problem of implementing a full-text index exploiting the LF-mapping. First we see how to emulate a traversal of the nodes of a 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 is necessary to locate the occurrences in the original text.

Suffix trie traversal

We can use the permutation LF to decode the intervals computed during a suffix trie traversal. This is possible because of the relationship between the cyclic shifts matrix M and the suffix array A. We show how to answer $sl^{-1}(v, c)$, i.e. given a node v and a character $c \in \Sigma$, return the node w such that $repr(w) = c \cdot repr(v)$.

We easily obtain the interval of the node labeled by c as $[C(c), C(c+1)]$. Now we suppose that we are in an arbitrary node v of known interval $[b_v, e_v]$ and we want to navigate to the node w of unknown interval $[b_w, e_w]$ such that $repr(w) = c \cdot repr(v)$ for some $c \in \Sigma$. Thus, we know all suffixes of t prefixed by $repr(v)$ and we are looking for all the suffixes of t prefixed by $c \cdot repr(v)$. All these characters c are in $l_{b_v \dots e_v}$, since l_i is the character $t_{A[i]-1}$ 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:

$$\begin{aligned} LF(b) &= C(l_b) + Occ(l_b, b) \\ &= C(c) + Occ(c, b) \\ &= C(c) + Occ(c, b_v - 1) + 1 \end{aligned} \tag{4.12}$$

Analogously, we can rewrite $LF(e)$ as $C(c) + Occ(c, e_v)$.

Locating occurrences

TODO.

4.3 Multiple backtracking

4.3.1 k-Mismatches

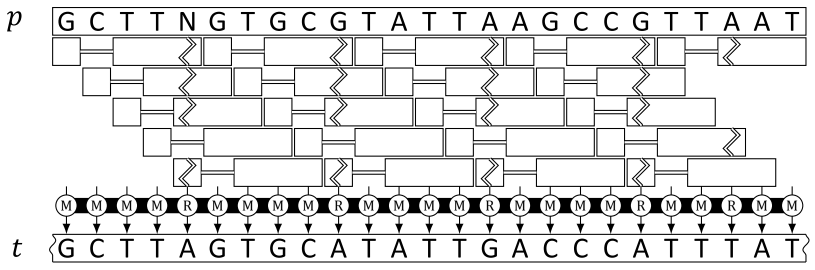
4.3.2 k-Differences

Chapter 5 Filtering Methods

5.1 Gapped q-grams

TODO.

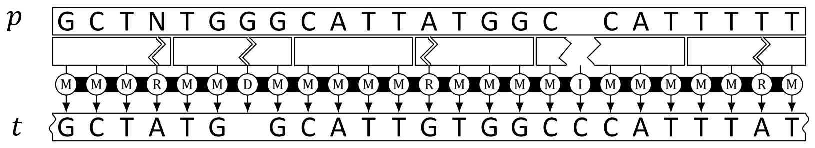
Figure 5.1: *Filtration with gapped q-grams.*


p
G C T T N G T G C G T A T T A A G C C G T T A A T
M M M M R M M M M R M M M M R M M M M R M M M R M M
t
G C T T A G T G C A T A T T G A C C C A T T T A T

5.2 Approximate seeds

TODO.

Figure 5.2: *Filtration with approximate seeds.*


p
G C T N T G G G C A T T A T G G C C A T T T T T
M M M R M M D M M M M M R M M M M I M M M M M R M
t
G C T A T G G C A T T G T G G C C C A T T T A T

5.3 Suffix filters

Part II

READ MAPPING

Chapter 6 Sequencing preliminaries

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 Σ and a read r sequenced at location l from the template $g_{l \ldots l+|r|-1}$. We define the base calling error ϵ_i at position i in the read r, as the probability ϵ_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 $g_{l \ldots l+|r|-1}$, is:

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

Chapter 7 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.

7.1 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 $g_{l...l+q-1}$ 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 $M_k^q : \mathbb{N} \rightarrow]0,1]$. 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:

$$2n \leq 4^q \tag{7.1}$$
$$\log 2n \leq \log 4^q \tag{7.2}$$
$$q \geq log_4 2n \tag{7.3}$$

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.

7.1.1 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.

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

	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

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.

7.1.2 Paired-end mappability

7.1.3 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.

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

7.2 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.

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

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:

$$p(l) = \frac{1}{|g| - |r| + 1} \tag{7.4}$$

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:

$$p(l|g, r) = \frac{p(r|g, l)}{\sum_{i=1}^{|g|-|r|+1} p(r|g, i)} \tag{7.5}$$

which in Phred-scale becomes:

$$Q(l|g, r) = -10 \log_{10}[1 - p(l|g, r)] \tag{7.6}$$

Computing the exact mapping quality as in equation 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 7.5 can be neglected. Therefore, equation 7.5 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 7; it is easy to see that the mapping probability in equation 7.5 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.

