Ghent University
Faculty of Sciences
Department of Applied Mathematics, Computer Science and Statistics

ALFALFA
Fast and Accurate Mapping of Long Next Generation Sequencing Reads

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Michaël Vyverman

September 2014

Supervisors: prof. dr. Peter Dawyndt
prof. dr. Bernard De Baets
prof. dr. Veerle Fack

Voor mijn ouders

Dankwoord

Deze thesis vormt het verslag van mijn reis doorheen het wetenschappelijk onderzoek die iets meer dan vier jaar geleden is gestart. Aan een doctoraat vertrek je op brede, vertrouwde en veel gebruikte wegen, maar al snel ontdek je nieuwe oorden. Ik ben het gebied van de bioinformatica binnengetrokken en heb exotische plaatsen verkend zoals sequeneringstechnologie, indexstructuren en *read mapping* algoritmen. Doorheen de tocht heb ik steeds smallere wegen moeten navigeren, waarbij ik de hulp kreeg van voorgaande verslagen en lokale gidsen. Zelf heb ik ook nieuwe gebieden in kaart gebracht die nu in dit werk staan beschreven, en in de toekomst misschien andere reizigers naar nieuwe oorden leiden.

De voorbije jaren vormden een fascinerende reis waarin ik vele interessante en boeiende ervaringen heb opgedaan. Toch zou dit verslag niet tot stand gekomen zijn zonder de steun en hulp van vele medereizigers, ervaren gidsen, en mijn familie en vrienden thuis.

Eerst en vooral wil ik mijn promotoren, Peter, Bernard en Veerle bedanken. Dankzij hen heb ik deze reis kunnen aanvangen en ben ik niet onderweg verdwaald. Ik wil Peter vooral bedanken voor alle hulp, zowel bij het onderzoek als bij het schrijven, en om mij te laten inzien dat je soms uitdagingen moet durven aangaan. Veerle wil ik vooral bedanken voor de wekelijkse afspraken die we hadden, waarbij ik alles eens op een rijtje kon zetten. Alhoewel we minder persoonlijk contact hebben gehad, zorgde Bernard altijd voor een nieuwe, heldere en kritische blik.

Dit reisverslag werd grondig nagelezen door de leden van mijn jury. Ik wil ze bedanken voor hun suggesties en opmerkingen die hebben geholpen om de inhoud van dit werk nog te verbeteren.

Ik draag deze thesis op aan mijn ouders. Zij hebben me altijd gesteund en gemotiveerd. Ook tijdens deze reis hebben ze me op alle mogelijke manieren geholpen. Zo heeft mijn vader bijvoorbeeld grafische ondersteuning verleend aan

enkele afbeeldingen die dit werk sieren.

Ik wil eveneens mijn familie en vrienden bedanken. Ze hebben steeds met veel interesse geluisterd naar mijn verhalen en nieuwe ontdekkingen, maar evenzeer naar mijn frustraties als er obstakels op de weg lagen.

Tijdens mijn expedities naar conferenties en buitenlandse verblijven heb ik heel wat mensen leren kennen waarmee ik boeiende gesprekken met heb gehad. *In particular, I would like to thank Veli Mäkinen and the members of his group for the very nice and interesting stay I had at the University of Helsinki.*

Aangezien het doorkruisen van bioinformatica-gebied vele uiteenlopende expertises vergt, ben ik blij dat ik deel uitmaakte van het multidisciplinaire onderzoeksplatform *Nucleotides to Networks* (N2N). Binnen dit kader wil ik vooral Yao-Chen Lin en Lieven Sterck bedanken voor hun expertise en hun feedback.

Verder kon ik altijd rekenen op de ervaren reizigers binnen de vakgroep Toegepaste Wiskunde, Informatica en Statistiek (TWIST). In het bijzonder wil ik mijn bureaugenoten Glad en Nico bedanken, die me niet alleen met hun ervaringen geholpen hebben, maar ook graag hun technische kennis met mij deelden, zodat ik nooit lang met pech onderweg bleef stilstaan.

Gelukkig was er tijd om onderweg regelmatig even te stoppen voor ontspanning en andere activiteiten. Zo waren er de lunches met de leden van de onderzoeksgroep Combinatorische Algoritmen en Algoritmische Grafentheorie (CAAGT), waar ik elke week naar uitkeek. Ook de regelmatige gesprekken met Jan zal ik zeker missen.

Met mijn medereizigers binnen de vakgroep TWIST heb ik heel wat leuke momenten beleefd, waaronder spelletjes- en filmavonden, en zelfs een avond rond het kampvuur op één van de TWI(ST)kends. Zelfs letterlijk onderweg, namelijk op de trein, heb ik nieuwe mensen leren kennen. Zij hebben ervoor gezorgd dat de lange treinrit Gent-Ressegem net iets korter leek. Ben en Jeroen, ik hoop dat we elkaar nu niet uit het oog verliezen, en we nog vele leuke spelletjesavonden beleven.

De nodige brandstof voor mijn reis werd voorzien door het agentschap voor Innovatie door Wetenschap en Technologie (IWT). Ik wil hen dan ook bedanken voor deze kans die zij mij geboden hebben. Een andere soort brandstof voor mijn reis werd geleverd in de vorm van computationele kracht van de STEVIN supercomputer van de Universiteit Gent en de goede technische ondersteuning geleverd door het HPC-team.
Aan allen veel dank!

Michaël Vyverman, december 2014

Contents

Summary

Bioinformatics research is currently dominated by the (r)evolution in sequencing technology that produces short fragments, called reads, containing the precise order of the bases in that sequence. New sequencing platforms produce biological sequence fragments faster and cheaper than ever before. The resulting growth in access to large amounts of data opens perspectives for new applications and prestigious projects, but simultaneously pushes existing sequence analysis tools beyond their limits as data storage, computational analysis and interpretation become true bottlenecks in life science research.

Mapping reads to reference genomes plays a key role in many genomics analysis pipelines. It consists of finding positions where sequencing reads best fit a reference sequence and identifying differences and similarities between the read sequence and region of the reference genome onto which the read is mapped. The Olympic motto *citius*, *altius*, *fortius* in the context of this computationally intensive problem drives read mappers into the algorithmically challenging quest to find an optimal balance between maximal speed, minimal memory footprint and maximal accuracy. Read mappers are also expected to shoot at a moving target, as reads produced by fast evolving technologies differ in length distribution and sequencing errors. Most of the current read mappers target short reads and allow for no or low numbers of mismatches and/or indels. This makes them vulnerable to the ongoing technological advances that feature increased read lengths, higher error rates and error models showing more and longer indels.

In this dissertation we develop advanced algorithms and index structures for fast and accurate mapping of long next generation sequencing reads. The developed read mappers rely on a combination of efficient index structures, search algorithms and a multitude of heuristics. The outline of this dissertation reflects the importance of each of those individual components, of which several are of independent interest to other sequence analysis problems.

The **first chapter** introduces basic notations and concepts that are used throughout this work, many of which will be familiar to readers with a background in bioinformatics. Most notations revolve around the concepts of strings, biological sequences and the wide variety of types of common substrings.

This chapter also provides background information on sequencing technology, including a historical overview of the technology and a comparison of the features of the currently available platforms. In addition, the chapter introduces key concepts related to the sequencing read data type, such as quality values and paired-end methodologies, that will affect the choices made in the development of our read mapping algorithms.

Similar to the major data type, this chapter provides background information on the algorithmic challenge tackled in this dissertation. In addition to basic definitions and notations concerning mapping and alignment of sequences, a taxonomy of read mappers is given. This section illustrates the diversity of the currently available read mappers and provides a basis for the selection of tools used in the evaluation of our algorithms. The theoretical and practical procedures for evaluating our methods are also discussed here.

Finally, the introductory chapter covers dynamic programming algorithms for aligning strings. These basic algorithms have become common knowledge in the field, but are still important subroutines in read mapping algorithms.

The **second chapter** contains a comprehensive review of full-text index structures, which are specialized data structures designed to speed up string searching. Index structures are widely used in life sciences research, as most applications in the field require basic string operations, most notably search operations. Read mappers in particular rely on advanced index structures to quickly identify short matching substrings between the reference genome and read that are used to prune the search space. The features of these substrings are usually related to the choice of index structure. Furthermore, memory requirements of read mappers are directly linked to the size of the index structure the employ.

Although the importance of index structures is generally known to the bioinformatics community, the design and potency of these data structures, as well as their properties and limitations, are less understood. Moreover, the last decade has seen a boom in the number of variant index structures featuring complex and diverse memory-time trade-offs. This chapter brings a comprehensive state-of-the-art overview of the most popular index structures and their recently developed variants. Their features, interrelationships, the trade-offs they impose, but also their practical limitations, are explained and compared.

In the **third chapter** we present a novel index structure, called an enhanced sparse suffix array. This index structure is based on suffix arrays, which encode the lexicographical ordering of the suffixes of a string. In addition to indexing only a sparse set of suffixes, it enhances suffix arrays with additional information about longest common prefixes of suffixes to boost performance of string searches.

In addition to the novel index structure, we present an algorithm for finding maximal exact matches between two sequences. Maximal exact matches are exact matches between two sequences that cannot be extended to the left or right without introducing a mismatch. They are widely used in genome comparison tools as anchor points for alignment, but can also be used as seeds for alignment of reads, especially for alignment of very long reads.

The algorithm for finding maximal exact matches using enhanced sparse suffix arrays is implemented and in a tool, called *essaMEM*. We show that *essaMEM* outperforms other commonly used tools for finding maximal exact matches, including *sparseMEM* [126], a tool that uses sparse suffix arrays. Furthermore, *essaMEM* is competitive with *backwardMEM* [201], which utilizes compressed suffix arrays. This final result indicates that, although compressed index structures have recently become very popular, the use of sparse index structures can be a viable option for further research.

In the **fourth chapter** we present a novel read mapper called *ALFALFA*. The read mapper *ALFALFA* is specifically designed to achieve high performance in accurately mapping long DNA reads that possibly contain numerous errors. Its implementation of the canonical seed-and-extend approach is empowered by the index structures and algorithms from Chapter 3. Enhanced sparse suffix arrays are used to find small exact matches between read and reference genome, called seeds. The techniques and heuristics used to filter and combine seeds and candidate mapping regions are designed to handle longer reads. Extension of candidate regions into alignments are handled by fast chain-guided dynamic programming routines, which are introduced in Chapter 1.

The performance, memory footprint and accuracy achieved by *ALFALFA* are evaluated and compared against other state-of-the-art read mappers. The benchmark includes a large variety of data sets, accuracy measures and read mapper configurations. Results clearly show that *ALFALFA* is one of the few read mappers that are able to cope with very long reads (> 1000bp). *ALFALFA* is extremely fast and accurate at mapping long reads (> 500bp), while still being competitive for moderately sized reads (> 100bp). Furthermore, flexible parameter tuning allows balancing performance, memory footprint and accuracy.

In the **fifth chapter** we extend the previous algorithms to the mapping and aligning of cDNA sequences and RNA-seq reads, which are used to study the RNA content of a cell. In comparison to DNA sequence mapping, alignment of cDNA sequences and RNA-seq reads to a eukaryotic reference genome poses additional algorithmic challenges due to the presence of large gaps in the alignment. These gaps are caused by removing (splicing) intronic segments from a transcribed sequence, and joining together the remaining segments (exons). Because introns are usually much larger than the deletions detected by traditional alignment algorithms, standard DNA read mappers fail to align reads spanning multiple exons.

We present *mesalina*, a prototype of a spliced alignment algorithm based on *essaMEM* and *ALFALFA*. The algorithm combines *ALFALFA* with powerful dynamic programming algorithms introduced by *GMAP* [258]. Preliminary results indicate that *mesalina* is competitive in terms of accuracy and has a high performance that is more robust with respect to increasing read length.

All aforementioned tools, algorithms and data structures are publicly available for download as `C++` source code at `https://github.com/readmapping`.

Chapter 1

Introduction

This chapter introduces basic notations and concepts that are used throughout this work, many of which will be familiar to readers with a background in bioinformatics. Section 1.1 introduces notations for the core concepts of strings, sequences and common substrings. Index structures, being more complex data structures, are introduced in Chapter 2. The need for the index structures and algorithms presented in this work is motivated by the constant evolution in sequencing technology. Section 1.2 provides background information on sequencing technology and introduces sequencing reads as the main input data type of our algorithms. Definitions and notations concerning mapping and alignment of strings are given in Section 1.3. Dynamic programming algorithms for aligning strings are covered in Section 1.4. These algorithms have become common knowledge in the field, but are still important subroutines in read mapping algorithms. Finally, Section 1.5 provides an overview of theoretical and practical evaluation procedures of the index structures and algorithms introduced in this dissertation.

1.1 Notations

This section contains notations for basic concepts that are used throughout this dissertation. Due to the interdisciplinary nature of the field of bioinformatics, some concepts might have a different definition than the one is commonly employed by some readers. We will discuss these differences and describe some properties of the biological sequences that are the subject of research in this work.

1.1.1 Basic notations

Let A be an array, list, set or string. The *size* of A is denoted by $|A|$.

For an ordered data structure A, the element at position or *index* i is given by $A[i]$. By convention, all indexes in this dissertation are zero-based and the first element can thus be found at position 0. If an array has more than one dimension, the comma is used as a separator, *e.g.* $A[i, j]$ denotes the element of the two-dimensional array A at the i-th row and j-th column. An *interval* in A from index i to (and including) index j, with $i \leq j$, is represented by $A[i..j]$.

Let the finite, totally ordered *alphabet* Σ be an array of size $|\Sigma|$. Furthermore, let Σ^k be the set of all *strings* composed of k characters from Σ. Such a string is also sometimes called a *k-mer*. The set of all strings composed of zero or more characters from Σ is denoted by Σ^*, with the *empty string* represented by ϵ.

Let $S \in \Sigma^n$ be a string of size n (S stands for string or sequence). Similar to arrays, for every $0 \leq i \leq j < n$, $S[i]$ denotes the character at position i in S, the interval $S[i..j]$ denotes a substring that starts at position i and ends at position j and $S[i..j] = \epsilon$ for $i > j$. A string P (for pattern) of length m is a substring of S if there exists an index $0 \leq i \leq n - m$ such that $S[i..i + m - 1] = P$. The entire set of positions for which this equality holds is denoted by $occ(P, S)$.

$S[i..]$ is the i-th *suffix* of S and is equal to the substring $S[i..n - 1]$. Likewise, the i-th *prefix* of S is denoted by $S[..i]$ and corresponds to the interval $S[0..i]$. The omission of the positions $n - 1$ and 0 focuses the attention on the fact that the intervals are suffixes or prefixes, which have specific properties in the data structures that are used in this dissertation. To ease notation, we state by convention that $S[-1..] = S[..n] = \epsilon$.

The *lexicographical order relation* between two elements S and P of Σ^* is represented as $S < P$.

As a final remark, note that all *logarithms* in this paper have base two, unless stated otherwise.

1.1.2 Common substrings

Many classical problems in string analysis and biological sequence analysis revolve around the identification of repeated elements within a string or common elements between two or more strings. In this section, we introduce several types of common substrings that are used throughout this dissertation.

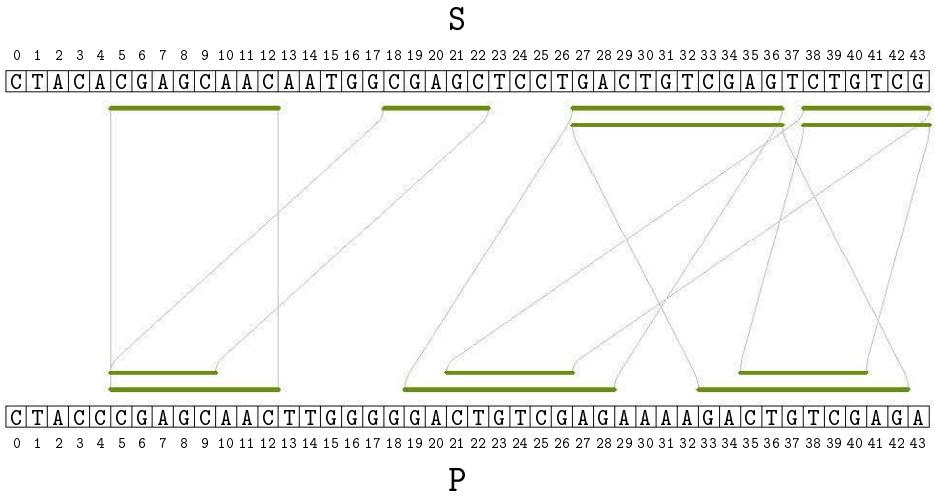


Figure 1.1: All maximal exact matches with minimal length five between two strings. Dark green lines represent pairs of intervals that are maximal exact matches.

Let S and P be two strings of respective sizes $|S| = n$ and $|P| = m$. A *common substring* C of size $\ell > 0$ between S and P is a string that appears as a substring in both S and P. Every pair of positions $i \in occ(C, S)$ and $j \in occ(C, P)$ defines a *match* of size ℓ between those strings, which is denoted by (i, j, ℓ). Note that a common substring refers to the string itself, whereas a match refers to a pair of intervals. For example, the common substring `CGAGC` corresponds to two matching interval pairs $(5, 5, 5)$ and $(18, 5, 5)$ in the example strings in Figure 1.1.

A substring C of size $\ell > 0$ that appears more than once in S is called a *repeat*. Similar to the definition of matches, *repeated pairs* of size ℓ can be defined as every pair of positions $i, j \in occ(C, S)$ and $i < j$. Repeated pairs are also denoted by (i, j, ℓ).

The set of all matches between two strings can be very large and does not discriminate enough to provide useful information. In practice, only matches are used that meet certain criteria. A widely used criterion is a restriction on the size of the common substring. For example, some applications search for matches with a fixed length k, called k-mers, or search for matches with a minimum length ℓ.

The *longest common substring* between two strings, for example, can be used to quickly identify possible contaminant substrings in a DNA sequence [97].

The longest common substring of the two example strings in Figure 1.1 is found at $(27, 33, 10)$ and equals the string `GACTGTCGAG`. Important in this work is the definition of the *longest common prefix* of two strings.

Definition 1.1. *The longest common prefix $LCP(S, P)$ of two strings S and P is the prefix $S[..k]$, such that $S[..k] = P[..k]$ and $S[k+1] \neq P[k+1]$.*

The longest common prefix of the example strings in Figure 1.1 is `CTAC`, which corresponds to the match $(0, 0, 4)$. The longest common prefix between two strings marks the point at which the strings start to differ. Although this concept seems of limited use at first, it is often used in indexing (see Chapter 2).

Another frequently used type of matches between two strings restricts the results to matches that are locally as large as possible by extending the match until mismatches are encountered.

Definition 1.2. *A triplet (i, j, ℓ) is called a maximal exact match (MEM) between two strings S and P of size $|S| = n$ and $|P| = m$ if:*

i) $S[i..i+\ell-1] = P[j..j+\ell-1]$

ii) $i = 0$, $j = 0$ *or* $S[i-1] \neq P[j-1]$

iii) $i+\ell = n$, $j+\ell = m$ *or* $S[i+\ell] \neq P[j+\ell]$

The second and third conditions are respectively called the left-maximality and right-maximality conditions. If only the first two conditions hold, the pair is a left-maximal exact match. Likewise if only the first and third condition hold, the pair is called a right-maximal exact match.

In Figure 1.1, all MEMs of minimum length five are shown in green. An example of a left-maximal exact match is the pair $(5, 5, 6)$. It is left-maximal because $S[4] = \texttt{A} \neq \texttt{C} = P[4]$, but not right-maximal because $S[11] = \texttt{A} = \texttt{A} = P[11]$.

A repeated pair for which the conditions in Definition 1.2 hold, is called a *maximal repeated pair*. Formally, a substring of length $\ell > 0$ that occurs at least at two positions i, j $(i < j)$ in S and that is both left-maximal ($i = 0$ or $S[i-1] \neq S[j-1]$) and right-maximal ($j+\ell = |S|$ or $S[i+\ell] \neq S[j+\ell]$) is called a *maximal repeat*; the pair (i, j, ℓ) is a maximal repeated pair.

Note that although there are fewer MEMs than the total number of matching intervals, the number of MEMs can still be very high. For example, in Figure 1.1 there are 273 MEMs of size one and 64 MEMs of size two. In practice, only MEMs of a certain minimum size are used.

In addition to a minimal length, extra cardinality constraints can be imposed on MEMs. A *maximal unique match (MUM)* is a maximal exact match for which the matching substring occurs only once in both strings. The only MUM of minimum length five in Figure 1.1 is the pair $(5, 5, 8)$. The MEM $(18, 5, 5)$ is not a MUM because the corresponding substring `CGAGC` occurs both on positions 5 and 18 in S.

A somewhat less strict definition is that of an *almost-unique maximal exact match (MAM)*. A MEM is called almost-unique if it occurs uniquely in one of the two strings, but not necessarily both. In practice, the concept of MAM is non-commutative. Thus it is possible that a MEM is a MAM between S and P, but not between P and S. From now on, we will impose that the uniqueness condition of a MAM must hold for string S. In Figure 1.1, examples of MAMs that are not MUMs are the intervals $(27, 19, 10)$ and $(27, 33, 10)$, as the corresponding substring `GACTGTCGAG` occurs only once in S. The MEM $(38, 21, 6)$ is not a MAM because `CTGTCG` appears at positions 29 and 38 in S and at positions 21 and 35 in P.

The concepts of MUMs and MAMs can further be generalized to MEMs with certain cardinality constraints. For example, it is possible to limit the set of MEMs to MEMs (i, j, ℓ) between S and P, with corresponding substring $C = S[i..i + \ell - 1] = P[j..j + \ell - 1]$, for which $occ(C, S) < a$ and $occ(C, P) < b$, for given a and b.

SMEMs [147] are another type of MEMs designed to define subsets of MEMs that are smaller, but contain the most "interesting" MEMs.

Definition 1.3. *A pair of intervals (i, j, ℓ) is called a super-maximal exact match or SMEM between two strings S and P if (i, j, ℓ) is a MEM that is not contained in another MEM in P, i.e. there does not exist another MEM (i', j', ℓ') between S and P such that $[j..j + \ell - 1]$ is a subinterval of $[j'..j' + \ell' - 1]$ and $\ell < \ell'$.*

Similar to MAMs, the definition of SMEMs is not commutative. In light of applications of MEMs in this work, the MAM uniqueness condition holds for string S, but the containment condition of SMEMs needs to hold for string P. Examples of MEMs that are not SMEMs in Figure 1.1 are the interval pairs $(18, 5, 8)$, $(38, 21, 6)$ and $(38, 35, 6)$. It is interesting to note that MAMs are always SMEMs, but the opposite is not always the case.

Lemma 1.1. *The set of MAMs between two strings S and P, where the uniqueness condition needs to hold for string S, is a strict subset of the set of SMEMs between those strings.*

Proof. If (i, j, ℓ) is a MAM between S and P, the substring $S[i..i+\ell-1]$ appears only once in S. Now suppose that (i, j, ℓ) is not an SMEM. By definition, there exists another MEM (i', j', ℓ') between S and P such that $[j..j + \ell - 1]$ is a subinterval of $[j'..j' + \ell' - 1]$ and $\ell < \ell'$. However, the substring $S[i..i + \ell - 1]$ would then appear twice in S, because $S[i'..i' + \ell' - 1] \neq S[i..i + \ell' - 1]$. This contradicts the fact that (i, j, ℓ) is a MAM. Thus, (i, j, ℓ) is also a SMEM.

