CHAPTER

1 Background

1.1 Introduction

1.1.1 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 **??** 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.*

1.1.2 Organization of this manuscript

1.2 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 **??**.

1.2.1 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, $x \in \Sigma^n$ with $x = x_1x_2 \dots x_n$, and $y \in \Sigma^m$ with $y = y_1y_2 \dots y_m$, their concatenation $x \cdot y$ (or simply denoted xy) is the string $z \in \Sigma^{n+m}$ consisting of the symbols $x_1x_2 \dots x_ny_1y_2 \dots y_m$.

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 1.1. These definitions allow us to model basic biological sequences. Let us consider the alphabet consisting of DNA bases: $\Sigma = \{\text{A}, \text{C}, \text{G}, \text{T}\}$. 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.

1.2.2 Transformations

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.

Example 1.2. TODO: example of edit transcript.

Edit transcripts lead us to the definition of distance functions between strings. The Hamming distance between two strings $x, y \in \Sigma^n$ is defined as the function $d_H : \Sigma^n \times \Sigma^n \to \mathbb{N}$ counting the number of substitutions necessary to transform x into y. More generally, the edit (or Levenshtein) distance between two strings $x, y \in \Sigma^*$ is defined as the function $d_E : \Sigma^* \times \Sigma^* \to \mathbb{N}$ counting the minimum number of edit operation necessary to transform x into y.

Example 1.3. TODO: example of edit and hamming distance.

1.2.3 Edit distance computation

The edit distance problem is to compute the edit distance between two given strings, along with an optimal edit transcript that describes the transformation **?**. The edit distance problem is a minimization problem and 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, for all $1 \leq i \leq |x|$ and $1 \leq j \leq |y|$ we define with $d_E(x_{1..i}, y_{1..j})$ the edit distance between their prefixes $x_{1..i}$ and $y_{1..j}$. The base conditions of the recurrence relation are:

$$d_E(\epsilon, \epsilon) = 0 \tag{1.1}$$

$$d_E(x_{1..i}, \epsilon) = i \text{ for } 1 \leq i \leq |x| \tag{1.2}$$

$$d_E(\epsilon, y_{1..j}) = j \text{ for } 1 \leq j \leq |y| \tag{1.3}$$

and the recursive case is defined as follows:

$$d_E(x_{1..i}, y_{1..j}) = \min\{d_E(x_{1..i-1}, y_{1..j}) + 1, d_E(x_{1..i}, y_{1..j-1}) + 1, d_E(x_{1..i-1}, x_{1..j-1}) + \delta(x_i, y_j)\} \tag{1.4}$$

The recurrence relation can be computed in time $\mathcal{O}(|x| \cdot |y|)$ using a table of $(n+1) \times (m+1)$ cells. However only $\mathcal{O}(\min\{n, m\})$ space is required.

The table can be filled in four different ways: column-wise, row-wise, diagonal-wise or antidiagonal-wise.

Figure 1.2: *DP table representing the computation of the edit distance $d_E(x_{1..5}, y_{1..4})$.*

An optimal alignment can be computed in time $\mathcal{O}(n + m)$.

1.2.4 Alignments

An alignment is a way of visualizing a transformation between two strings.

The problem of finding the optimal alignment between two strings is the dual of the edit distance problem.

Example 1.4. TODO: example of alignment.

1.3 Overview of string matching

1.3.1 Problem definition

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 $|t| = n$) called the *text*, the exact string matching problem is to find all occurrences, if any, of pattern p into text t **?**. 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.

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 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 **?** and approximate **?** online string matching methods, as well as to a succint survey on indexed methods **?**.

1.3.2 Online methods

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

Automata

Exact search of one pattern. Boyer-moore 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 edit 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.

Let pose $x = t$ and $y = p$ and consider equations **??**. Since an occurrence of the pattern can start anywhere in the text, we change the initialization of the top row as:

$$d_E(\epsilon, y_{1..j}) = 0 \text{ for } 1 \leq j \leq |y| \tag{1.5}$$

and since an occurrence of the pattern can end anywhere in the text, we check every cell of the bottom row for the condition:

$$d_E(x_{1..m}, y_{1..j}) \leq k \text{ for } 1 \leq j \leq |y|. \tag{1.6}$$

Figure 1.3: *DP table representing the match of $p = ...$ in $t = ...$*

1.3.3 Indexed methods

No matter how fast 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. We build an index of the text beforehand and use it to speed up subsequent searches. To this intent, we first introduce *suffix trees*, optimal data structures to index strings. Later on, we consider algorithms solving string matching problems on indices.

