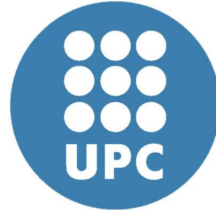

Universitat Politècnica de Catalunya

Statistical models for genome sequence mapping

Master of Science thesis

in partial Fulfillments for the Degree of
Electronic Engineering

at the
Universitat Politècnica de Catalunya

Author:
Eduard Valera i Zorita

Advisors:
Dr. Guillaume Filion
Dr. Josep Vidal Manzano

September 14, 2016

Contents

Resum **11**

Resumen **13**

Abstract **15**

1 DNA and sequence alignment **17**

1.1 Biological information is stored using DNA . . . 17

1.1.1 The molecule . . . 17

1.1.2 Origins of DNA . . . 18

1.1.3 Genes and genomes . . . 20

1.1.4 Evolution of genomes . . . 21

1.1.5 DNA sequencing technologies . . . 22

1.2 Sequence alignment . . . 22

1.2.1 Needleman-Wunsch algorithm . . . 24

2 Finding needles in a haystack **28**

2.1 Sequence mapping . . . 28

2.2 Efficient string search structures . . . 30

2.2.1 Hash tables . . . 30

2.2.2 Trees . . . 31

2.2.3 Suffix arrays . . . 32

2.2.4 BW Transform and FM index . . . 33

2.3 The sequence neighborhood . . . 40

2.3.1 Neighborhood annotation algorithm . . . 40

2.3.2 *k*-mer uniqueness . . . 41

3 Mapping algorithm **44**

3.1 Seeding . . . 45

3.1.1 MEM seeding . . . 45

3.1.2 Threshold seeding . . . 46

3.1.3 Inexact seeding . . . 46

3.2 Filtering significant seeds . . . 46
3.2.1 Filter on seed length . . . 46
3.2.2 Filter on number of hits . . . 47
3.2.3 Filter on seed significance . . . 47
3.3 Aligning . . . 48
3.3.1 Sequence alignment . . . 49
3.3.2 Split reads and breakpoint algorithm . . . 49
3.4 Mapping quality . . . 51
3.4.1 Alignment score model . . . 51
3.4.2 The neighbors model . . . 52

4 Results **56**

5 Conclusions and future work **61**

List of Figures

1.1 DNA molecule. a) DNA nucleotides are formed by a Phosphate group, a 5-carbon sugar and a nitrogenous base. b) Sugars that compose RNA/DNA differ by an OH/H group on the second carbon. c) The nitrogenous bases are Guanine, Adenine, Cytosine, Thymine (DNA only) and Uracil (RNA only). 18

1.2 Structure of the DNA double helix. The nitrogenous bases hold together through hydrogen bonds: A and T with two bonds, C and G with three. The nucleotide pairs form stacks through sugar-phosphate bonds. 19

1.3 Single-stranded RNA molecules can form secondary structures. a) Self-annealing of complementary nucleotides from the same RNA strand. b) Representation of the secondary structure. Source [1]. 20

1.4 Human karyotype. Human cells have 23 pairs of chromosomes. Each cell contains two copies of chromosomes 1 to 22 and two copies of chromosome X (female) or one copy of chromosome X and one copy of chromosome Y (male). Source [1]. . . . 21

1.5 Organization of genes in the human genome. a) Representation of chromosome 22. b) A ten-fold expansion of a fragment of chromosome 22 with about 40 genes indicated in red. c) An expanded portion of (b) showing four genes. d) Representation of one of the genes of chromosome 22 where the regulatory region and its 9 exons are indicated. Source [1]. 22

1.6 A representation of the nucleotide sequence content of the sequenced Human genome. Source [1]. 23

1.7 BLOSUM62 matrix, used to score alignments between evolutionarily divergent protein sequences. 24

1.8 Initialization of a Needleman-Wunsch alignment between `ATGCAA` and `ACGCTTTAA`. . 25

1.9 Needleman-Wunsch matrix for q =`ATGCAA` and t =`ACGCTTTAA` (top row) and alignment path based on backtracking through valid cell transitions (bottom row). Green: match/mismatch. Red: Gap open/extend. **a)** Score matrix for Levenshtein distance ($P_m = 0$, $P_s = 1$, $P_o = 1$, $P_e = 1$). **b)** Score matrix for $P_m = -1$, $P_s = 4$, $P_o = 6$, $P_e = 1$. 25

1.10 Smith-Waterman matrix initialization for q =`ATGCAA` and t =`TGTGCATGGAAAGCAGCT`. 26

1.11 Smith-Waterman alignment for q =`ATGCAA` and t =`TGTGCATGGAAAGCAGCT` using *min-gap* score model. 27

2.1 Representation of error free subsequences modeled as a stick breaking problem. The mismatched nucleotides and the longest error-free stretch are highlighted in red. 29
2.2 Seeding probability for error rate $q = 0.02$ and read lenghts $m = 25, 50, 75, 100, 125, 150$ nt. 31
2.3 Process of construction of a tree structure containing all the suffixes of the text `ATGAC`. The position of the suffixes are stored in the leaf nodes. 32
2.4 Suffix array of `GATGCGAGAGATG`. The numbers on the top represent the positions of the suffixes (the suffix array). Below, the suffixes sorted in lexicographical order. . 33
2.5 The preceding character of the suffixes shown in Figure 2.4. The preceding character of the first suffix is the last character of the text ($). 34
2.6 The correspondence between the preceding character and its order of appearance in the suffix array. Note that the oder of appearance is conserved for each nucleotide set. 35
2.7 Backward search using the preceding character information. The inteval of the matching suffix is shrinked every iteration, when the preceding nucleotide is added. 36
2.8 Backward search using only the C and Occ tables. The suffix intervals are updated with (2.8) reading the query backwards. 37
2.9 Subintervals of the suffix `A`. The subintervals are consecutive in the suffix array. The interval of the suffix `A` is denoted by $\mathtt{I_A}$, the subintervals are $\mathtt{I_{AA}}, \mathtt{I_{AC}}, \mathtt{I_{AG}}, \mathtt{I_{AT}}$ and f_p is the position of the first suffix of the interval. 39
2.10 Representation of the recurive block search algorithm. The first step (top-bottom) divides the intervals in two blocks. When the bottom is reached, the algorithm starts a bottom-up search and extension. 43

3.1 Seed and extend algorithm. Seeding consists in finding short exact matches between the read and the reference. The extension step performs a full alignment between the read and the reference at the position of the seeds. 49
3.2 Seed and extension over a split read. The split read is composed of two distant fragments of the reference (blue and red). At the extension step, the alignment proceeds passed the fragment border, where the sequences do not coincide anymore (represented in gray on the reference). 50
3.3 Local neighborhood of the read T_R, its best match T_B and the closest neighbor of the best match T_N. The Lenvenshtein distance between sequences is denoted by $\mathrm{d}(\cdot, \cdot)$. 52
3.4 On our hypothesis for the mapping score, T_R originated from T_N. This figure summarizes the three possible scenarios when a nucleotide of T_R is modified. . . . 53

4.1 Sensitivity vs error rate for all the possible mapping quality thresholds. Illumina simulated reads on Human Genome v19, with 1% (left), 2% (right) and 5% (bottom) error rate. Comparison between bowtie2 (blue), bwa-mem (red) and our mapper (black). 59

4.2 Throughput (correctly mapped sequences per core per second) vs error rate for all the possible mapping quality thresholds. Illumina simulated reads on Human Genome v19, 1% (left) and 5% (right) error rate. Comparison between bowtie2 (blue), bwa-mem (red) and our mapper (black). 60
4.3 Sensitivity vs error rate for all the possible mapping quality thresholds. Illumina simulated reads on Drosophila Melanogaster genome v3, 2% (left) and 15% (right) error rate. Comparison between bowtie2 (blue), bwa-mem (red) and our mapper (black). 60

List of Tables

1.1 Genome size and protein-coding gene count of model organisms. 20
1.2 DNA sequencing technologies in current use. 23
1.3 Examples of alignment scores. 24
1.4 Alignments generated with different score models for q = `AAATCA` and t = `AAAGAATTCA`. 26

3.1 Upper bound probability of incorrect seed ($l = 19$). 48

Resum

En aquest projecte hi presentem un algoritme de mapping. Els mappers són algoritmes que s'utilitzen per trobar seqüències curtes d'ADN en textos de referència molt grans. El nostre algoritme utilitza la tècnica estàndard de *seed-and-extend*, utilitzada per la majoria de mappers actuals, combinada amb una nova anotació del genoma que hem anomenat *neighborhood annotation*. Aquesta anotació consisteix en una estructura de dades que emmagatzema informació sobre les similaritats entre les seqüències del text de referència. Basant-nos en aquesta estructura, hem dissenyat un model estadístic que utilitzem per assistir els processos de *seeding* i d'estimació de la qualitat de mapping. Finalment, hem implementat i mesurat el rendiment del nostre algoritme en seqüenciacions simulades d'Illumina. Els resultats obtinguts determinen millor sensitivitat i estimacions més acurades de la fiabilitat de mapping, a la mateixa velocitat que els mappers de l'estat de l'art. El codi font de la implementació en C està disponible en open-source al web `http://github.com/ezorita/mapper`.

Resumen

En este proyecto presentamos un algoritmo de mapping. Los mappers son algoritmos utilizados para encontrar secuencias cortas de ADN en textos de referencia mucho más largos. Nuestro algoritmo utiliza la técnica estándar de *seed-and-extend*, utilizada por la mayoria de mappers actuales, combinada con una nueva anotación del genoma: el *neighborhood annotation*. Esta anotación es una estructura de datos que almacena información sobre las similitudes entre las secuencias del texto de referencia. Basandonos en esta estructura, hemos diseñado un modelo estadístico que utilizamos para favorecer los procesos de *seeding* y de estimación de la calidad de mapping. Finalmente, hemos implementado y testeado el rendimiento de nuestro algoritmo en secuencias simuladas de Illumina. Los resultados obtenidos muestran una mejor sensitividad y estimaciones más precisas de la fiabilidad de mapping, a la misma velocidad que los mappers del estado del arte. El código fuente de la implementación en C está disponible en open-source en `http://github.com/ezorita/mapper`.

Abstract

In this work we present a mapper, an algorithm to find short DNA sequences in large reference texts. Our algorithm uses the standard seed-and-extend approach, utilized by most modern mappers, combined with a novel genome annotation called neighborhood annotation. The neighborhood annotation is a data structure that contains information of similarity between sequences of the same reference. Based on this annotation, we build a statistical model to aid the processes of seeding and mapping quality estimation. Overall, our algorithm achieves higher sensitivity and more accurate estimation of mapping reliability with simulated Illumina reads, at the same speed compared to the state-of-the art algorithms. The C source code of the algorithm implementation is available at `http://github.com/ezorita/mapper`.

Chapter 1

DNA and sequence alignment

1.1 Biological information is stored using DNA

1.1.1 The molecule

The deoxyribonucleic acid (DNA) is the molecule that provides all living organisms the ability to store, retrieve and pass from generation to generation the genetic instructions required to make and maintain a living organism. A molecule of DNA consists of two long strands of complementary poly-nucleotide chains. The building blocks which compose a chain of DNA are called nucleotides. The nucleoties are molecules with simple structure composed of a five-carbon sugar, a nitrogen-containing base and one or more phosphate groups (Figure 1.1a).

The nucleotides are covalently linked together building a backbone of sugar-phosphate bonds, in which the phosphate groups that are attached to the 5th carbon of the sugars (5') form a covalent bond with the 3rd carbon (3') of the next nucleotide in the strand (Figure 1.2). Following this fashion, DNA nucleotides can be enchained to form an arbitrarily long DNA strands. Note that only one of the ends of a DNA molecule will have a dangling phosphate group on its backbone (the 5' end), giving an inherent polarity to the strand. The base group is the only subunit that differs in each of the four types of nucleotides. There are four different nitrogen-containing bases that give name to their respecive nucleotide: Adenine (A), Thymine (T), Guanine (G) and Cytosine (C) (Figure 1.1c). The bases can hold together two strands of DNA through hydrogen bonds. However, they do not pair at random: the chemical structure of the bases only permits the efficient formation of hydrogen bonds in the interior of the double helix between A and T with 2 hydrogen bonds, and G and C with 3 hydrogen bonds. The latter pairing of bases is not a strict limitation, nevertheless, in terms of free energy, it's the best combination to bring the atoms close enough to form hydrogen bonds without distorting the structure of the DNA backbone. One last additional requirement to form the double helix molecule is that the bases can only pair if the complementary strands are antiparallel, i.e. the polarities of the strands (5' and 3') are oriented in an opposed manner (see Figure 1.2).

The discovery of the double helix structure of the DNA [2] in 1953 was crucial to answer

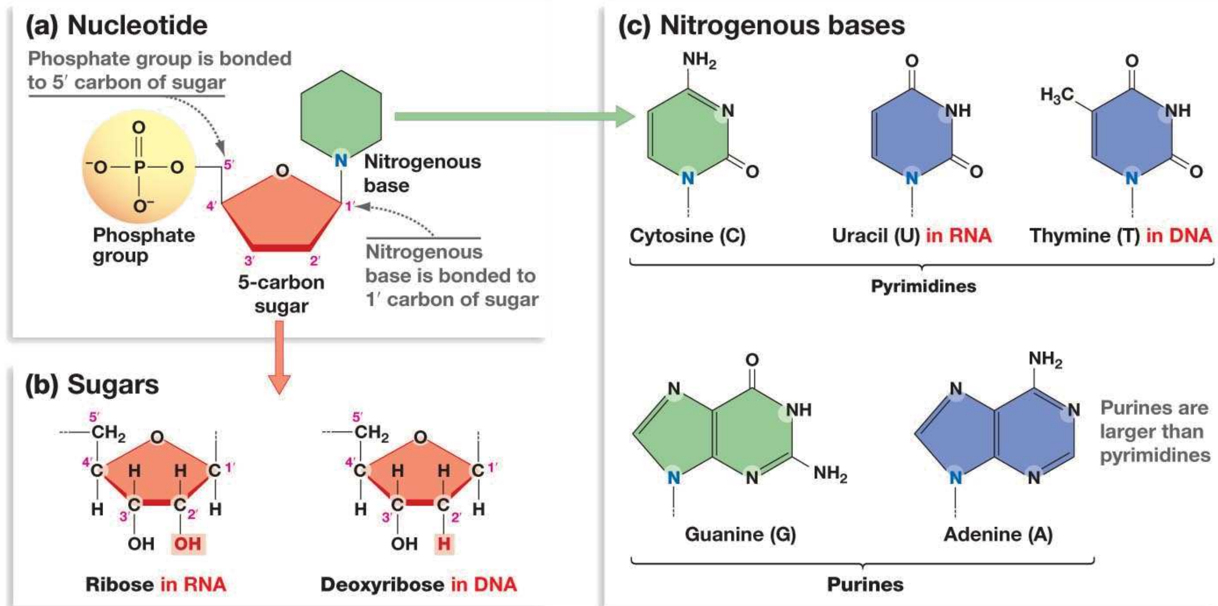

Figure 1.1: DNA molecule. a) DNA nucleotides are formed by a Phosphate group, a 5-carbon sugar and a nitrogenous base. b) Sugars that compose RNA/DNA differ by an OH/H group on the second carbon. c) The nitrogenous bases are Guanine, Adenine, Cytosine, Thymine (DNA only) and Uracil (RNA only).

two important questions: (i) how the information is stored in a chemical form and (ii) how this information is passed from the mother cell to its daughters. The answer to the first question is that the biological information in living organisms is encoded in the sequence of nucleotides along DNA strands. Therefore, the genetic information is analogous to a text with a 4-letter alphabet: A, C, G and T. In consequence, different organisms differ from each other because they have different nucleotide sequences. The discovery of the DNA structure also shed some light to the second question. The fact that the DNA molecule contains two strands bearing identical information rapidly suggested a mechanism of DNA replication: unwinding the two strands and filling them separately with the complementary nucleotide sequence would give rise to two new double stranded DNA molecules with the same information.