On the other hand, it is clear that not all SMEMs are necessarily MAMs. A counterexample can be found in Figure 1.1. The SMEM $(1, 1, 1)$ between $S = \texttt{CACAC}$ and $P = \texttt{TAT}$ is not a MAM, because the corresponding substring `A` appears twice in S, at positions 1 and 3. □

1.1.3 Biological sequences

In the previous sections, we introduced concepts dealing with strings in the context of computer science. In biology, however, the term *sequence* is used instead of string. The term sequence is used for different concepts in the field of computer science and biology. What is called a sequence in biology is usually a string in standard computer science parlance. The distinction between strings and sequences becomes especially prominent in computer science when introducing the concepts of substrings and subsequences. The former refer to contiguous intervals from larger strings, whereas the latter do not necessarily need to be contiguous intervals from the original string. As index structures work with substrings and to avoid ambiguity, we will stick to the standard computer science term string throughout this dissertation, unless we explicitly want to stress the biological origin of the sequence.

The group of biological sequences includes several types of sequences that use different alphabets, including DNA, RNA and the amino acid alphabet for proteins. The algorithms in this dissertation were designed for handling sequences using the DNA-alphabet, which has size four and is given by

$$\Sigma_{\text{DNA}} = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}.$$

The four letters of the alphabet represent the four *nucleotides* or *bases* that make up a DNA sequence: `A` stands for *adenine*, `C` stands for *cytosine*, `G` for *guanine* and `T` for *thymine*. The RNA alphabet is given by

$$\Sigma_{\text{RNA}} = \{\texttt{A},\texttt{C},\texttt{G},\texttt{U}\},$$

which is the same as the DNA alphabet, except for thymine, which is replaced by *uracil* (`U`).

The amino acid alphabet has size 20 and is given by

$$\Sigma_{\text{AA}} = \{\texttt{A},\texttt{R},\texttt{N},\texttt{D},\texttt{C},\texttt{E},\texttt{Q},\texttt{G},\texttt{H},\texttt{I},\texttt{L},\texttt{K},\texttt{M},\texttt{F},\texttt{P},\texttt{S},\texttt{T},\texttt{W},\texttt{Y},\texttt{V}\}.$$

Sometimes the *IUPAC nucleotide code*[1] is used as an extension of the standard DNA alphabet. It contains special characters that express uncertainty about the real base present at that position. For instance, the letter `R` can either be an `A` or `G` in the DNA alphabet. We will use an alphabet that is a subset of the IUPAC code of size five, namely the DNA5 alphabet, given by

$$\Sigma_{\text{DNA5}} = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T},\texttt{N}\}.$$

Here, the letter `N` stands for an unknown character which could be any base, and which is usually treated as a mismatch or wildcard.

In Σ_{DNA}, cytosine (`C`) pairs with guanine (`G`) and adenine (`A`) pairs with thymine (`T`) (in RNA adenine pairs with uracil (`U`)). This pairing comes from the complementary *strands* of genomic sequences in which the *forward* strand is read in the direction from the $5'$ to the $3'$ end and the *reverse complement* strand is read in the reverse direction. For example, the reverse complement of the forward sequence `ATGC` is `GCAT`.

Definition 1.4. *If S is a DNA string of size n, its reverse complement is represented by $\bar{S}$ and is given by $\overline{S[i]} = \overline{S[n-i-1]}, 0 \leq i < n$, where $\bar{\texttt{A}} = \texttt{T}$, $\bar{\texttt{T}} = \texttt{A}$, $\bar{\texttt{C}} = \texttt{G}$, $\bar{\texttt{G}} = \texttt{C}$ (and $\bar{\texttt{N}} = \texttt{N}$ in the DNA5 alphabet).*

Characters in DNA or RNA sequences are also referred to as *nucleotides* (nt) or *base pairs* (bp). The size of a sequence is often described in terms of the number of base pairs (bp), thousands of base pairs: kilobases (kbp), millions of base pairs: megabases (Mbp) or billions of base pairs: gigabases (Gbp).

In addition to the small alphabet, biological sequences have certain features that need to be taken into account. For example, they usually contain a lot of repeated patterns, which form obstacles for alignment and *de novo* assembly of short sequences. In addition to repeats, DNA sequences contain special patterns that partition long sequences in logical parts, such as start and stop codons that mark the beginning of genes, and splice sites that mark the boundary between introns and exons. Furthermore, the size of biological sequences, such as complete genomes, ranges from millions to billions of nucleotides and, in contrast to natural language texts, cannot be divided into small parts, such as words, sentences and paragraphs.

[1] `http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html`(last accessed September 2014)

1.2 Sequencing technology

The analysis of DNA samples first requires the determination of the precise order of bases in that sequence. Methods that perform this "reading" of sequences are collectively called *sequencing technology* platforms. These platforms do not, however, produce complete genomes or chromosomes. Instead, they produce sequence fragments, called *reads*, whose sizes range from a few tens of bases to several thousands of bases and which may contain a small number of erroneous bases. As such, an important part of the analysis of sequencing reads consists of the correction of errors and to determine the origin of the small fragments in the genome, either by mapping them onto a reference genome, if available, or *de novo* assembly of the reads. Therefore, the demand for new bioinformatics solutions is closely related to evolutions in sequencing technology.

In this section, we provide an overview of evolutions in sequencing technology, covering major technologies and marking several milestone events. We will not, however, go into the biological or technical details. Instead, we introduce the generic properties of sequencing reads that need to be taken into account for the development and evaluation of our algorithms. Finally, we give an overview of the major sequencing technologies currently used based on these properties. More in-depth information on (evolution in) sequencing technology can be found in several reviews [48, 161, 166, 184, 236].

1.2.1 Historical overview

The method dominating the first generation of sequencing technology was the chain-termination sequencing method of Frederick Sanger [223]. Early sequencing efforts using this technology were slow and cumbersome and mainly focused on determining the sequence of genes and micro-organisms such as viruses, bacteria and fungi. Over the course of three decades several enhancements were made to Sanger sequencing, including the automation and parallelization of the process. The Human Genome Project [116,244] further boosted the development of Sanger sequencing. Despite of its high cost and low throughput, Sanger sequencing is still widely used today due to its long read length and high quality reads.

By the mid-2000s, a collection of several new technologies appeared that revolutionized the field of sequencing and were collectively called *next generation sequencing* technologies. The main feature these technologies have in common is their ability to massively parallelize the sequencing process. The decrease in cost and increase of throughput of sequencing ranged in several orders of magnitude.

Table 1.1: Overview of major sequencing technology platforms, mostly adapted from [161, 236]. The sequencing technologies are roughly divided by generation. The numbers in this table are often approximations and can change rapidly due to constant evolution and improvements in the technology. The values that are not available (N.A.) are either due to company restrictions or technologies still under development.

	major error modality	accuracy	read length (bp)	sample throughput
Sanger sequencing	substitutions	99.9%	300 to 900	low
454	indels	99.9%	700	medium
Illumina/Solexa	substitutions	98%	50 to 300	high
SOLiD	substitutions	99.9%	$50 + 35$ or $50 + 50$	high
Complete Genomics	N.A.	N.A.	35	high
Ion Torrent	indels	98%	200 to 400	medium
Pacific Biosciences	indels	87%	> 2000	medium
Nanopore sequencing	N.A.	N.A.	N.A.	N.A.

For example, the Human Genome Project took almost ten years to complete at the cost of approximately $3 billion. This amount has dropped by several orders of magnitude, towards the symbolic boundary of $1000 [99].

The major technologies of this generation include the 454, Illumina/Solexa, SOLiD, Complete Genomics and Ion Torrent platforms. Methods employed by these platforms vary in both sequencing principle, sample preparation and detection principle. More important for analysis of the produced reads, however, the output of these platforms also has varying properties, such as read length and type and number of sequencing errors. A summary of the different features of these technologies, including advantages and disadvantages can be found in Table 1.1 and Section 1.2.3.

Higher throughput and lower cost of next generation sequencing allowed projects at a much larger scale than ever before [1, 139] and gave rise to new applications. Apart from standard DNA sequencing (*DNA-seq*), next generation sequencing also gave rise to sequencing of the presence of RNA in cells using *RNA-seq* [252], sequencing methods for functional analysis and analysis of interactions [256] and sequencing entire environments [237] using *metagenomics*.

Since 2008, another generation of sequencing technology has emerged, currently referred to as *third generation sequencing*. Where second generation sequencing introduced massively parallel consensus sequencing, the third generation features a technique called *single-molecule sequencing*. This technique eliminates the need of an amplification step in the procedure, which is known to cause biases, and allows real quantitative DNA or RNA to be obtained. Major technologies in this generation include platforms by Pacific BioSciences and Oxford Nanopore. Another advantage of some of these platforms is the length of the reads they produce, which is much larger than that of their second generation counterparts (see Table 1.1).

The increase in amount of data has caused bottlenecks in other parts of the research pipeline, pushing existing sequence analysis tools beyond their limits as data storage [52, 235], computational analysis and interpretation become true bottlenecks in life sciences research. Tools have to adopt to the new scale of data, as well as changes in read length, error models, *etc.* The read mapping algorithms presented in this work were designed to contribute to alleviating part of this analysis bottleneck.

1.2.2 Sequencing reads

The constant evolution in sequencing and a multitude of available sequencing platforms have led to a wide variety of sequencing reads with different features. The choice of sequencing platform depends on many factors, including cost, speed, throughput and chosen applications. For the sequence analysis algorithms in this work, however, we focus only on those features of sequencing and the available sequencing platforms that have a direct impact on the performance of the algorithms. These features are: *i*) the size of the data sets, *ii*) length of the reads, *iii*) types and number of sequencing errors and *iv*) special types of sequencing libraries, such as paired-end reads.

The size of read data sets is governed by the size of the sequenced fragment and the *coverage*, which is the average number of reads representing a given nucleotide in the fragment. Depending on application, technology (which dictates read length) and other factors, coverage can rise up to 50, resulting in very large data sets. Reads in the data set also originate from both forward and reverse complemented strand of the fragment and contain very similar sequences originating from homologous chromosomes in polyploid organisms.

Read length varies depending on the used technology from as few as 35bp to approximately 10 000bp with some technologies, with even longer reads expected

in the near future. The majority of sequencing reads currently produced range from 100bp to about 600bp with given technology.

As with most technology, sequencing technology is not infallible and occasionally produces reads containing errors. Types of errors include *substitutions* of a single base, called *mutations*, and *insertion* or *deletion* of one or more successive bases. The latter type or errors are collectively called *indels*. An overview of the error rate and major type of errors present in several sequencing platforms in given in Table 1.1.

Quality values

To help identify sequencing errors, read sequences are accompanied by a series of *quality values*. In practice, quality values are not represented as the raw base-calling error probabilities, but by the *Phred quality score*[2]. Phred quality scores Q are defined as a property that is logarithmically related to the base-calling error probabilities P [57]:

$$Q = -10 \log_{10} P.$$

For example, if the Phred score of a base equals 30, the chance that this base is called incorrectly is 0.1%. For Sanger sequencing, the cut-off reflecting high-quality bases is given by bases having a minimum score of 20, which represents an error probability of 1%. For next generation sequencing data, a standard cut-off is not yet established, but each manufacturer provides quality scores that should reflect values equivalent to or greater than 20 [255].

As the score values are assigned to each individual nucleotide, the scoring system is useful for tracking mutation-type errors, but is less suited to identify indel errors. Furthermore, quality values double storage requirements of sequencing reads. As a result, lossy storage of quality values has been debated due to increasing costs of the storage of sequencing data [80].

Paired-end reads

By default, sequencing libraries produce *single-end* sequencing reads. These sequences are produced by reading the order of nucleotides from the 5′-end of a sequence to the 3′-end of a larger DNA template that was inserted in the sequencing device. Due to limited read length, only part of the template at the 5′-end is contained in the read. In contrast, *paired-end* and *mate-pair* sequencing libraries

[2]Named after the program that developed these values to help automation of DNA sequencing in the Human Genome Project.

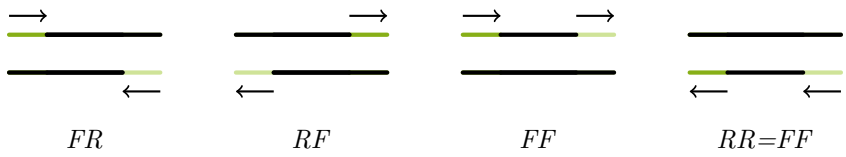


Figure 1.2: Possible orientations for paired-end reads. The DNA fragment is shown in black, with green ends showing the part of the fragment that can be part of the reads. Top lines represent the forward strand and bottom lines represent reverse complement strand. Arrows indicate the orientation of the reads.

produce reads from both ends of the same DNA template. The use of paired-end reads is especially useful to improve mappability of reads and help with assembly in repetitive regions of a genome and to discover rearrangements in a sequence.

Both paired-end and mate-pair sequencing libraries produce pairs of reads with an approximately known distance between them, but differ in the technological approach [48]. Although the processes vary technologically, the main difference between the final read sequences are the distance between the reads and their relative orientation. Therefore, we will refer to the reads produced by both protocols as *paired-end reads*. For our purposes, a data set of paired-end reads consists of pairs of reads, consisting of the *first mate* (originating from the 5′-end) and the *second mate* (originating from the 3′-end). The size of the initial DNA template is called the *insert size*, which is equal to the sum of the read lengths together with the distance between them.

Depending on the sequencing library, both reads can have the same orientation, or be reverse complemented. There are three possible orientations, which are depicted in Figure 1.2. From left to right these orientations are:

- forward-reverse (*FR*): the mates are found on opposite strands and point towards each other,
- reverse-forward (*RF*): the mates are found on opposite strands and point away from each other,
- forward-forward (*FF*): the mates are found on the same strand, with the first mate being on the left if on the forward strand, and on the right if on the reverse complement strand.

The case reverse-reverse is identical to the forward-forward orientation.

Figure 1.3: Representation of special cases in the relative position of paired-end reads in *FR* orientation, depending on the insert size and size of the reads. The DNA template is indicated in black, with the upper line being the forward strand and the bottom the reverse complemented strand. The darker green line represents the first mate, whereas the lighter green is the second mate. In *(a)*, there is a positive gap between the mates, *(b)* represents an overlap, *(c)* shows mate 2 being fully contained in mate 1 and case *(d)* shows both mates extending past each other in normal orientation.

In practice, the size of the sequenced DNA templates is not equal for all reads, but lies in an interval between given minimum insert size and maximum insert size. Furthermore, the length of both mates is not necessarily the same.

In the default case, the insert size is large enough to have a positive gap or small overlap between both mates in a pair. However, during alignment (see Section 1.3) it is possible that one mate could be fully contained in the other, or that both mates extend past each other, resulting in a *dovetail*. These cases are shown in Figure 1.3.

Read file formats

Two file formats are typically used for representing sequence information: FASTA and FASTQ. Both file formats include a header that can contain additional information, but FASTQ also stores the quality values for the sequence. Multiple sequences in the same file are separated by distinguishable characters in the header of each sequence entry.

In FASTA, information about the sequence is stored in a header, starting with the '`>`' character. The sequence is split over the next lines, in such a way that each line, except possibly the last one contains the same number of characters. Common file extensions of FASTA files are `.fasta` and `.fa`. An example FASTA sequence is:

```
>chromosome1
AGGGTCACGTAATGCTGATCCAGTCTTGTTTTTATTTTCATTCATGTTCC
CGCTCTTGCTTTGATTCCGACTTCTAACGTTTAACCTGTGATCAGACGTT
```

A sequence in FASTQ format consists of four lines, structured as follows:

- First line contains the header information and always starts with '@'.
- Second line contains the sequence.
- Third line starts with '+' and is optionally followed by the same identifier given in line 1.
- Fourth line consists of the encoded quality scores and should be of equal length as line 2.

Encoded quality values are represented by ASCII characters in the range '!' to '~'. Although some variants appear between sequencing technologies, usually ASCII characters 33 to 126 represent Phred quality scores from 0 to 93. In practice, however, quality values above 40 (Illumina technology) or 60 are rarely seen. Common FASTQ file extensions are `.fastq` and `.fq`. An example of a sequence in FASTQ is:

```
@reads.000000000
CGAAAGTAAGAACGCGAAAAGCGGAAAAAAGCAGCAGAGAAGAAACGACG
+
67U/'41P,0E(46-7_B.(-1H)_^9.(&/78223467G(0X1(,>/0/
```

Paired-end reads can either be stored in a single file, where the second mate directly follows the first, or using two separate files, where the two mates have the same order in both files. In addition, the header of the first mate of a pair usually ends with '_1', '/1' or using similar tokens. Likewise, the header of the second mate usually ends with '_2' or '/2'. In some cases, the first mate is also identified with the number 0, whereas the second mate is assigned number 1.

1.2.3 Sequencing technology comparison

This section contains a comparison between the major sequencing technologies of all generations, which is mostly adapted from several excellent review articles [48, 161, 166, 184, 236], and complemented by other sources containing additional changes since publication of the reviews. The comparison is mainly focused towards features that are important for mapping and alignment of reads. A comprehensive summary of the most important features is given in Table 1.1.

Sanger sequencing was the dominating technology of first generation sequencing. It is still used due to its relative high read length (300bp - 900bp) and low

error rate. As the oldest technology listed, it has the lowest throughput and highest cost.

One of the earliest breakthroughs in massively parallel sequencing was obtained by the *454 Life Sciences* technology, obtained by Roche. Its pyrosequencing technology currently features reads up to 1000bp[3], rivaling that of Sanger sequencing. It is therefore ideal for *de novo* sequencing, resequencing of known genomes and metagenomic studies. The major source of errors in 454 reads can be found in homopolymer sequences (long single-nucleotide repeats). For example, the technology cannot discriminate between sequence `AAAAAAAAAAAAA` (13 successive `A`'s) and sequence `AAAAAAAAAAAAAA` (14 successive `A`'s). Recently, the manufacturer has announced that it will discontinue support for its sequencers[4].

The *Illumina* sequencing platform by Solexa/Illumina is the current market leader in next generation sequencing technology. It features a very high throughput at a low cost per sequenced base. Substitution errors appear most frequently in this technology and errors are also more frequent at the end of the read. Initially, it featured high-quality reads of only 30bp to 36bp, which rendered it near impossible to be used for *de novo* assembly, but did not prohibit its success in resequencing applications. This led to an era of short-read mapping algorithms (see Section 1.3.3). The read length of Illumina technology is anticipated to increase to 2×400bp using paired-end sequencing [181]. Recently, Illumina also offers a long read technology, built from overlapping smaller reads that are locally assembled to reads reaching 10kbp [246].

Other NGS technologies include those of *SOLiD* (Sequencing by Oligonucleotide Ligation and Detection), *Complete Genomics* and *Ion Torrent*. SOLiD features very high quality reads, but its long runtime and short read length limit its usefulness for some applications. Complete Genomics focuses on in-house human genome sequencing and analysis. It features low sequencing costs [55], but also short read lengths. Ion Torrents ion semiconductor sequencing technology features lower costs and faster runtimes and is comparable to the 454 technology in terms of sequencing errors. However, the raw error rate of Ion Torrent reads is higher than that of 454 reads to the point that no reads can be generated in regions containing long homopolymers [208]. In addition, the size of reads produced by Ion Torrent is smaller than that of 454 reads.

Third generation sequencing technology has introduced single-molecule DNA sequencers, who have several advantages over the previous methods (see Sec-

[3]Source: `http://454.com/products/gs-flx-system/`(last accessed May 2014)

[4]Source: `www.genomeweb.com/sequencing/roche-shutting-down-454-sequencing-business` (last accessed November 2014)

tion 1.2.1). The most used technology from this generation is the Single-Molecule Real-Time (SMRT) sequencing, developed by *Pacific Biosciences* (PacBio). This technology features both high quality short reads and lower quality long reads. In 2012, average long read length was approximately 3kbp, but reads up to 15kbp could be generated [181]. Sequencing errors in this technology consist primarily of small indels that are uniformly distributed over the read sequence [132]. This poses challenges for existing mapping algorithms [36]. It has been shown, however, that many errors in these long reads can be corrected using shorter Illumina reads [132].

In the near future, *Oxford Nanopore* [236] might become an important sequencing read provider player. Its technology promises very long read lengths, over 5kbp, with relatively low error rates of less than 4% [181]. The company furthermore claims that there would be no theoretical limit on read length, and 50kbp reads would be easily obtainable.

The main focus of the algorithms developed in this work follow the general trend of sequencing technologies towards longer reads, which could possibly contain more errors. Therefore, evaluation of our methods mainly use (simulated) reads from Illumina, 454 and (error corrected) Pacific Biosciences technologies.

1.3 Alignment

A key process in the field of biological sequence analysis is the comparison of sequences, including the identification of differences and matching substrings, which in turn can be arranged in an alignment between those sequences. Many applications are geared towards finding an alignment between two or more sequences that is optimal given a predefined distance measure.

1.3.1 Sequence alignment

An *alignment* between two sequences is an arrangement of the nucleotides to identify regions of similarity (matches), and dissimilarity (mismatches or mutations and gaps). It can also be seen as a way of transforming the first sequence into the second by changing, inserting and removing characters. In biological sequences this corresponds to the introduction of point mutations, insertions and deletions. The alignment can be used to indicate regions of structural, functional or evolutionary similarity.

```
T C T A C A G T T G C G A T G G T T C T A G G T A C A T T G C C T A - A T T A
| |   |    | | | |     | | | | | | | | | | | | |  | | | | | |  |   |   | |
T C - A - - G T T G - - - T G G T T C T A G G T A A A T T G C C - A C A C T A
```

Figure 1.4: Representation of a possible alignment between two sequences. The dashes in the top and bottom row represent gaps, whereas the pipes in the middle row indicate matching bases. The alignment is a possible optimal global alignment using a match score of 1 and a mismatch and gap penalty of -1.

A representation of an alignment between two DNA sequences can be found in Figure 1.4. The representation consists of a matrix that includes a single row for each sequence, separated by a row containing a '|' symbol in a column when the bases of the sequences in that column are equal. A dash symbol '-' in a column represents a gap in the alignment. Dashes in the first and second sequence respectively represent deletions and insertions. Unaligned bases that are part of the sequences are printed in lower case (example in Figure 1.6b).

The definition of *optimal alignment* depends on a given *distance measure* or *scoring function* defined on the set of matches, mismatches and gaps between the sequences. In computer science, the Hamming distance and edit distance are commonly used distance measures, whereas in bioinformatics more complex distance measures are employed that take into account different types of mutations and gaps.

The *Hamming distance* is defined for strings of equal length and measures the number of mutations required to change one string into the other. The cost of a mismatch is one, whereas the bonus of a match is zero. The *edit distance* counts the number of edit operations, which includes changing a character and inserting or deleting a single character. Thus, the costs of mutations and gaps both equal one, whereas the bonus for a match is still zero.

In bioinformatics, some mutations are considered more likely than others. Therefore, a *substitution matrix* is used in the scoring function, which includes the score (or penalty) for each possible mutation. The score for matching bases is usually constant over all nucleotides, except for the unknown base match `N`, which can be considered as a mismatch. For the alignment of sequencing reads, the mismatch penalty can also depend on the Phred quality scores [141].