Suffix tree and suffix trie

The suffix tree [**?**] 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[1]. 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 $l_1 ... l_n$. 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 **??** 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 1.4: *Suffix tree and suffix trie.*

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

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

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 **??** 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 $l_i \in \mathbb{L}(x)$ points to a distinct suffix $t_{i..n}$ such that $t_{i..i+m} = p$. Algorithm **??** 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 1.1 Exact string matching on a suffix trie.

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

Figure 1.5: *Exact string matching on a suffix tree.*

Backtracking k-mismatches

We can solve k-mismatches by backtracking [**??**] on the suffix trie $\mathcal{T}$, in average time sublinear in n [**?**]. 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 **??**.

Backtracking k-differences

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.

Algorithm 1.2 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 ∈ 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
```

Algorithm 1.3 k-differences on a suffix trie.

```
1:  procedure KDifferences(x, p, e)
2:      if e = 0 then
3:          ExactSearch(x, p)
4:      else
5:          KDifferences(x, p_{2..|p|}, e − 1)
6:          for all c_x ∈ ℂ(x) do
7:              KDifferences(c_x, p, e − 1)
8:              if label(c_x) = p_1 then
9:                  KDifferences(c_x, p_{2..|p|}, e)
10:             else
11:                 KDifferences(c_x, p_{2..|p|}, e − 1)
12:             end if
13:     end if
14: end procedure
```

1.3.4 Filtering methods

Why filtering

Motivate filtering methods.

Pigeonhole principle

Exact seeds

A simple solution to the problem is provided by a filtering algorithm proposed in **?** which reduces an approximate search into smaller exact searches. A pattern p is partitioned into $k + 1$ non-overlapping seeds which are searched in t with the help of $\mathcal{T}$. Since each edit operation can affect at most one seed, for the pigeonhole principle each approximate occurrence of p in t contains an exact occurrence of some seed. However the converse

Algorithm 1.4 k-difference on a suffix trie.

```
1: procedure KDIFFERENCES(x, p, e)
2:     for all c_x ∈ ℂ(x) do
3:         KDIFFERENCES(c_x, p_{2..|p|}, e − d_E(repr(x), p))
4: end procedure
```

is not true, consequently we must verify whether any candidate location induced by an occurrence of some seed corresponds to an approximate occurrence of p in t.

Filtration specificity in terms of candidate locations to verify is strongly correlated to seed length. Since we want to maximize the length of the shortest seed, we let the minimum seed length be $\lfloor |p|/(k+1) \rfloor$. If we want to improve filtration specificity by increasing seed length, we can resort to approximate seeds.

Figure 1.6: *Filtration with exact seeds.*

Approximate seeds

A more involved filtering algorithm proposed in **?** reduces an approximate search into smaller approximate searches. We partition p into $s \leq k+1$ non-overlapping seeds. According to the pigeonhole principle each approximate occurrence of p in t then contains an approximate occurrence of some seed within distance $\lfloor k/s \rfloor$.

Approximate seeds are searched via backtracking on $\mathcal{T}$. We search $(k \bmod s) + 1$ seeds within distance $\lfloor k/s \rfloor$ and the remaining seeds within distance $\lfloor k/s \rfloor - 1$. To prove full-sensitivity it suffices to see that, if none of the seeds occurs within its assigned distance, the total distance must be at least $s \cdot \lfloor k/s \rfloor + (k \bmod s) + 1 = k+1$. Hence all approximate occurrences of p in t within distance k will be found.

Figure 1.7: *Filtration with approximate seeds.*

q-Gram lemma

1.4 Related problems

1.4.1 Local similarity search

Define score and scoring scheme.

Define local similarity.

Online methods

Give dynamic programming solution.

Indexed methods

Backtracking over substring index. BWT-SW.

Filtering methods

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

1.4.2 Dictionary search

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

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.

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 $||\mathbb{D}||$.

Algorithm 1.5 Exact dictionary search on a radix trie.

Algorithm 1.6 Approximate dictionary search on a radix trie.

Filtering methods

1.4.3 Overlaps computation

Define problem.

Online methods

DP solution.

Indexed methods

Indexed solution, exact and approximate.