1.1.2 Origins of DNA

There are two main types of nucleic acids, which differ in the type of sugar contained in their sugar-phosphate backbone (Figure 1.1b). The acid based on the sugar deoxyribose is known as deoxyribonucleic acid (DNA) and contains the bases A, G, C and T (Sec. 1.1.1). On the other hand, the acid based on sugar ribose, known as ribonucleic acid (RNA), contains the bases A, C, G and Uracil (U), see Figure. 1.1c. Despite their structural similarities, the RNA is commited to many more cellular tasks than its sister nucleic acid. For instance, RNA is usually found in cells in the form of a single-stranded molecule, whereas DNA is almost always in the form of a double

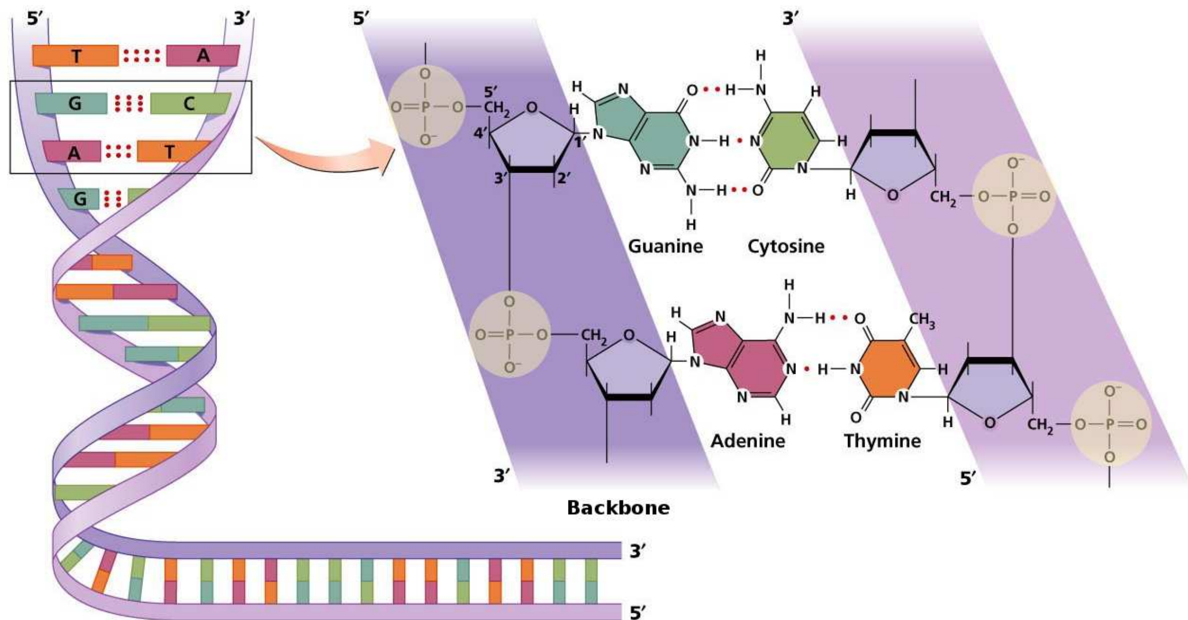

Figure 1.2: Structure of the DNA double helix. The nitrogenous bases hold together through hydrogen bonds: A and T with two bonds, C and G with three. The nucleotide pairs form stacks through sugar-phosphate bonds.

stranded polynucleotide chain. Being single stranded, the RNA backbone is flexible, allowing the polymer chain to bend back on itself and form weak bonds between different parts of the same molecule. In fact, these types of internal associations form specific shapes that are dictated by its sequence (Figure 1.3). The shape of the RNA molecule, in turn, may recognize other molecules and bind to them selectively or act as a catalyst in some chemical reactions. However, besides its enzymatic roles, the RNA can also store and replicate genetic information in the same manner as DNA. Most notably, RNA works in all living organisms as an intermediate in the transfer of genetic information in the form of messenger RNA (mRNA), bearing the information and guiding the synthesis of proteins.

The triple role of RNA (enzymatic, protein synthesis and information-bearing) strongly suggests that early stages of life emerged from a combination of RNA and proteins. Such early RNA-protein world would imply the existence of genetic code prior to DNA, which is consistent with the ubiquitous use of messenger RNA (mRNA) and transfer RNA (tRNA) as protein-building machinery. Given the success of this early form of life, an immediate question arises: why would the transition from RNA to DNA be so complete as to eradicate all RNA-based genomes of living organisms? The straightforward answer is that DNA genomes would have had substantial advantage with respect to RNA genomes in terms of the reliable production of progeny genotypes. In effect, the absence of the $-OH$ group on desoxyribose renders DNA much more structurally stable than RNA. In addition, the use of Thymine in place of Uracil evades one of the most common sources of mutations in RNA: the accidental production of Uracil via the deamination of Cytosine (see the similarities in Figure 1.1c). These advantages for DNA-based organisms may

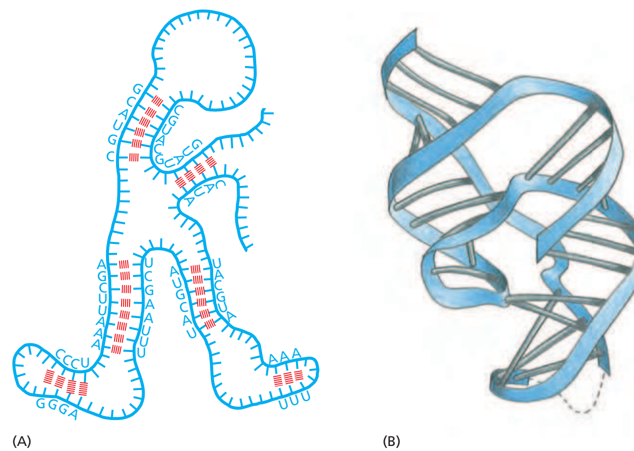

Figure 1.3: Single-stranded RNA molecules can form secondary structures. a) Self-annealing of complementary nucleotides from the same RNA strand. b) Representation of the secondary structure. Source [1].

have facilitated its survival in less permisive environments where they could develop more complex biological functions.

1.1.3 Genes and genomes

Table 1.1: Genome size and protein-coding gene count of model organisms.

Organism	**Genome size** [bp]	**Coding genes**
Escherichia coli (Bacterium)	$4.6 \cdot 10^6$	~4.400
Sacchromyces cerevisiae (Yeast)	$12.5 \cdot 10^6$	~6.000
Neurospora crassa (Fungus)	$39.9 \cdot 10^6$	~10.000
C. Elegans (Nematode)	$100.2 \cdot 10^6$	~20.000
Arabidopsis Thaliana (Plant)	$135 \cdot 10^6$	~27.000
Drosophila Melanogaster (Fly)	$180 \cdot 10^6$	~13.000
Danio Rerio (Zebra Fish)	$1.7 \cdot 10^9$	~26.000
Mus musculus (Mouse)	$2.8 \cdot 10^9$	~23.000
Homo Sapiens (Human)	$3.3 \cdot 10^9$	~20.000

In eukaryotic cells, the large amounts of DNA required to encode all the information needed to sustain cellular life, are packaged into **chromosomes**. Chromosomes are very long double-stranded DNA molecules found in the cell nucleous (Figure 1.4). These DNA molecules fit readily inside the nucleous and, after they are replicated, they can be easily distributed between the two daughter cells. On the other side, prokaryotic cells typically carry their genes on a single, circular DNA molecule called *bacterial chromosome*, found in the cytoplasm.

The DNA in human species is packed into 24 different chromosomes, each consisting of a fine

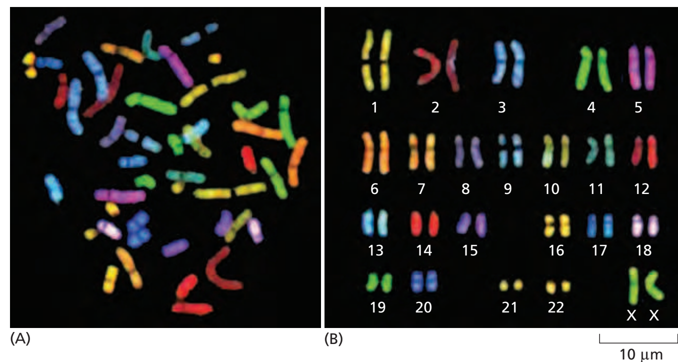

Figure 1.4: Human karyotype. Human cells have 23 pairs of chromosomes. Each cell contains two copies of chromosomes 1 to 22 and two copies of chromosome X (female) or one copy of chromosome X and one copy of chromosome Y (male). Source [1].

thread of DNA and a set of proteins that fold and pack it into a compact structure. Such complex of protein and DNA is called **chromatin**. The main function of the chromosomes is to carry the genes (Figure 1.5). A gene is a segment of DNA that is transcribed, i.e. that is converted to RNA either to be used as is (structural, catalytic, regulatory RNA...) or to guide the synthesis of a protein. The total genetic information carried by all the chromosomes of an organism constitutes its **genome**. As it may be expected, there exists a correlation between the genome size and the complexity of the species, Table 1.1 summarizes the genome sizes of the most widely used model organisms.

1.1.4 Evolution of genomes

The evolution of the current living species from the ancestral forms of life has been an impressive journey. The living organisms had been continously adapting to get through the selective pressure of the environment, a process that wouldn't have been possible without the capacity of the genomes to change and *generate* new genes. Remarkably, the genes that provide an organism with selective advantage do not arise as a whole; they are instead the product of thousands or even millions of years of collection of non-deleterious single nucleotide changes, combined with the *horizontal transfer* of bigger sequence blocks.

Single nucleotide changes or *mutations* may be produced by the exposure to radiation, double-strand breaks on DNA, replication errors, etc. However, the evolution of genomes due to single mutations is an extremely slow process, considering that error rates during DNA synthesis in humans are in the 10^{-6} to 10^{-8} range. Besides, only an insignificant part of such mutations will remain, given that deleterious mutations are rapidly displaced by selective pressure. On the other side, more sophisticated mechanisms, which include: sequence duplication and recombination, virus infections or plasmid transformation, among others, allow the transfer of bigger blocks of DNA, a process called horizontal gene transfer (HGT). As a result of HGT, the genomes of the more

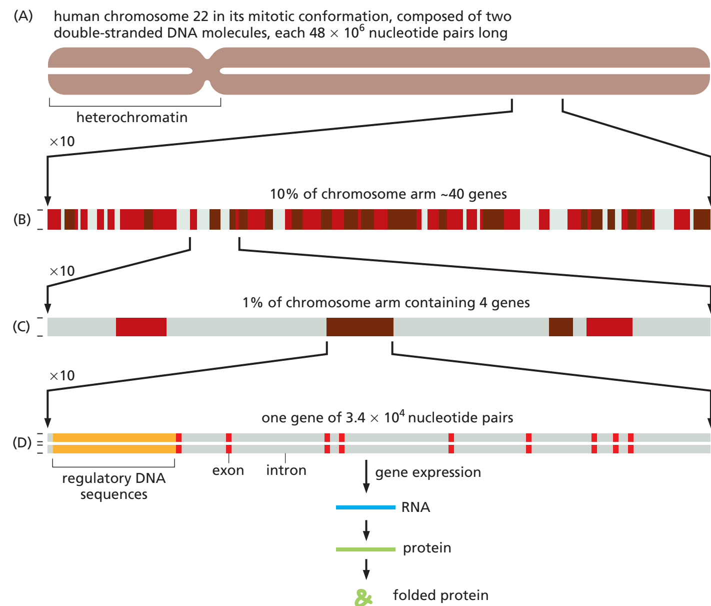

Figure 1.5: Organization of genes in the human genome. a) Representation of chromosome 22. b) A ten-fold expansion of a fragment of chromosome 22 with about 40 genes indicated in red. c) An expanded portion of (b) showing four genes. d) Representation of one of the genes of chromosome 22 where the regulatory region and its 9 exons are indicated. Source [1].

complex species are full of duplicated sequences that may belong, for instance, to a functional substructure present in many genes, a massive infection from ancestral virus or an accidental duplication. Such repeated sequences are so significant that fill up to 50% of the Human genome, whereas the protein coding sequences only represent the 2% (Figure 1.6). We will see in the following chapters that such repetitive structure of the genomes is the major source of difficulties in the field of sequence mapping.

1.1.5 DNA sequencing technologies

DNA sequencing is a method or technology that is used to measure the precise order of the four bases (A,C,G,T) in a strand of DNA. The first sequencings methods: *Chemical* and *Sanger* sequencing were invented in 1977. These methods were very slow and expensive, requiring enormous human power, but were able to generate reads of several hundreds of nucleotides [3]. Later in the early 2000's, *Next Generation sequencing technologies* were developed, currently yielding bilions of reads in less than 24 hours. A summary of the current sequencing technologies along with their properties is presented in Table 1.2.

1.2 Sequence alignment

Sequence alignment is a process in which sequences of DNA, RNA or protein are arranged to identify the regions of highest similarity based on a score model. In other words, alignments are

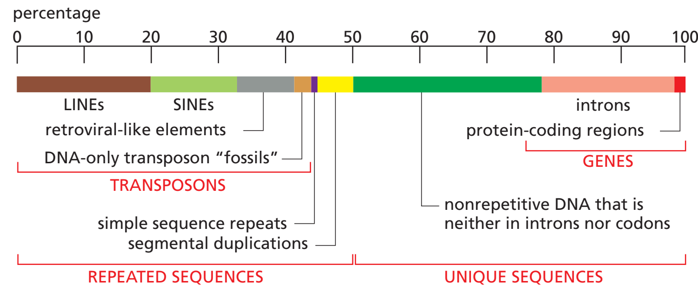

Figure 1.6: A representation of the nucleotide sequence content of the sequenced Human genome. Source [1].

Table 1.2: DNA sequencing technologies in current use.

Name	**Read length** [bp]	**Reads**	**Time**	**Error rate**	**Cost/Mb**
Illumina	150+150bp	>100M	24hrs	~1%	0.1$
PacBIO	up to 30kbp	~100K	1-4hrs	~15%	0.5$
Pyrosequencing	700bp	1M	24hrs	<1%	10$
SOLiD	50+50bp	1G	1 week	<0.5%	0.1$
Sanger	900bp	<100	hours	<0.1%	2000$

measurements of similarities between sequences. Alignments can be *local* or *global*, depending on whether subsequences of the query sequence are expected to be found split in distinct regions along the reference (local alignment) or the query sequence is expected to be found as a whole (global alignment). An example of a *local* alignment would be to align the mRNA sequence of a gene that contains introns against its reference genome, whereas *global* alignments would be used to align sequences of DNA that have been extracted from the genome itself. On the other side, *sequence mapping* is the process of finding the most probable original locus of a short sequence in a much larger reference, e.g. its reference genome. In the following sections we describe and discuss the complexity and purpose of some basic algorithms used for sequence alignment. Sequence mapping will be descripted in the following chapter.

Even though there are a myriad of sequence alignment algorithms in the literature, the vast majority are based on the same approach: *dynamic programming* methods. The alignment between two sequences is computed by applying a recurrence relation throughout a matrix of $m \cdot n$ terms, called *edit matrix* or *alignment matrix*, where m and n are the respective sequence lengths. Such relation is based on a substitution matrix, the *score matrix*, that establishes a relationship between the possible sequence modifications and their respective penalties. The technique of dynamic programming can be applied to produce *global alignments* via the *Needleman-Wunsch* algorithm, and *local alignments* via the *Smith-Waterman* algorithm, described below.

1.2.1 Needleman-Wunsch algorithm

The Needleman-Wunsch algorithm [4] is used to compute the *global distance* between two sequences. Let q be a query sequence and t be the reference sequence, with lengths m and n, respectively. Hence, the alignment matrix S is an integer matrix with dimensions $m \times n$, i.e. $S \in \mathbb{Z}^{m \times n}$. We will refer to the i-th character of either sequence with a subindex, i.e. as t_i and q_i. Let us also introduce a simple score matrix, which defines the update values for character match, mismatch, gap open and gap extend (Table 1.3).

Table 1.3: Examples of alignment scores.

	Levenshtein	**min-gap**
P_m (match)	0	-1
P_s (mismatch)	1	4
P_o (gap open)	1	6
P_e (gap extend)	1	1

Other, more sophisticated matrices, take into account the probability of nucleotide transition (or aminoacid, in case of protein) drawn from empirical models of sequence evolution (Figure 1.7).

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	
C	9																				C
S	-1	4																			S
T	-1	1	5																		T
P	-3	-1	-1	7																	P
A	0	1	0	-1	4																A
G	-3	0	-2	-2	0	6															G
N	-3	1	0	-2	-2	0	6														N
D	-3	0	-1	-1	-2	-1	1	6													D
E	-4	0	-1	-1	-1	-2	0	2	5												E
Q	-3	0	-1	-1	-1	-2	0	0	2	5											Q
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8										H
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5									R
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5								K
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5							M
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4						I
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4					L
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4				V
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6			F
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7		Y
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11	W
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	

Figure 1.7: BLOSUM62 matrix, used to score alignments between evolutionarily divergent protein sequences.

Once the score matrix is defined, the Needleman-Wunsch algorithm is initialized as follows: each character of the query sequence q_i is associated with the $i+1$-th row, similarly the reference characters t_i are placed one per column, following the same trend. The first row and column are not associated to any character because they represent the first run of gaps. Hence, the first row and column are initialized by recursively applying a gap extension penalty P_e, as shown in Figure 1.8.