Different scoring schemes are also used for *gap penalties*. For instance, it is possible to define a different penalty for insertions and deletions, which might, for example, be helpful in error correcting reads that contain mostly deletion or

```
T C T A C A G T T G C G A T G G T T C T A G G T A C A T T G C C T A A T T A
      | | | | | |     | | | | | | | | | | | | |  | | | | | |    |  | |
- - - T C A G T T G - - - T G G T T C T A G G T A A A T T G C C A C A C T A
```

Figure 1.5: Representation of an optimal alignment between two sequences using affine gap penalties. The scores used are $+1$ for matches, -3 for mismatches, -5 for opening a gap, and -1 for extending a gap.

insertion type errors. Instead of a constant penalty for each single-nucleotide indel, the *affine gap penalty* model assumes a fixed (large) gap opening penalty and another (smaller) penalty for each consecutive base in the gap.
This model favors mismatches and larger gaps at the expense of small gaps. As an example, Figure 1.4 shows an optimal alignment between two sequences using fixed gap penalties and Figure 1.5 shows an optimal alignment for the same sequences using affine gap penalties. The alignment in Figure 1.4 consists of two mismatches and five gaps which in total consist of eight nucleotides. In contrast, the alignment in Figure 1.5 consists of five mismatches and two gaps which in total consist of six nucleotides.

In addition to affine gap penalties, certain gaps can also be declared costless. In *global alignment*, an attempt is made to align all bases of both sequences. This type of alignment is preferred when both sequences are roughly of equal size. If the size of the sequences is highly dissimilar, however, *local alignment* or a type of *semi-global alignment* is preferred. Local alignment allows free gaps at the ends of both sequences and is especially used to locate local regions of high similarity. As an example, Figure 1.6 shows optimal alignments between two sequences using global and local alignment.

In between global and local alignment, there are other alignment methods that allow free gaps at some ends or sequences, but not everywhere. These methods are collectively called semi-global alignment methods. In theory, there are four *ends* that can either fix the alignment or allow free gaps, resulting in sixteen different alignment methods. However, most of these cases are symmetric.

There are three semi-global alignment methods that we will use in this dissertation. The first, called *substring alignment*, allows free gaps at the beginning and end of a single sequence, but requires the other sequence to be aligned end-to-end. As the name suggests, the method is used when aligning a smaller sequence that is assumed to be a substring of a larger sequence. This is typically the case in read mapping. An example substring alignment is given in Figure 1.7, using

```
C A T G A A C A T G A A G A G A C G T G T G C T C T C C A T A A T G G C C C C
  |   |  |   |  |  | | | |  | | |  | |  |    | |  |
- A - - A - C - - G C A - A G A C A T G T T C T G T T - - T A - T - - - - - -
```

(a) global alignment

```
c a t g a a c a t g a a g A G A C G T G T G C T C T C C A T a a t g g c c c c
                      | | | |  | | |  | |  |   | |
            a a c g c a A G A C A T G T T C T G T T T A T
```

(b) local alignment

Figure 1.6: Representation of (a) an optimal global alignment and (b) an optimal local alignment between two sequences.

```
c a t g A A C A T G A A G A G A C G T G T G C T C T C C A T a a t g g c c c c
      | | |   |  |  | | | |  | | |  | |  |   | |
      A A C - - G C A - A G A C A T G T T C T G T T T A T
```

Figure 1.7: Representation of an optimal substring alignment between the same sequences shown in Figure 1.6.

the same sequences as in Figure 1.6.

The other two semi-global alignment methods that will be used fix one end of the alignment, disallowing gaps, but allow free gaps at the other end of the alignment in a single or both sequences. The former method is defined as *extension alignment* and the latter as *local extension alignment*. These alignment methods will be used as part of a chain-guided global or local alignment (see Section 1.4.2 and Chapter 4).

The figures in this section represent possible optimal alignments, but are not unique. For example, a substring can appear multiple times in a larger string, resulting in alignments with equal optimal score. In the case of alignments containing gaps, however, an alignment of equal score can sometimes be obtained by a change in placement of a single indel. It should be noted that, although these alignments are different by definition, they are not considered different by many alignment algorithms and different definitions for the concept of *different alignment* are used in practice. In addition, *suboptimal alignments* can also be of interest if the alignment score is close to the optimal one.

The alignment scoring function used in this work consists of a positive match

bonus, a fixed mismatch penalty and an affine gap penalty consisting of a gap opening penalty and a gap extension penalty. In addition, depending on the type of alignment, some gaps will be without penalty.

1.3.2 Read mapping

The main contributions presented in this dissertation are algorithms for *read mapping*. The biological read mapping problem consists of finding positions and alignments where sequencing reads best fit a reference sequence. If the sequencing reads would originate from the reference sequence, this results in finding the sample position from where the read originates. However, as this is not the case in practice, the biological problem is ambiguous and approximated by a mathematical problem [104].

Technically, the read mapping problem consists of performing substring alignments between a data set of sequencing reads and a reference sequence. Each read is *mapped* independently to the reference sequence, obtaining positions on which the substring alignments starts, called the *mapping positions*, and the alignments proper. As the strand of origin is unknown, both the forward sequence and the reverse complement of the read are mapped.

As discussed in Section 1.2, the sequencing reads are usually tens to thousands of bases long, whereas the reference sequences range from millions to billions of bases in length. In practice, multiple reference sequences are used, such as the set of chromosomes of a genome. However, these reference sequences are often treated as being concatenated into a single sequence, resulting in the search for an optimal substring alignment in a single large sequence.

An abstract definition of the read mapping problem can be given as follows (adapted from [104]):

Definition 1.5. *The read mapping problem takes as input a reference sequence S, a set R of reads $r_i \in R$, $0 \leq i < |R|$, a score function f and a minimum score d. The score function is used to assign an alignment score for substring alignments between reads and S. For each read r, the problem is to find a set of feasible matches of r in S, which are locations in S where the read aligns with a minimum score of d. The set of feasible matches can further be limited in size and/or to only the highest scoring matches, which are called best matches.*

Mapping quality

The quality of read mappings is usually assessed using a numerical value, called the *mapping quality*. The mapping quality value was introduced by *MAQ* [155], and is the Phred-scaled [57] probability that a read alignment may be wrong. Definition 1.6 is transcribed from Lee and Schatz [144].

Definition 1.6. *The mapping quality score q of an alignment is a probabilistic measure that a read is correctly mapped. It is typically expressed in Phred-scaled [57] form. For a reference genome S of length $|S| = n$ and a read P of length $|P| = m$, the probability $p(P|S, i)$ of observing a read alignment of P on position i in S is given by the product of the probabilities of errors recorded in the quality values of the bases that disagree with the reference genome. The posterior error probability $p_s(i|S, P)$ of a position is defined as:*

$$p_s(i|S,P) = \frac{p(P|S,i)}{\sum_{j=0}^{n-m} p(P|S,j)}$$

The mapping quality score q for the alignment starting at position i is then defined as:

$$q = -10 \log_{10}(1 - p_s(i|S,P))$$

Accordingly, higher values of q represent a more confident alignment, and the mapping quality score q will be lower or zero for reads that could be mapped to multiple locations with nearly the same number of mismatches. Mapping quality values of paired-end reads are either the sum of the single-end quality values for uniquely mapped pairs, or the quality values of the individual ends for pairs with multiple alignments.

In a probabilistic view, each read alignment is an estimate of the true alignment and is therefore also a random variable. If the mapping quality of a read alignment is q, the probability p that the alignment is wrong can be calculated as:

$$p = 10^{\frac{-q}{10}}$$

For $q = 30$, this means that on average one in every thousand alignments would be wrong.

Calculation of mapping quality using Definition 1.6 is impractical, as it requires mapping the read at every position of the reference genome. Therefore, approximations of this measure are used in practice. In addition, a different

definition is essential as it does not take into account indels and some read mapping algorithms do not rely on quality values for alignment. The approximation used in this work is adapted from the *BWA-SW* [151] mapper and is given in Definition 1.7.

Definition 1.7. *The mapping quality value is approximated by* $m \times \frac{s_1 - s_2}{s_1}$*, where* s_1 *is the alignment score of the highest scoring alignment,* s_2 *is the alignment score of the second highest scoring alignment and* m *is the maximum allowed mapping quality value. The alignment score is the score given by the aligner to the alignments, using a given distance measure or scoring function (usually obtained through dynamic programming). If only one alignment has been found,* s_2 *is set to* 1*. Mapping quality of paired-end reads is defined based on the sum of the alignment scores of the mates.*

The mapping quality is often used to compare the sensitivity and specificity of read mapping algorithms. Reads mapped with a high mapping quality indicate that the mapper perceives this alignment to be the best one, whereas reads mapped with low mapping quality indicate that similar alignments were found on other positions in the reference genome. Matches with low mapping quality are more likely to contain false positives than matches with high mapping quality.

Evaluation of read mapping tools using mapping quality is preferable to a simple count of the number of mapped reads, but still might not provide a fair comparison between the tools. For example, mapping quality zero indicates a mapper has found two alignments with equal alignment score. Another mapper might be less sensitive and find only one of both alignments, and reports the alignment with a high mapping quality. As many evaluations in the literature discard reads mapped with mapping quality zero, the more sensitive mapper is at a disadvantage. In contrast, reporting mapping quality zero could overestimate the false-positive rate of tools that utilize a low alignment score threshold.

The mapping quality metric is also very sensitive to small changes in read positions, read qualities, differences between local and end-to-end alignments, and the used alignment scoring function. Clipping few bases of an alignment can result in a significant drop in mapping quality [144]. Scoring functions that contain large differences between penalties and bonuses for matches, mismatches and gaps will produce different mapping qualities compared to scoring functions that contain smaller differences between bonuses and penalties.

1.3.3 Read mappers

The advent of next generation sequencing has made large-scale (re)sequencing possible, resulting in a huge increase in the size of sequencing read data sets and changing the field of read mapping considerably. First, due to the size of the data sets, read mapping algorithms do not produce exact solutions to the problem stated in Definition 1.5. Instead, many heuristics are used to speed up the process at the cost of accuracy.
However, since read mapping is one of the first steps in many next generation sequencing pipelines, and is crucial to the accuracy of downstream analysis, it has gained a lot of attention during the last decade. Many read mapping algorithms, also referred to as *read mappers*, obtain a different trade-off between accuracy, performance and memory requirements. Second, next generation sequencing has widened the field of sequence analysis to gave rise to new applications, each featuring different data and/or different algorithmic challenges. Third, in contrast to the domination of Sanger sequencing in the first generation, the presence of multiple sequencing technologies in the second generation sequencing technology requires adaption to technology-specific features. Finally, the need for faster methods has inspired more emphasis on parallelization and hardware optimized implementations.

The multitude of read mapping algorithms that have recently been published can be found in Figure 1.8. As a result of the above mentioned features of next generation sequencing read mapping, few mappers presented in Figure 1.8 outright outperform older mappers. Instead, many have unique features, or present a unique trade-off between accuracy, performance and memory requirements.

The remainder of this section provides an overview of the different types of read mappers, their features and limitations. This review is mainly based on several review articles [77, 153].

Applications and read features

A first major category in which read mappers can be divided, is the type of sequencing application the mapper is designed for. Most mappers in Figure 1.8 are DNA mappers (shown in blue), designed for analysis of the DNA content of an organism. The mappers in red represent RNA mappers, designed for RNA-seq. RNA mappers have to take into account large deletions in the alignment which are due to intronic regions in eukaryotic cells being spliced out after transcription. Mappers in green represent miRNA mappers, as part of miRNA-seq analysis

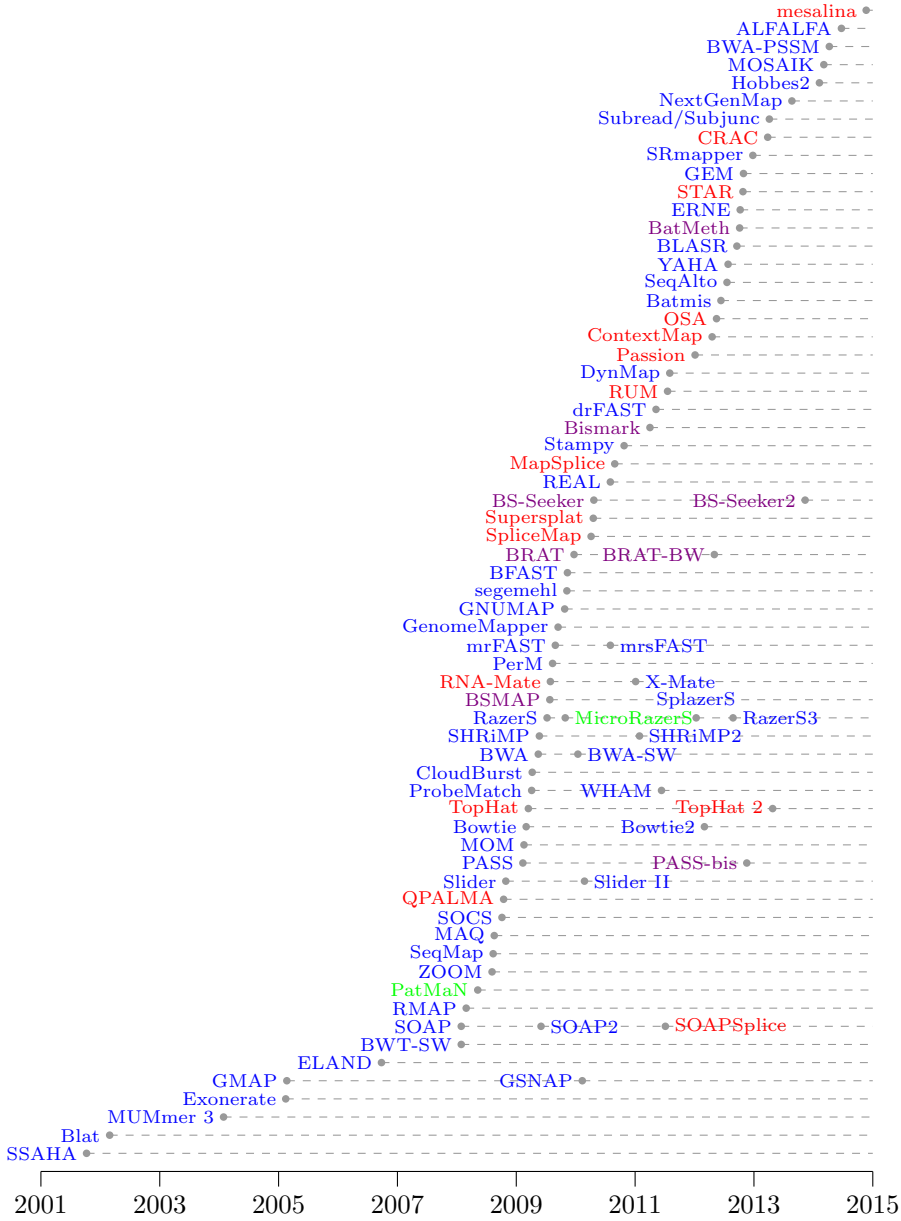


Figure 1.8: Time line showing publication date of most of the read mapping algorithms that have been developed since 2001. The figure is mostly adapted from a recent version (`http://wwwdev.ebi.ac.uk/fg/hts_mappers/#timeline`, last accessed June 2014) of data originally published by Fonseca *et al.* [77]). Tools on the same line show major updates of previously released tools. DNA mappers are plotted in blue, RNA mappers in red, miRNA mappers in green and bisulphite mappers in purple.

pipelines that examine MicroRNAs. MiRNA mappers have to be able to map very short reads, ranging from 16 to 30 bases [77]. Finally, mappers in purple are bisulphite mappers, used in bisulphite sequencing [42] analysis pipelines, which is used for determination of DNA methylation patterns. In bisulphite sequencing reads, non-methylated `C`'s (respectively `G`'s) are converted to `T`'s (respectively `A`'s), but others are not.

The length of reads has a major impact on the accuracy and performance of read mapping algorithms.
Many mappers are therefore optimized or even restricted to a certain range in the size of reads. Early next generation sequencing technology produced reads that are much shorter than those produced by Sanger sequencing, resulting in a large number of *short read mappers* being developed since 2007. Recently, however, read lengths have increased again for some technologies and this trend is expected to continue in the near future [181]. The increase of read length has led to the development of *long read mappers*, of which *ALFALFA* (see Chapter 4) and *mesalina* (see Chapter 5) are examples of. An overview of the range of read lengths for most mappers was produced by Fonseca *et al.* [77].

The trade-off between accuracy and performance of read mappers is influenced greatly by the number of differences between the reads and reference genome. These differences can be caused by sequencing errors or natural differences, arising from the fact that the reads do not originate from the reference genome's DNA directly. Like in general alignment, most differences include mutations and indels. Other structural variations, such as duplication, inversion or relocations of an entire substring, are more difficult to detect and are often left to post processing tools.

Some read mappers limit the total number of mutations and indels, limiting their ability to map certain reads, but greatly increasing their performance for the reads they do map. For example, *Vmatch* [136] only allows 5 mutation-based differences, and the number of indels allowed in *SOAP* [156] is limited to 3. Other mappers also allow a single large gap in the alignment, such as *SOAP2* [157] and *SeqAlto* [187]. Other mappers use different techniques [77] to take advantage of the Phred quality scores of sequencing reads, as it has been shown to reduce alignment errors if mutations are given a lower penalty when the base has a low quality value [153]. In practice, most mappers utilize a user-set maximum score or minimum identity percentage, resulting in a trade-off between accuracy and performance.

Both read length and error modalities are usually unique to certain sequencing

technologies. As a result, several mappers have been specifically designed for mapping reads produced by a given technology. For example, *Slider* [175] is designed for Illumina reads, *TAPyR* [60] for 454 reads, *SOCS* [203] for SOLiD reads and *BLASR* [36] is a mapper for reads produced by Pacific Biosciences technology. In addition, most read mappers are evaluated on Illumina reads, as Illumina technology currently dominates the next generation sequencing market.

Type and size of the alignment and output

Read mappers differ not only in the target input data, but also in the type and size of the alignment output. For example, most mappers perform substring alignments in which the read is aligned end-to-end. Other mappers, such as *BWA-SW* [151] and *SSAHA2* [195] produce local alignments. Local alignments have the advantage of being more robust against differences located at the end of the read sequence and even highly dissimilar reads can be matched partially to the reference genome. In other cases, however, local alignment might miss important variants located at the end of reads. Several mappers either allow users to switch between end-to-end and local alignment [141,249] or automatically choose between a global and local alignment, depending on the difference in alignment score for both situations [148].

Another difference between mappers is the number of alignments that are produced and reported. Mappers can either report one or multiple *matches*, *feasible matches* or *best matches* (see Definition 1.5). Mappers that report only a single alignment per read are called *best-mappers* [104], and include *SOAP*, *Blat* [125] and *QPalma* [47]. Other mappers, such as *BWA-MEM* [148], offer users the option to report all alignments found during the matching process. Another option is to set a maximum threshold on the number of alignments reported by the mappers. Examples of mappers with this feature are *Bowtie 2* [141] and *ALFALFA*. Finally, several mappers can be categorized as true *all-mappers* [104], in the sense that they will attempt to find all possible different alignments with a given minimum similarity or alignment score. Mappers in this category usually operate on the edit distance and include *RazerS3* [253] and *GEM* [179]. The bulk of the available read mappers are either best-mappers or mappers that allow the user to set a fixed number of alignments. For many applications, a single or few alignments are enough or even desirable. In other applications that require a more exhaustive search, such as metagenomics, all-mappers can provide more detailed information [179]. In addition, all-mappers are usually more sensitive overall, but much slower than best-mappers.

A definition that is especially important for mappers producing multiple alignments per read is when two alignments are considered "different". The *Rabema* benchmarking method [104] features a formal definition of *match equivalence*, as well as an exhaustive discussion on the concept. Informally, two matches are considered equivalent if their alignment shares a common trace on the dynamic programming matrix (see Section 1.4), their end-positions are neighboring, or their end-positions are separated by feasible or other equivalent matches.

Algorithmic aspects

Together with input and output, read mappers also differ in the algorithms and data structures they use to calculate alignments. To the end-users, the algorithmic aspects of mappers are usually less important. They can, however, have a large impact on the trade-off between accuracy, performance and memory requirements on specific data sets. This trade-off is not always obvious from the description and limitations imposed by the mappers themselves.

In general, read mapping algorithms consist of a three-step procedure. The first step consist of the construction of an index structure for the reference genomes and/or the set of reads. This data structure considerably speeds up string searches in the indexed sequence, has a high impact on performance, but is also the main contributor to the memory footprint of the analysis. The second step in mapping usually consists of searching the index for small (in)exact matches between read and reference genome, called *seeds*. Using seeds or another method, the alignment search space is considerably trimmed to only a few candidate regions. These candidate regions are thus explored further in the third step, which usually consists of an exhaustive alignment algorithm, such as the dynamic programming algorithms described in Section 1.4.

A more detailed discussion on the specific steps used in mapping algorithms can be found elsewhere. Index structures will be discussed in Chapter 2. A taxonomy of mapping algorithms based on their alignment strategy can also be found in the reviews by Li and Homer [153] and Fonseca *et al.* [77]. Equally important to the main algorithmic aspects are the heuristics they employ. Although these heuristics contribute greatly to the trade-off obtained by mappers on different data, they do not facilitate easy comparison between mappers.

Some design choices are made to obtain certain time-accuracy trade-offs. All-mappers, such as *RazerS3*, focus on high accuracy, reaching full sensitivity for given thresholds [253]. Best-mappers focus on speed, such as suggested by their reference's title (*e.g.* "Ultrafast..." [143]). Another important trade-off includes

memory requirements. Some mappers are designed to be run on standard desktop computers and utilize memory-efficient index structures, such as the FM-index [143, 150, 163] and sparse or compressed suffix arrays [249]. Other mappers employ larger index structures to increase performance or accuracy, such as *segemehl* [102] and *Hobbes* [6].

Implementations and technical details

In addition to the purely algorithmic aspects given in the previous section, read mapping tools also employ more technical traits to set them apart from others. These traits include both extra tunable features due to additional parameters and performance optimization through parallelization or hardware acceleration.

To increase performance for high-throughput read mapping, most mappers provide support for some level of parallelization. The most common forms of parallelization are those of multi-threading in multi-core CPUs, bit-level parallelism using bit-vector operations or Single Instruction, Multiple Data (SIMD) operations. More rare are the hardware-specific implementations for Graphics Processing Unit (GPU) [8], including some ports of CPU mappers to GPU [160, 225] and several mappers specifically designed for the architecture [130, 164]. Several high-throughput read mappers also exist for use with Cloud Computing, using the MapReduce implementation Hadoop [204, 224].

A full-blown read mapper contains a lot of parameters that can be used to enable or disable special features, specify input and output details, extra preprocess and post process scripts or tune the performance, accuracy and memory trade-off. For example, mappers can be set to only map reads to the forward or reverse complement strand, orientation and insert size restrictions can be set for paired-end read mapping, *etc.* Tuning the trade-offs can especially be daunting for many end-users, as usually many parameters exist for this cause.

Accuracy and performance tuning parameters can vary from parameters with global effect, such as a minimum alignment score or maximum number of alignments to calculate, to very local parameters that adjust the setting of a single heuristic in the algorithm. The large variety in available parameters also complicate a fair evaluation and comparison of read mappers. Moreover, the lack of adjustable traits in tools is not always caused by the use of different algorithms, but the value can just be hard coded, it can be dependent on other parameters, or it is automatically tuned during the alignment process. In practice, evaluations frequently utilize the default parameters settings given by the developers. In some cases, these default settings work well for many settings, and parameter

tuning has marginal effects. In other cases, parameter tuning is a prerequisite to achieve good performance.

Finally, the choice of a read mapper can depend on practical or technical issues, such as the availability and license, the supported operating systems, input and output formats, documentation and ease of installation. The bulk of read mappers are open source and most popular mappers have been successfully tested on all major operating systems. Some notable exceptions are *GEM*, *SSAHA2* and *Vmatch* which are only available as precompiled binaries. Standard input formats are FASTA for reference genomes and FASTQ for read data sets. The SAM/BAM format has become the standard output format and is discussed in the next section. Some mappers offer an additional output format, developed for their tool, which contains additional information that does not fit the SAM/BAM standard. A list of these technical details can be found in the review article by Fonseca *et al.* [77].