7.3 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 7.2) 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]:

$$p(l|g) = \sum_{r \in \mathcal{R}(l)} \frac{p(l|g,r)}{|\mathcal{R}(l)|} \tag{7.7}$$

which in Phread-scale becomes:

$$Q(l|g) = \sum_{r \in \mathcal{R}(l)} \frac{1 - 10^{-\frac{Q(l|g,r)}{10}}}{|\mathcal{R}(l)|} \tag{7.8}$$

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

$$\text{GMS}(l) = 100Q(l|g) \tag{7.9}$$

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 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

[2] Equation 3 in [Lee and Schatz, 2012] is not precise, please refer to our equation 7.7.

CHAPTER 8

Read mappers engineering

8.1 Masai

8.1.1 Approximate seeds

8.1.2 Multiple backtracking

8.1.3 Single-end mapping

All-mapping

Mapping by strata

8.1.4 Paired-end mapping

8.2 Yara

8.2.1 Single-end mapping

All-mapping

Mapping by strata

8.2.2 Paired-end mapping

All-mapping

Mapping by strata

8.2.3 Parallelization

8.2.4 Hardware acceleration

Chapter 9
Read mappers evaluation

9.1 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.

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.1.4 SHRiMP

SHRiMP 2 TODO.

9.1.5 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.

9.1.6 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.

9.1.7 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.

9.1.8 Masai

9.1.9 Yara

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

9.2 Rabema results

9.3 Variant detection results

9.4 Runtime results

0

Chapter 10 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. □

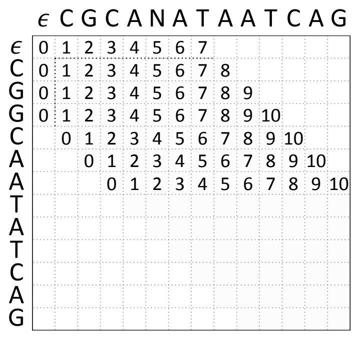
A.1.2 Indexed methods

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 $t =$ ....*



	ϵ	C	G	C	A	N	A	T	A	A	T	C	A	G
ϵ	0	1	2	3	4	5	6	7						
C	0	1	2	3	4	5	6	7	8					
G	0	1	2	3	4	5	6	7	8	9				
G	0	1	2	3	4	5	6	7	8	9	10			
C		0	1	2	3	4	5	6	7	8	9	10		
A			0	1	2	3	4	5	6	7	8	9	10	
A				0	1	2	3	4	5	6	7	8	9	10
T														
A														
T														
C														
A														
G														

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

```
1: procedure ExactSearch($x, p$)
2:     if $p = \epsilon$ then
3:         report $\mathbb{E}(x)$
4:     else if $\exists\, c_x \in \mathbb{C}(x) : label(c_x) = p_1$ then
5:         ExactSearch($c_x, p_{2..|p|}$)
6:     end if
7: end procedure
```

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.

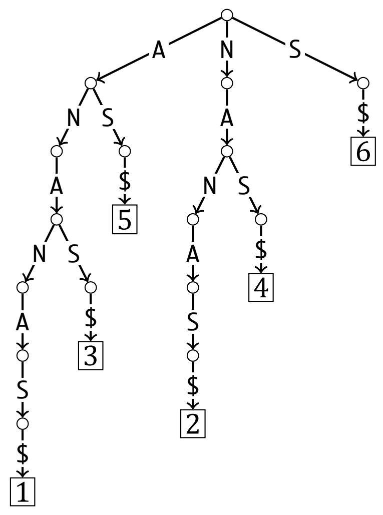
A.2.3 Filtering methods

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

A.3 Overlaps computation

Define problem.

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



A.3.1 Online methods

DP solution.

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
February 15, 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.

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.

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.

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

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*.

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.

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.

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., 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.

LIST OF FIGURES

1.1 NGS pipeline. 1

2.1 Example of edit transcript. 6
2.2 Example of dotplot. 7
2.3 Example of DP table. 8
2.4 Example of approximate string matching via DP. 10
2.5 Suffix tree and suffix trie for the string ANANAS. 11
2.6 Exact string matching on a suffix tree. 12
2.7 Approximate string matching on a suffix tree. 13
2.8 Filtration with exact seeds. 15
2.9 Filtration with q-grams. 16

4.1 Suffix array for the string ANANAS. 20
4.2 Exact string matching on a suffix array. 21
4.3 q-Gram index for the string ANANAS. 21

5.1 Filtration with gapped q-grams. 27
5.2 Filtration with approximate seeds. 27

A.1 Example of k-differences global aligment via DP. 50
A.2 Exact dictionary search on a suffix trie. 51

LIST OF TABLES

LIST OF AlgorithmS

2.1 Exact string matching on a suffix trie. 12
2.2 k-mismatches on a suffix trie. 13
2.3 k-differences on a suffix trie. 14
2.4 k-difference on a suffix trie. 14
4.1 Exact string matching on a suffix array. 21
4.2 Exact string matching on a q-gram index. 22
A.1 Exact dictionary search on a radix trie. 50