Then, the matrix cells are computed row-wise or column-wise starting from the top left corner with the following update rules:

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

Figure 1.8: Initialization of a Needleman-Wunsch alignment between `ATGCAA` and `ACGCTTTAA`.

$$S[i,j] = \begin{cases} S[i-1,j-1] + P_m, & \text{if } q_i = t_j. \\ \min(S[i-1,j-1] + P_m, S[i-1,j] + P_e, S[i,j-1] + P_e), & \text{if } q_i \neq t_j \text{ and gap is open.} \\ \min(S[i-1,j-1] + P_m, S[i-1,j] + P_o, S[i,j-1] + P_o), & \text{otherwise.} \end{cases} \tag{1.1}$$

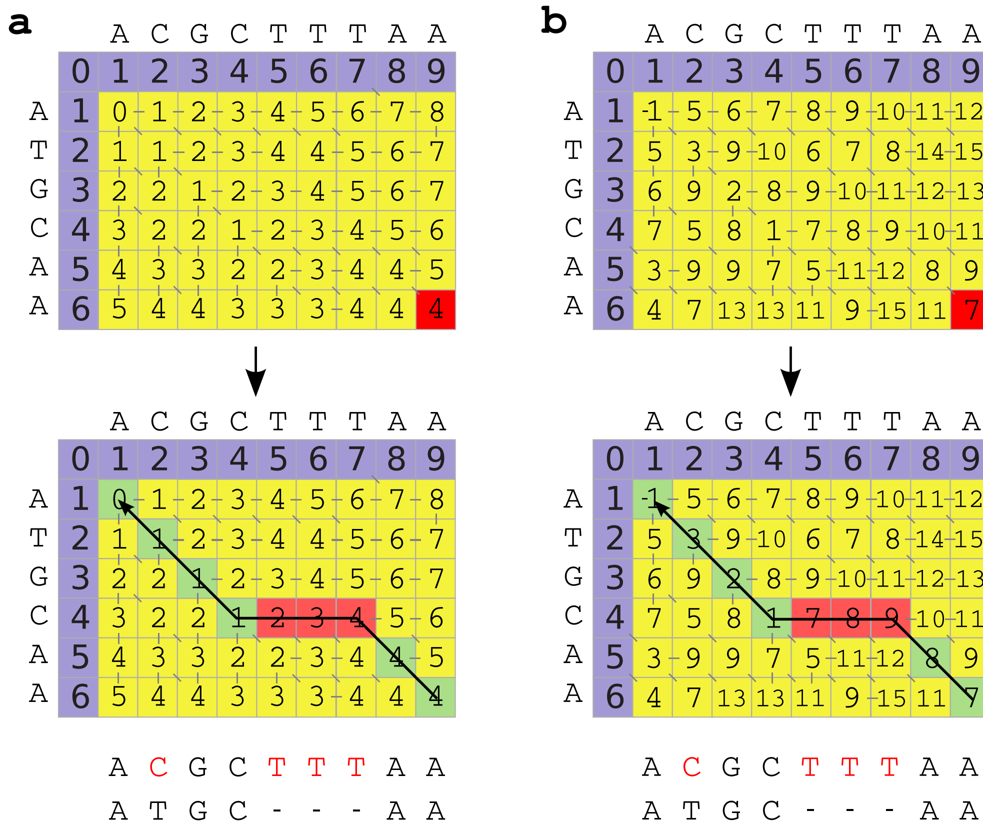

Figure 1.9: Needleman-Wunsch matrix for $q =$`ATGCAA` and $t =$`ACGCTTTAA` (top row) and alignment path based on backtracking through valid cell transitions (bottom row). Green: match/mismatch. Red: Gap open/extend. **a)** Score matrix for Levenshtein distance ($P_m = 0$, $P_s = 1$, $P_o = 1$, $P_e = 1$). **b)** Score matrix for $P_m = -1$, $P_s = 4$, $P_o = 6$, $P_e = 1$.

Applying (1.1) for $q =$`ATGCAAA` and $t =$`ACGCTTTAAA`, one obtains the matrix shown in Figure 1.9. The final score of the alignment corresponds to the value of the bottom-right cell (highlighted in red). The small lines connecting adjacent cells represent the transitions that satisfy the update rules.

The final alignment is computed following the transition trace from the last cell back to the upper-left corner. Any path that connects these two cells through valid transitions represents a valid best alignment (Figure 1.9, second row). Note that different score models may produce different alignments. For instance, the alignment of `AAATCA` against `AAAGAATTCA` generates the best alignments represented in Table 1.4. In this example, the *min-gap* score would be preferred because it produces only one alignment in which all the gaps appear in succession.

Table 1.4: Alignments generated with different score models for $q =$ `AAATCA` and $t =$ `AAAGAATTCA`.

Levenshtein score	**min-gap score**
AAAGAATTCA	AAAGAATTCA
--A-AA-TCA	AAA----TCA
-A--AA-TCA	
A---AA-TCA	
AA--A--TCA	
AA--A-T-CA	
AAA---T-CA	
...	

Smith-Waterman

The Smith-Waterman algorithm [5] is a simple variant of the Needleman-Wunsch to compute local alignments. Local alignment is preferred when the query sequence is much smaller, partially present or split in the reference, e.g. the alignment of a mRNA sequence against its full gene sequence. The exons will be aligned *locally*, each producing an independent score that will not be affected by the low score of the intronic region gaps. On the other side, a global alignment will report a low score given the huge amount of gaps that would need to be introduced to fill the introns.

		T	G	T	G	C	A	T	G	G	A	A	A	G	C	A	G	C	T
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0																		
T	0																		
G	0																		
C	0																		
A	0																		
A	0																		

Figure 1.10: Smith-Waterman matrix initialization for $q =$`ATGCAA` and $t =$`TGTGCATGGAAAGCAGCT`.

Smith-Waterman and Needleman-Wunsch are very similar methods, both use the same matrix and the same update rules, however there are two differences between them. First, the initial gap runs are initialized with all 0 (Figure 1.10). Second, since this method computes local alignments, the final score for a given alignment may not coincide with the lower-right corner. Hence, one should define a criterion to select good alignments. Usually, the criterion is to select scores greater than some threshold with a minimum alignment path length, e.g. match score greater than 30 (matrix score -30 with *min-gap* score model) and alignments of, at least, 30 nucleotides. An example of a local alignment using Smith-Waterman along with the best 3 local alignments is shown in Figure 1.11.

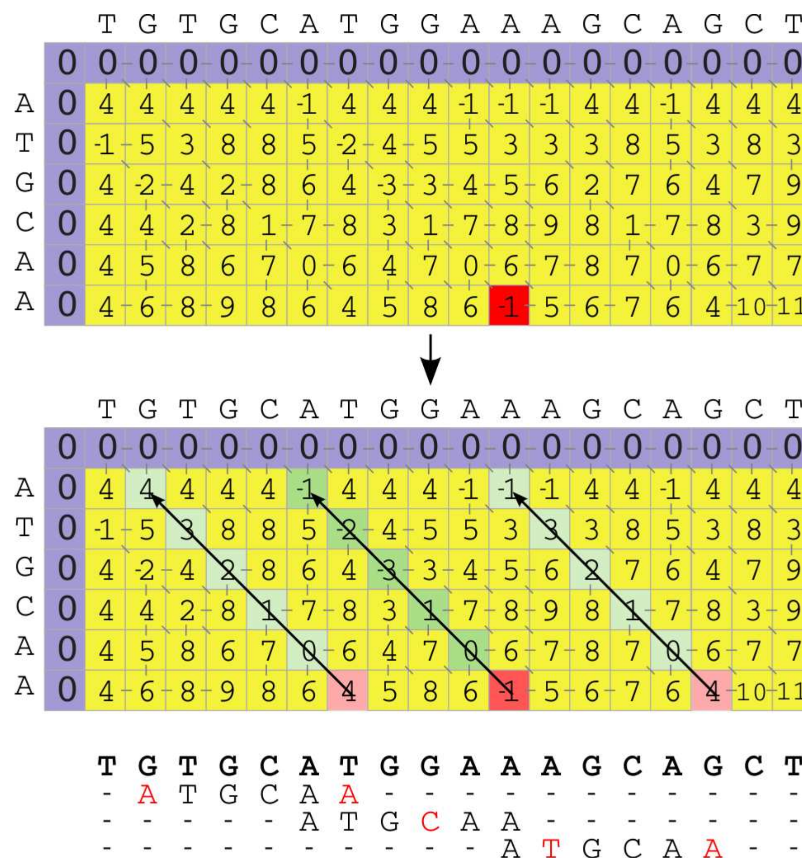

Figure 1.11: Smith-Waterman alignment for q =`ATGCAA` and t =`TGTGCATGGAAAGCAGCT` using *min-gap* score model.

Chapter 2

Finding needles in a haystack

2.1 Sequence mapping

Mapping: formal definition

Let t and q be sequences of DNA (e.g. a genome and a sequencing read) of lengths m and n, respectively. The alphabet of t and q will be denoted as Σ and contains the four DNA nucleotides plus the unknown base N, $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$. Let a_i denote the 1-based i-th nucleotide of an arbitrary sequence a, and $a_{i,j}$ its substring of $j - i$ nucleotides starting at position i, e.g. if $a = \mathtt{ACGGCAGTAT}$, $a_2 = \mathtt{C}$ and $a_{5,9} = \mathtt{CAGT}$. We then define the mapping process as:

$$l = \min_j (S_{qt}[n, j]) \tag{2.1}$$

where S_{qt} is the Smith-Waterman alignment matrix of the query sequence q against the reference t. In other words, mapping is the process of finding the position(s) l on the reference for which the local alignment between q and $t_{l-n,l}$ yields the best score.

Aligning the fast way

In Chapter 1 we reviewed the basic alignment algorithms. These algorithms solve elegantly the alignment problem and allow us to find similarities between sequences, like conserved regions of a protein in diverging species. In this section, however, we will present methods with the aim of finding *where* a short sequence aligns well in a much larger reference text (eq. (2.1)). Speaking in numbers, the typical length of a high-throughput seqeuencing read is on the order of hundreds of nucleotides (Table 1.2), whereas the size of the reference (human genome) is over 10^9 nucleotides (Table 1.1). In such context, one could use the Smith-Waterman algorithm to find the matrix positions l that satisfy equation (2.1) with complexity $O(nm)$. Running this process in a modern computer takes around ten seconds. This is a reasonable performance if one needs to map several reads, but recall that an Illumina MiSeq run yields more than 200 milion reads in less than 24 hours (Table 1.2). Hence, aligning a full lane with this method would take longer than 60 years.

To deal with this bottleneck, it is possible to query the genome at a much faster rate using an *index*. An index is a data structure generated from the reference sequence that allows very fast sequence lookup operations, e.g. the faster indexes require only $O(1)$ or $O(n)$ (the length of the query) operations per lookup. This means that, with only a few operations, one may obtain the complete list of positions of the reference where the query sequence is present. However, most indices only allow lookups of exact sequences, whereas high-throughput sequencing technologies have error rates ranging between 1% and 15% (Table 1.2). Therefore, most of the time we will not find a complete read in the index because it is unlikely that a sequence of hundreds of nucleotides does not contain any mismatch. More concretely, the probability of finding a perfect read of 100 nt, considering an error rate of 1% and binomial distribution of the errors is:

$$\text{pr}(\text{no error}|n = 100 \text{ nt}, P_e = 0.01) = (1 - P_e)^n = 0.37 \tag{2.2}$$

In effect, looking for exact matches of 100 nt would work only in 37% of the reads. Hence, instead of querying the index with the aim of finding long exact matches, we use a better method called *seeding*. This method consists in looking up short subsequences of the query q to increase the probability of finding a perfect match, because short sequences are less likely to contain mutations. However, one has to be conservative and avoid extremely short sequences because these may appear frequently in the reference and yield too many candidate loci. In general, assuming a random reference of length m, the lookup of a sequence of length l yields $m/4^l$ matching loci. As the lookup length increases, the number of reported loci reduces exponentially.

Seeding methods allow us to quickly find tentative positions in the reference where the query sequence may align well. The seeding step is then followed by local alignments of the whole query sequence at all the candidate loci (see seed-and-extend in Chapter 3). Since the seed length is adjusted to yield only a few matches in the genome, the positions of the best alignments are found much faster compared to a Smith-Waterman alignment over the whole reference.

Figure 2.1: Representation of error free subsequences modeled as a stick breaking problem. The mismatched nucleotides and the longest error-free stretch are highlighted in red.

Although it is fast, the seeding method is an heuristic. This means that there is a certain probability of missing the best aligning position l. For instance, imagine that we split the sequence in Figure 2.1 in two halves (seeds) of 30 nucleotides. Since the sequence contains two mutations (highlighted in red), each seed would contain one mutation and neither seed would be found exactly in the reference. Or even worse, despite being mutated, the seed sequence may still exist in the reference and report an incorrect locus, different than the real position where the query sequence was taken from.

We can model the probability of successful seeding, which is the probability of choosing a

subsequence of q without any mutation. This probability depends on the global error rate[1], the length of the query sequence and the error distribution, which will be assumed binomial. Consider a sequencing read q extracted from a reference t and modified with respect to the original reference at k different positions. Choosing the position of the k differences at random would split the read in $k+1$ subsequences without errors. This process is analogous to inserting k breaks at random in a stick of length n. Figure 2.1 illustrates this concept with a read of length $n = 60$ and $k = 2$, the longest error-free stretch is highlighted in red. For a fixed binomial distribution of errors with $q = 1 - p$, the cummulative distribution of the longest error-free stretch X_m in a read of length m is:

$$P(X_m \leq x) = \sum_{k=0}^{m} \binom{m}{k} q^k p^{m-k} P\left(Z_k \leq \frac{x}{m-k}\right) \tag{2.3}$$

where

$$P(Z_k \leq x) = \sum_{j=0}^{k+1} \binom{k+1}{j} (-1)^j (1 - jx)_+^k \tag{2.4}$$

is the cummulative distribution of the longest fragment when k breaks are inserted ($a_+ = a$ if $a > 0$ and 0 otherwise). The cummulative distribution of the seeding probability (2.3) is shown in Figure 2.2 for $q = 0.02$ and several read lengths.

A good tradeoff for the seed length on Illumina reads from Homo Sapiens is between 17 and 20 nucleotides. With these seed lengths, the expected number of perfect matches in a random reference is $3 \cdot 10^9/4^{17} < 1$, with a seeding probability over 95% (Figure 2.2) even with reads as short as 50 nucleotides.

2.2 Efficient string search structures

In the following section we will review the most common data structures (indices) used to efficiently search on long reference texts. The goal of an index structure is to provide very fast lookup operations on a reference text with complexities of $O(1)$ or $O(n)$ and memory footprint $O(m)$ (comparable to the raw genome size).

2.2.1 Hash tables

The first and simplest indexing technique is based in hash tables [6, 7, 8]. The idea is simple, build a hash table with all the substrings in t of fixed length l:

$$H\{t_{i,i+l}\} = [H\{t_{i,i+l}\}, i] \qquad \text{for } x = 1 \ldots m - l \tag{2.5}$$

where $H\{\cdot\}$ is the hash table and $\cdot$ is the table key. Note that the seed length is fixed in this index and this restricts the search to sequences of length l. Therefore, each hash table is built

[1] This accounts for the technology error rate, the divergence between the sequenced organism and the reference genome and the rate of single nucleotide polymorphisms (SNP).

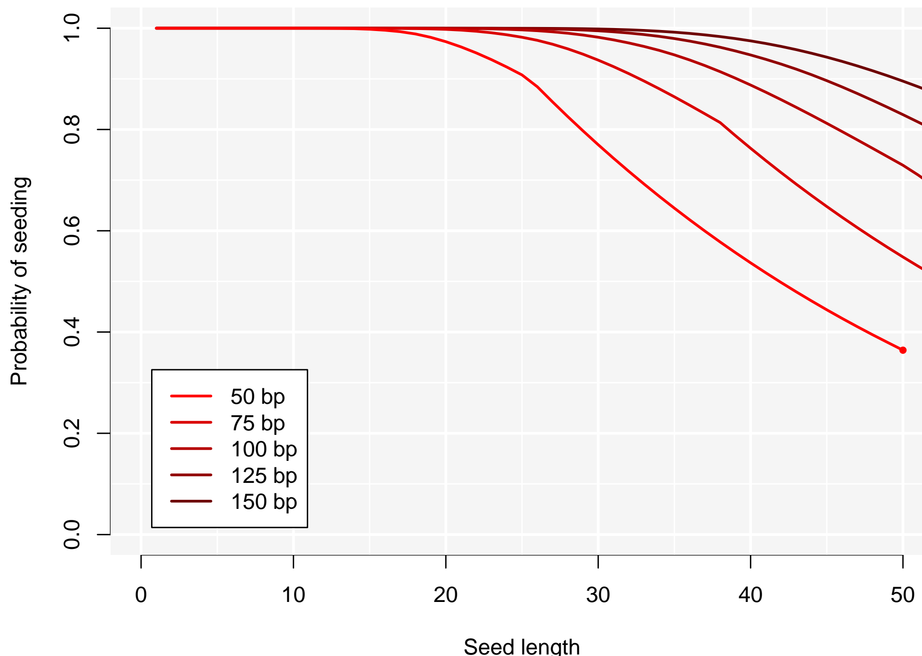

Figure 2.2: Seeding probability for error rate $q = 0.02$ and read lenghts $m = 25, 50, 75, 100, 125, 150$ nt.

for a concrete error rate and sequence length. Note also that the arbitrary substring $t_{i,i+l}$ may be repeated in different positions of the reference, therefore we use the array append operation $a = [a, \cdot]$ to keep a complete list of the positions matching $t_{i,i+l}$. Storing all the l-mers in a hash table achieves the fastest lookup time $O(1)$, with a good memory footprint of $O(m)$. However, the fact that l is determined at the building time and cannot be modified is a major drawback, especially when the error rate is variable or unknown. Nonetheless, querying seeds of length $l' < l$ is possible by appending all the combinations of sequence termination, yielding a lookup cost of $O(4^{l-l'})$.