1.4 Dynamic programming

One of the most commonly used techniques for solving the various sequence alignment problems is that of *dynamic programming*, originally introduced to sequence alignment by Needleman-Wunsch [194] for global alignment and Smith-Waterman [234] for local alignment. In addition, these algorithms can be easily adjusted for semi-global alignments.

In their classical forms, these dynamic programming algorithms produce optimal alignments for two sequences S and P, with $|S| = n$ and $|P| = m$, with quadratical theoretical time and memory complexity $\mathcal{O}(nm)$. As these algorithms have become common knowledge in the field of bioinformatics, this section contains only a summary of the basic algorithm, together with the adjustments needed for semi-global alignments. Finally, this section also contains a summary of several optimizations to the classical algorithms, of which some are used in Chapters 4 and 5.

The algorithm for finding an optimal alignment between the two sequences S and P is based on the fact that the optimal alignment for any two prefixes $S[..i]$ and $P[..j]$, $0 < i < n$ and $0 < j < m$, can be derived from the optimal alignments for $S[..i-1]$ and $P[..j-1]$, $S[..i-1]$ and $P[..j]$, and $S[..i]$ and $P[..j-1]$. This defines a recursive formula, which can be solved using the trivial solutions for the case of aligning a sequence to an empty string. The implementation of this technique for sequence alignment fills a $(n+1) \times (m+1)$ matrix row-by-row or

$$
\begin{aligned}
A[i,0] &= i \cdot c \\
A[0,j] &= j \cdot c \\
A[i,j] &= \max \begin{cases} A[i-1,j] + c \\ A[i,j-1] + c \\ A[i-1,j-1] + M[S[i],P[j]] \end{cases} \\
\text{Begin alignment} &= A[0,0] \\
\text{End alignment} &= A[n,m]
\end{aligned}
$$

Figure 1.9: Dynamic programming global alignment recursion formula. In the base case, the positions have the following range: $0 \leq i \leq n$ and $0 \leq j \leq m$. For the recursion formula, they have the following range $1 \leq i \leq n$ and $1 \leq j \leq m$.

column-by-column with all intermediate solutions calculated. The final alignment score for an end-to-end alignment is found in the corner opposite of the starting position. The actual alignment then needs to be traced back from this corner using the intermediate solutions that gave rise to the final optimal score.

As an example, the recursion relation, starting situations and begin and end of a global alignment $A[i,j]$ of sequences $S[..i-1]$ and $P[..j-1]$, using scoring matrix M and constant gap penalty c, are given in Figure 1.9. The scoring matrix M holds the match and mismatch score or penalty for each combination of characters from the alphabet. An illustration of the algorithm is shown in Figure 1.10.

1.4.1 Variants for different alignment methods

The recursion relation in Figure 1.9 is used for global alignment using the constant gap model. This recursion relation and/or boundary conditions need to be altered in order to be used for the other alignment problems given in Section 1.3.

The use of an affine gap model requires the use of two extra matrices in the dynamic programming algorithm. These matrices are used to keep track of insertions and deletions separately, whereas the main matrix still holds the optimal alignment scores of any two prefixes of the sequences that need to be aligned. The recursion relation and boundary criteria for the global alignment

		A	A	C	G	C	A	A	G	A	C	A	T	G	T	T	C	T	G	T	T	T	A	T
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
C	-1	-1	-2	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21
A	-2	0	0	-1	-2	-3	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19
T	-3	-1	-1	-1	-2	-3	-3	-3	-4	-5	-6	-7	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17
G	-4	-2	-2	-2	0	-1	-2	-3	-2	-3	-4	-5	-6	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15
A	-5	-3	-1	-2	-1	-1	0	-1	-2	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-12	-13
A	-6	-4	-2	-2	-2	-2	0	1	0	-1	-2	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13
C	-7	-5	-3	-1	-2	-1	-1	0	0	-1	0	-1	-2	-3	-4	-5	-4	-5	-6	-7	-8	-9	-10	-11
A	-8	-6	-4	-2	-2	-2	0	0	-1	1	0	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-8	-9
T	-9	-7	-5	-3	-3	-3	-1	-1	-1	0	0	0	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-7
G	-10	-8	-6	-4	-2	-3	-2	-2	0	-1	-1	-1	1	3	2	1	0	-1	-2	-3	-4	-5	-6	-7
A	-11	-9	-7	-5	-3	-3	-2	-1	-1	1	0	0	0	2	2	1	0	-1	-2	-3	-4	-5	-4	-5
A	-12	-10	-8	-6	-4	-4	-2	-1	-2	0	0	1	0	1	1	1	0	-1	-2	-3	-4	-5	-4	-5
G	-13	-11	-9	-7	-5	-5	-3	-2	0	-1	-1	0	0	1	0	0	0	-1	0	-1	-2	-3	-4	-5
A	-14	-12	-10	-8	-6	-6	-4	-2	-1	1	0	0	-1	0	0	-1	-1	-1	-1	-1	-2	-3	-2	-3
G	-15	-13	-11	-9	-7	-7	-5	-3	-1	0	0	-1	-1	0	-1	-1	-2	-2	0	-1	-2	-3	-3	-3
A	-16	-14	-12	-10	-8	-8	-6	-4	-2	0	-1	1	0	-1	-1	-2	-2	-3	-1	-1	-2	-3	-2	-3
C	-17	-15	-13	-11	-9	-7	-7	-5	-3	-1	1	0	0	-1	-2	-2	-1	-2	-2	-2	-2	-3	-3	-3
G	-18	-16	-14	-12	-10	-8	-8	-6	-4	-2	0	0	-1	1	0	-1	-2	-2	-1	-2	-3	-3	-4	-4
T	-19	-17	-15	-13	-11	-9	-9	-7	-5	-3	-1	-1	1	0	2	1	0	-1	-2	0	-1	-2	-3	-3
G	-20	-18	-16	-14	-12	-10	-10	-8	-6	-4	-2	-2	0	2	1	1	0	-1	0	-1	-1	-2	-3	-4
T	-21	-19	-17	-15	-13	-11	-11	-9	-7	-5	-3	-3	-1	1	3	2	1	1	0	1	0	0	-1	-2
G	-22	-20	-18	-16	-14	-12	-12	-10	-8	-6	-4	-4	-2	0	2	2	1	0	2	1	0	-1	-1	-2
C	-23	-21	-19	-17	-15	-13	-13	-11	-9	-7	-5	-5	-3	-1	1	1	3	2	1	1	0	-1	-2	-2
T	-24	-22	-20	-18	-16	-14	-14	-12	-10	-8	-6	-6	-4	-2	0	2	2	4	3	2	2	1	0	-1
C	-25	-23	-21	-19	-17	-15	-15	-13	-11	-9	-7	-7	-5	-3	-1	1	3	3	3	2	1	1	0	-1
T	-26	-24	-22	-20	-18	-16	-16	-14	-12	-10	-8	-8	-6	-4	-2	0	2	4	3	4	3	2	1	1
C	-27	-25	-23	-21	-19	-17	-17	-15	-13	-11	-9	-9	-7	-5	-3	-1	1	3	3	3	3	2	1	0
C	-28	-26	-24	-22	-20	-18	-18	-16	-14	-12	-10	-10	-8	-6	-4	-2	0	2	2	2	2	2	1	0
A	-29	-27	-25	-23	-21	-19	-17	-17	-15	-13	-11	-9	-9	-7	-5	-3	-1	1	1	1	1	1	3	2
T	-30	-28	-26	-24	-22	-20	-18	-18	-16	-14	-12	-10	-8	-8	-6	-4	-2	0	0	2	2	2	2	4
A	-31	-29	-27	-25	-23	-21	-19	-17	-17	-15	-13	-11	-9	-9	-7	-5	-3	-1	-1	1	1	1	3	3
A	-32	-30	-28	-26	-24	-22	-20	-18	-18	-16	-14	-12	-10	-10	-8	-6	-4	-2	-2	0	0	0	2	2
T	-33	-31	-29	-27	-25	-23	-21	-19	-19	-17	-15	-13	-11	-11	-9	-7	-5	-3	-3	-1	1	1	1	3
G	-34	-32	-30	-28	-26	-24	-22	-20	-18	-18	-16	-14	-12	-10	-10	-8	-6	-4	-2	-2	0	0	0	2
G	-35	-33	-31	-29	-27	-25	-23	-21	-19	-19	-17	-15	-13	-11	-11	-9	-7	-5	-3	-3	-1	-1	-1	1
C	-36	-34	-32	-30	-28	-26	-24	-22	-20	-20	-18	-16	-14	-12	-12	-10	-8	-6	-4	-4	-2	-2	-2	0
C	-37	-35	-33	-31	-29	-27	-25	-23	-21	-21	-19	-17	-15	-13	-13	-11	-9	-7	-5	-5	-3	-3	-3	-1
C	-38	-36	-34	-32	-30	-28	-26	-24	-22	-22	-20	-18	-16	-14	-14	-12	-10	-8	-6	-6	-4	-4	-4	-2
C	-39	-37	-35	-33	-31	-29	-27	-25	-23	-23	-21	-19	-17	-15	-15	-13	-11	-9	-7	-7	-5	-5	-5	-3

Figure 1.10: Illustration of global alignment using dynamic programming. The alignment was performed for the sequences in Figure 1.6 using a match bonus of 1 and mismatch and constant gap penalties of -1. The grey path shows the trace of an optimal alignment.

$$
\begin{aligned}
U[i,0] &= -\infty \\
L[0,j] &= -\infty \\
A[i,0] &= o + i \\
A[0,j] &= o + j \\
A[0,0] &= 0 \\
U[i,j] &= \max \begin{cases} U[i-1,j] + e \\ A[i-1,j] + o + e \end{cases} \\
L[i,j] &= \max \begin{cases} L[i,j-1] + e \\ A[i,j-1] + o + e \end{cases} \\
A[i,j] &= \max \begin{cases} U[i,j] \\ L[i,j] \\ A[i-1,j-1] + M[S[i],P[j]] \end{cases} \\
\text{Begin alignment} &= A[0,0] \\
\text{End alignment} &= A[n,m]
\end{aligned}
$$

Figure 1.11: Recursion formula for global alignment using affine gap penalties. In the base case, the positions have the following range: $0 \leq i \leq n$ and $0 \leq j \leq m$. For the recursion formula, they have the following range $1 \leq i \leq n$ and $1 \leq j \leq m$.

with affine gap model $A[i,j]$ of sequences $S[..i-1]$ and $P[..j-1]$ are given in Figure 1.11. Mismatch penalties and match scores are stored in the scoring matrix M, the gap opening penalty is o and the gap extension penalty is e. Two auxiliary matrices are used: U stores vertical gaps, which correspond to gaps in S (deletions), whereas L stores horizontal gaps, which correspond to gaps in P (insertions).

For the remainder of the alignment problems in this section we will again use the constant gap model.
It is, however, easy to combine the model (Figure 1.11) with the new boundary conditions.

The Smith-Waterman local alignment algorithm changes the recursion relation by allowing free gaps at the start and the end of the alignment. Free

$$
\begin{aligned}
A[i,0] &= 0 \\
A[0,j] &= 0 \\
A[i,j] &= \max \begin{cases} 0 \\ A[i-1,j]+c \\ A[i,j-1]+c \\ A[i-1,j-1]+M[S[i],P[j]] \end{cases} \\
\text{Begin alignment} &= \text{ first encountered zero in trace} \\
\text{End alignment} &= A[i_{max},j_{max}] \geq A[i,j]
\end{aligned}
$$

Figure 1.12: Recursion formula for local alignment. In the base case and for the end alignment value, the positions have the following range: $0 \leq i \leq n$ and $0 \leq j \leq m$. For the recursion formula, they have the following range $1 \leq i \leq n$ and $1 \leq j \leq m$.

gaps at the start are achieved by setting the start conditions in Figure 1.9 to zero. In addition, the alignment can start at any position by disallowing negative scores and tracing back until the first zero is found. Free gaps at the end can be gained by starting the trace at $A[i_{max}, j_{max}]$, for which holds that $A[i_{max}, j_{max}] \geq A[i,j]$, for any $i,j \in [0..n, 0..m]$. The local alignment relation is given in Figure 1.12.

The semi-global alignment variants can be obtained by using a combination of the global and local alignment recursion relations.
The substring alignment allows free gaps in the beginning and end of a single sequence, which is reflected in the beginning condition of the rows or columns of the matrix and the end of the alignment (beginning of the traceback), which starts at the cell with the highest alignment scores among those of the last row or column:

$$
\begin{aligned}
A[i,0] &= 0 && 0 \leq i \leq n \\
A[0,j] &= j \cdot c && 0 \leq j \leq m \\
\text{End alignment} &= A[i_{max},m] \geq A[i,m] && 0 \leq i \leq n
\end{aligned}
$$

Extension alignment is almost identical to global alignment, except for the start cell of the trace of the alignment:

$$\text{End alignment} = A[i_{max},m] \geq A[i,m] \quad 0 \leq i \leq n$$

Similar to extension alignment, local extension alignment differs from global alignment only in the start cell of the trace of the alignment:

$$\text{End alignment} = A[i_{max}, j_{max}] \geq A[i, j], \quad 0 \leq i \leq n, 0 \leq j \leq m$$

The difference between global, local, substring and extension alignment is also illustrated in Figure 1.13.

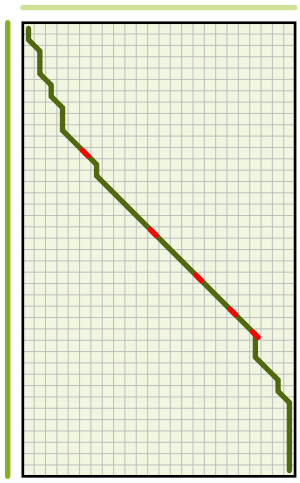
1.4.2 Optimizations

Since the initial publication of the Needleman-Wunsch and Smith-Waterman algorithms, several theoretical and practical improvements have been suggested to both time and memory consumption. Some of these improvements maintain the full sensitivity of the original algorithms, whereas others sacrifice some of it to achieve better practical performance. For example, the memory footprint of the divide-and-conquer algorithm of Hirschberg [101] is linear instead of quadratic, while maintaining theoretical time complexity and sensitivity, at the cost of a practical runtime. Banded alignment and chain-guided alignments improve on both time and memory, but are not fully sensitive. Other optimizations rely on parallelization of the algorithm using bit-level parallelism or specialized hardware, such as GPUs and FPGAs [8, 262]. As the algorithms in this dissertation are designed for standard hardware, we do not cover methods for GPU or other hardware accelerations.

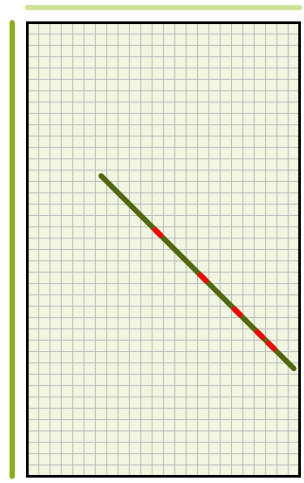
Banded alignment

If an estimate of the maximum alignment distance or minimum alignment score is known beforehand, the dynamic programming matrix does not have to be fully computed to obtain the optimal alignment. In *banded sequence alignment* [73] only a number of diagonals around the main diagonal are computed for an optimal end-to-end alignment. Only a few changes are required to the original algorithm: computation of rows or columns should start and end at the band, cells in the recursion relation that fall outside the band have value $-\infty$, and the trace should not go beyond the band.

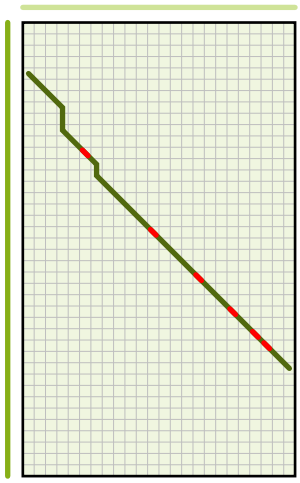
The theoretical time and memory complexity of banded alignment is $\mathcal{O}(md)$, where m is the length of the shortest aligned sequence and d is the *band size*. The memory requirements can also be further reduced using Hirschberg's algorithm to $\mathcal{O}(d)$. An illustration of the effect of banded alignment can be seen in Figure 1.14.



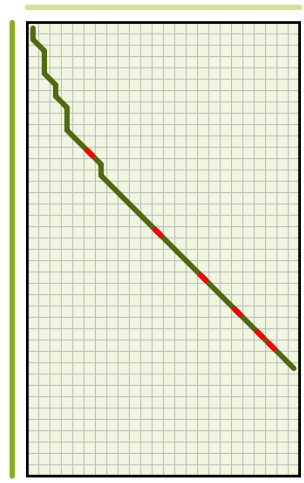
(a) global alignment



(b) local alignment



(c) substring alignment



(d) extension alignment

Figure 1.13: Illustration of the difference between several types of alignment problems solved with dynamic programming. Rows and columns represent respectively reference sequence S and query sequence P from the example also used in figures from Section 1.3.1 and Figure 1.10. The trace of the alignment is indicated in green, with red dots indicating mismatches in the alignment. Alignment (a) represents the global alignment from Figure 1.6a, (b) is the local alignment from Figure 1.6b, (c) represents the substring alignment from Figure 1.7 and (d) is an extension alignment.

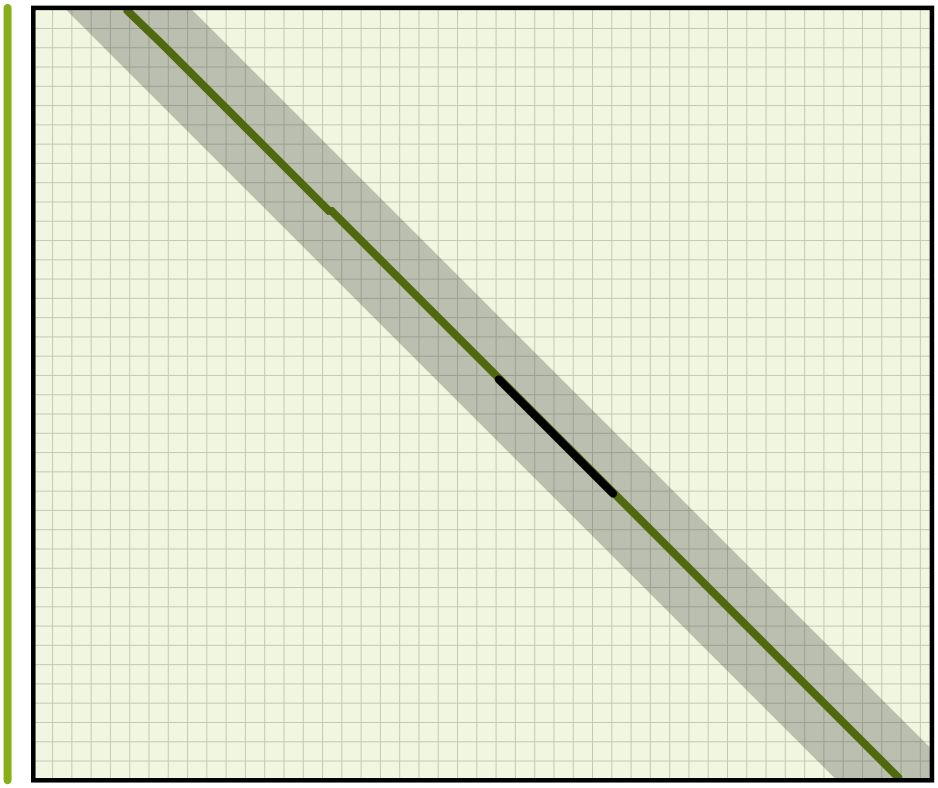


Figure 1.14: Illustration of the effect of banded alignment. The shaded area around the main diagonal of the matrix and the trace of the alignment shows the area of cells that are computed in banded alignment. The black line on the trace of the alignment is an exact match between the aligned sequences that was used to position the band.

Banded alignment guarantees an optimal solution of all possible alignments that fall within the band. If the optimal global alignment would fall outside of the band, however, the optimal solution is obviously not found. In addition, banded alignment is mostly useful for global alignment because a range of diagonals should be known beforehand in which an optimal alignment is to be expected. Furthermore, if both sequences are of different length, care should be taken that both ends of the alignment fall within the given band.

In some situations, such as chain-guided alignment, non-global alignments can also make use of banded alignment if long exact matching substrings are used as anchors to fix the position of the band (see also Chapter 4).

Chain-guided alignment

A more heuristic optimization makes use of a group of exact matches between both sequences to guide the alignment. These matches represent certain diagonals in the dynamic programming matrix that can be part of an optimal alignment. A subset of these matches can be ordered in a (non-overlapping) chain, resulting in a partial alignment. The full alignment is obtained by solving smaller alignment problems in between two consecutive elements of the chain (global alignment) and between the first (last) element of the chain and the start (end) cell of the matrix (extension alignment).

An optimal alignment is not guaranteed, especially in the event of many spurious matches. The accuracy and the time and memory performance of this optimization depend on the number and length of the exact matches found between the sequences. The chain can also be used in combination with banded alignment, and thus shares its time and memory complexity. Examples of chain-guided alignment are used in *MUMmer* [137] and *ALFALFA*. An example of a chain-guided alignment can be found in Figure 1.15. More information on this type of dynamic programming can be found in Chapter 4.

Bit-vector alignment

Whereas the previously discussed optimizations try to limit the number of cell values that need to be computed, the bit-vector algorithm of Myers [188] improves the performance of the classical dynamic programming algorithm by calculating many cell values simultaneously using bit-vectors. The time complexity of the algorithm is $\mathcal{O}(\lceil \frac{m}{w} \rceil n)$, with n and m the lengths of the aligned sequences and w the word size of the machine, which is typically 32, 64 or 128 bit. If the size of a

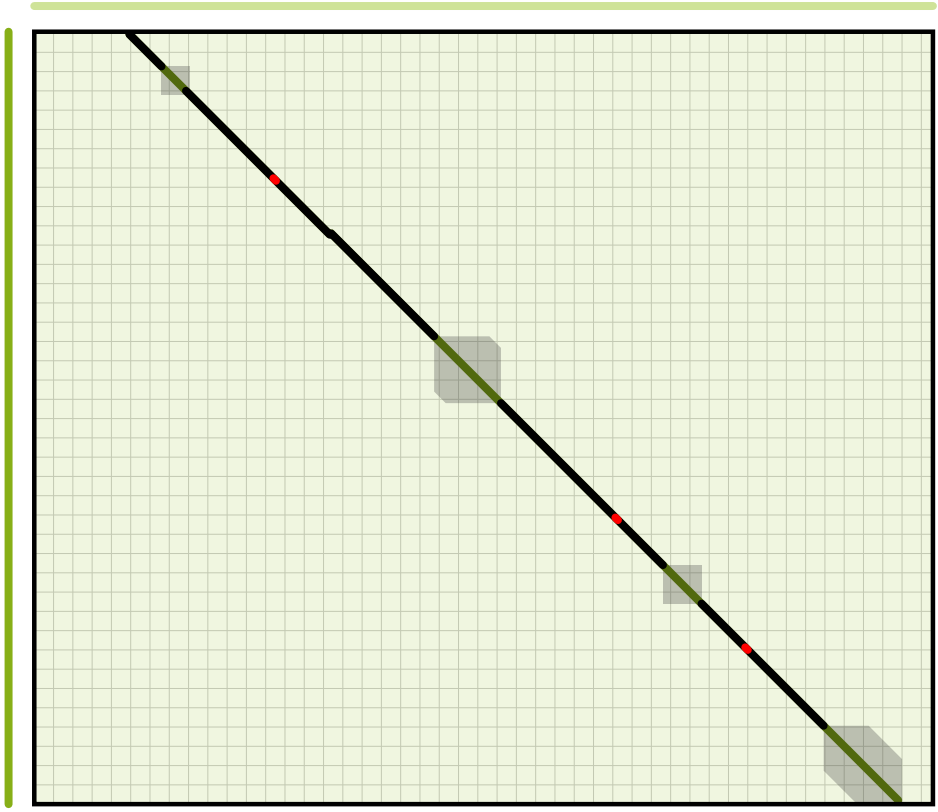


Figure 1.15: Illustration of the effect of chain-guided banded alignment. The only cells that need to be computed are indicated by a darker shaded area around the trace of the alignment (in green). Black line segments on the trace indicate exact matches that guide the alignment and red dots between segments indicate single nucleotide gaps between two consecutive matches. The size of the shaded area is bound by the size of the gap between two matches and the band size.

column is smaller than the computer word size, an entire column of the dynamic programming matrix can be calculated using bit-vector arithmetic.

