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 Fundamental stringology

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 **??**.

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 $x = x_1 x_2 \dots x_n$, and $y \in \Sigma^m$ with $y = y_1 y_2 \dots y_m$, their concatenation $x \cdot y$ (or simply denoted xy) is the string $z \in \Sigma^{n+m}$ consisting of the symbols $x_1 x_2 \dots x_n y_1 y_2 \dots y_m$.

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: $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. 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.

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 $d_H : \Sigma^n \times \Sigma^n \to \mathcal{N}$ 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 $d_E : \Sigma^* \times \Sigma^* \to \mathcal{N}$ counting the minimum number of edit operation necessary to transform x into y.

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

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:

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

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

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

The recursive case is defined as follows:

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

The recurrence relation can be computed in time $\mathcal{O}(|x| \cdot |y|)$ using a table of size $(n + 1) \times (m + 1)$.

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 $\mathcal{O}(n + m)$.

Answering the question whether the distance $d_E(x, y) \leq k$ is an easier problem: a band of size $k + 1$ is sufficient.

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.

1.2 Overview of string matching

1.2.1 Problem definition

We can now define exact string matching, perhaps the most fundamental problem in stringology. Given a string p called the pattern and a longer string t 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 reader is referred to [?] for an extensive treatment of the subject.

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 $k \in \mathcal{N}$, 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.

Existing methods to solve approximate string matching problems can be classified in three categories: online, indexed and filtering. An extensive survey on online methods is provided by [?], while a more succint survey on indexed methods is given in [?]. In the following of this section we give a brief overview of the state of the art in this field, including most recent methods not treated by [?] nor [?].

1.2.2 Online methods

Dynamic Programming

Finding all occurrences of a pattern p in a text t...

Change initialization.

Change traceback to start from all $j : D[m, j] \leq k$.

Automata

1.2.3 Indexed methods

Suffix tree

Backtracking

1.2.4 Filtering methods

Why filtering

Pigeonhole principle

q-Gram lemma

1.3 Related problems

1.3.1 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. Lastz resembles a suffix filter.

1.3.2 Dictionary search

Define problem.

Trie

Backtracking

1.3.3 Overlaps computation

Define problem.

DP solution.

Indexed solution, exact and approximate.

CHAPTER 2

Online Methods

2.1 Myers' bit-vector algorithm

2.2 Banded Myers' bit-vector algorithm

2.3 Increased bit-parallelism using SIMD instructions

CHAPTER 3

Indexed Methods

3.1 Classic Full-Text Indices

3.1.1 Trie

3.1.2 Suffix trie and suffix tree

3.1.3 Suffix array

3.1.4 q-Gram index

3.2 Compressed Full-Text Indices

3.2.1 Burrows-Wheeler transform

3.2.2 FM-index

3.2.3 Rank dictionaries

3.3 Backtracking

3.3.1 Pruning methods

3.3.2 Multiple backtracking

CHAPTER 4

Filtering Methods

4.1 q-Gram filters

4.1.1 Exact seeds

4.1.2 Gapped seeds

4.2 Factor filters

4.2.1 Exact seeds

4.2.2 Approximate seeds

4.3 Suffix filters

CHAPTER 5

Read Mapping

5.1 Related work

5.1.1 Best mappers

5.1.2 All mappers

5.2 The Masai mapper

5.2.1 Single-end mapping

5.2.2 Paired-end mapping

5.3 The Masai 2 mapper

5.3.1 Single-end mapping

5.3.2 Paired-end mapping

5.3.3 Parallelization

5.3.4 Hardware acceleration

5.4 Experimental results

5.4.1 Comparison of filtration strategies

5.4.2 Rabema benchmark results

5.4.3 Variant detection results

5.4.4 Runtime results

5.5 Discussion

CHAPTER

6 String Similarity Search / Join

6.1 The competition

6.2 Our method

6.2.1 Implementation

6.2.2 Parallelization

6.3 Related methods

6.4 Experimental results

6.4.1 Search

6.4.2 Join

6.5 Discussion