2.2.2 Trees

Another possible index structure is the tree of suffixes. A tree is nothing but the tree-structure arrangement [9, Section 2.3] of all the suffixes of the reference text. If t is a reference text of length m, then the i-th suffix is the end of t starting from position i, i.e. $t_{i,m+1}$, and will be denoted as $t_{i,\cdot}$.

A tree structure is built using nodes that contain links to other nodes, called *branches*. Each node has as many branches as the cardinality of the alphabet $\|\Sigma\|$. The suffixes are added to the tree starting from the *root* node. The different *leaves* of the tree represent all the suffixes of the

text. The tree is built following the procedure illustrated in figure 2.3: to insert the suffix $t_{i,}$, the root node is selected, if the branch t_i does not exist, a new node is added to this branch. Then, the same process is recursively applied starting at the new node and following t_{i+1}. The position of the suffix i is stored at the last node of the branch (the *leaf*).

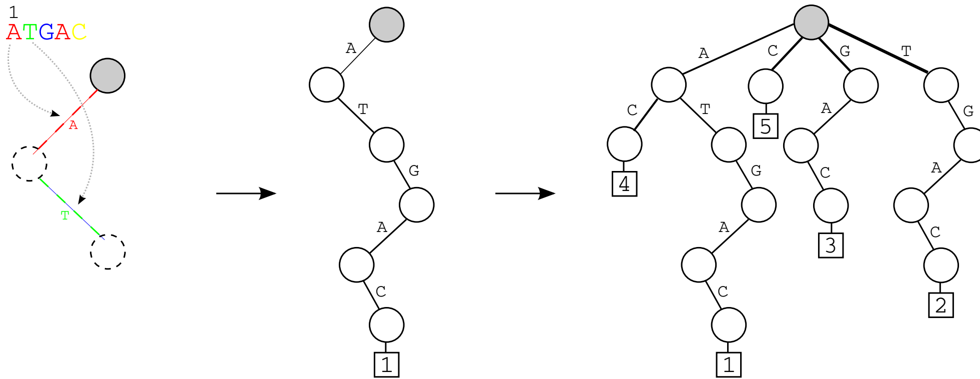

Figure 2.3: Process of construction of a tree structure containing all the suffixes of the text `ATGAC`. The position of the suffixes are stored in the leaf nodes.

To retrieve the occurrences of a seed in the index, the branches corresponding to the nucleotides of the seed are followed. The sequence is not present in the reference if, during this process, a branch is not found. Otherwise, the seed exists in the reference but the search may terminate in an intermediate node. When this happens, all the branches downstream must be followed to retreive the positions of the seed in the text. This index presents an expensive memory footprint ($O(m^2)$) if the suffixes are stored at full. Besides, the exploration until the leaf becomes very demanding, with a worst case of $O(m)$. To solve these problems and make this structure efficient, the suffixes are pruned at a predefined length l.

The tree structure has a worse overall performance and footprint compared to the hash table. Assuming that the tree is pruned at length l, each query has a cost $O(l)$ and the memory footprint is asymptotically $O(lm)$. However, when querying subseeds of length $l' < l$, the cost is $O(4^{l-l'})$ only in the worst case, because only the existing seed terminations (branches) will be followed. A much more efficient implementation of the tree-like index is the *suffix tree* [10]. Suffix trees store more than one letter per branch and reuse repeated parts of the text to boost the compression, achieving $O(m)$ space and $O(m)$ access time.

2.2.3 Suffix arrays

The suffix array is a powerful data structure used to index and compress big texts [11]. It is an array containing the start positions of the suffixes after sorting them in lexicographical order. In other words, the suffix array of a text t of length m is:

$$\mathrm{SA}_t = [i_1, i_2, \ldots, i_m] \qquad \text{where} \qquad t_{i_j,} < t_{i_k,} \text{ iif } j < k. \tag{2.6}$$

in equation (2.6), the symbol $<$ denotes the lexicographical order comparison between two strings. Before computing the suffix array, a unique, lexicographically smallest symbol (usually \$), must be appended at the end of the text ($t_m = \$$).

An example of the suffix array of `GATGCGAGAGATG` is shown in Figure 2.4. Note that $t =$`GATGCGAGAGATG$` has the \$ symbol at the end of the text, which increases the text length by one ($m = 14$). The list of numbers at the top of Figure 2.4 is the suffix array of t, each with its corresponding suffix shown below. Noteworthy, the suffix array can be computed by traversing a *suffix tree* of t from

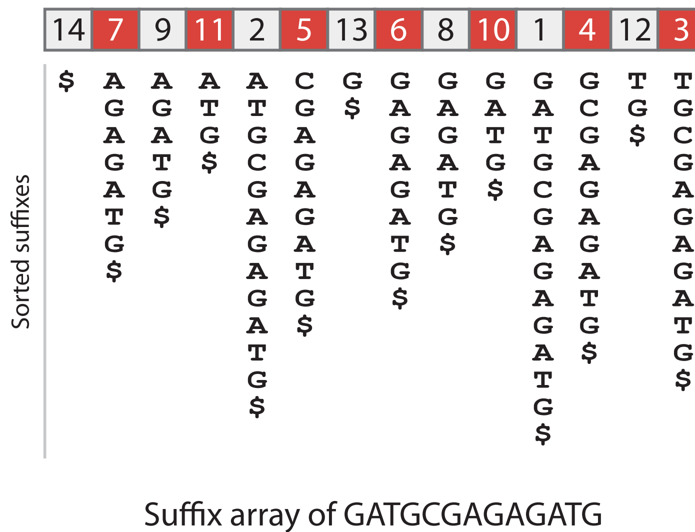

Figure 2.4: Suffix array of `GATGCGAGAGATG`. The numbers on the top represent the positions of the suffixes (the suffix array). Below, the suffixes sorted in lexicographical order.

left (branch `AAAA...A`) to right (branch `TTTT...T`). The positions stored at each leaf coincide with the indices stored at the suffix array.

We can use the suffix array to query exact sequences from the text. Since the suffixes are sorted, one possible method is to use the bisection algorithm [**?**] on the suffix array, which takes $O(\log_2 m)$ for a query q of **any** length. The memory footprint is compact, taking only $O(m)$ space.

2.2.4 BW Transform and FM index

Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is a lossless text transform based on text permutation [12]. The transform of a text t of length m will be denoted as $\mathrm{BW}_t = \mathrm{BWT}(t)$. The BWT is performed in two steps: (i) Compute the suffix array of t (SA_t). (ii) The BWT of t is the list of preceding characters of each position of the suffix array. Formally,

$$\mathrm{BW}_t[i] = t[\mathrm{SA}_t[i] - 1] \tag{2.7}$$

for $\mathrm{SA}_t[i] > 1$ and $\mathrm{BW}_t[i] = t[m]$ if $\mathrm{SA}_t[i] = 1$.

Following the previous example shown in Figure 2.4, we construct the Burrows-Wheeler transform by taking the preceding character of each suffix (Figure 2.5). A fundamental property of the Burrows-Wheeler transform is:

Property 1. *Each nucleotide in the BWT appears in the same relative order as in the Suffix Array.*

That means that the first `A` in the BWT will be also the first `A` appearing in the SA, the fourth `G` in the BWT is the same as the fourth `G` in the SA, and so on.

Figure 2.5: The preceding character of the suffixes shown in Figure 2.4. The preceding character of the first suffix is the last character of the text ($).

To prove Property 1, consider the first character of the BWT shown in Figure 2.5, $\mathrm{BW}_t[1] = \texttt{G}$. Note that the full suffix starting at this position is the concatenation of $\mathrm{BW}_t[1]$ with the suffix $t_{\mathrm{SA}_t[1],}$. Since $t_{\mathrm{SA}_t[i],} > t_{\mathrm{SA}_t[j],}$ for any $j > i$, $\mathrm{BW}_t[1] = \texttt{G}$ will be also the first `G` to appear in the suffix array. The property is illustrated in Figure 2.6.

The Burrows Wheeler transform has other interesting properties, like the rearrangement of the text into runs of similar characters. This allows for effective text compression using simple algorithms like run-length encoding.

The Ferrangina-Manzini Index and Backward Search

The Burrows-Wheeler transform is an interesting structure but it cannot be used as is for the purpose of sequence mapping. The Ferrangina-Manzini (FM) index is a combination of the BWT with other structures which allows to query any sequence of length n in $O(n)$ time, regardless of the text size [13].

Let us illustrate how the Burrows-Wheeler transformed text can be used to look for an arbitrary sequence q in the text, for instance, $q = \texttt{GAGA}$ (Figure 2.7). All the occurrences of `GAGA` in the text

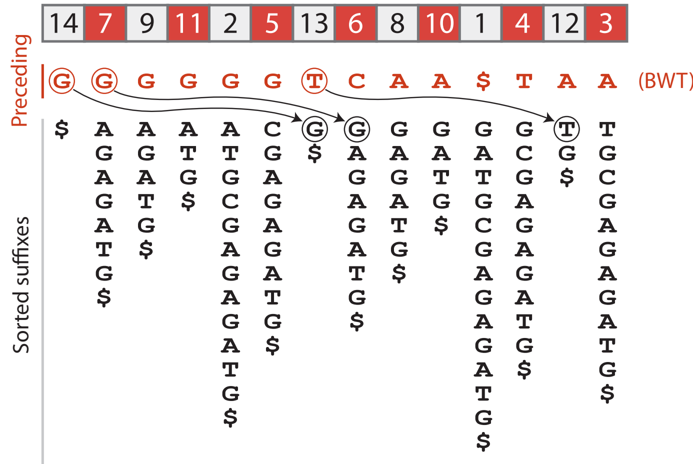

Figure 2.6: The correspondence between the preceding character and its order of appearance in the suffix array. Note that the oder of appearance is conserved for each nucleotide set.

end with A. Each A is also the first letter of some suffix of the text. Because such suffixes all start with A, they are stored next to each other in the suffix array, namely between positions 2 and 5 (Figure 2.7). However, only those suffixes preceded by a G can potentially contain the query. This is exactly the information encoded in the Burrows-Wheeler transformed text BW_t: the preceding character. In this concrete example, all the As are preceded by a G, so the text contains 4 suffixes that start with GA.

So far we know there are 4 suffixes in the text that start with GA, but how do we continue extending the query? Now we need to search for all the A preceding the suffix GA. Indeed, we just learned that the set of G from the suffix GA are the 2nd, the 3rd, the 4th and 5th on the BWT. Therefore, they also will be the 2nd, 3rd, 4th and 5th G in the Suffix Array (Property 1). So now we can continue from the range in the SA covering from the 2nd until the 5th G, and check for the preceding A in the BWT. Now we find two A that correspond to the 1st and 2nd A in the Suffix array. We would continue in the same fashion until the completion of the query q. This process is called **backward search** because the query starts with the last nucleotide and iteratively queries the preceding nucleotides.

The BWT is necessary but not sufficient to perform the backward search on an index. We need additional information to: (i) get the positions in the suffix array where a suffix is present and (ii) the order of occurrence of each nucleotide in the BWT.

Since the Suffix Array is a representation of the sorted suffixes, all the occurrences of the same letter will appear together, i.e. first all the A, then all the C, and so on (Figure 2.4). Therefore, to know the position of the first appearance of each letter, we will directly store them in an array called C. It is straightforward to retrieve the full range of any nucleotide from C, for instance the range of A is $C[\texttt{A}]$ to $C[\texttt{C}]-1$. In general, the full range of the i-th letter of the alphabet Σ is $[C[\Sigma[i], C[\Sigma[i+1]-1]]$.

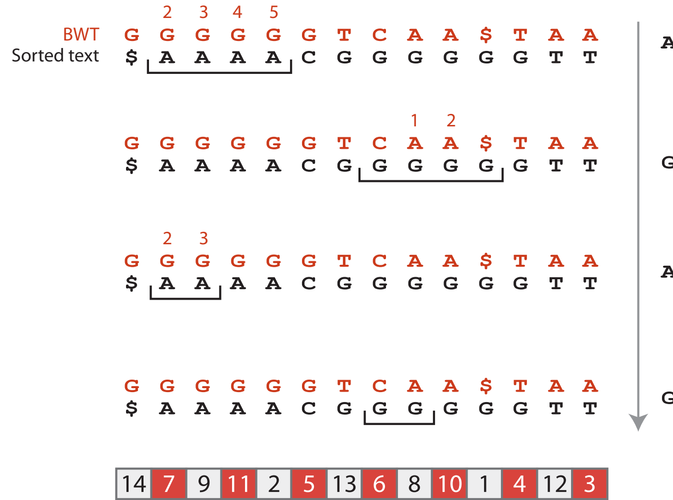

Figure 2.7: Backward search using the preceding character information. The inteval of the matching suffix is shrinked every iteration, when the preceding nucleotide is added.

To store the order of occurrence of the nucleotides in the BWT we will use the *Occ* structure. This structure, shown in Figure 2.8, is the individual cummulative count of each nucleotide in the BWT. The cummulative count directly indicates the order of appearance of a nucleotide in any position of the BWT.

Figure 2.8 illustrates the query of $q = \texttt{GAGA}$ in t, now using the complete index with C and *Occ*. First we start looking for the full range of A, this is from $C[\texttt{A}]$ until $C[\texttt{C}] - 1$. Then we check the occurrences of G within this range: (i) we find one appearance of G before $C[\texttt{A}]$ checking $Occ(\texttt{G}, C[\texttt{A}] - 1) = 1$ and (ii) at the end of the range, we have $Occ(\texttt{G}, C[\texttt{C}] - 1) = 5$. With a simple subtraction we conclude that the range contains 4 G, from the 2nd until the 5th. Now we would like to obtain the range of the suffix array covering these G. Since $C[\texttt{G}]$ points to the 1st G, the range covering from the 2nd until the 5th G is from $C[\texttt{G}] + 1$ until $C[\texttt{G}] + 4$. Once we got a new range, we can start again the process by checking the occurrences of the next nucleotide of the query.

In general, we define the range update process for an arbitrary query q of length n as

$$\begin{aligned} sp[i+1] &= C[q[n-i]] + Occ(q[n-i], sp[i] - 1) \\ ep[i+1] &= C[q[n-i]] + Occ(q[n-i], ep[i]) - 1 \end{aligned} \tag{2.8}$$

for $i = 1 \ldots n-1$ and $sp[0] = C[q[n]]$, $ep[0] = C[q[n]+1] - 1$. The range at the i-th step of the query is completely determined by $[sp[i], ep[i]]$, where sp and ep denote the start and end points of the BWT ranges, respectively. Once the update process finishes, we obtain the range $[sp[n], ep[n]]$, which contains the positions of q in the suffix array. The number of occurrences of q in the text is $ep[n] - sp[n] + 1$ and their positions in t will be the values of the Suffix Array stored at positions

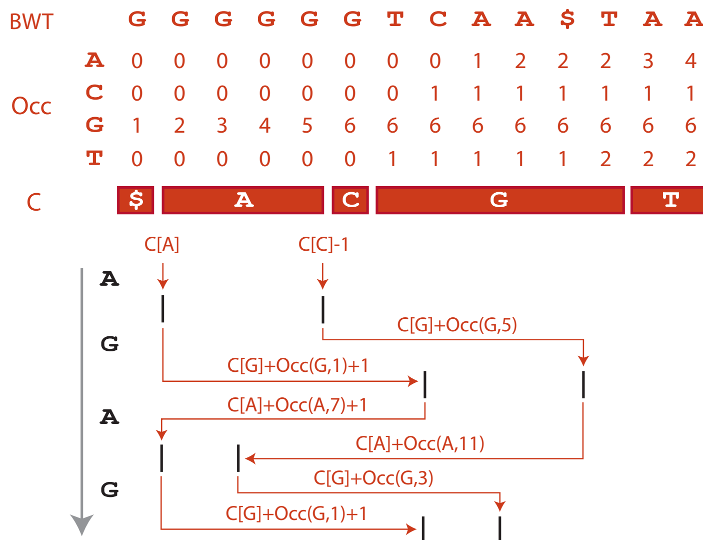

Figure 2.8: Backward search using only the C and Occ tables. The suffix intervals are updated with (2.8) reading the query backwards.

from $sp[n]$ to $ep[n]$. Note that the update algorithm described in (2.8) takes exactly n update steps, which only require the Suffix Array, the *Occ* table and the C array as auxiliary structures. Hence, we conclude that the backward search algorithm has a complexity of $O(n)$. The memory footprints for the different structures are: $O(m)$ for the Suffix Array, $O(\|\Sigma\|)$ for the C array and $O(\|\Sigma\|m)$ for the *Occ* table, resulting in an overall memory footprint $O(\|\Sigma\|m)$. However, with the use of simple compression techniques, the final size of the index may be reduced to less than m bytes.