The bit-vector algorithm of Myers is designed for the edit distance and several other scoring functions that use constant, single point penalties to differences. The main observations made by Myers are that *i)* in these systems, the difference between a cell's value and the value of the cells on which it depends falls in the range $[-1..1]$, and can thus be easily encoded using bit-vectors, and *ii)* the dynamic programming recursion relation can be expressed using these bit-vectors and dependencies within a single column can be resolved.

In essence, the bit-vector algorithm proceeds as follows. First, preprocessing is performed for the alphabet and query sequence. The scores in the dynamic programming matrix are represented by the difference to their horizontal, vertical and diagonal predecessors. For edit distance, the difference values are -1, 0 or 1 and can be encoded using a maximum of two bit-vectors per dependency. The algorithm proceeds as the normal dynamic programming algorithm, calculating column by column. For each column, all bit-vectors are updated and the difference value for the last row is used to update the global alignment score.

Several additions and improvements to the original bit-vector algorithm have been proposed. Hyyrö has designed a banded version of the bit-vector algorithm [112], and has shown how to trace back the alignment [113], as the original algorithm was only designed to return the alignment score. Recently, Benson *et al.* have developed another bit-vector algorithm for scoring functions that use constant integer bonuses and penalties for matches, mismatches and gaps [27].

The bit-vector optimizations improve the performance of the classic dynamic programming algorithms considerably, at no loss of sensitivity. Furthermore, the algorithm does not impose restrictions on the length of the sequences. However, these bit-vector algorithms cannot handle more complex scoring functions with variable penalties for mismatches or affine gap penalties. Furthermore, it is unclear if it can easily be modified to find local or semi-global alignments. Finally, the bit-vector encoding of the alphabet makes it more suited for small alphabets, such as Σ_{DNA} and the need for preprocessing the query sequence makes it unsuitable for very small alignment problems.

Implementations of the bit-vector algorithms have been used in several bioinformatics tools, such as the alignment programs *GEM*, *Hobbes*, *RazerS3* and *Vmatch*.

SIMD alignment

Among the hardware-accelerated optimizations, the use of *Single Instruction, Multiple Data* (SIMD) instructions is the most widely available in common hardware (in all x86 processors). The type of alignment optimized by SIMD operations is local alignment, whereas the bit-vector method described above optimizes global alignment.

Similar to the bit-vector encoding, the algorithms using SIMD instructions pack several matrix elements into a single word (SIMD register) and use efficient CPU operations (SSE, SSE2, AVX, AVX512 operations) to obtain the value of many cells in few CPU cycles. In contrast to the bit-vector method, the actual values are encoded into the register. This typically means that for a register of 128 bits, 16 cells of 8-bit words are grouped (corresponding to values in the range [0..255]) or 8 cells of 16-bit values (corresponding to the values in the range [0..65 535]), allowing for a theoretical speed-up of 8 or 16. Similar to the bit-vector method, however, dependencies between cells in the same column need to be resolved. A possible strategy [58] to handle these dependencies is to ignore the contribution of the vertical values in (1.12), but track them separately. In a typical case, vertical contributions would be less than or equal to zero and local alignment would not be influenced by them. If the vertical values are greater than zero, a second pass through this column can correct errors in the values for the given column.

Several implementations have been suggested for the SIMD accelerated Smith-Waterman algorithm. They differ in, for example, the order of traversing the matrix and which values are packed together. One of the most successful implementations is that of Farrar [58], which is used in alignment tools such as *BWA-SW*, *Bowtie 2* and *SHRiMP* [211]. Another implementation [259] has been used in the mapping algorithm *MOSAIK* [145] and has been shown to improve mapping speed two-fold. In addition, this last implementation improves that of Farrar by also providing an alignment (instead of just the alignment score) and a second, suboptimal score that can help in calculating alignment quality and uniqueness.

Similar to the bit-vector optimization, SIMD implementations of the Smith-Waterman algorithm require preprocessing of the query sequence, which makes it less useful for very small sequences. An advantage over bit-vector methods is that it allows a much more varied scoring system, including affine gap penalties. A remark should be made, however, that the integer scores should not be chosen too high, as scores should optimally be bound to 255. In addition, the less optimal setting of 16-bit matrix elements is always required for longer sequences.

To the best of our knowledge, all SIMD implementations are designed for Smith-Waterman local alignment. The adaption to other alignment problems should be possible, but the performance of these variants compared to regular banded alignment is uncertain. The method can process many matrix elements simultaneously to calculate the alignment score, but the strategy to recover the alignment [259] uses a basic banded alignment. For local alignment, this still provides a performance boost, as the alignment area is usually much smaller than that of the entire matrix, but if the beginning and end of the alignment are indeed the edges of the matrix (*i.e.* global alignment), the gain in performance is lost.

1.5 Evaluation and Testing

This section contains information on the various ways to evaluate the time and memory performance of the index structures and algorithms. The definition of accuracy measurement of algorithms is given in the corresponding chapters. Theoretical performance is expressed in terms of theoretical complexity. The tools used to measure the practical performance are given below. Finally, this section also contains a description of the test environment used in chapters 3, 4 and 5.

1.5.1 Theoretical complexity

The theoretical performance of algorithms and data structures is usually expressed in terms of their *theoretical complexity*, indicated by the *big-$\mathcal{O}$ notation*. Although a theoretical measure of the worst-case scenario, it contains valuable practical information about the qualitative and quantitative performance of algorithms and data structures. For example, some data structures, such as index structures, contain an alphabet-dependency, while others do not. Thus, alphabet-independent index structures theoretically perform string searches equally well on DNA sequences (4 different characters) as on protein sequences (20 different characters). The qualitative information of the theoretical complexity usually categorizes the dependency of input parameters in terms of logarithmic, linear, quasilinear, quadratic or exponential dependency. Intuitively, this means that even if several algorithms nearly have the same execution time or memory requirements for a given input sequence, the execution time and memory requirements of some algorithms will grow much faster than those of others when the

input size increases. In practice, quasilinear algorithms (complexity $\mathcal{O}(n \log n)$) are sometimes much faster than linear algorithms (complexity $\mathcal{O}(n)$), because of the lower order terms and constants involved. These are usually omitted in the big-$\mathcal{O}$ notation. In general, however, the big-$\mathcal{O}$ notation is a good guideline for algorithm and data structure performance. Furthermore, this measure of algorithm and data structure efficiency is timeless and is not dependent on hardware, implementation and data specifications, as opposed to benchmark test results which can be misleading and may quickly become obsolete over time.

1.5.2 Memory model

Practical performance of index structures is not only governed by their algorithmic design, but also by the hardware that holds the data structure. Computer memory in essence is a hierarchical structure of layers, ordered from small, expensive, but fast memory to large, cheap and slow memory types. The hierarchy can roughly be divided into *main memory*, most notably RAM memory and caches, and secondary or *external memory*, which usually consists of hard disks or in the near future solid state disks. Most index structures and applications are designed to run in main memory, because this allows for fast *random access* to the data, whereas hard disks are usually 10^5-10^6 times slower for random access [117]. As the price of biological data currently decreases much faster than the price of RAM memory and bioinformatics projects are becoming much larger, comparing more data than ever before, algorithms and data structures designed for cheaper external memory become more important [245]. These external memory algorithms usually read data from external memory, process the information in main memory and report the result again to disk. As mentioned above, these *input/output* (I/O) operations are very expensive. As a result, the algorithmic design needs to minimize these operations as much as possible, for example by keeping key information that is needed frequently into main memory. This technique, known as *caching*, is also used by file systems. File systems usually load more data into main memory than requested because it is physically located close to the requested data and may be predicted to become needed in the near future. The physical *locality* of data organized by index structures is thus of great importance. Moreover, data that is often logically requested in sequential order, should also be physically ordered sequentially, because sequential disk access is almost as fast as random access in main memory. More information about index structure design for the different memory settings is found in Chapter 2.

Chapter 2

Full-text Index Structures

Sequence data form a large fraction of the data processed in life sciences research. Although the type of sequences and applications varies widely, they all require basic string operations, most notably search operations. Given the sheer number and size of the sequences under consideration and the number of search operations required, efficient search algorithms are important components of genome analysis pipelines. For this reason, specialized data structures, generally bundled under the term *index structures*, are required to speed up string searching. The use of specialized algorithms and data structures is motivated by the fact that the data flow has already surpassed the flow of advances in computer hardware and storage capabilities. The type and implementation of the index structure used directly affects the memory and time performance of many bioinformatics tools. Examples of those tools can be found, among others, in read mapping, alignment, repeat detection, error correction and genome assembly.

There are many types of index structures. The most commonly known index structures are inverted indexes and lookup tables. These work in a similar way to the indexes found at the back of books. However, biological sequences generally lack a clear division in words or phrases, a prerequisite for inverted indexes to function properly. Two alternative index structures are used in bioinformatics applications. *k-mer indexes* divide sequences into substrings of fixed length k and are used, among others, in the *BLAST* [7] alignment tool. *Full-text indexes*, on the other hand, allow fast access to substrings of any length. Full-text indexes come at a greater memory and construction cost compared to k-mer indexes and are also far more complex. However, they contain much more information and allow for faster and more flexible string searching algorithms [76]. The in-

dex structures developed and used in this dissertation belong to the category of full-text index structures. As such, this chapter solely covers full-text index structures. An overview of sequence alignment algorithms based on hash tables can be found elsewhere [154].

Although index structures are already widely used to speed up bioinformatics applications, there are two major factors that limit the performance of index structures in current tools. On the one hand, they too are challenged by the recent data flood. Index structures provide a wide variety of efficient string searching algorithms, but also require an initial construction phase and impose extra storage requirements. Traditionally, this has led to a dichotomy between search efficiency and reduced memory consumption. On the other hand, there is a gap between the field of index structure research and its application domains. Concepts such as suffix trees, suffix arrays or FM-indexes are introduced in general terms in bioinformatics courses, but most of the time, these index structures are applied as black boxes having certain properties and allowing certain operations on strings at a given time. In scientific literature, the description of bioinformatics tools often bypasses a detailed description about the specifications of the index structures used. This does injustice to the vast and rich literature available on index structures and does not present their complex design, possibilities and limitations. As a result, most tools are designed using basic implementations of these index structures, without taking full advantage of the latest advances in indexing technology.

This twofold limitation of index structure performance formed the motivation of the review article "*Prospects and limitations of full-text index structures in genome analysis*" [247], on which this chapter is based. The chapter follows the outline of the article, which covers a comprehensive review of the basic ideas behind classical full-text index structures and an overview of the limitations of these data structures as well as the research done in the last decade to overcome these limitations. Furthermore, in light of recent advances made in both sequencing technology as well as computing technology, some prospects on future developments in index structure research were made in the review article. We reflect on these prospects and list a comprehensive update on the developments made since publication of the article.

In detail, this chapter is structured according to the following outline. Section 2.1 reviews some of the most popular index structures currently in use. These include suffix trees, enhanced and compressed suffix arrays and FM-indexes, which are based on the Burrows-Wheeler transform.

Section 2.2 gives an overview of current state-of-the-art main (RAM) memory index structures, with a focus on memory-time trade-offs. Several memory saving techniques are discussed, including compression techniques utilized in compressed index structures. The aim of this section is to provide insight into the complexity of the design of these compressed index structures, rather than to give their full details. It is shown how their design is composed of auxiliary data structures that govern the performance of the main index structure. On a larger scale, practical results from the bioinformatics literature illustrate the performance gain and limitations of search algorithms. Furthermore, a comparison between index structures, together with an extensive literature list, acts as a taxonomy for the currently known main memory full-text index structures.

While main memory index structures are the focus of the Section 2.2, Section 2.3 discusses the design, limitations and improvements of external memory index structures. The difference between index structures for internal and external memory is most prominent in their use of compression techniques, which are (still) less important in external memory. However, because hard disk access is much slower than main memory access, data structure layout and access patterns are much more important.

The second biggest bottleneck of index structure usage is the initial construction phase, which is covered in the Section 2.4. Both main memory as well as secondary memory construction algorithms are reviewed. The main conceptual ideas used for construction of the index structures discussed in previous sections are provided together with examples of the best results of construction algorithms found in the literature.

A summary of the findings presented in this chapter and some prospects on future directions of the research on index structures and its impact on bioinformatics applications is given in Section 2.5. These prospects include variants and extensions of classical index structures, designed to answer specific biological queries, such as the search for structural RNA patterns, but also the use of new computing paradigms, such as the Google MapReduce framework [50]. Finally, an update is given on the state of index structures research since the publication of the original review article [247]. The update supplements the chapter with current state-of-the-art results and compares these to the previously made prospects in the field.

2.1 Popular Index Structures

Index structures are data structures used to preprocess one or more strings in order to speed up string searches. As the examples in this section will illustrate, the types of searches can be quite diverse, yet some index structures manage to achieve an optimal performance for a broad class of search problems. The ultimate goal of index structures is to quickly capture maximal information about the string to be queried and to represent this information in a compact form. It turns out that both requirements often conflict in practice, with different types of index structures providing alternative trade-offs between speed and memory consumption. However, the speedup achieved over classical string searching algorithms often makes up for the extra construction and memory costs.

The type of index structures discussed here are *full-text index structures*. Unlike natural language, biological sequences do not show a clear structure of words and phrases, making popular *word-based* index structures such as inverted files [29] and B-trees [23] less suited for indexing genomic sequences. Instead, full-text indexes that store information about all variable length substrings are better suited to analyze the complex nature of genome sequences.

The three most commonly used full-text index structures in bioinformatics today are suffix trees, suffix arrays and FM-indexes. The *raison d'être* of the latter two is the high memory requirements of suffix trees. In this section, it is shown how those smaller indexes actually are reduced suffix trees and can be enhanced with auxiliary information to achieve complete suffix tree functionality.

2.1.1 Suffix trees

Suffix trees have become the archetypical index structure used in bioinformatics. Introduced by Weiner [254], who also gave a linear time construction algorithm, they are said to efficiently solve a myriad of string processing problems [97]. Complex string problems such as finding the longest common substring can be solved in linear time using suffix trees. The suffix tree of a string S contains information about all suffixes of that string and gives access to all prefixes of those suffixes, thus effectively allows fast access to all substrings of the string S.

Definition 2.1. *The suffix tree $ST(S)$ is formally defined as the radix tree [186], i.e. a compact string search tree data structure, built from all suffixes of S. The edges of $ST(S)$ are labeled with substrings of S and the leaves are numbered 0 to $n-1$. The one-to-one correspondence between leaf i of $ST(S)$ and suffix i of S is found by concatenating all edge labels on the path from the root to the leaf: the*

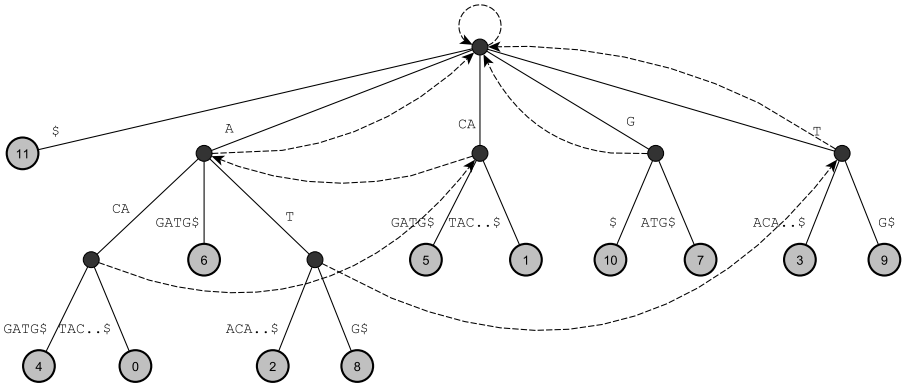


Figure 2.1: Suffix tree for string S = `ACATACAGATG$`, where $ is the special end-character. Each number i inside a leaf represents suffix $S[i..]$ of the string S. Dashed arrows correspond to suffix links. Edges are arranged in lexicographical order. For the sake of brevity, only the first characters followed by two dots and the special end-character $ are shown for edge labels that spell out the rest of the suffix corresponding to the leaf the edge is connected with.

concatenated string ending in leaf i equals suffix $S[i..]$. Moreover, internal nodes correspond to the LCP of suffixes of S, such that labels of all outgoing edges from an internal node start with a different character and every internal node has at least two children.

The last property in Definition 2.1 allows to distinguish suffix trees and non-compact suffix *tries* whose nodes can have single children because edge label lengths are all equal to one. In order for Definition 2.1 to hold for a string S, the last character of S has to uniquely appear in S. In practice, this problem is solved by appending a special end-character $ to the end of string S, with $\$ \notin \Sigma$ and $\$ < c, \forall c \in \Sigma$. This special end-character plays the same role as the virtual end-of-string symbol used in regular expressions (also represented as $ in that context). Hereafter, for every indexed string S it is assumed $S[n-1] = \$$ or, equivalently, $S \in \Sigma^*\$$ holds. As a running example, the suffix tree $\mathrm{ST}(S)$ for the string S = `ACATACAGATG$` is given in Figure 2.1.

The *label* $\ell(v)$ of a node v of $\mathrm{ST}(S)$ is defined as the concatenation of edge labels on the path from the root to the node. From this definition it follows

that $\ell(\text{root}) = \epsilon$. The *string depth* of v is defined as $|\ell(v)|$. The *suffix link* $sl(v)$ of an internal node v with label cw ($c \in \Sigma$ and $w \in \Sigma^*$) is the unique internal node with label w. Suffix links are represented as dashed lines in Figure 2.1.

Most suffix tree algorithms boil down to (partial or full) top-down or bottom-up traversals of the tree, or traversals using suffix links [3]. These different types of traversals are further illustrated using some classical string algorithms.

Top-down traversal

In the exact string matching problem, $occ(P, S)$ has to be found, *i.e.* all positions of a substring P in string S. Exact string matching is an important problem on its own and is also used as a basis for more complex string matching problems. Because P is a substring of S if and only if P is a prefix of some suffix of S, it follows that matching every character of P along a path in $\text{ST}(S)$ (starting at the root) gives the answer to the existential question. This algorithm thus requires a partial top-down traversal of $\text{ST}(S)$ and has a time complexity of $\mathcal{O}(m)$. Because suffixes of S are grouped by common prefixes in $\text{ST}(S)$, the set of leaves in the subtree below the path that spells out P represents $occ(P, S)$. This set can be obtained in $\mathcal{O}(|occ(P, S)|)$ time.

As an example, consider matching pattern $P = \texttt{AC}$ to the running example in Figure 2.1. The algorithm first finds the edge with label `A` going down from the root and then continues down the tree along the edge labeled `CA`. After matching the character `C`, the algorithm decides that P is a substring of S. Furthermore, $occ(P, S) = \{0, 4\}$ and thus $P = S[0..1] = S[4..5]$. This classical example already demonstrates the true power of suffix trees: the time complexity for matching k patterns of length m to a string of length n is $\mathcal{O}(n + km)$. String matching algorithms that preprocess pattern P instead of string S (Boyer-Moore [30] and Knuth-Morris-Pratt [131], among others) require $\mathcal{O}(k(n + m))$ time to solve the same problem. Since k and n are usually very large in most bioinformatics applications, for example in mapping millions ($= k$) short ($= m$) reads to the human genome ($= n$), this speedup is significant.

Bottom-up traversal

Bottom-up traversals through suffix trees are mainly required for the detection of highly similar patterns, such as common substrings or (approximate) repeats. This follows from the fact that internal nodes of $\text{ST}(S)$ represent the LCP of suffixes in their subtree. Internal nodes with maximal string depth correspond to

suffixes with the largest LCP, which makes it easy to find maximal repeats and longest common prefixes using a full bottom-up search of ST(S).

In detail, the longest common substring of two strings S_1 and S_2 of lengths n_1 and n_2 is found by first building a suffix tree for the concatenated string $S_1\$S_2$, called a *generalized suffix tree* (GST), and then traversing the GST twice. In GSTs, special characters, such as \$, are used to separate the suffixes of the individual strings. During an initial top-down traversal, string depths are stored at the internal nodes (if this information is gathered during construction of ST(S_1S_2), the top-down traversal can be skipped). A consecutive bottom-up traversal determines whether leaves in the subtree of an internal node all originate from S_1, S_2 or both. This information can percolate up to parent nodes. In case leaves from both S_1 and S_2 have the current node as their ancestor, the corresponding suffixes have a common prefix. Because every internal node is visited at most once during each traversal, and calculations at every internal node can be done in constant time, this algorithm requires $\mathcal{O}(n_1 + n_2)$ time. The details of the algorithm can be found in [97].

Maximal repeats, such as calculated in *Vmatch* [136], are found in a similar fashion. Labels of the internal nodes of ST(S) represent all repeated substrings that are right-maximal. There are, however, node labels that correspond to repeats that are not left-maximal. Similar to finding the longest common substring, a bottom-up traversal of ST(S) uses information in the leaves to check left-maximality and forwards this information to parent nodes.

As an example, the maximal repeats in the running example (Figure 2.1) are `ACA`, `AT`, `A` and `T`. The first internal node v visited by a bottom-up traversal has $\ell(v) =$ `ACA` and v has two leaves: 0 and 4. Because leaf 0 is a child of v, left-maximality is guaranteed for v and every parent of v. The internal node w with label $\ell(w) =$ `CA` has leaves 5 and 1 as children, but because $S[5-1] = S[1-1] =$ `A`, $\ell(w) =$`CA` is not a maximal repeat.

Suffix link traversal

A final way of traversing suffix trees is by following suffix links. Suffix links can both be used in suffix tree construction and algorithms for searching MEMs or matching statistics. Intuitively, suffix links maintain a sliding window when matching a pattern to the suffix tree. Furthermore, suffix links act as a memory-efficient alternative to generalized suffix trees. Because constructing, storing and updating suffix trees is a costly operation, the utilization of suffix links offers an important trade-off. The following algorithm demonstrates how suffix links enable

a quick comparison between all suffixes of string S_1 and the suffix tree $\mathrm{ST}(S_2)$ of another string S_2. Suppose the first suffix $S_1[0..]$ has been compared up to a node v with $\ell(v) = S_2[0..i]$. After following $sl(v) = w$, the second suffix $S_1[1..]$ is already matched to $\mathrm{ST}(S_2)$ up to w, with $\ell(w) = S_2[1..i]$. In this way, $i = |\ell(w)|$ characters do not have to be matched again for this suffix. This process can be repeated until all suffixes of S_1 are matched to $\mathrm{ST}(S_2)$. Hence, the MEMs between S_1 and S_2 can be found again in $\mathcal{O}(n_1 + n_2)$ time, but using less memory to store only the suffix tree of S_2 plus its suffix links.