Bi-directional FM index

There exists a variant of the FM-index that supports both *backward* and *forward* search, as well as switching from backward search to forward search, or vice versa. This index is called **bi-directional FM index** [14]. This index is just an extension of the *FM index* described above, in which the reverse complement of the text is appended to the original text. That is, if t is the original text to be indexed and $(\cdot)^R$ denotes the reverse complement operation, then the input to the FM index is "$tt^R\$$". The main advantages of having the reverse complement of the text stored in the index are:

- the possibility of searching in both DNA strands with just one query.
- performing a *backward search* operation on the reverse strand is equivalent to a *forward search* on the forward strand, and vice versa.

For that, we need to know the position of the start and end pointers for both strands for a given query, e.g. if we perform a backward search of `ACTGCA` and compute its *sp* and *ep* pointers in a

given index using (2.8), the pointers of the reverse complement of the query (reverse strand) sp_{q^R} and rp_{q^R} are necessary to perform bi-directional search. To keep track of the positions of the pointers in the forward query and its reverse complement, we define the following property:

Property 2. *For any given sequence q, the number of occurrences of q and q^R on a bi-directional FM index structure is the same.*

As an example, we will search the query `ACTGCA` starting from the middle, i.e first do backward search on `ACT` and then extend the query forward querying `GCA`. We start the backward search with the nucleotide `T`, but we need to store the pointers both for the forward and reverse complement of the query. For the first nucleotide this process is straightforward; we find the *sp* and *ep* pointers as: $sp_{\mathtt{T}} = C[\mathtt{T}]$, $ep_{\mathtt{T}} = C[\mathtt{T}{+}1]-1$. And the same for the reverse complement: $sp_{\mathtt{T}^R} = C[\mathtt{T}^R] = C[\mathtt{A}]$ and $ep_{\mathtt{T}^R} = C[\mathtt{A}{+}1] - 1$. Since we know from Property 2 that $ep_q - sp_q = ep_{q^R} - sp_{q^R}$, we do not need to store the four pointers. Instead, the search state in the bi-directional FM index is completely defined by a triplet: the forward pointer $fp_q = sp_q$, the reverse pointer $rp_q = sp_{q^R}$ and the interval size $sz_q = ep_q - sp_q + 1$. Using this notation, we derive a second property:

Property 3. *For any given sequence q, $fp_q = rp_{q^R}$ on a bi-directional FM index structure.*

Once the triplet $fp_{\mathtt{T}}, rp_{\mathtt{T}}, sz_{\mathtt{T}}$ has been computed, one can extend the search either forward or backward. To continue with our example, we extend the forward strand of the query backwards, adding a `C`. We do so with the backward search update step described in (2.8). This will yield the updated values $fp_{\mathtt{CT}}$ and $sz_{\mathtt{CT}}$, but we still need to keep simultaneous track of the pointer in the reverse complement strand: $rp_{\mathtt{CT}}$. Note that a backward extension on the forward strand is equivalent to a forward extension on the reverse strand (the equivalent search of `CT` in the reverse strand is `AG`). Thus, to know $rp_{\mathtt{CT}}$ we need to perform a forward search of the nucleotide `G` on the reverse strand of the query. The process to update $rp_{\mathtt{T}}$ to $rp_{\mathtt{CT}}$ relies on very basic properties of the bi-directional FM index and the suffix array. Our current interval $I_{\mathtt{T}^R} = [rp_{\mathtt{T}}, rp_{\mathtt{T}} + sz_{\mathtt{T}} - 1]$ are all the positions of the suffix array with suffixes starting with $\mathtt{T}^R = \mathtt{A}$ (Figure 2.8). Since the forward-extended suffix `AG` also contains the suffix `A`, the new pointer $rp_{\mathtt{CT}}$ must lie in the current interval $I_{\mathtt{T}^R}$, i.e. $I_{\mathtt{CT}^R} = [rp_{\mathtt{CT}}, rp_{\mathtt{CT}} + sz_{\mathtt{CT}} - 1]$ is a subinterval of $I_{\mathtt{T}^R}$. More generally, we can divide any interval I_q in 5 subintervals, one for each possible forward extension: $I_{q\mathtt{A}}$, $I_{q\mathtt{C}}$, $I_{q\mathtt{G}}$, $I_{q\mathtt{T}}$ and $I_{q\mathtt{N}}$, as shown in Figure 2.9. In such context, the following identities verify:

$$\begin{aligned} I_q &= I_{q\mathtt{A}} \cup I_{q\mathtt{C}} \cup I_{q\mathtt{G}} \cup I_{q\mathtt{T}} \cup I_{q\mathtt{N}} \\ |I_q| &= |I_{q\mathtt{A}}| + |I_{q\mathtt{C}}| + |I_{q\mathtt{G}}| + |I_{q\mathtt{T}}| + |I_{q\mathtt{N}}| \end{aligned} \tag{2.9}$$

where $|I_q| = sz_q$. Provided that $sz_{\mathtt{CT}} = sz_{\mathtt{AG}}$ from Property 2, we can obtain the interval sizes for the forward extension as $|I_{q\mathtt{X}}| = sz_{q\mathtt{X}} = sz_{(q\mathtt{X})^R} = sz_{\mathtt{X}^R q^R}$, therefore the sizes obtained with the backward search can be reutilized to compute the subinterval sizes

$$|I_{q\mathtt{X}}| = sz_{\mathtt{X}^R q^R} \tag{2.10}$$

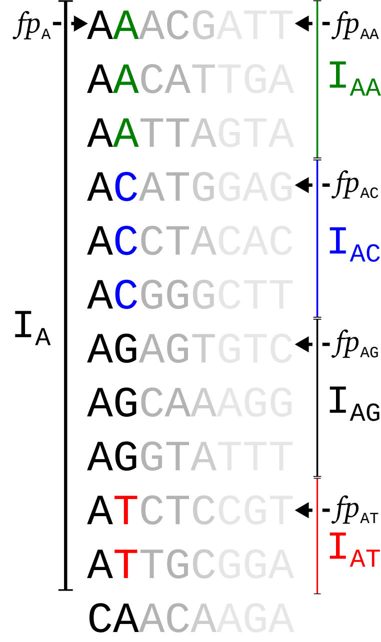

Figure 2.9: Subintervals of the suffix A. The subintervals are consecutive in the suffix array. The interval of the suffix A is denoted by $\mathtt{I}_{\mathtt{A}}$, the subintervals are $\mathtt{I}_{\mathtt{AA}}, \mathtt{I}_{\mathtt{AC}}, \mathtt{I}_{\mathtt{AG}}, \mathtt{I}_{\mathtt{AT}}$ and f_p is the position of the first suffix of the interval.

where $\mathtt{N}^R = \mathtt{N}$. Once the subinterval sizes are known, we use Property 3 to derive the update rules for rp_q:

$$
\begin{aligned}
rp_{\mathtt{T}q} &= rp_q \\
rp_{\mathtt{G}q} &= rp_q + sz_{\mathtt{T}q} \\
rp_{\mathtt{C}q} &= rp_q + sz_{\mathtt{T}q} + sz_{\mathtt{G}q} \\
rp_{\mathtt{A}q} &= rp_q + sz_{\mathtt{T}q} + sz_{\mathtt{G}q} + sz_{\mathtt{C}q} \\
rp_{\mathtt{N}q} &= rp_q + sz_{\mathtt{T}q} + sz_{\mathtt{G}q} + sz_{\mathtt{C}q} + sz_{\mathtt{A}q}
\end{aligned}
\tag{2.11}
$$

Back to our example, we already computed $fp_{\mathtt{CT}}$ and $sz_{\mathtt{CT}}$ with the backward search update rules, we can now follow Equation 2.11 and obtain the reverse strand query pointer as $rp_{\mathtt{CT}} = rp_{\mathtt{T}} + sz_{\mathtt{T}q} + sz_{\mathtt{G}q}$.

To perform a forward search, one only needs to apply the same procedure to the reverse strand. In effect, assume that we already computed the triplet $fp_{\mathtt{ACT}}$, $rp_{\mathtt{ACT}}$, $sz_{\mathtt{ACT}}$ for the first half of the query and we would like to continue extending the query (ACTGCA) forward. To do so, we perform a forward extension of G in the forward strand. Again, performing a forward extension on the forward strand is equivalent to a backward extension to the reverse strand, so we only need to apply the same process but swapping the strands. We update the reverse pointer $rp_{\mathtt{ACTG}} = fp_{\mathtt{CAGT}}$ with a backward search on $\mathtt{G}^R = \mathtt{C}$ and compute the forward pointer as $fp_{\mathtt{ACTG}} = fp_{\mathtt{ACT}} + sz_{\mathtt{ACTA}} + sz_{\mathtt{ACTC}}$.

2.3 The sequence neighborhood

In this section we will introduce the concepts of sequence neighborhood and neighbor sequences. The neighbor annotation of a text t will be very useful to determine sequence similarities and will be exploited in some steps of the mapping algorithm to speed-up the process and improve the quality of the classification.

Let us first define the **sequence neighborhood**. A k-neighborhood is the complete set of sequences that are k nucleotides long, i.e. the permutation with replacement of k elements taken from the set $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. A k-neighborhood contains 4^k different sequences of length k (k-mers). For instance, the 1-neighborhood is {A, C, G, T}, the 2-neighborhood {AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, TT}, and so on...

We say that two k-mers are **τ-neighbors** *iif* their distance under a certain metric is equal to τ. We consider two different metrics for DNA sequences, namely the *Hamming distance* where only substitutions are accounted as differences, and the *Levenshtein distance* where both substitutions and indels (insertions and deletions) are accounted as differences[2]. Both metrics are suitable for the methods introduced below.

2.3.1 Neighborhood annotation algorithm

In future sections, we will assist the detection of repeats and accurately compute the probability of correct mapping with a neighborhood annotation. The annotation consists in computing the τ-neighbors of all the k-mers of a reference text and store, for each of them, (i) the distance to the closest neighbor and (ii) the number of neighbors at such distance. This information very is valuable because sequences that have, for instance, a high count of 1-neighbors are more prone to be incorrectly mapped compared to the ones whose closest neighbor is at greater distance.

The design of an efficient algorithm to annotate the neighborhood of every k-mer of a long text is not straightforward. A naive approach would be to compare all the k-mers of the text with each other. Such algorithm has a complexity of $O(m^2k)$ for a genome of size m. In the human genome, where $m \sim 10^9$, this approach would take ages to complete.

In this section we propose a divide-and-conquer algorithm to efficiently compute a neighborhood annotation. This algorithm exploits the advantages of the bi-directional FM index and can compute a full annotation of the human genome with $k = 48$, $\tau = 4$ in less than 100 CPU hours.

The algorithm proceeds as follows:

1. Select the next k-mer from the suffix array (in lexicographical order).
2. Query the sequence using backward search. If the sequence has more than one matching position, set $\tau = 0$.
3. Otherwise, perform a divide and conquer search up to τ mismatches, using bi-directional search and starting the search by the interval without mismatches.

[2] Introduced in Section 1.2

4. Annotate the distance to the closest neighbor $\hat{\tau}$ and the number of matches at distance $\hat{\tau}$ in the position of the genome (available at the suffix array) where the k-mer was taken from.

Block search algorithm

For these cases in which the query sequence is not exactly repeated in the genome, we have designed a divide and conquer algorithm to perform inexact k-mer search, up to τ mismatches. This algorithm is called *block search* because it divides the k-mer in $\tau + 1$ blocks and performs a recursive inexact matching using the bi-directional FM index.

The algorithm starts by dividing the sequence in $\tau + 1$ blocks, at this level `blockcount` is $\tau + 1$. Then, the blocks are grouped in two sides, left and right, and we recursively apply the same division to both sides. Once `blockcount` is 1, an exact search is performed and the result of the search is returned to the calling function. When the block search call on the left side has returned, we extend the search until the end of the right side, allowing up to `blockcount-1` mismatches. The same procedure is applied once the recursive call to the right side has returned, extending until the end of the left side. On the top level, the recursive call will return all the matches from the left side with up to $\tau/2$ differences, and each one of these matches will be extended forward until the end of the query, allowing a total of τ mismatches. The other recursive call will return all the matches of the right side with up to $\tau/2$ mismatches and they will also be extended using backward search until the beginning of the query, allowing up to τ mismatches. The aggregate of the forward and backward extension is the set of sequences in the index that match the query with up to τ mismatches. The block division and extend process is illustrated in Figure 2.10 and the algorithm is described in Algorithm 1.

The algorithm presented achieves a notable speedup over the naive algorithm described above. The improved efficiency of the FM index reduces the complexity from $O(m^2k)$ to $O(m\binom{k}{\tau}3^\tau)$, where the binomial-exponential term represents the exhaustive tree-like traversal of the index to find inexact matches. The block search algorithm (described in Algorithm 1) is efficient because it reduces the search space of the index where the expensive inexact search is performed. In other words, since the block search always starts by an exact search of $k/(\tau + 1)$ nucleotides, the subsequent dead-end mismatched extensions are more likely to fail immediately unless the queried sequence is a true neighbor. In a best case scenario (random genomes), the search space and hence the complexity would be consistently reduced by at least a factor $4^{k/\tau+1}$ after the first exact search.

2.3.2 k-mer uniqueness

A k-mer s is τ-unique in a reference text t if neither d-neighbor (for $d = 0 \ldots \tau$) of s is a suffix of t. It is straightforward to check the uniqueness of s using the previously described neighborhood annotation. We can obtain the genomic locus of s with a backward search on t and then check the distance of the closest neighbor in the annotation. If the annotated distance is greater than τ then s is τ-unique in t.

Algorithm 1 Block Search algorithm

```
1:  function BLOCK_SEARCH(query,blocks):
2:      if len(blocks) == 1 then                                          ▷ Last block.
3:          matches = FW_SEARCH on query[blocks] with τ = 0
4:          return matches
5:      end if
6:                                                                        ▷ Split blocks
7:      middle = (blocks[0] + blocks[len(blocks)-1])/2
8:      left_blocks = blocks[0]:middle
9:      right_blocks = (middle+1):blocks[len(blocks)-1]
10:                                                     ▷ Recursively process left blocks
11:     left_matches = BLOCK_SEARCH(query, left_blocks)
12:                                            ▷ Extend left blocks to the right (fw search)
13:     for all left_matches do
14:         matches += FW_EXTEND left_match on query[right_blocks] with τ = blocks-1
15:     end for
16:                                                    ▷ Recursively process right blocks
17:     right_matches = BLOCK_SEARCH(query, right_blocks)
18:                                            ▷ Extend right blocks to the left (bw search)
19:     for all right_matches do
20:         matches += BW_EXTEND right_match on query[left_blocks] with τ = blocks-1
21:     end for
22:     return matches
23: end function
```

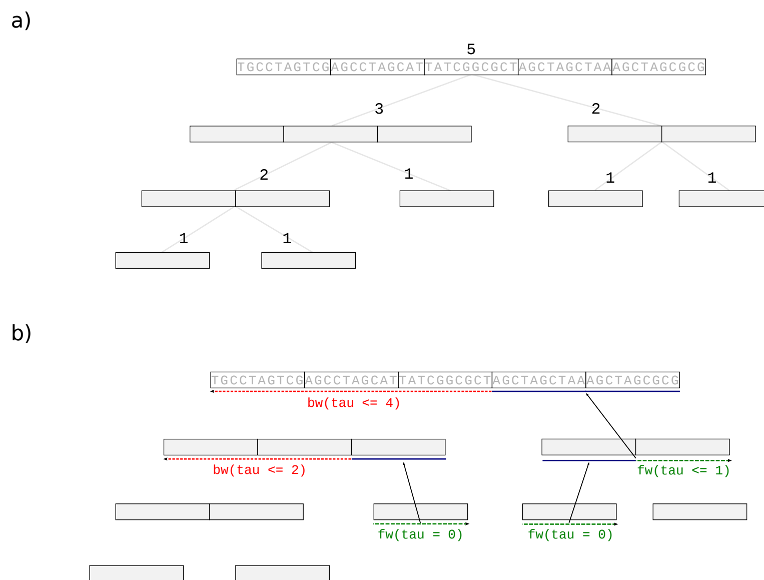

Figure 2.10: Representation of the recurive block search algorithm. The first step (top-bottom) divides the intervals in two blocks. When the bottom is reached, the algorithm starts a bottom-up search and extension.