Memory requirements

Given enough fast memory, suffix trees are probably the best data structure ever invented to support string algorithms. For large-scale bioinformatics applications, however, memory consumption really becomes a bottleneck. Although the memory requirements of suffix trees are asymptotically linear, the constant factor involved is quite high, *i.e.* up to ten [84] to twenty times [182] higher than the amount of memory required to store the input string. However, state-of-the-art suffix tree implementations are able to handle sequences of human chromosome size [137]. During the last decade, a lot of research focused on tackling this memory bottleneck, resulting in many suffix tree variants that show interesting memory versus time trade-offs.

2.1.2 Suffix arrays

The most successful and well-known variants of suffix trees are the so-called suffix arrays [176]. They are made up of a single array containing a permutation of the indexes of string S, making them extremely simple and elegant. In terms of performance, expressiveness is traded for lower memory footprint and improved locality. Suffix arrays in general only require four times the amount of storage needed for the input string, can be constructed in linear time and can exactly match all occurrences of pattern P in string S in $\mathcal{O}(m \log n + |occ(P, S)|)$ time using a binary search.

Definition 2.2. *Suffix array $SA(S)$ stores the lexicographical ordering of all suffixes of string S as a permutation of its index positions:*

$$S[SA[i-1]..] < S[SA[i]..], \quad 0 < i < n.$$

The last column of Table 2.1 shows the lexicographical ordering for the running example. $\mathrm{SA}(S)$ itself can be found in the second column. The uniqueness of

the lexicographical order is determined by the fact that all suffixes have different lengths, and the use of the special end-character \$ $< c$, $c \in \Sigma$. By definition, S[SA[0]] always equals the string \$. The relationship between suffix trees and suffix arrays becomes clear when traversing suffix trees depth-first and giving priority to edges with lexicographically smaller labels. Leaf numbers encountered in this order spell out the suffix array. All edges were lexicographically ordered on purpose in Figure 2.1, so that leaf numbers, read from left to right, form SA(S) as found in Table 2.1.

Exact matching of substring P is done using two binary searches on SA(S). These binary searches locate $P_L = \min\{k|P \leq S[\ \mathrm{SA}[k]]\}$ and $P_R = \max\{k|P \geq S[\ \mathrm{SA}[k]]\}$, which form the boundaries of the interval in SA(S) where $occ(P, S)$ is found. Note that finding the boundaries P_L and P_R requires $\mathcal{O}(m \log n)$ time, but finding $occ(P, S)$ only requires an additional $\mathcal{O}(|occ(P, S)|)$ time.

Although conceptually simple, suffix arrays are not just reduced versions of suffix trees [94, 200]. Optimal solutions for complex string processing problems can be achieved by algorithms on suffix arrays without simulating suffix tree traversals. An example is the all pairs suffix-prefix problem in which the maximal suffix-prefix overlap between all ordered pairs of k strings of total length n can be determined by both suffix trees [97] and suffix arrays [200] in $\mathcal{O}(n + k^2)$ time.

2.1.3 Enhanced suffix arrays

Suffix arrays are not that information-rich compared to suffix trees, but require far less memory. They lack LCP information, constant time access to children and suffix links, which makes them less fit to tackle more complex string matching problems. Abouelhoda *et al.* [3] demonstrated how suffix arrays can be embellished with additional arrays to recover the full expressivity of suffix trees. These so-called *enhanced suffix arrays* consist of three extra arrays that, together with a suffix array, form a more compact representation of suffix trees that can also be constructed in $\mathcal{O}(n)$ time. Furthermore, the next paragraphs demonstrate how the extra arrays of enhanced suffix arrays enable efficient simulation of all traversal types of suffix trees [3].

A first array LCP(S) supports bottom-up traversals on suffix array SA(S).

Definition 2.3. *The array LCP(S) stores LCP lengths of consecutive suffixes from the suffix array:*

$$LCP[i] = |LCP(S[SA[i-1]..],\ S[SA[i]..])|, \quad 0 < i < n.$$

By definition, $LCP[0] = -1$.

Table 2.1: Arrays used by enhanced suffix arrays (columns 2 to 5), compressed suffix arrays (columns 2, 6 and 7) and FM-indexes (columns 8 to 14) for string $S =$ `ACATACAGATG$`. From left to right: index position, suffix array, LCP array, child array, suffix link array, inverse suffix array, Ψ-array, BWT text, *rank* array, LF-mapping array and suffixes of string S. FM-indexes also require an array $C(S)$.

			ESA		CSA		FM-index							
								rank						
i	SA	LCP	*child*	*sl*	SA^{-1}	Ψ	BWT	$	A	C	G	T	LF	S[SA[i]..]
0	11	-1			2	2	G	0	0	0	1	0	8	$
1	4	0	6	[0..11]	7	6	T	0	0	0	1	1	10	ACAGATG$
2	0	3	2	[6..7]	4	7	$	1	0	0	1	1	0	ACATACAGATG$
3	6	1	4	[0..11]	10	9	C	1	0	1	1	1	6	AGATG$
4	2	1	5		1	10	C	1	0	2	1	1	7	ATACAGATG$
5	8	2	3	[10..11]	6	11	G	1	0	2	2	1	9	ATG$
6	5	0	8		3	3	A	1	1	2	2	1	1	CAGATG$
7	1	2	7	[1..5]	9	4	A	1	2	2	2	1	2	CATACAGATG$
8	10	0	10		5	0	T	1	2	2	2	2	11	G$
9	7	1	9	[0..11]	11	5	A	1	3	2	2	2	3	GATG$
10	3	0			8	1	A	1	4	2	2	2	4	TACAGATG$
11	9	1	11	[0..11]	0	8	A	1	5	2	2	2	5	TG$

An example LCP array for the running example is shown in the third column of Table 2.1. Originally, Manber and Myers [176] utilized LCP arrays to speed up exact substring matching on suffix arrays to achieve an $\mathcal{O}(m+\log n+|occ(P,S)|)$ time bound. Recently, Grossi [94] proved that the $\mathcal{O}(m+\log n+|occ(P,S)|)$ time bound for exact substring matching can be reached by using only S, SA(S) and $\mathcal{O}(\frac{n}{\log^2 n})$ sampled LCP array entries. Furthermore, it is possible to encode those sampled LCP array entries inside a modified version of SA(S) itself. However, the details of this technique are rather technical and fall beyond the scope of this review. Later, Kasai *et al.* [124] showed how all bottom-up traversals of suffix trees can be mimicked on suffix arrays in linear time by traversing LCP arrays. In fact, LCP(S) represents the tree topology of ST(S).

Recall that internal nodes of suffix trees group suffixes by their longest common prefixes. In enhanced suffix arrays, internal nodes are represented by *LCP intervals* ℓ-$[i..j]$.

Definition 2.4. *An interval* ℓ-$[i..j]$, $0 \leq i < j < n$ *is an LCP interval with LCP value* ℓ *if*

- $LCP[k] \geq \ell$, $\forall k, i < k \leq j$
- $\exists k, i < k \leq j$: $LCP[k] = \ell$
- $LCP[i] < \ell$
- $LCP[j+1] < \ell$

The LCP interval 0-$[0..n-1]$ *is defined to correspond to the root of* $ST(S)$.

Intuitively, an LCP interval is a maximal interval of LCP length that corresponds to an internal node of ST(S). As an illustration, LCP interval 1-[1..5] with LCP value 1 of the example string S in Table 2.1 corresponds to internal node v with label $\ell(v) = \texttt{A}$ in Figure 2.1. Similarly, subinterval relations among LCP intervals relate to parent-child relationships in suffix trees.

Abouelhoda *et al.* [3] have shown that the boundaries between LCP subintervals of LCP interval ℓ-$[i..j]$ are given by the *ℓ-indexes* for which it holds that LCP$[k] = \ell$, $i < k \leq j$. Singleton intervals correspond to leaves in the suffix tree and non-singleton intervals correspond to internal nodes. Consider, for example, the LCP interval 1-[1..5] in the running example. Its ℓ-indexes are 3 and 4. The resulting subintervals are LCP intervals 3-[1..2] and 2-[4..5] and singleton interval [3..3]. The above definitions thus generate a virtual suffix tree called the *LCP*

interval tree. Note that the topology of this tree is not stored in memory, but is traversed using the arrays SA(S) and LCP(S).

Fast top-down searches of suffix trees not only require their tree topology, but also constant time access to child nodes. This means constant time access to the ℓ-indexes of LCP intervals. This information can be precomputed in linear time for the entire LCP interval tree and stored in another array of enhanced suffix arrays, the *child array*.

Definition 2.5. *The array child(S) stores the ℓ-indexes of LCP intervals ℓ-[i..j]. The first ℓ-index is either stored in i or j. If $i < child[j] \leq j$, the value child[j] corresponds to the first ℓ-index of ℓ-[i..j]. Otherwise this value is found in child[i]. The next ℓ-indexes are stored at the location of the previous ℓ-indexes.*

The child array for the running example is given in the fourth column of Table 2.1. As an example, again consider LCP interval 1-[1..5]. The first ℓ-index (3) is stored at position 5 and the second ℓ-index (4) is stored at position 3. Because $child[4] = 5$ is equal to the right boundary of the interval (which cannot equal ℓ by definition), 4 is the last ℓ-index. The child array allows enhanced suffix arrays to simulate top-down suffix tree traversals.

As a final step towards complete suffix tree expressiveness, suffix arrays can be enhanced with *suffix link arrays* that store suffix links as pointers to other LCP intervals. These pointers are stored at the position of the first ℓ-index of an LCP interval because no two LCP intervals share the same position as their first ℓ-index [3]. This property and the suffix link array for the running example can be checked in Table 2.1.

With three extra arrays added, enhanced suffix arrays support all operations and traversals on suffix trees using the same time complexity. However, the simple modular structure allows memory savings if not all traversals are required for an application. Furthermore, array representations generally show better locality than most standard suffix tree representations, which is important when converting the index to disk, but also improves cache usage in memory [93]. Practical implementation improvements have further reduced memory consumption [74] of enhanced suffix arrays and have speeded up substring matching for larger alphabets [129]. In practice, several state-of-the-art bioinformatics tools make use of enhanced suffix arrays for finding repeated structures in genomes (*Vmatch*), short read mapping [102] and genome assembly [100]. If memory is a concern, enhanced suffix arrays occupy about the same amount of memory as regular suffix trees and are thus equally inapplicable for large strings. Suffix arrays (without enhancement) are preferred for exact substring matching in very large strings.

2.1.4 Compressed suffix arrays

Although suffix arrays are much more compact than suffix trees, their memory footprint is still too high for extremely large strings. The main reason stems from the fact that suffix arrays (and suffix trees) store pointers to string positions. The largest pointer takes $\mathcal{O}(\log n)$ bits, which means that suffix arrays require $\mathcal{O}(n \log n)$ bits of storage. This is large compared to $\mathcal{O}(n \log |\Sigma|)$ bits needed for storing uncompressed strings. A demand for smaller indexes that remain efficient gave rise to the development of *succinct indexes* and *compressed indexes*. Succinct indexes require $\mathcal{O}(n)$ bits of space, whereas the memory requirements of compressed indexes is in the order of magnitude of the compressed string [192].

Many types of compressed suffix arrays [96] have already been proposed (see Navarro and Mäkinen for a recent review [192]). They are usually centered around the idea of storing suffix array samples, complemented with a good compressible *neighbor array* $\Psi(S)$. To understand the role of the array $\Psi(S)$, the concept of *inverse suffix arrays* $\mathrm{SA}^{-1}(S)$ is introduced, for which holds that

$$\mathrm{SA}^{-1}[\mathrm{SA}[i]] \equiv \mathrm{SA}[\mathrm{SA}^{-1}[i]] = i.$$

$\Psi(S)$ can then be defined as

$$\Psi[i] \equiv \ \mathrm{SA}^{-1}[\mathrm{SA}[i] + 1 \mod (n-1)], \quad 0 \leq i < n.$$

This definition closely resembles that of suffix links and it will thus come as no surprise that in practice Ψ can be used to recover suffix links [219]. Consequently, the array Ψ can be used to recover suffix array samples from a sparse representation of $\mathrm{SA}(S)$. This is illustrated using the running example string from Table 2.1. Assume that only SA[0], SA[6] and SA[11] are stored and that the value of SA[10] is unknown. Note that $\Psi[10] = 1$ and $\mathrm{SA}[1] = 4 = 3 + 1$, *i.e.* the requested value plus one. A sampled value of $\mathrm{SA}(S)$ is reached by repeatedly calculating $\Psi[\Psi[\ldots\Psi[10]]] = \Psi^k[10]$. In the example $k = 2$, because $\Psi[\Psi[10]] = 6$. Consequently, $\mathrm{SA}[10] = \ \mathrm{SA}[6] - k = 5 - 2 = 3$. A more detailed discussion about compressed suffix arrays is given in Section 2.2.

2.1.5 The Burrows-Wheeler transform

Several compressed index structures, most notably the FM-index [70], are based on the Burrows-Wheeler transform [34] $\mathrm{BWT}(S)$. This reversible permutation of the string S is also known to lie at the core of compression tools such as *bzip2*.

The Burrows-Wheeler transform itself does not compress a string, rather it enables an easier and stronger compression of the original string by exploiting regularities found in the string. Unlike SA(S) that is a permutation of the index positions of S, BWT(S) is a permutation of the characters of S. As a result, BWT(S) only occupies $\mathcal{O}(n \log |\Sigma|)$ bits of memory in contrast to $\mathcal{O}(n \log n)$ bits needed for storing SA(S). Because it contains the original string itself, the Burrows-Wheeler transform does not require an additional copy of S for string searching algorithms. Index structures having this property are called *self-indexes*.

Intuitively, the Burrows-Wheeler transformation orders the characters of S by the context following the characters. Thus, characters followed by similar substrings will be close together.

Definition 2.6. *Let M be a $n \times n$ matrix whose rows are formed by the characters of the lexicographically sorted n cyclic shifts of a string S. $BWT(S)$ is the string represented by the last column of M, or $BWT[i] \equiv M[i, n-1]$, $0 \leq i < n$.*

The rows of M also represent the suffixes of S in suffix array order. Thus, the first column of M equals the first characters of the suffixes in suffix array order. $BWT(S)$ can be defined using $SA(S)$ as

$$BWT[i] \equiv S[SA[i] - 1 \mod n], \quad 0 \leq i < n,$$

where the modulo operator is used for the case $SA[i] = 0$.

From Definition 2.6 it immediately follows that BWT(S) can be constructed in linear time using SA(S). BWT(S) for the running example can be found in Table 2.1, column 8, while the complete matrix M is given in Table 2.2.

The inverse transformation that reconstructs S from BWT(S) is key to uncompression algorithms and the string matching algorithm utilized in compressed index structures. It recovers S back-to-front and is based on a few simple observations. First, although BWT(S) only stores the last column of M, the first column of M is easily retrieved from BWT(S) because it is the lexicographical ordering of the characters of S (and thus also BWT(S)). Moreover, the first column of M can be represented in compact form as an array $C(S)$ that stores the number of characters in S that are lexicographically smaller than character $c \in \Sigma$. More precisely:

$$C[c] \equiv \sum_{c_i < c} |occ(c_i, S)|, \quad c_i \in \Sigma.$$

For the running example, $C(S) = [0, 1, 6, 8, 10]$ can be retrieved from Table 2.2.

Table 2.2: Conceptual matrix M containing the lexicographically ordered n cyclic shifts of S = `ACATACAGATG$`. $M[0..11, 0]$ contains the lexicographically ordered characters of S and $M[0..11, 11]$ equals BWT(S). The last two columns are required for the inverse transformation. *offset*[i] stores the number of times BWT[i] has appeared earlier in BWT(S). The last column LF[i] contains pointers used during the inverse transformation algorithm: if $S[i] =$ BWT[j], then BWT[LF[j]] = $S[i-1]$.

i	S[SA[i]]		BWT[i]	*offset*[i]	LF[i]
0	`$`	`ACATACAGAT`	`G`	0	8
1	`A`	`CAGATG$ACA`	`T`	0	10
2	`A`	`CATACAGATG`	`$`	0	0
3	`A`	`GATG$ACATA`	`C`	0	6
4	`A`	`TACAGATG$A`	`C`	1	7
5	`A`	`ATG$ACATAC`	`G`	1	9
6	`C`	`AGATG$ACAT`	`A`	0	1
7	`C`	`ATACAGATG$`	`A`	1	2
8	`G`	`$ACATACAGA`	`T`	1	11
9	`G`	`ATG$ACATAC`	`A`	2	3
10	`T`	`ACAGATG$AC`	`A`	3	4
11	`T`	`G$ACATACAG`	`A`	4	5

A second observation is that BWT(S) stores the order of characters preceding the suffixes in suffix array order. As a result, if the character at position i ($S[i]$) has been decoded and the lexicographical order of suffix $S[i..]$ is known to be j, character $S[i-1]$ is found in BWT[j].

Finally, the most important observation that allows for the retrieval of S from BWT(S) is that identical characters preserve their relative order in the first and last columns of M. To see the correctness of this observation, let BWT[i] = BWT[j] = c for $i < j$. The lexicographical ordering of the cyclic permutations means that the suffix in row i of M corresponding to SA[i] is lexicographically smaller than the suffix in row j corresponding to SA[j]. From cS[SA[i]..] < cS[SA[j]..] it then follows that the location of character c corresponding to BWT[i] precedes the location of character c corresponding to BWT[j] in the first column of M.

The relative order of identical characters in BWT(S) is captured in the array *offset*(S): *offset*[i] stores the number of times that character BWT[i] occurs in

BWT(S) before position i, *i.e.*

$$\mathit{offset}[i] \equiv |\mathit{occ}(\mathrm{BWT}[i], \mathrm{BWT}[..i-1])|, \quad 0 < i < n.$$

Given a position i in BWT(S), the corresponding character in the first column of M can then be found at position $\mathrm{LF}[i] = C[\mathrm{BWT}[i]] + \mathit{offset}[i]$. The array LF($S$) is called the *last-to-first column mapping*.

The above observations allow the back-to-front recovery of S from BWT(S) utilizing a zig-zag algorithm. Starting in row i_0 of BWT(S) containing character `$`, the position of the previous character of S is found in row $\mathrm{LF}[i_0] = i_1$. The next preceding character is found on row $i_2 = \mathrm{LF}[i_1]$ in BWT(S), and so on. Thus, to find the row of the next preceding character, the algorithm looks horizontally in Table 2.2 and the actual character is retrieved from the BWT column on that row in Table 2.2. Note that neither M nor its first column are ever used explicitly during the algorithm. They only serve to understand the procedure for the inverse transformation.

In practice, $C(S)$ and $\mathit{offset}(S)$ are first constructed from BWT(S). During each step, $\mathrm{LF}[i_k]$ is calculated using $C(S)$ and $\mathit{offset}(S)$ and $\mathrm{BWT}[\mathrm{LF}[i_k]]$ is returned as the preceding character.

As an example, M, $\mathit{offset}(S)$ and LF(S) for the running example can be found in Table 2.2 and $C(S) = [0, 1, 6, 8, 10]$. $S[\mathrm{SA}[0]] =$ `$` is preceded by the character $\mathrm{BWT}[i_0] =$ `G` in the running example. Consequently, `G$` is the lexicographical first suffix that starts with `G`, which translates into $\mathit{offset}[i_0] = 0$. The first row of M whose corresponding suffix starts with `G` has row number $C[\mathtt{G}] = 8$. Adding the number of suffixes that also start with `G`, but are lexicographically smaller than `G$` ($= 0$), returns the position in BWT(S) of the next character that will be decoded. $\mathrm{BWT}[8 + 0] = \mathrm{BWT}[\mathrm{LF}[0]] =$ `T` $= S[9]$. In the next step, $S[8]$ is retrieved by computing $\mathrm{LF}[8] = 11$ and $\mathrm{BWT}[11] =$ `A`. Eventually, S is retrieved in $\mathcal{O}(n)$ time using the LF-mapping.

The Burrows-Wheeler transform by itself only permutes strings without compressing them. It is however easier to compress BWT(S) than the original string S, as the order of the characters in BWT(S) is determined by similar contexts following the characters, analogous to the way suffixes are grouped by longest common prefixes in suffix trees. An immediate consequence is that run-length encoding, which encodes runs of identical characters by their length, shows good compression results for BWT(S). Apart from run-length encoding [70,171], move-to-front lists [70], wavelet trees [72,171,192] and several entropy encoders, such as Huffman codes [69, 91], have also been used successfully to compress

BWT(S). For a complete overview on compression techniques based on the Burrows-Wheeler transform, we refer to the book of Adjeroh *et al.* [5].

Analogous to suffix arrays, BWT(S) can be used to find exact matches of substrings by applying binary search. Similar to compressed suffix arrays, binary searching BWT(S) requires auxiliary data structures, including $\Psi(S)$ and (sampled) SA(S) [5], resulting in compressed suffix arrays. Given the relation between BWT(S) and SA(S), BWT(S) can also be utilized for constructing other compressed suffix arrays [107]. Moreover, suffix trees, suffix arrays and other non-self-indexes require a copy of the indexed string S, which can be replaced by a compressed form of BWT(S) to reduce space.

2.1.6 FM-indexes

Another search method for exact string matching can be applied to Burrows-Wheeler transformed strings, using ideas from the inverse transformation algorithm. This method is referred to as *backward searching* and forms the basic search mechanism of *FM-indexes* [70]. FM-index is the short name given by Ferragina and Manzini to their *f*ull-text self-indexes that require "*m*inute amount of space". The space requirement is proportional to and sometimes even smaller than that of the indexed string. FM-indexes can be constructed in $\mathcal{O}(n)$ time and all occurrences of pattern P can be located in $\mathcal{O}(m+|occ(P,S)|\log n)$ time. Note that finding $|occ(P,S)|$ only requires $\mathcal{O}(m)$ time, which makes that FM-indexes have theoretical optimal time and space requirements for counting the number of occurrences of a pattern in a string.

The backward search algorithm employed by FM-indexes requires BWT(S), $C(S)$ and a two-dimensional $n \times |\Sigma|$ array $rank(S)$[1]. This array is defined as

$$rank[i,c] \equiv |occ(c, \mathrm{BWT}[..i])|, \quad 0 \leq i < n,\ c \in \Sigma.$$

For the running example, $rank(S)$ is shown as columns 9 to 13 in Table 2.1. The role of $rank(S)$ is similar to the role $\mathit{offset}(S)$ plays in the inverse transformation of BWT(S). However, while $\mathit{offset}(S)$ only stores information on the number of occurrences of one character for each index position, $rank(S)$ contains this information for all the characters in the alphabet in all index positions. The extra information contained in $rank(S)$ compared to $\mathit{offset}(S)$ gives it the advantage of granting random access to LF(S). Furthermore, $rank(S)$ is easier to compress than $\mathit{offset}(S)$ or LF(S) [5].

[1] In many papers, $rank(S)$ is referred to as $Occ(S)$, but to avoid confusion with $occ(P,S)$, the name *rank* is used.

During the course of the search algorithm, P is matched from right to left. For every step i, $0 \leq i < m$, an interval BWT$[s_i..e_i]$ is maintained that contains all occurrences of $P[m-i..]$. Initially, $[s_0..e_0] \equiv [0..n-1]$, and after m steps $[s_m..e_m]$ contains the suffix array interval corresponding to $occ(P, S)$. Given $[s_i..e_i]$ and $c = P[m-i-1]$, the next interval is found using the formulas

$$s_{i+1} = C[c] + rank[c, s_i - 1] \text{ and } e_{i+1} = C[c] + rank[c, e_i + 1] - 1.$$

Here, array $C(S)$ is used to locate the interval of suffixes starting with c in SA(S) and array $rank(S)$ is used to find the number of suffixes starting with c that are lexicographically smaller and larger than the ones prefixed by $cP[m-i..]$.