Chapter 3

Mapping algorithm

The field of sequence alignment and mapping started right after the appearance of the first computers. During the 80s there were no assembled genomes but biologists were making efforts to reveal the DNA and amino-acid sequences of the genes and their proteins. However, at that stage, there was no apparent need to find the origin of the sequence. Instead, the main interest was to investigate whether there were potential sequence similarities and/or divergences between species. To do so, big databases containing the set of known protein and DNA sequences were released so that scientists could start comparing their findings. The growth of the available annotations triggered the need for fast and accurate computer algorithms to find local sequence similarities of relatively short queries in a much bigger reference. The first approaches, like FASTA [15], were based on optimized word-to-word heuristic search methods which selected potential matches before performing exhaustive alignments. On the 90s, BLAST [16] succeeded the previous algorithms with a novel combination of statistics and seeding strategies, where the high scoring words of the query were extracted and used to filter the most significant seeds in the database. Also, BLAST introduced a statistical model to assess the significance of the alignments.

The field of sequence mapping evolved rapidly with the appearance of new-generation sequencing technologies (NGS). The modern sequencing technologies are nowadays able to produce data of the order of thousands of millions of base-pairs per day. This changed the use and the needs dramatically. Notably, some established experiments like Chromatin Immuno-Precipitation (ChIP) [17] or Chromosome Conformation Capture (3C) [18] were adapted to take advantage of high-throughput sequencing (ChIP-seq[19], Hi-C[20]) and the immediate need was to identify where, in a reference genome, the sequenced reads were coming from. Even though the new mapping algorithms use different kinds of indices, namely hash-tables or suffix arrays/FM indices, the vast majority are still based on the seed-and-extend approach. This heuristic, nonetheless, is not new, but it has been continuously improved with more sophisticated seeding strategies and faster alignment algorithms.

Our algorithm is also a standard seed-and-extend approach with several improvements in the seeding strategy and the statistical evaluation of the mapping reliability. Overall, our algorithm achieves higher sensitivity and better estimation of mapping reliability with Illumina reads, at the

same speed compared to the state-of-the-art mappers.

3.1 Seeding

As discussed before in Section 2.1, the main difficulty of mapping is to deal efficiently with the divergence between the query sequence and its best match in the reference. These differences may be caused by the natural divergence of the species, sequencing errors or polymorphism, among others. Seeding is the most widely-used heuristic to solve this problem and has become a critical part of any aligning/mapping algorithm. The technique consists in identifying tentative query-reference matches by finding small regions of the query that partially match the reference with high significance. Seeding is very fast in combination with modern indices (Sec 2.2) and significantly speeds up the overall mapping process. This speed up is due to the fact that exhaustive alignments are reserved only for the positions of the reference that show significant similarity with the query. Moreover, seeding is a flexible heuristic and offers a well characterized trade-off between speed and sensitivity. There are several widely used strategies in the literature, namely fixed-length seeding, usually for indices based on hash tables, maximal exact match (MEM) seeding, threshold seeding, etc. In our algorithm we use a combination of three seeding strategies that are suitable for the bi-directional FM index: MEM seeding, threshold seeding and inexact seeding.

3.1.1 MEM seeding

A maximal exact match (MEM) is any longest uninterrupted exact match between the query and the reference. In other words, a MEM is an exact match between two sequences that cannot be extended in either direction. MEMs are a form of seed especially suited to the FM index, given that forward and backward search allow arbitrarily long extensions in either direction. A query sequence may have only one (if the query exactly matches the reference) or many MEMs. The fact that a query has more than one MEM indicates that it necessarily differs from the reference, but the contrary does not guarantee the real correspondence between the original sequence (prior to sequencing) and its exact match, provided that a mutation in the original sequence may still exactly match a different sequence of the reference.

MEMs become computationally advantageous seeds when there is a high correspondence between the reads and the reference. In such conditions, the search space is considerably reduced because MEMs tend to generate long seeds that exploit the high identity of the compared sequences. On the other side, however, long MEMs perform poorly on the job of detecting repeats. Detecting highly repeated subsequences in the query is utterly important since these reveal a potentially larger set of match candidates in the reference. To handle this issue, usually MEM seeding is accompanied with reseeding of long MEMs to avoid seeding only over one variant of a highly repeated sequence. Reseeding a MEM with L hits consists in finding the longest seed inside the MEM that matches at least $L + 1$ positions in the reference.

3.1.2 Threshold seeding

Threshold seeding consists in performing arbitrarily long seed extensions until the number of hits in the reference drops below a certain threshold. Every time a seed is extended, the number of hits is either mantained or decreased, therefore one should start a forward extension from the beginning of the seed and continue extending until the last seed whose number of hits is greater than the defined threshold.

This seeding method is very simple but powerful. Its main strength is that it generates seeds of relatively high sensitivity (the higher the threshold the more sentitive) while permitting a very precise control of the number of alignments and hence the mapping time.

3.1.3 Inexact seeding

Taking advantage of the bi-directional FM index and the block search algorithm, it is feasible to compute mismatched seeds with small τ (usually 1 or 2 mismatches). Inexact seeding can be applied in many ways: inexact MEMs, to find potential repeats or close neighbors, inexact seeds of fixed length to increase sensitivity, or even inexact seeding on short subsequences of the query that do not exist in the reference, as a way to perform error correction. The method is simple, the subsequence is passed to the block search algorithm (described in Sec 2.3), which returns the set of neighbors at distance τ. The neighbors are then treated as normal seeds and the algorithm proceeds with the alignments. This is the most sensitive but the slowest seeding method and should be reserved for those reads that fail to find significant matches with the other seeding methods.

3.2 Filtering significant seeds

A step prior to alignment is to classify the seeds based on their significance. Given that the number of seeds obtained at the seeding stage may be large, it is critical to filter out those that are not significant enough to be worth spending a complete alignment. The aim of this step is, therefore, to enhance the throughput of the algorithm while maintaining the sensitivity. The filter is applied on several seed metrics: the seed length, the number of hits and the significance.

3.2.1 Filter on seed length

The first step is to discard the seeds shorter than l nucleotides. The parameter l depends on the size of the reference text, and the choice is usually greater than 17 nucleotides for the Human genome. Assuming a perfectly random genome of size m, the expected number of appearances of an arbitrary sequence of length l is:

$$E_{match}(l, m) = \frac{m - l + 1}{4^l} \approx \frac{m}{4^l} \tag{3.1}$$

where the approximation holds for $m \gg l$. Applied to the Human genome ($m \sim 3 \cdot 10^9$), we expect to find $3 \cdot 10^9/4^{17} = 0.17$ matches per random search of length $l = 17$, $E_{match}(15, 3 \cdot 10^9) = 2.79$ for $l = 15$ and $E_{match}(19, 3 \cdot 10^9) = 0.01$ for $l = 19$. This important result indicates that seeds shorter than 17 nucleotides are not significant due to their high likelihood to randomly match the human genome, given that almost all the possible 16-mers exist in the reference. The threshold value $\bar{l}$ is 19 for our default Illumina setting.

3.2.2 Filter on number of hits

Once the short seeds have been discarded, we expect the remaining seeds to match a single position in a random reference. Real genomes, however, are not random. They are mostly filled with repetitive and conserved sequences, which notably increase the chances of finding the same sequence in several distinct positions even if the sequence is apparently significant. At this step, a filter is applied on the maximum permitted alignments per seed. Our consideration is that seeds matching more than $\bar{N}_h$ different positions are likely to originate from long repeats. Hence, performing a full alignment in all the matching positions would be redundant. Thus, if a sequence matches more than $\bar{N}_h$ positions, $\bar{N}_h$ randomly selected matches will be aligned and the rest are discarded. The default value is $\bar{N}_h = 20$ in our Illumina settings.

3.2.3 Filter on seed significance

The evaluation of the seed significance based on the sequence neighborhood is a novel contribution of this work. This method is very powerful in the sense that it permits a quantitative evaluation of the seed significance. Significant seeds are those that associate a read to a single (or a few) position in the reference with high confidence. This method utilizes the knowledge of the seed neighborhood to measure its significance. For instance, a seed that has only one hit but many 1-neighbors will be less significant that a seed whose closest neighbor is at distance 3. The measurement of the significance is based on the probability that the seed contains errors and is the result of a fortuitous divergence of the original sequence towards another existing sequence. Consider that our seed of length l has N_c closest neighbors at distance τ_c. Then, the conditional probability $P(\text{neighbor}|\tau \text{ differences})$ that the original sequence is not the seed, but one of its closest neighbors, given that the original sequence suffered a divergence of τ differences (including insertions, deletions and substitutions) is

$$P(\text{neighbor}|\tau \text{ differences}) = \frac{N_c}{\binom{l}{\tau} 8^\tau} \tag{3.2}$$

where the 8 (3 for Hamming distance) accounts for all the possible divergences: 3 substitutions, 4 insertions and one deletion. Applying the law of total probability, we derive the probability that

the true sequence is not the seed but originates from one of its neighbors:

$$P(\text{neighbor}) = \sum_{\tau} P(\text{neighbor}|\tau \text{ differences}) \cdot P(\tau \text{ differences}) \tag{3.3}$$

We observe from the neighborhood annotation that the closest neighbor is at distance τ_c, therefore $P(\text{neighbor}|\tau \text{ differences}) = 0$ for $\tau < \tau_c$. Assuming also that

$$P(\text{neighbor}|\tau_{c+1}\text{differences}) \cdot P(\tau_{c+1} \text{ differences}) \ll P(\text{neighbor}|\tau_c \text{ differences}) \cdot P(\tau_c \text{ differences})$$

we approximate

$$\begin{aligned} P(\text{neighbor}) &= \textstyle\sum_{\tau \geq \tau_c} P(\text{neighbor}|\tau \text{ differences}) \cdot P(\tau \text{ differences}) \\ P(\text{neighbor}) &\approx P(\text{neighbor}|\tau_c \text{ differences}) \cdot P(\tau_c \text{ differences}) \end{aligned} \tag{3.4}$$

this lets us obtain an upper bound for $P(\text{neighbor})$, considering that $P(\tau_c \text{ differences}) \ll 1$,

$$P(\text{neighbor}) < P(\text{neighbor}|\tau_c \text{ differences}) \tag{3.5}$$

or, equivalently

$$P(\text{neighbor}) < \frac{N_c}{\binom{l}{\tau} 8^{\tau}} \tag{3.6}$$

Table 3.1 summarizes this probability for various combinations of N_c and τ_c at the minimum seed distance for the Illumina setting ($l = 19$).

Table 3.1: Upper bound probability of incorrect seed ($l = 19$).

	$\tau_c = 1$	$\tau_c = 2$	$\tau_c = 3$
$N_c = 1$	$6.6 \cdot 10^{-3}$	10^{-4}	$2 \cdot 10^{-6}$
$N_c = 10$	0.07	10^{-3}	$2 \cdot 10^{-5}$
$N_c = 100$	0.66	0.01	$2 \cdot 10^{-4}$

After the first round of seeding, if a seed shows a $P(\text{neighbor})$ smaller than a predefined threshold ($\sim 10^{-4}$) it is considered significant enough, thence only this seed is aligned and the rest are discarded.

3.3 Aligning

Once a seed has passed the significance filter, it is used as an anchor to start a more exhaustive comparison between the query and the reference. Such comparison is called *sequence alignment*. Standard methods for sequence alignment were described in Section 1.2.

3.3.1 Sequence alignment

The alignment process starts at the seed, i.e. coordinates where the read and the reference partially match. Then two Needleman-Wunsch (NW) alignments are performed, extending the seed forward (towards the end of the read) and backwards (towards the beginning of the read). The overall score of the alignment is computed as the sum of both extensions. In the current version of the software we used a naive non-optimized implementation of the NW algorithm, however, SIMD [21, 22] and GPU [23] optimized versions of this algorithm have been published, reporting speedups of several orders of magnitude.

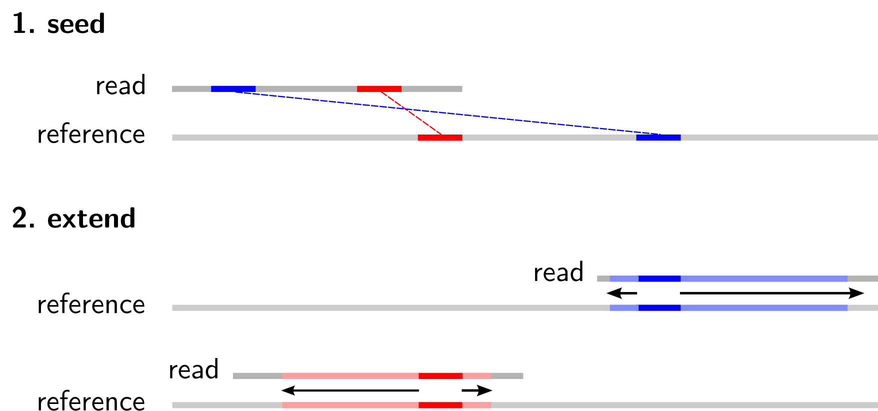

Figure 3.1: Seed and extend algorithm. Seeding consists in finding short exact matches between the read and the reference. The extension step performs a full alignment between the read and the reference at the position of the seeds.

3.3.2 Split reads and breakpoint algorithm

Another interesting feature of our mapping algorithm is its ability to map split reads. Split reads are sequences constituted by more than one DNA fragment. Such reads are deliberately generated in experiments like Hi-C [20], in which the different fragments that constitute the sequenced DNA molecule may originate from very distant positions of the genome.

In this context, the problem generalizes to map an unknown number of subsequences of unknown lengths. One possible way to detect the breakpoint between two different fragments is to infer the location of an abrupt decrease of sequence identity during alignment. In effect, during the Needleman-Wunsch extension, one expects high identity between the read and the reference as long as the alignment extends over the seeded fragment. However, when the alignment reaches the breakpoint between fragments, the rest of the extension behaves approximately like an alignment between random sequences. We exploit this fact and design an optimal breakpoint detection algorithm based on the Maximum Likelihood (ML) breakpoint detection model presented in [24].

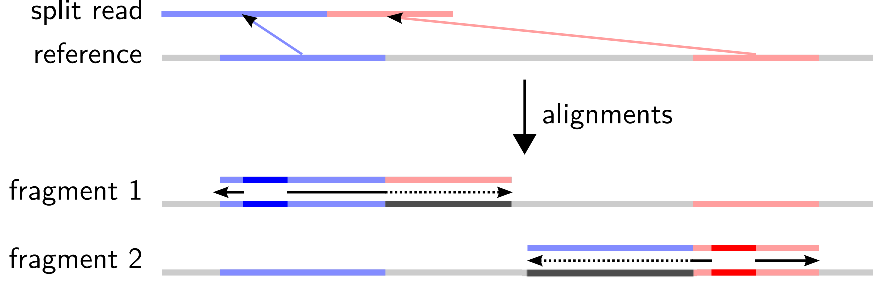

Figure 3.2: Seed and extension over a split read. The split read is composed of two distant fragments of the reference (blue and red). At the extension step, the alignment proceeds passed the fragment border, where the sequences do not coincide anymore (represented in gray on the reference).

In the breakpoint detection algorithm we model our read as a sequence $S = (S_1, \ldots, S_n)$, where the S_i are independent random variables following a binomial probability distribution

$$\mathrm{pr}(S_i = 1) = \begin{cases} \theta_0 & (i = 1, \ldots, b) \\ \theta_1 & (i = b+1, \ldots, n) \end{cases} \tag{3.7}$$

and $\mathrm{pr}(R_i = 0) = 1 - \mathrm{pr}(R_i = 1)$; and θ_0 and θ_1 are the known alignment mismatch probabilities but the change-point b is unknown. The random variable outputs 0 and 1 represent matches and mismatches between the read and the reference, respectively. For our model of alignment over a split-read, we estimate θ_0 to be the expected error probability of two truly-matching sequences, which depends on the sequencing technology error rate, the mutation rate and the divergence of the reference text and the real genome. We expect the value of θ_0 for Illumina reads mapped on a well annotated genome to be $\theta_0 < 0.05$. On the other side, θ_1 represents the error read past the split-read boundary, which we approximate to a comparison of two random sequences, being $\theta_1 > 0.35$ for an alignment with insertions, deletions and substitutions.