As an example of backward searching, again consider matching $P =$ `CA` to the running example in Table 2.1. Initially, the backward search interval is [0..11]. Because $C[\texttt{A}] = 1$ and $C[\texttt{C}] = 6$, the backward search interval narrows down to $[s_1..e_1] = [1..5]$ in the next step, which corresponds to the suffix array interval containing suffixes starting with `A`. Note that BWT[3] = BWT[4] = `C`, so there are two suffixes starting with `A` that are preceded by `C`. Consequently, $s_2 = C[\texttt{C}] + rank[0, \texttt{C}] = 6+0 = 6$ and $e_2 = C[\texttt{C}] + rank[5, \texttt{C}] - 1 = 6+2-1 = 7$. The answer $|occ(P, S)| = 7-6+1 = 2$ is found in $\mathcal{O}(m)$ time. $rank[0, \texttt{C}] = 0$ means that there are no suffixes starting with `C` located in SA[0..0] and $rank[5, \texttt{C}] = 2$ means that there are 2 suffixes starting with `C` located in SA[0..5].

Also note the resemblance between LF-mapping and backward search: s_2 also could have been found as the first occurrence of `C` in BWT[1..5], which is 3: LF[3] $= 6 = s_2$. Likewise, e_2 could have been found as the last occurrence of `C` in BWT[1..5]. However, instead of locating these occurrences, note that $offset[3] = rank[3, \texttt{C}] - 1 = rank[1, \texttt{C}] - 1$. Thus, the $offset(S)$ values are stored in $rank(S)$ at the boundaries of every interval, allowing search intervals to be narrowed down in constant time. As a result, the reverse search algorithm of the FM-index simulates a top-down search in a suffix *trie*, *i.e.* a suffix tree where every edge label contains only a single character.

After backward searching has terminated, $occ(P, S)$ is still unknown. Using LF-mapping, this set can be retrieved from the interval BWT$[s_m..e_m]$. One possibility is to count the number of backward searches it takes to reach character \$ for every $s_m \leq i \leq e_m$. However, this would require too much time. To achieve better performance, FM-indexes mark additional positions with suffix array values in BWT(S). The number of suffix array values stored constitutes a time-space trade-off. Recall that LF$[i]$ returns the position in SA(S) of suffix $S[\text{SA}[i] - 1..]$.

Thus $\mathrm{SA}[\mathrm{LF}[i]] = \mathrm{SA}[i] - 1$, such that $\mathrm{LF}(S)$ can be used to find the next smaller suffix array value. The ability of $\mathrm{LF}(S)$ to find smaller suffix array values is used as an argument to classify FM-indexes as compressed suffix arrays [70]. Moreover, $\mathrm{LF}(S)$ and $\Psi(S)$ are each others' inverse: $\mathrm{SA}[\mathrm{LF}[i]] = \mathrm{SA}[i] - 1$ and $\mathrm{SA}[\Psi[i]] = \mathrm{SA}[i] + 1$, hence $\mathrm{LF}[\Psi[i]] = \Psi[\mathrm{LF}[i]] = i$.

FM-indexes combine fast string matching with low memory requirements. Their original design [70] compresses $\mathrm{BWT}(S)$ using move-to-front lists, run-length encoding and a variable-length prefix code. In the original paper, $rank(S)$ was compressed using the *Four-Russians* technique [10]. Roughly speaking, this technique comes down to subdividing the problem into small enough subproblems and indexing all solutions to these small problems in a global table. The subdivision into smaller subproblems is done by recursively splitting arrays into equally sized blocks and storing answers to queries relative to the larger parent block. Other compression methods have been proposed that show better performance in practice [69] or that give different space-time trade-offs [66, 71, 72, 91, 171].

Because they allow fast pattern matching while having small memory requirements, FM-indexes have become a very popular tool for different types of genome analyses. Compressed full-text index structures are mainly used for exact string matching, but algorithms for inexact string matching exist [5, 214]. FM-indexes have started to become used as part of *de novo* genome assembly algorithms [230] and are supporting popular tools for mapping reads to reference genomes such as *Bowtie* [143], *BWA* [150] and *SOAP2* [157].

2.2 Time-memory trade-offs

The increase in sequencing data requires efficient algorithms and data structures to form the backbone of computational tools for storing, processing and analyzing these sequences. Without the use of index structures, many algorithms that rely on string searching would become unfeasible due to a long execution time. However, index structures also incur a memory overhead to sequence analysis.

Over the last decade, much energy has been put into decreasing the memory consumption of index structures. The proposals differ in the performance overhead incurred by lowering the memory footprint. Some index structures suffer from a logarithmic slowdown, while others allow for tuning the space-time trade-off. There are indexes that have been especially designed for certain types of data, whereas others are tweaked for particular hardware architectures. An example of a data-specific property influencing index structure performance is

the alphabet size of the sequences.

Another major factor that allows classifying index structures is their expressiveness. Suffix trees are considered to have full expressiveness [97], supporting a large variety of string algorithms. Conversely, the bulk of recent compressed self-index structures are limited to performing mainly (in)exact string matching. These string matching self-indexes are often compared on the basis of four criteria: their performance of extracting a random substring of S, calculating $|occ(P, S)|$ and $occ(P, S)$ and their size.

An overview of the memory taken by several index structures discussed in this section can be found in Table 2.3. This table represents memory requirements both in general terms of number of bits required per indexed character, as well as in terms of its size for indexing full genomes. Note, however, that the list of index structures in Table 2.3 is not complete nor gives a full overview of the memory-time trade-offs. For example, external memory index structures were omitted, but can be found in Section 2.3. Additionally, peak memory requirements during construction can be much higher than the figures described here (see Section 2.4). Furthermore, index structures contain parameters that allow manual tuning of the memory-time trade-off. Finally, because the expressiveness differs greatly between index structures, Table 2.3 does not include any time-related results. Partial results for some algorithms can be found elsewhere [13, 66, 93, 243].

The remainder of this section focuses on the basic principles behind these index structures and the memory-time trade-offs induced by design choices and confounding factors such as application and data types.

2.2.1 Uncompressed index structures

Choosing appropriate data structures for implementing the different components of suffix trees forms a basic step in lowering their memory requirements. These components include nodes, edges, edge labels, leaf numbers and suffix links. The topological information of $\mathrm{ST}(S)$ and the edge labels are traditionally stored as pointers, resulting in suffix trees that require $\mathcal{O}(n)$ words of usually 32 bits. Note that for very large strings ($n > 2^{32} \approx 4 \cdot 10^9$) 32 bits is insufficient for storing the pointers, thus larger representations are required. This factor is often overlooked when presenting theoretical results.

There is only one major $\mathcal{O}(|\Sigma|)$-sized memory-time trade-off in this traditional representation. This trade-off comes from the data structure that handles access to child vertices. Most implementations make use of — roughly ordered from high memory requirements to low access time — static arrays, dynamic arrays [93],

Table 2.3: Representative memory requirements for different index structure implementations, expressed both as bits per indexed character (column 2) and estimated size in megabytes for several known genomes (columns 3-5). Column 6 contains references to the original theoretical proposals and an additional reference to the articles from which these practical estimates originate. For ease of comparison, the index structures are sorted by increasing memory requirements. As a reference, the original (non-indexed) sequence is also included (bold), both stored using 2-bit encoding and byte encoding. [§]Genome sizes were taken from the NCBI genome information pages `http://www.ncbi.nlm.nih.gov/genome` of *Saccharomyces cerevisiae* (yeast), *Drosophila melanogaster* (fruit fly) and *Homo Sapiens* (human). [†]mean of the interval of possible memory requirements given in [35].

Name index structure	bits/ char	size for genome in MB yeast	fruit fly	human	reference
2-bit encoded string	2	3	35	775	NCBI[§]
CSA Grossi *et al.*	2.4	4	42	931	[78, 95]
FM-index	3.36	5	59	1302	[70, 93]
SSA (best)	4	6	70	1551	[13, 171]
CST Russo *et al.*[†]	5	8	87	1939	[35, 213]
CSA Sadakane (best)	5.6	8	98	2171	[218, 220]
LZ-index (best)	6.64	10	116	2574	[13]
byte encoded string	8	12	139	3102	NCBI[§]
CST Navarro[†]	12	18	209	4653	[35]
SSA (worst)	20	30	349	7754	[13, 171]
CST Sadakane[†]	30	45	523	11 632	[35, 219]
LZ-index (worst)	35.2	53	614	13 648	[93, 189]
suffix array	40	60	697	15 509	[176]
enhanced SA	72	109	1255	27 916	[3]
WOTD suffix tree	76	115	1325	29 467	[84]
ST McCreight	232	350	4045	89 952	[84, 182]

hash tables, linked lists and layouts with only pointers towards the first child and next sibling. Furthermore, mixed data structures that represent vertices with different numbers of children have also been proposed [135]. Note that for DNA sequences, $|\Sigma|$ is very small, turning array implementations into a workable solution. Also note that algorithms that perform full suffix tree traversals, such

as repeat finding and many other string problems [97], do not suffer from a performance loss when implemented with more memory-efficient data structures.

In practice, suffix trees and suffix arrays require between $34n$ and $152n$ bits of memory. The suffix tree implementations described by Kurtz [135] perform very well and are implemented in the latest release of *MUMmer* [137], an open-source sequence analysis tool. The implementation in *MUMmer* allows indexing DNA sequences up to 250Mbp on a computer with 4GB of memory. Single human chromosomes are thus well within reach of standard suffix trees. Another implementation by Giegerich *et al.* [84] is even smaller, but lacks suffix links. Enhanced suffix arrays [3] also reach full expressiveness of suffix trees, as described in Section 2.1. When carefully implemented, they require anything between $40n$ and $72n$ bits.

Enhanced suffix arrays use a linked list to represent the vertices of the tree. However, the $\mathcal{O}(|\Sigma|)$ performance penalty for string matching can be reduced to $\mathcal{O}(|log\Sigma|)$ [129]. Furthermore, enhanced suffix arrays form the basis of the *Vmatch* program that finds different types of exact and approximate repeats in sequences of several hundreds of Mbp in a few seconds. Moreover, according to a comparison between several implementations of suffix trees and enhanced suffix arrays [93], enhanced suffix arrays show the best overall performance for both the memory footprint and the traversal times. Finally, their modular design allows replacing some arrays by a compressed counterpart to further reduce space.

2.2.2 Sparse indexes

An intuitive solution for decreasing index structure memory requirements is sparsification or sampling of suffixes or array indexes. *sparse suffix trees* [123] and *sparse suffix arrays* [64] adopt the idea of utilizing a sparse set of suffixes, whereas compressed suffix arrays and trees sample values in $\Psi(S)$, $C(S)$, $rank(S)$ and other arrays involved in their design. As a consequence of sparsification, more string comparisons and sequential string searches are required. This, however, gives the opportunity to optionally tweak the size of the index structure based on the available memory.

We have combined techniques from enhanced suffix arrays and sparse suffix arrays to create a new index structure, called *enhanced sparse suffix array*, and used this index structure for finding MEMs between two sequences. This index structure will be presented in Chapter 3.

In general, sparse index structures have received less attention in bioinformatics applications than compressed index structures. However, sparse suf-

fix arrays have been successfully used for exact pattern matching, retrieval of MEMs [126, 248] and read alignment [133, 249]. Furthermore, splitting indexes over multiple sparse index structures has been used for index structures that reside on disk [19] and for distributed query processing [180].

Word-based index structures are special cases of sparse index structures which only sample one suffix per word. Although word-based index structures are most popular in the form of inverted files, word-based suffix trees [9, 115] and suffix arrays [64] also exist. Although it is possible to divide biological sequences into "words", word-based index structures are generally designed to answer pattern matching queries on natural language data. On natural language data, Transier and Sanders [239] found that inverted files outperformed full-text indexes by a wide margin. Unfortunately, the inverted files were not compared against word-based implementations of suffix trees and suffix arrays. A somewhat dual approach was taken by Puglisi *et al.* [206], who adapted inverted files to become full-text indexes able to perform substring queries. They found compressed suffix arrays to generally outperform inverted files for DNA sequences, but the opposite conclusion was drawn for protein sequences. It turns out that compressed suffix arrays perform relatively better compared to inverted files when searching for patterns having fewer occurrences. Note that both comparative studies were performed in primary memory.

2.2.3 Compressed index structures

Compressed and succinct index structures are currently the most popular forms of index structures used in bioinformatics. Index structures such as compressed suffix arrays and FM-indexes are gradually built into state-of-the-art read mapping tools and other bioinformatics applications. Where traditional index structures require $\mathcal{O}(n \log n)$ bits of storage, succinct index structures require $\mathcal{O}(n)$ bits and the memory footprint of compressed index structures is defined relative to the *empirical entropy* [178] of a string. Furthermore, these self-indexes contain S itself, thus saving again $\mathcal{O}(n)$ bits. Theoretically, this means that the size of compressed index structures can become a fraction of S itself. In practice, however, DNA and protein sequences do not compress very well [63, 133]. For this reason, the size of compressed index structures is roughly similar to the size of storing S using a compact bit representation. The major disadvantage of compressed index structures is the logarithmic increase in computation time for many string algorithms. This is, however, not the case for all string algorithms. For example, calculating $|occ(P, S)|$ can still be done in $\mathcal{O}(m)$ time for some compressed

indexes. These internal differences between compressed index structures result from their complex nature, as they combine ideas from classical index structures, compression algorithms, coding strategies and other research fields. In the following paragraphs, the conceptual differences of state-of-the-art compressed index structures are surveyed, illustrated with theoretical and practical comparisons wherever possible. A more technical review is found in [192].

Auxiliary data structures

Understanding the organization details and properties of compressed index structures requires prior knowledge of the auxiliary data structures involved in their design. Compressed indexes consist of many auxiliary structures that influence their memory-time trade-off, and have properties that dictate their expressiveness and performance for certain types of data. Representation of these auxiliary structures forms an active field of research. What follows is a brief summary of several commonly used auxiliary structures, not including the rather technical implementation details.

Almost all compressed index structures make use of bit vectors B to support random access and $rank(B)$ and $select(B)$ queries. Intuitively, $rank(B)$ queries count the number of zeros or ones before a certain index in the vector. Dual to this, $select(B)$ queries return the position in B of the i-th zero or one. They often play a role in granting random access to a compressed or permutated string. Their usefulness, however, goes further than being mere building blocks of compressed index structures. For example, they can also be used to succinctly represent de Bruijn graphs [44], a typical data structure used in *de novo* genome assembly.

Formally, $rank(B)$ is represented as a two-dimensional array defined by

$$rank[i, c] \equiv |occ(c, B[..i])|, \quad 0 \leq i < |B|, \ c \in \{0, 1\},$$

similar to $rank(S)$ for FM-indexes. $select(B)$ is defined as

$$select[i, c] \equiv j \text{ iff } i = rank[j, c], \quad 0 \leq i < |occ(c, B)|, \ c \in \{0, 1\}.$$

These data structures and their generalizations to non-binary strings strongly influence the memory-time trade-off of compressed index structures [72]. As an example, the array $rank(S)$ used in FM-indexes takes up to half of its size. As is the case for other data structures, there is no single optimal implementation for every application, but many proposals exist [40, 202, 209]. The performance also depends on the restrictions imposed by the compressed index structure or the

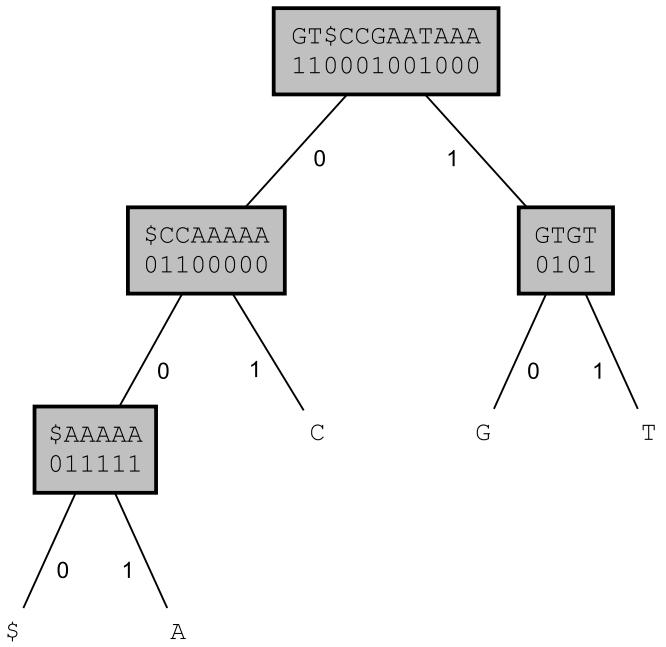


Figure 2.2: Wavelet tree for indexing string S = `GT$CCGAATAAA`. Only the binary strings are stored in practice. Subsequences of S are shown only to ease the interpretation. This figure does not include data structures for resolving rank and select queries for every bit vector. For this small example, however, the answer to these queries is straightforward.

properties of the data, such as the sparsity of the original bit vector. From extremely sparse to more balanced, the best implementations require $0.2n$ bits [202] (1% ones), $0.8n$ bits [40] (20% ones) and $1.4n$ bits [202] (50% ones).

The results for bit vectors have been generalized to non-binary strings [40, 72], as worked with in many applications, including FM-indexes. A simple idea towards such a generalization is to create $|\Sigma|$ bit vectors B_c, with $B_c[j] = 1$ iff $S[j] = c$. However, this entails an overhead both in time (random access to S) and memory. A careful implementation allows eliminating this overhead [72], but *wavelet trees* [95] form an even more elegant solution.

Wavelet trees are balanced binary trees with $|\Sigma|$ leaves. Every node v in

the tree represents a subsequence S' of S formed by the concatenation of all characters that belong to some interval $\Sigma[i..j]$. The two children of v are the subsequences formed by the concatenation of all characters of S' that belong to $\Sigma[i..\lceil\frac{i+j}{2}\rceil]$ and $\Sigma[\lceil\frac{i+j}{2}\rceil+1..j]$, respectively. Vertex v itself is represented by a bit vector B of size $|S'|$ that is defined as $B[i] = 0$ iff $S'[i] \in \Sigma[i..\lceil\frac{i+j}{2}\rceil]$. Furthermore, B is preprocessed as to resolve $rank(B)$ and $select(B)$ in constant time.

The wavelet tree for BWT(S) of the running example is shown in Figure 2.2, and has the same functionality as BWT(S) and $rank(S)$. From this figure, BWT[9] can be found as follows. The root bit vector learns that $B_\Sigma[9] = 0$, meaning that BWT[9] is a character from the first half of the alphabet. Because $B_\Sigma[9]$ is the sixth occurrence of 0 in B_Σ ($rank[9, 0] = 5$), it corresponds to $B_{\texttt{\$AC}}[5]$ (zero-based index). Repetition of this procedure for the vertices corresponding to $S_{\texttt{\$AC}} = \texttt{\$CCAAAAA}$ and $S_{\texttt{\$A}} = \texttt{\$AAAAA}$ yields BWT[9] = `A`. $rank(S)$ queries can be resolved in a similar way.

Further research on wavelet trees gave rise to non-binary wavelet trees [72] and Huffman-shaped wavelet trees [78]. This elegant, yet somewhat complex data structure, has become very popular in index structure design. As an example result, all maximal repeats occurring in the complete human genome could be found in less than 17 hours on a desktop PC [127] with 8 GB internal memory using an index structure based on the Burrows-Wheeler transform combined with a sparse wavelet tree implementation of the LCP array. Similar tests using suffix trees or enhanced suffix arrays failed due to the memory bottleneck.

Other important index structure building blocks are auxiliary tree representations. Index structures use various types of trees, but a common design problem is the representation of their topology. As an example, suffix tree topology is traditionally implemented using pointers, requiring $\mathcal{O}(n \log n)$ bits of memory. In contrast, a popular way to succinctly represent tree topology by a sequence of balanced parentheses [118] only requires $2n + o(n)$ bits of memory. This implementation represents nodes in the tree as a pair of parentheses '()'. The nested structure of the parentheses then represents the tree [11], similar to a reduced form of the known Newick Tree Format [59]. More tree operations are generally supported in constant or near-constant time by succinct tree topology representations compared to classical pointer-based representations, which only supports top-down traversals in constant time. Node depth, subtree size and the lowest common ancestor of two nodes [87] are examples of properties that can be retrieved in constant time from succinct representations where pointer-based representations require additional data structures to achieve the same performance.

In theory, this means that a highly expressive suffix tree topology can be stored using $4n+o(n)$ bits instead of $64n$ bits using 32-bit pointers. Note, however, that the $o(n)$-term may become large in practice and even surpass the higher order term. For biological sequences, tests [11] show that these representations require something in between 2.1 and 4.84 bits per node, which has to be multiplied by $2n$ nodes in the worst case.

Retrieval of the lowest common ancestor of two nodes v_1 and v_2, mentioned in the previous paragraph, is a fundamental operation for inexact string matching algorithms [97]. Denoted by $\mathrm{LCA}(v_1, v_2)$, it is defined as the unique node v_3 for which holds that $\ell(v_3) = \ell(\mathrm{LCA}(v_1, v_2)) \equiv \mathrm{LCP}(\ell(v_1), \ell(v_2))$. This operation is supported by a combination of LCP arrays and data structures for resolving range minimum queries [28]. This directly follows from the definition of $\mathrm{LCA}(v_1, v_2)$.

Range minimum query data structures return the positions of the smallest values in any interval of an array. In LCP arrays, they return the length of the longest common prefix of any two suffixes. Furthermore, range minimum query data structures can replace the child array in enhanced suffix arrays [74], because ℓ-indexes are the positions of minimal values in LCP intervals.

Compressed suffix arrays

Compression of suffix arrays is based on storing a sparse representation of $\mathrm{SA}(S)$ and storing $\Psi(S)$ in compressed form. $\Psi(S)$ has the property that it is increasing in areas of $\mathrm{SA}(S)$ that point to suffixes starting with the same character [96], which makes it compressible. The first real compressed suffix array was designed by Grossi and Vitter [96]. They built on a hierarchical decomposition of $\mathrm{SA}(S)$ that halves the size of $\mathrm{SA}(S)$ in every level by removing values pointing to odd suffixes and dividing even suffix array values by two. $\Psi(S)$ is stored in every level for odd suffix array values. $rank(S)$ and $select(S)$ data structures are used to retrieve the parity of suffixes on every level of the hierarchy and in an encoding of $\Psi(S)$ [96,216]. The number of levels stored in this representation is a parameter that tunes the memory-time trade-off.

Sadakane [218] further improved the above implementation by incorporating the compressed string into the index structure. A basic version of this self-index does not allow direct access to $\mathrm{SA}[i]$, but instead allows access to $S[\mathrm{SA}[i]]$, which is sufficient for pattern matching and finding $|occ(P, S)|$. Direct access to $\mathrm{SA}[i]$ and $\mathrm{SA}^{-1}[i]$ and random access to S is achieved by incorporating the hierarchical structure by Grossi and Vitter. Sadakane's compressed suffix array was implemented [220] and constructed for the human genome. The index required about

$5.6n$ bits of memory, resulting in an overall memory footprint of less than 2 GB. Additionally, Sadakane designed a backward search algorithm, similar to that used by FM-indexes, for counting patterns [217]. This strategy is much faster than the traditional binary search used by suffix arrays.

Other compressed suffix array designs incorporated wavelet trees [95]. In practice, an example implementation [78] required $2.4n$ bits of memory for real DNA sequences.