Under the model (3.7), the likelihood function of $(S_1, \ldots, S_n)$ is

$$\prod_{i=1}^{b} \theta_0^{S_i} (1-\theta_0)^{1-S_i} \prod_{i=b+1}^{n} \theta_1^{S_i} (1-\theta_1)^{1-S_i} \tag{3.8}$$

so that the log likelihood conditional on $b = t$, with known θ_0 and θ_1, can be written as

$$L(t) = \sum_{i=1}^{t} \left\{ S_i \log\left(\frac{\theta_0}{\theta_1}\right) + (1-S_i)\log\left(\frac{1-\theta_0}{1-\theta 1}\right) \right\} + \sum_{i=1}^{n} \{ S_i \log \theta_1 + (1-S_i)\log(1-\theta_1) \} \tag{3.9}$$

Hence the maximum likelihood estimate of the breakpoint $\hat{b}$ is the value of t which maximizes the

sequence

$$X_t = \sum_{i=1}^{t} \left\{ S_i \log\left(\frac{\theta_0}{\theta_1}\right) + (1 - S_i) \log\left(\frac{1-\theta_0}{1-\theta_1}\right) \right\} \qquad (t = 1, \ldots, n) \tag{3.10}$$

The maximization step described in (3.10) is applied periodically during the alignment process to track the position and the likelihood of a potential breakpoint. The algorithm stops when $X_{\hat{b}}$ exceeds a defined threshold, or at the end of the alignment otherwise. A detailed relationship between the the breakpoint confidence and the likelihood $X_{\hat{b}}$ can be found in [24].

3.4 Mapping quality

At this point of the algorithm we obtained many candidate mapping positions and computed the alignment score for them. But there is still one crucial step: to provide a mapping confidence score, i.e. the probability that the association between the read and the reported locus is correct. The mapping quality model is what ultimately makes the difference between a good and an average mapper, because the final users will process their reads and trust the assignments based on a threshold confidence score. Hence, the mappers should be characterized by: (i) *how many of the input reads is the mapper able to find in the reference?* (sensitivity) and (ii) *how many of the high confidence mappings are actually wrong assignments?* (false positive rate). In this section we present two simple models to evaluate the mapping confidence, which used in combination yield better confidence estimates compared to other widely-used mappers [25, 26].

The mapping quality reported by our mapper is computed as:

$$\text{MAPQ} = -10 \log\left(\text{pr(wrong assignment)}\right) \tag{3.11}$$

3.4.1 Alignment score model

This widely-used mapping quality model is based on the alignment scores of the extended seeds. The idea is simple: sort the alignment scores of the seed extensions. The locus with highest alignment score is considered the true mapping position of the read. In case there are many $n_1 > 1$ best matches, each match is considered equally likely to be the true origin of the sequence. Hence, the probability of incorrect assignment is pr(wrong assignment) $= 1 - 1/n_1$. As the gap between the best and the second best alignment scores grows, the probability that the best alignment is indeed the correct mapping position becomes evident. In effect, this is analogous to the neighbors model, a big gap in the alignment score indicates that the reference sequence is somehow unique and therefore it is unlikely that its similitude with the read is due to an artifact.

When the alignments report only one $n_1 = 1$ best alignment, the model computes the mapping quality with an heuristic which takes into account the difference between the best and the second best alignment scores Δ_{12}, the number of second best score matches n_2 and the identity (1 minus

the mismatch rate) of the best alignment η:

$$Q_A = f(\eta) \cdot (c\Delta_{12} - 10\log_{10}(n_2)) \tag{3.12}$$

where f is a function and c a constant. This model is powerful but may show substantial performance differences depending on the sequencing technology and the reference genome, because the parameters of the heuristic f, c are usually calibrated by fitting a curve on simulated data. This model alone, however, already gives extraordinary results for well-calibrated inputs [25].

3.4.2 The neighbors model

The neighbors model follows the same principle of the alignment score model: quantify the reliability of the read-locus assignment through the exploration of the close neighborhood. This model, however, is both faster and more precise thanks to the availability of exhaustive neighborhood information. It also allows a notable reduction of the necessary alignments at the extension step, since one can assess the state of the neighborhood with a single seed extension, whereas the same is not possible using the alignment score model, in which at least two alignments are required.

Consider the following sequences T_R, T_B,and T_N, which represent the read, the sequence in the reference that best matches the read and its closest neighbor, respectively. T_B is known and has been obtained through a seed-and-extend process applied to T_R. The closest neighbor of T_B is generally unknown but we can infer some information about T_N using the neighborhood annotation. Since T_B and T_N exist in the reference, we will denote their positions with l_B and l_N. We also denote the Levenshtein distance between two sequences T_1 and T_2 as $\mathrm{d}(T_1, T_2)$.

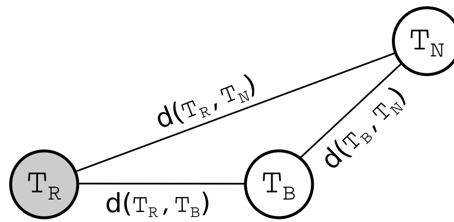

Figure 3.3: Local neighborhood of the read T_R, its best match T_B and the closest neighbor of the best match T_N. The Lenvenshtein distance between sequences is denoted by $\mathrm{d}(\cdot,\cdot)$.

The diagram of Figure 3.3 illustrates the local neighborhood composed by T_R, T_B and T_N. Note, however, that this neighborhood does not represent the annotation since, in general, T_R does not belong to the reference. The distance $\mathrm{d}(T_R, T_B)$ is known because T_R and T_B had been aligned previously during the seed-and-extend algorithm. The distance $\mathrm{d}(T_B, T_N)$ can be obtained from the neighborhood annotation, but the remaining distance $\mathrm{d}(T_R, T_N)$ is generally unknown.

The fact that $\mathrm{d}(T_R, T_S)$ is unknown greatly complicates the construction of a model to evaluate the mapping quality.

The goal of the model is to estimate the probability that T_R did not originate from T_B, which we approximate by the probability that T_R originated from any of its closest neighbors T_N. To derive an estimation of such probability, we make several reasonable assumptions on $\mathrm{d}(T_R, T_N)$:

Assumption 1. *The distance* $\mathrm{d}(T_R, T_B)$ *is small enough to assume that the whole* $\mathrm{d}(T_R, T_B)$*-neighborhood of* T_R *has been explored during the seeding stage. In other words, we assume that the seeding strategy is fully sensitive up to* $\mathrm{d}(T_R, T_B)$ *mismatches.*

Assumption 2. *As a result of assumption 1, we conclude that* T_B *is the closest neighbor of* T_R *and the inequality* $\mathrm{d}(T_R, T_N) > \mathrm{d}(T_R, T_B)$ *must hold. Otherwise, if* $\mathrm{d}(T_R, T_N) \leq \mathrm{d}(T_R, T_B)$ *we would have performed a seed extension of* T_R *on* T_N *and* $\mathrm{d}(T_R, T_N)$ *would be known.*

We start to build our model leveraging on assumptions 1 and 2, with the hypothesis that the true origin of T_R is l_N instead of l_B. From our definition of mapping quality, it is reasonable to approximate the probability of incorrect mapping by the probability that T_R originated from l_N even when T_B lies closer in the neighborhood space.

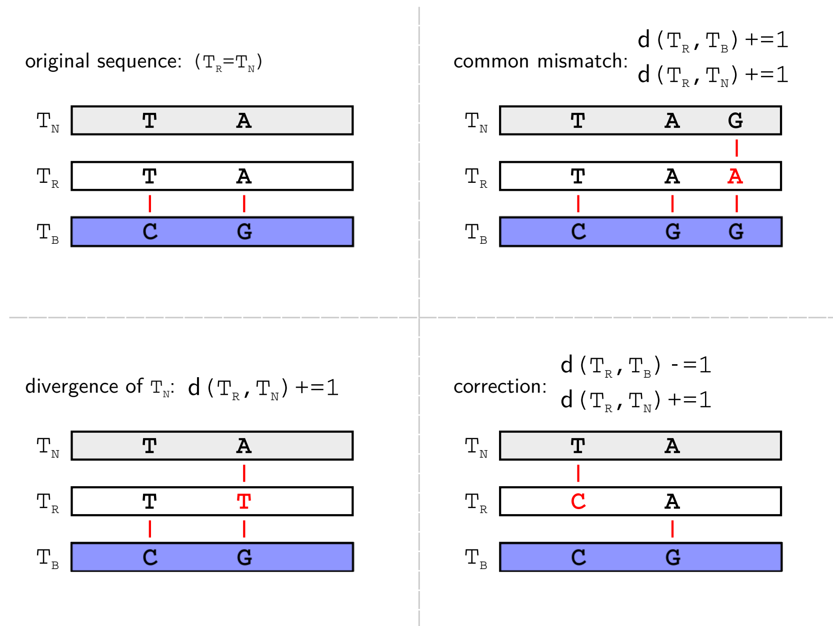

Figure 3.4: On our hypothesis for the mapping score, T_R originated from T_N. This figure summarizes the three possible scenarios when a nucleotide of T_R is modified.

From our hypothesis we conclude that the physical sequence taken from the reference is T_N but it has been exposed to modifications during the experiments and/or the sequencing steps, causing the final read to lie closer to T_B. In Figure 3.4 we summarize the possible modifications that the molecule can suffer and how do they affect the position of T_R relative to T_N and T_B. The top-left panel illustrates the original sequence, where T_R is copied from T_N, and their differences

with T_B. The only way to push T_R towards T_B is through nucleotide modifications, with three possible scenarios:

- The first is a *common mismatch*, when the modified nucleotide is not a mismatch between T_B and T_N, the distances from both references to T_R increase by one.
- The second case is when a mismatched nucleotide between T_N and T_B is modified but it is not set equal to the one in T_B. So, T_R *diverges from* T_N but not from T_B, because the mismatch already existed between T_R and T_B at that position.
- The third case is the same as case 2 but the modified nucleotide is set to match the one in T_B, causing a *correction* with respect to T_B. In this case the distance to T_N increases by one and $\mathrm{d}(T_R, T_B)$ is decreased by one.

Now the question is how these modification events combined to finally situate T_R closer to T_B. Following Figure 3.3, we can express the distances of the local neighborhood as

$$\begin{aligned} \mathrm{d}(T_R, T_N) = &\quad M + C + D \\ \mathrm{d}(T_R, T_B) = &\quad \mathrm{d}(T_B, T_N) + M - C \end{aligned} \tag{3.13}$$

where the distances and all the other terms are non-negative: M are the common mismatches, D the divergent mismatches and C the corrections. Besides, the sum of corrections and divergences must be equal or smaller than the distance between T_B and T_N:

$$D + C \leq \mathrm{d}(T_B, T_N) \tag{3.14}$$

Plugging (3.13) into assumption 2, it is clear that at least one correction or two divergences must happen to verify $\mathrm{d}(T_R, T_B) < \mathrm{d}(T_R, T_N)$:

$$\mathrm{d}(T_B, T_N) + M - C < M + C + D \tag{3.15}$$

which simplified, yields

$$\mathrm{d}(T_B, T_N) < 2C + D \tag{3.16}$$

where, in effect, for the smallest distance $\mathrm{d}(T_B, T_N) = 1$ we need $C \geq 1$ or $D \geq 2$.

Our interest is to find the probability of the most likely divergence of T_R towards T_B. Since divergences are more likely to happen compared to corrections, we will start by determining the minimum number of corrections necessary to verify the hypothesis. The value of C has two restrictions, the first is that M on (3.13) must be non-negative. The limiting case is $M = 0$, which plugged into (3.13) yields the inequality $\mathrm{d}(T_R, T_B) - \mathrm{d}(T_B, T_N) + C \geq 0$, hence

$$C \geq \max(1, \mathrm{d}(T_B, T_N) - \mathrm{d}(T_R, T_B)) \tag{3.17}$$

The second constrain is derived by applying the triangle inequality to Figure 3.3, i.e. $\mathrm{d}(T_R, T_N) \leq$

$\mathrm{d}(T_R, T_B) + \mathrm{d}(T_B, T_N)$,

$$2C + D \leq 2\mathrm{d}(T_B, T_N). \tag{3.18}$$

The two constrains together give a lower bound (3.17) and upper bound (3.18) for C, that we will denote C_L and C_U respectively. Once the range $[C_L, C_U]$ has been found, we can determine D_i and M_i for $i = [L, U]$ from and (3.13):

$$\begin{aligned} D &= \max(0, \mathrm{d}(T_B, T_N) - 2C + 1) \\ M &= \mathrm{d}(T_R, T_B) - \mathrm{d}(T_B, T_N) + C \end{aligned} \tag{3.19}$$

Finally, we model the probability that T_R originated from l_N given C_i corrections and D_i divergences on a sequence of length L as:

$$\mathrm{pr}_i(\text{origin is } T_N) = \frac{\binom{\mathrm{d}(T_B,T_N)}{C_i}\binom{\mathrm{d}(T_B,T_N)-C_i}{D_i} \cdot 3^{D_i}}{\binom{L}{M_i+D_i+C_i} \cdot 3^{M_i+D_i+C_i}} = \frac{\binom{\mathrm{d}(T_B,T_N)}{C_i}\binom{\mathrm{d}(T_B,T_N)-C_i}{D_i}}{\binom{L}{M_i+D_i+C_i} \cdot 3^{M_i+C_i}} \tag{3.20}$$

and the mapping quality is obtained from the maximum of these probabilities

$$\mathrm{MAPQ} = -10\log(\max_i(\mathrm{pr}_i(\text{origin is } T_N))). \tag{3.21}$$

Final remarks on the mapping quality

The mapping qualities are computed independently for the neighbors and the alignment score models. The final mapping quality is the minimum of these two. Also, the neighbors model does not have neighbor annotation for any length L, since the neighborhood annotations are computed for discrete values of k. Therefore, L is selected to be the largest $k < n$, where n is the length of the query. The neighbor model for the mapping quality may be then computed for all the k-mers of the read. In such case, one should keep the maximum among all the qualities obtained.

Chapter 4

Results

To test the results of this work, we implemented all the algorithms described above in C language. The source code of the mapper is available at `http://github.com/ezorita/mapper`. The mapping performance of our algorithm is tested and discussed in this chapter. We compare all the results of our mapper with two of the most widely used mappers: `bwa-mem` [25] and `bowtie2` [26]. The performance of the three mappers has been tested on simulated Human and Drosophila Illumina data generated with wgsim (`http://github.com/lh3/wgsim`).

Datasets of 1 million single-end 50bp reads were generated for different sequencing error rates. Namely, we considered three usual sequencing scenarios: good (1% error rate), standard (2%) and average (5%). Wgsim was run in command line with parameters `-e R -N 1000000 -1 50 -2 0`, where `R` was set to 0.01, 0.02 or 0.05 depending on the scenario. The simulated reads are presented in fastq format without meaningful quality information (all bases are assigned the same quality score). The sequence headers contain the real locus and the strand of the read. After running the three mapping algorithms on the datasets the read headers are parsed and compared to the mapped locus and strand. Reads are considered well mapped if the estimate in the same strand and within 100 basepairs of the actual locus.

The algorithms were benchmarked on a dual-processor Intel Xeon E5-2687W v2 system with 256GB of DDR3-RAM at 1866 Mhz. All softwares were set to run single-core. Our mapper and `bwa-mem` were ran using default parameters, `bowtie2` was setup to use the `very-sensitive` configuration.

Our benchmark includes three measurements:

- **Sensitivity** expressed as the fraction of correctly mapped reads with respect to the total input read count.
- **Positive error rate** is the rate of false positives per true positive.
- **Throughput** expressed as the rate of true positives per second.

We believe that these three magnitudes provide a good characterization of the performance of a mapper. Almost all benchmarks evaluate the sensitivity, but it is equally important to provide

an accurate estimate of the expected number of incorrect mappings for each quality score. This will ultimately help the users decide which quality score threshold is better suited for their needs. We combine these magnitudes in two-dimensional plots to show the comparison of sensitivity versus positive error rate, and throughput versus positive error rate. The first provides valuable information to tune the acceptable quality threshold, clearly showing how many correct mappings one must disregard in order to reduce the false positive rate. The latter is a representation of how efficiently the computational power is spent.

Classification performance

The classification and mapping performence achieved by the different algorithms is represented in Figure 4.1. This result compares the sensitity versus the false positive rate achieved in the Human datasets with 1% (left), 2% (right) and 5% (bottom) error rate. Each point represents the values for a different mapping quality threshold, i.e. a threshold on mapping quality is applied on the mapping output and all the reads below the threshold are discarded. The sensitivity and false positive rates are therefore measured with the reads whose quality is above the threshold. This process is repeated for all the possible quality values, starting from 0 (the rightmost points). The general trend of the mappers is the one expected: the false positive rate decreases as the quality increases, but so does the sensitivity.

Our algorithm outperforms the `bwa-mem` and `bowtie2` in the three configurations, achieving a good response even in the worst configuration. Remarkably, our algorithm consistently reports 100.000 more well-mapped sequences with false positive rates below 10^{-4} at the standard Illumina setting (2%). Moreover, our mapper shows a smooth response on average sequencing (5% error), recovering $> 65\%$ of the reads in all the possible configurations, whereas the other mappers drop well below 50% as soon as they reach a false positive rate of 10^{-4}.

Computational efficiency

In this section we assess the computational efficiency of the algorithms. To do so, we measure the rate of true positives produced at the output for a target error read. We prefer a direct measurement of how well the computational resources are spent rather than the absolute running time, since an algorithm may run very fast but produce poor results. The measurements of computational efficiency are shown in Figure 4.1 for Human Genome reads in the two extreme cases of 1% error rate (left) and 5% error rate (right). We expect the algorithms to perform faster when the data contains few errors, because the seeding is more effective and it is more likely that a few alignments will suffice to produce an estimate. This is what we observe if we compare the average true positive rates. All the algorithms perform faster when the error rate is small. On the other side, our algorithm takes advantage of the seed significance filter to speed up the mapping process when the seed is considered highly significant, which happens more often if the read quality is good. On the other side, for 5% error, both bwa-mem and bowtie2 outperform our mapper for false positive rates above 10^{-4}. Hence, the good performance of our mapper observed

in Figure 4.2 is at the expense of a decreased computational efficiency. This lower computational efficiency at higher error rates is attributed to the time spent processing difficult reads. The difficult reads, which contain many errors, take orders of magnitude longer to map because they can only be assigned the right locus using high-sensitivity seeding strategies.

Other genomes

We have also benchmarked the mappers on a different reference: Drosophila Melanogaster. Figure 4.3 summarizes the same results shown in Figure **??** for error rates of 2% and 5%. The results show the same trend as in the Human genome, but all the algorithms show diminished sensitivity at the highest mapping qualities. This is due to the nature of the genome and generally indicates that the repeats have more copies compared to the average number in the Human genome.

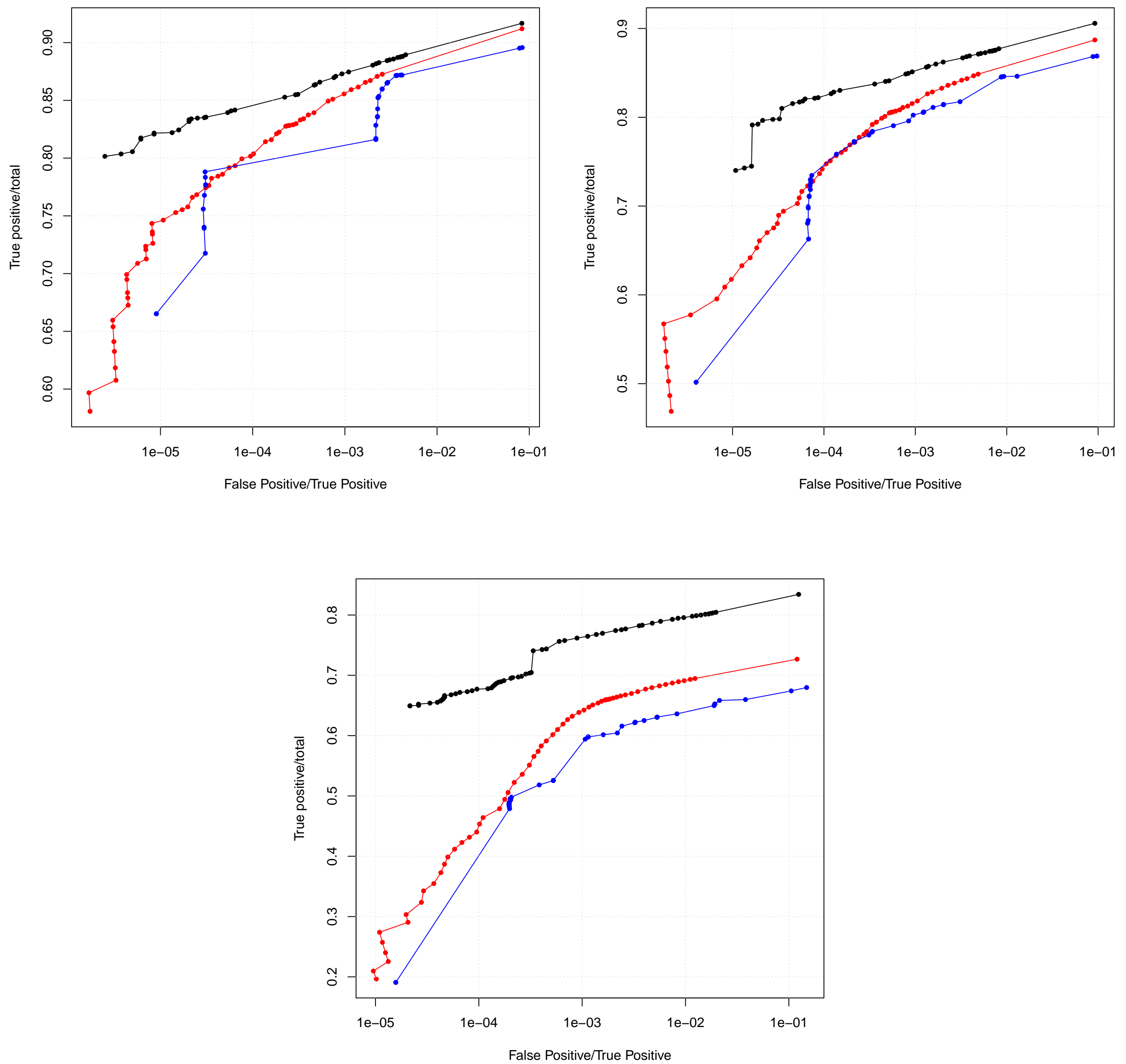

Figure 4.1: Sensitivity vs error rate for all the possible mapping quality thresholds. Illumina simulated reads on Human Genome v19, with 1% (left), 2% (right) and 5% (bottom) error rate. Comparison between bowtie2 (blue), bwa-mem (red) and our mapper (black).

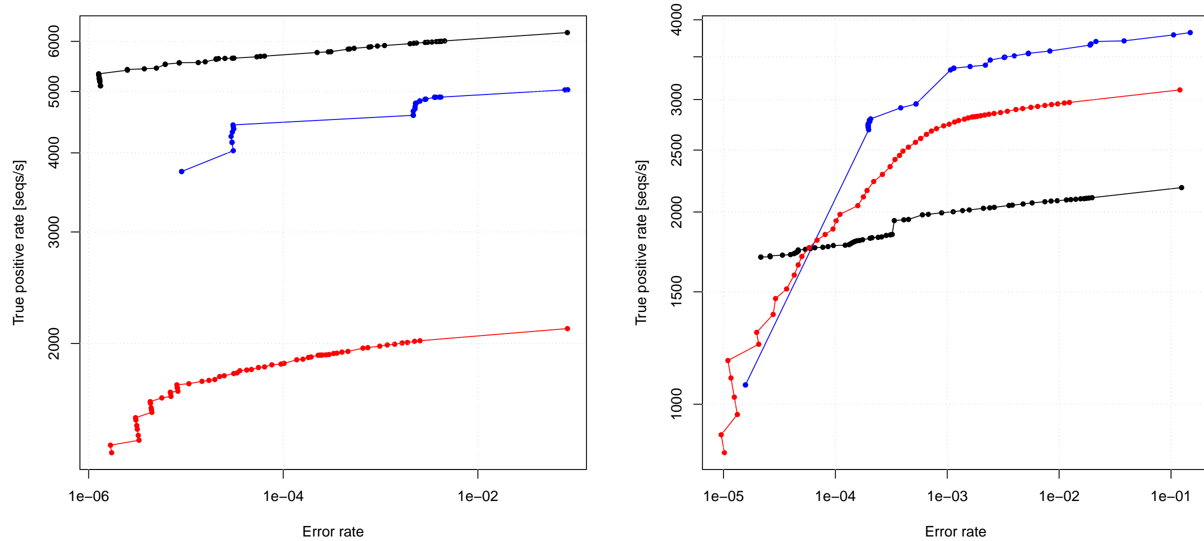

Figure 4.2: Throughput (correctly mapped sequences per core per second) vs error rate for all the possible mapping quality thresholds. Illumina simulated reads on Human Genome v19, 1% (left) and 5% (right) error rate. Comparison between bowtie2 (blue), bwa-mem (red) and our mapper (black).

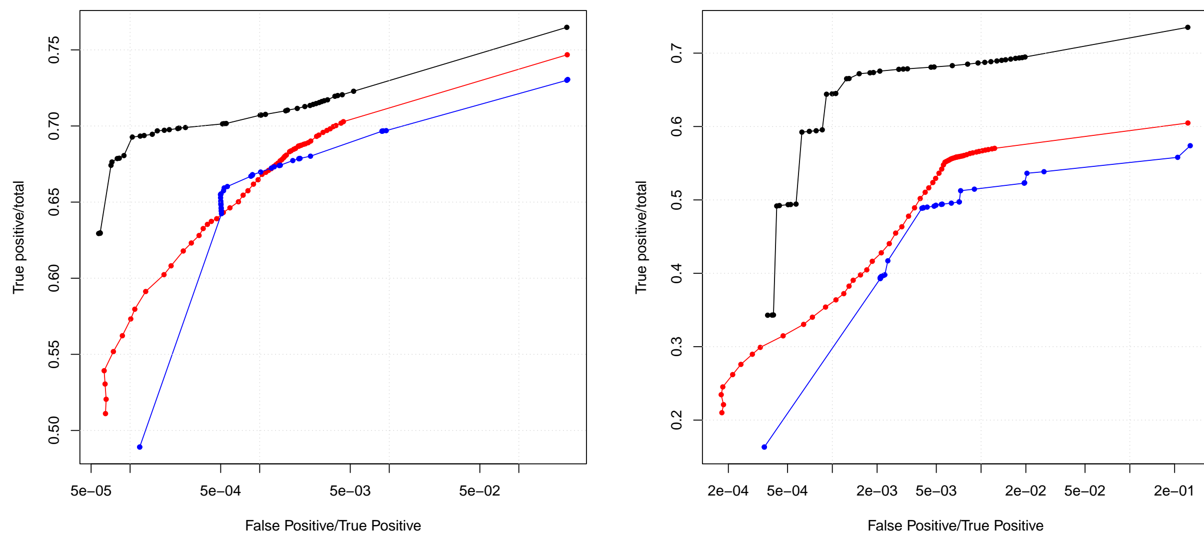

Figure 4.3: Sensitivity vs error rate for all the possible mapping quality thresholds. Illumina simulated reads on Drosophila Melanogaster genome v3, 2% (left) and 15% (right) error rate. Comparison between bowtie2 (blue), bwa-mem (red) and our mapper (black).

Chapter 5

Conclusions and future work

Overall, we have shown that even though the current mapping algorithms have good performance, there is still room for innovation. Complementing the search indices with annotations of genomic features, such as the k-mer neighborhood, can enhance the overall performance and efficiency of the algorithm. This enhancement also shows that statistical models built upon annotations, like the one presented, can be exploited to recursively evaluate the computational cost of the on-going computations and feedback this information to make efficient decisions. In this work, we have shown a simple and straightforward use of the neighborhood model. To better characterize the advantages of this annotation, we would have to evaluate its performance in a more extensive benchmark, with varying sequence lengths, error rates and other sequencing technologies. In the context of our research line, this mapper was ideated as the previous step to build a split-read mapper for Hi-C reads. Therefore, our goal was to achieve high sensitivity and improved mapping score estimations on short reads, with running times comparable to the state-of-the-art mappers. In this sense, we have clearly succeeded.

The next steps would be to improve the mapper to work reliably with the latest sequencing technologies (PacBio and Oxford Nanopore), which produce reads on the order of kilobasepairs long, but at the expense of a much higher error rate (around 15%). To do so, we would need to generalize the Neighbor model to greater neighborhood spaces, in which the presented assumptions may not hold. Other future tasks include the design of optimized-decision algorithm based on the seed significance to increase even more the computational efficiency.

Bibliography

[1] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter, *Molecular biology of the cell*, 6th ed. Garland Science, Nov 2014.

[2] J. Watson and F. Crick, "Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid," *Nature*, vol. 171, pp. 737–738, Apr. 1953.

[3] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Science*, vol. 74, pp. 5463–5467, Dec. 1977.

[4] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970. [Online]. Available: http://dx.doi.org/10.1016/0022-2836(70)90057-4

[5] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981. [Online]. Available: http://dx.doi.org/10.1016/0022-2836(81)90087-5

[6] T. D. Wu, "Bitpacking techniques for indexing genomes: I. hash tables," *Algorithms for Molecular Biology*, vol. 11, no. 1, pp. 1–13, 2016. [Online]. Available: http://dx.doi.org/10.1186/s13015-016-0069-5

[7] W. J. Kent, "Blat-the blast-like alignment tool," *Genome Res*, vol. 12, 2002. [Online]. Available: http://dx.doi.org/10.1101/gr.229202. Article published online before March 2002

[8] Z. Ning, A. J. Cox, and J. C. Mullikin, "Ssaha: a fast search method for large dna databases," *Genome Res*, vol. 11, 2001. [Online]. Available: http://dx.doi.org/10.1101/gr.194201

[9] D. Knuth, *The art of computer programming: Fundamental algorithms*, 3rd ed. Addison-Wesley, 1997.

[10] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF01206331

[11] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '90. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327. [Online]. Available: http://dl.acm.org/citation.cfm?id=320176.320218

[12] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," HP labs technical reports, Tech. Rep., May 1994. [Online]. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

[13] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, vol. 0. IEEE, 2000, pp. 390–398. [Online]. Available: http://dx.doi.org/10.1109/sfcs.2000.892127

[14] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu, "High throughput short read alignment via bi-directional BWT," in *Bioinformatics and Biomedicine, 2009. BIBM '09. IEEE International Conference on*, vol. 0. IEEE, Nov. 2009, pp. 31–36. [Online]. Available: http://dx.doi.org/10.1109/bibm.2009.42

[15] D. Lipman and W. Pearson, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985. [Online]. Available: http://science.sciencemag.org/content/227/4693/1435

[16] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022283605803602

[17] V. Jackson, "Studies on histone organization in the nucleosome using formaldehyde as a reversible cross-linking agent," *Cell*, vol. 15, no. 3, pp. 945–954, 2016/09/14 XXXX. [Online]. Available: http://dx.doi.org/10.1016/0092-8674(78)90278-7

[18] J. Dekker, K. Rippe, M. Dekker, and N. Kleckner, "Capturing chromosome conformation," *Science*, vol. 295, no. 5558, pp. 1306–1311, 2002. [Online]. Available: http://science.sciencemag.org/content/295/5558/1306

[19] D. S. Johnson, A. Mortazavi, R. M. Myers, and B. Wold, "Genome-wide mapping of in vivo protein-dna interactions," *Science*, vol. 316, no. 5830, pp. 1497–1502, 2007. [Online]. Available: http://science.sciencemag.org/content/316/5830/1497

[20] E. Lieberman-Aiden, N. L. van Berkum, L. Williams, M. Imakaev, T. Ragoczy, A. Telling, I. Amit, B. R. Lajoie, P. J. Sabo, M. O. Dorschner, R. Sandstrom, B. Bernstein, M. A. Bender, M. Groudine, A. Gnirke, J. Stamatoyannopoulos, L. A. Mirny, E. S. Lander, and J. Dekker, "Comprehensive mapping of long-range interactions reveals folding principles of the human genome," *Science*, vol. 326, no. 5950, pp. 289–293, 2009. [Online]. Available: http://science.sciencemag.org/content/326/5950/289

[21] M. Farrar, “Striped SmithWaterman speeds database searches six times over other SIMD implementations,” *Bioinformatics*, vol. 23, no. 2, pp. 156–161, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btl582

[22] T. Rognes, “Faster smith-waterman database searches with inter-sequence simd parallelisation,” *BMC Bioinformatics*, vol. 12, no. 1, pp. 1–11, 2011. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-12-221

[23] S. A. Manavski and G. Valle, “Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment,” *BMC Bioinformatics*, vol. 9, no. Suppl 2, pp. S10–S10, Mar 2008, 1471-2105-9-S2-S10[PII]. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2323659/

[24] D. V. Hinkley and E. A. Hinkley, “Inference about the change-point in a sequence of binomial variables,” *Biometrika*, vol. 57, no. 3, pp. 477–488, 1970. [Online]. Available: http://biomet.oxfordjournals.org/content/57/3/477.abstract

[25] H. Li, “Aligning sequence reads, clone sequences and assembly contigs with bwa-mem.” *arXiv:1303.3997v2*, 2013. [Online]. Available: http://arxiv.org/abs/1303.3997

[26] B. Langmead and S. L. Salzberg, “Fast gapped-read alignment with bowtie 2,” *Nat Meth*, vol. 9, no. 4, pp. 357–359, Apr 2012, brief Communication. [Online]. Available: http://dx.doi.org/10.1038/nmeth.1923