A different solution to lower the memory requirements of suffix arrays was used for *compact suffix arrays* [169]. Here, the compression is based on self-repetitions, so-called *runs*, in $\mathrm{SA}(S)$. These are suffix array intervals $[i..i+\ell]$ for which another interval $[j..j+\ell]$ exists such that $\mathrm{SA}[i+k] = \mathrm{SA}[j+k]+1$ for $0 \leq k \leq \ell$. In practice, compact suffix arrays take up more memory than existing compressed suffix arrays, but are also faster. It was shown that the number of self-repetitions in $\mathrm{SA}(S)$ is related to the number of equal-characters runs in $\mathrm{BWT}(S)$ [170], which can be compressed by run-length encoding. In terms of compression, however, this technique was superseded by other FM-indexes [171].

The above compressed suffix arrays are especially geared towards pattern matching. Some compressed index structures [192] are able to find $|occ(P,S)|$ in $\mathcal{O}(m)$ time, but in practice they all require at least $\mathcal{O}(|occ(P,S)| \log n)$ time for retrieving the actual occurrences of pattern P. Furthermore, locating the patterns requires a lot of random accesses to the index structures, resulting in degrading performance due to cache misses. This becomes even more severe when ported to secondary memory [89]. This also holds for FM-indexes, as discussed further. González and Navarro [89] designed locally compressed suffix arrays to cope with this problem. Their index structures are based on sampling exact suffix array values, differentially encoding $\mathrm{SA}(S)$ and encoding this array using dictionaries. However, these index structures are not self-indexes and have to be incorporated into existing compressed suffix arrays or FM-indexes. In practice, the speed for locating patterns is indeed much faster, even compared to the Lempel-Ziv index structures described further. However, their compression rate is not that high, as it requires up to 85% of the size of regular suffix arrays for DNA sequences and 70% for protein sequences.

A practical performance comparison between compressed suffix arrays and plain suffix arrays was made by Sadakane and Shibuya [220]. They both tested for the application of approximate string matching. Compressed suffix arrays required one sixth of the memory typically needed by plain suffix arrays, but were 2 to 20 times slower.

FM-indexes

As previously stated, FM-indexes are compressed full-text indexes based on the Burrows-Wheeler transform. Different memory-time trade-offs are reached for FM-indexes by using different techniques for compressing BWT(S) and *rank*(S). As a reminder, in the original proposal [70,71], BWT(S) is compressed by applying move-to-front transformation, run-length compression and a version of Elias-γ prefix codes [56]. *rank*(S) is encoded by cutting the array in blocks and using the Four-Russians technique. In the original practical implementation [69], the dictionary used for the Four-Russians technique is replaced by a linear scan of a bit vector.

The above representation of FM-indexes is heavily dependent on the alphabet size. A simple way to reduce this dependence is to use a wavelet tree over BWT(S) and use any representation of *rank*(S) for bit vectors in every internal node [192]. Huffman-shaped wavelet trees are used by *succinct suffix arrays* [171]. In a recent practical survey [66], this implementation shows the best known practical time-memory trade-offs for the most used basic operations on compressed index structures when applied to DNA and protein sequences. Although its memory footprint is somewhat higher ($4n$-$20n$ bits) than that of the standard FM-index, it is 20 times faster than its classical counterpart [93]. Compared to suffix trees, however, it is 20 times slower. There exist even smaller FM-indexes, such as *run-length FM-indexes* [171] that apply run-length compression to BWT(S) prior to building a wavelet tree. A more recent proposal by Ferragina *et al.* [72], the *alphabet-friendly FM-indexes*, theoretically supersedes all previous FM-index implementations. In practice [66], however, the alphabet-friendly FM-index is superseded by the succinct suffix array for biological sequences. Only for strings with a large alphabet and small high-order entropy (making them highly compressible), such as natural language strings or XML files, alphabet-friendly FM-indexes outperform other FM-indexes.

Another possibility for lowering the memory dependence of FM-indexes was explored by Grabowski *et al.* [91]. They first Huffman-encoded S and then applied the Burrows-Wheeler transform. They require sampling some characters from S additionally to the sampling of SA(S). Their best implementation slightly outperforms succinct suffix arrays on biological sequences and requires $3.28n$ bits of memory on average.

Note that locating patterns using FM-indexes is done by sampling suffix array values, which turns out to be rather slow in practice. A memory-time trade-off is imposed by the sampling rate. Improvements on the pattern locating performance

can be made by using more complex sampling strategies, different from basic evenly spaced sampling [66, 69]. An alternative is to incorporate another index structure that supports fast locating of patterns [71, 89].

Lempel-Ziv index structures

Similar to the above compressed full-text index structures, Lempel-Ziv index structures [122] are mainly designed for pattern matching. Unlike the above compressed index structures, however, Lempel-Ziv indexes are not based on suffix arrays or the Burrows-Wheeler transform. Instead, they build on the dictionary-based Lempel-Ziv [146] compression technique. Briefly, LZ78 [261] compression is achieved by traversing S and replacing substrings of S with tuples (w, c), where w is a word from the dictionary and $c \in \Sigma$. Assume that at some point, $S[..i-1]$ has been compressed and the next tuple in the compressed string is (w, c). w equals the code word for the longest prefix of $S[i..]$, say $S[i..j]$, that is already part of the dictionary and $c = S[j+1]$. Furthermore, $S[i..j+1]$ is added to the dictionary. Note that there are other variants of Lempel-Ziv compression, similar to the technique described here, which are omitted for the sake of brevity.

Details on the structure and search algorithms of Lempel-Ziv indexes are omitted, but can be found elsewhere [192]. What is important to note about their structure, however, is that Lempel-Ziv indexes contain many building blocks: compressed or sparse (suffix) tree data structures to compactly represent the dictionaries of forward and reverse code words, data structures for linking those trees and several other auxiliary data structures that answer $rank(S)$ queries and data structures to answer orthogonal range queries. As a direct consequence, further improvements in these building blocks will improve the performance of Lempel-Ziv indexes. Compared to other compressed index structures, Lempel-Ziv index structures require more memory than other self-indexes on average and they are not competitive for counting occurrences of patterns ($\mathcal{O}(m^2)$ time). They, however, excel at retrieving the exact set of all occurrences $occ(P, S)$.

Lempel-Ziv indexes have been turned into self-indexes by Navarro [189], who also designed an efficient implementation [190]. Further improvements in counting occurrences were made by Ferragina and Manzini [71], who attached FM-indexes to Lempel-Ziv indexes. Other approaches [15, 215] have minimized the redundancy caused by an overload of building blocks and have experimented with new auxiliary data structures. Recent tests [13, 66, 215] show that those new implementations have made Lempel-Ziv indexes more competitive compared to compressed suffix arrays and FM-indexes, but succinct suffix arrays are still reported

to have better memory-time trade-offs. In the near future, however, Lempel-Ziv indexes could outperform other indexes for highly compressible strings because all building blocks of Lempel-Ziv index structures can be compressed, while other compressed indexes contain sampled suffix array values, which are incompressible [15].

Compressed suffix trees

The above compressed index structures were mainly designed for exact string matching. As such, they do not reach the full expressiveness of suffix trees. Examples of this expressiveness have been previously given as illustration of the different traversal types of suffix trees. In recent years, efforts have been made to increase the flexibility of compressed index structures either by designing index-specific algorithms or by implementing additional auxiliary data structures. Analogous to enhanced suffix arrays, the main auxiliary data structures used for augmenting compressed suffix arrays are succinct representations of LCP arrays [217], data structures for top-down tree traversals and suffix link support. As an example, the combination of Burrows-Wheeler index structures and wavelet trees for succinct LCP arrays was used for locating all maximal repeats in the whole human genome [127]. Ohlebush *et al.* [201], among others, noted that the backward search mechanism mimics top-down suffix *trie* traversal. Using additional data structures to simulate suffix links, they calculated maximal exact matches between DNA sequences, using less memory than, for example, *MUMmer*.

Instead of developing application-specific compressed index structures, several *compressed suffix trees* [219] or *compressed enhanced suffix arrays* [199] have been designed that even surpass the expressiveness of classical suffix trees. Furthermore, because compressed suffix trees extend compressed self-indexes, they are self-indexes themselves. The difference between these data structures and the compressed suffix arrays and FM-indexes on which they are built, is their ability to directly implement suffix tree algorithms. Although the extra data structures increase their memory footprint, compressed suffix trees are still smaller than classical suffix arrays. Furthermore, space-time trade-offs can be tuned to a certain extent, similar to the sparsification parameter in compressed suffix arrays and FM-indexes.

Over the last years, several compressed suffix tree designs have been proposed. These can be classified by their choice of auxiliary data structures, especially the representation of the suffix tree topology [198]. They either use sequences of

balanced parentheses or implicit representation by LCP intervals. Additional building blocks are succinct representations of LCP arrays and data structures for performing lowest common ancestor queries, which in turn support suffix links.

As an example, the first compressed suffix tree reaching full expressiveness was given by Sadakane [219]. It consists of a compressed suffix array, succinct LCP array, balanced parentheses representation for suffix tree topology and additional data structures for solving range minimum queries. In practice, an engineered version [243] of this compressed suffix tree required $25n$-$35n$ bits of memory and was able to index the complete human genome using only 8.5GB. Compared to classical suffix trees, this compressed variant is two orders of magnitude slower on average. Nevertheless, compressed suffix trees are still much faster than brute force algorithms. Furthermore, many auxiliary data structures used in the design offer a memory-time trade-off which can be optimized for the available memory.

Advancements made in representing auxiliary data structures have led to index structures with even smaller memory requirements [87]. The smallest compressed suffix tree we know of [213] requires only $4n$-$6n$ bits of memory and is based on sampling the suffix tree. This low memory footprint, however, is paid for by giving up performance, and it is several orders of magnitude slower than Sadakane's compressed suffix tree [35]. Another compressed suffix tree proposed by Fischer *et al.* [75] has a memory-time trade-off which lies between the two previously mentioned compressed suffix trees. Cánovas and Navarro [35] engineered an implementation of this compressed suffix tree and compared the impact of different LCP array implementations on the compressed suffix tree. Depending on the implementation of the LCP arrays used, the compressed suffix tree requires between $8n$ and $16n$ bits of memory. A compressed enhanced suffix array reaching full expressiveness is given by Ohlebusch and Gog [199]. However, it does not support lowest common ancestor queries.

Prospects are that space-time trade-offs of compressed index structures will keep improving due to advances in auxiliary data structures, especially advances in compressed suffix arrays and compressed LCP arrays.

2.3 Index structures in external memory

The solution for the memory bottleneck suffered by (main memory) index structures are index structures in external or secondary memory, such as hard disks. This paradigm shift is necessary when even the smallest compressed index struc-

tures cannot be stored in main memory. This limit is usually reached when even a compressed form of S cannot be stored in main memory.

Secondary or external memory has the advantages of low cost, abundance and the persistence given to index structures. However, random access to secondary memory (disk) is much slower than random access to primary memory (RAM). In practice, this difference can be up to five orders of magnitude [117]. Because index structures, such as suffix trees, intrinsically access data structures and input strings in a random manner, this leads to a so-called *I/O bottleneck*.

Several techniques are used to minimize the effect of this bottleneck, both in hardware and in algorithm and data structure design. Solid State Disks, for example, are one order of magnitude faster than classical hard disks. Also, sequential disk access is almost as fast as random access on RAM. Another solution is to limit the number of I/O operations altogether by, for example, decreasing the size of the index structure.

Buffering is another strategy commonly employed, as well as improving locality of information that is closely connected. To achieve this locality, redundancy is often introduced in the data structure, which is opposite to the space-saving techniques seen in main memory indexes. These techniques are not only applied for designing the spatial layout of index structures, but also for their traversal algorithms.

In this section, existing index structures for external memory are reviewed with an emphasis on the high-level strategies employed. Other, more technical, reviews on this topic can be found elsewhere [19, 108, 245].

2.3.1 Suffix arrays

Both suffix trees and suffix arrays perform poorly when naively implemented in secondary memory. Because of their simple design, however, suffix arrays are easier to implement on disk. The basic idea is to use levels of sparse suffix arrays in faster memory to guide searches in the full suffix array stored on disk.

Baeza-Yates *et al.* [18] proposed a two-level index structure. They also augmented the sparse suffix array, stored in RAM, with exact prefixes of the suffixes represented in the sparse suffix array. This has the advantage that no random access to S is needed for matching in the sparse suffix array. Tests revealed that this implementation is five times faster than a naive implementation [231] of a single level suffix array on disk.

Later, Sinha *et al.* [231] replaced sparse suffix arrays by pruned suffix *tries* for the first level of the hierarchy. Again, labels on the pruned suffix *trie* are

explicitly stored instead of pointers to S. Sinha *et al.* also improved the second level of the hierarchy by storing SA(S), LCP(S) and substrings of S, to minimize random access to S. Note that in primary memory, redundancy is eliminated, whereas in secondary memory it is introduced to increase performance. Tests showed that this method is five times faster than the two-level method of Baeza-Yates *et al.* and requires about ten times less non-sequential I/O operations for pattern matching.

A larger number of levels is used in the design of *string B-trees* [67]. These index structures act as conceptual B-trees [23] over suffix arrays. Similar to B-trees, internal nodes are B-ary and the final suffix array values are found in the leaves. To speed up the search through the B-tree, each internal node v contains a *Patricia tree* or *blind tree* for the suffixes in v. Blind trees are suffix tree variants for which edge labels are stored as the first character of the label and its length. Pattern matching in blind trees consists of two phases. A first phase, similar to pattern matching in suffix trees, finds candidate positions according to the matched characters on the edges of the tree. A second phase explicitly compares the pattern to the candidate substrings in S. This type of edge labeling followed by a blind search can also be applied to all external memory suffix tree implementations to minimize random access to S. This data structure has the advantage that pattern matching is theoretically I/O optimal and updates are supported due to its B-tree nature. Furthermore, succinct cache-oblivious string B-trees have been developed [68]. Note that string B-trees are not suffix trees and thus do not reach full expressiveness. Another disadvantage is that the blind search method used is impractical for inexact string matching [111].

Distribution of suffix arrays has also been proposed [180]. This allows processing batches of queries in parallel by dividing SA(S) in intervals or by interleaving suffix array values. This interleaving can be done by grouping every k-th suffix to a single computing unit or by grouping the suffixes of a substring of S together in one node, thus minimizing access to S. Although these designs look promising, we have no knowledge of any recent performance results for string matching algorithms on biological data using any of the above external memory suffix arrays.

2.3.2 Suffix trees

Because of the underlying tree data structure, efficient implementation on disk is more difficult for suffix trees than for suffix arrays. Although many papers about external memory suffix trees exist, most of them focus on construction in

external memory. Less attention has been given to optimizing suffix tree layout for traversals and even fewer performance tests are available for algorithms that make use of external memory suffix trees. The most important factor in designing external memory representations of suffix trees is the grouping of nodes into blocks and the layout of these blocks onto disk. Other important aspects are node and edge label representations. For locality reasons, array-based representations are superior to other implementations [24] and nodes contain more information than their primary memory counterparts, while edge labels can be compactly represented by their first character and length as in blind trees. An example of this strategy is one of the earliest external suffix trees, the *compact Patricia tree* [39], which uses a topology representation similar to the balanced parentheses representation.

A very intuitive external memory suffix tree layout is that of partitioning by prefixes. The suffix tree is split into an upper root-block and blocks containing the subtrees of a given prefix. This layout is similar to the two-level hierarchical layout for suffix arrays. For top-down traversals of the suffix tree, it works well in practice. Furthermore, this layout is created naturally during construction [24, 111]. A disadvantage, however, is its scalability. Although these indexes can be constructed for the human genome [238], larger sequences or data sets suffer from either a large growth in the size of the partitions or an exponential growth in the number of partitions. Moreover, data skewness results in decreasing performance, as some partitions are much larger than others. In theory, a multi-level hierarchical structure could alleviate the scalability problem and data skewness has already been tackled by using variable length prefixes [82, 205].

Another weakness of external memory suffix trees are suffix links. These links imply a lot of random access and are thus optional [82, 205] or completely omitted [111, 238]. On the other hand, some authors [205] claim that the use of suffix links in external memory improves performance of some search algorithms, such as finding maximal exact matches. Clifford [41] designed *distributed suffix trees*, which contain a local version of suffix links, called *sparse suffix links*. These links point to the local root if the normal suffix link would point to a node in a different partition. Clifford points out that prefix partitioning allows traversals on the suffix tree to be run in parallel on the distributed subtrees. Furthermore, he claims that most bioinformatics applications do not require traversals that require communications between the different prefix-partitioned parts. Thus prefix-partitioning enables the parallelization of most search algorithms on suffix trees.

For exact pattern matching, prefix partitioned suffix trees work well. For other queries, however, transforming the tree layout to an already constructed prefix-partitioned suffix tree has been proposed. The goal of changing layouts is to increase scalability and improve the locality of the nodes. For pattern matching, however, the new layout could increase the number of I/O operations. Different techniques have been proposed to achieve this goal. Clark and Munro [39] focused on minimizing the number of blocks required to store suffix trees using a greedy bottom-up algorithm. *STELLAR* [25], on the other hand, focused on improving locality of nodes for both parent-child links as well as suffix links. Other layouts introduce redundancy of data by having the subtrees stored in blocks on disk overlap [20,33]. Although the redundancy introduced increases the memory footprint of the index structures, it improves locality of the nodes and improves the scalability of the index structures. Care has to be taken, however, not to destroy some of the expressiveness of suffix trees, including longest common prefix values and suffix links.

In practice, the largest indexed single DNA sequence found in the literature contains 12 billion base pairs [21]. Although no extensive performance results for string algorithms on this index were given, disk-based index structures are known to be several times faster than non-indexed methods for string matching on the scale of the human genome. Compared to string B-trees, disk-based suffix trees require a similar number of I/O operations [108] for pattern matching. Furthermore, Halachev *et al.* [98] showed that for protein data, pattern matching on disk-based suffix trees can be almost as fast as pattern matching on enhanced suffix arrays. As an example of other applications, a disk-based enhanced suffix array has been used to locate repeats in human chromosomes [16].

2.3.3 Compressed index structures

Data compression and indexing are very important in computational biology, although they seem to be opposites at first sight. With the rise of compressed index structures, this dichotomy can be considered solved [63] for the RAM model. However, designing a disk-based version of these indexes is non-trivial, because compressed suffix arrays and FM-indexes perform many random accesses and show a poor locality [89]. Nevertheless, some compressed index structures for external memory do exist.

Mäkinen *et al.* [172] designed a secondary memory version of the compressed suffix array by Sadakane [216] using a multi-level hierarchical structure. They also designed a distributed compressed suffix array. External memory variants of

FM-indexes have been developed by González and Navarro [90]. They proposed external memory versions for auxiliary data structures for calculating $rank(B)$ and $select(B)$ and proposed a two-level hierarchy for storing $rank(S)$. Different structures were designed for representing BWT(S) on disk, all having different trade-offs depending on the size of the available main memory. For fast locating, they adopted the locally compressed suffix array designed for fast locating [89]. Arroyuelo and Navarro [12] designed an external memory Lempel-Ziv index based on the Lempel-Ziv index structure proposed by Navarro [189]. A recent article by Russo *et al.* [212] shows how parallel and distributed compressed suffix arrays can efficiently answer more advanced queries such as longest common substrings. Furthermore, they designed parallel and distributed compressed suffix trees.

Although the idea of reducing space in external memory to reduce the number of I/O-operations is interesting, it is not known how this affects performance in practice. Some tests on natural language data suggest that compressed index structures are competitive in practice, although they are somewhat slower than string B-trees [90].

Recently, Chien *et al.* [37] proposed a new transformation, called the *geometric Burrows-Wheeler transform*, which connects index structures with range searching. It translates characters of a string into 2D points and vice versa and uses the vast research on 2D range queries to answer pattern matching queries. To achieve a succinct representation, sparsification is used by grouping substrings in meta characters. For external memory purposes it uses a string B-tree to find ranges in the sparse suffix array, while 2D search can be done using a wavelet tree. Tests [108] show that these compressed index structures are smaller compared to other external memory index structures, but they require more I/O operations. Another application opened by these index structures is the possibility to answer relevance queries [108]. As an example, it would be possible to retrieve only the top k most similar sequences in a database.

2.4 Construction

Before index structures can be used, they first need to be constructed. Although construction is fast in theory, it is not always the case in practice. The current bottlenecks in constructing disk-based index structures for very large strings are memory limitations in the working space, cache misses and a high number of random accesses to secondary memory. The working space is the amount of memory required by the construction algorithm, which is usually higher than the

memory required by the final index. Apart from dealing with these issues, some research has focused on parallelizing construction algorithms. In this section, an overview of existing construction algorithms for various index structures is given, illustrated with practical results found in the literature. Note that the figures in this section represent some of the historical breakthroughs in index structure construction, and are not meant as a comparison between the cited implementations. As a general reference, reported index structure construction times for the human genome, or for sequences in the same order of magnitude, were in the range of a few hours on desktop computers and in the range of minutes on clusters and specialized hardware.

2.4.1 Suffix trees

Historically, suffix tree construction goes back to Weiner [254], who gave a first $\mathcal{O}(n)$ algorithm. Later, Ukkonen [242] gave a simpler $\mathcal{O}(n)$ algorithm, which has the nice property of being online, *i.e.* a new string can be added to the suffix tree by appending it to the back of the previous strings. The WOTD suffix tree by Giegerich *et al.* [84] comes with a lazy construction algorithm, in the sense that suffix tree nodes are added the first time that a traversal algorithm requires these nodes. Thus, suffix trees can also be efficiently used for smaller applications that do not require information about the whole tree. The suffix links that are a by-product of Ukkonen's algorithm have very nice features, as discussed in Section 2.1, but they are omitted in other construction algorithms. To retrieve these suffix links, some post-processing algorithms exist [167]. Although the above mentioned suffix tree construction algorithms only scale up to chromosome level, they form the basis for many external memory construction algorithms. Although a main memory suffix tree for the whole human genome was constructed by Kurtz [135], most main memory index structure construction algorithms focus on suffix arrays and compressed index structures.

2.4.2 Suffix arrays

Originally, linear time suffix array construction required the construction of the suffix tree [97]. During the last decade, however, many direct suffix array construction algorithms have been proposed. A taxonomy of existing suffix array construction algorithms is given by Puglisi *et al.* [207]. Because suffix array construction consists of sorting all suffixes of S, many algorithms are based on known sorting algorithms. One of the most popular algorithms is the recursive $\mathcal{O}(n)$ KS3

algorithm of Kärkkäinen and Sanders [120]. It can be modified to a parallel and external memory version, called DC3 [121], which can construct SA(S) for the whole human genome using only 1GB RAM and for which a Message Passing Interface (MPI) version exists that has indexed the human genome in only a few minutes (on specialized hardware) [134]. However, it was noted elsewhere that DC3 is unable to index strings longer than 4Gbp [65].

Other algorithms try to minimize the working space in internal memory. So-called *lightweight* [177, 197] construction algorithms have a working space that approaches the theoretical minimum. Furthermore, according to extensive tests on biological sequences made by Mori[2], among others, they are the fastest construction algorithms in practice. Another trick utilized is to only sort suffixes up to a certain LCP value, leading to *partial suffix arrays*. Although the expressiveness of partial suffix arrays is unclear, they have already been applied for error correction of sequencing reads [260]. For the construction of enhanced suffix arrays, efficient LCP array construction algorithms have been developed [124] and $\mathcal{O}(n)$ algorithms exist for the construction of the other tables [3, 167].

2.4.3 Compressed index structures

Working space is even more important for compressed full-text index structures. Compressed suffix arrays, FM-indexes and regular suffix arrays can easily be obtained from one another. However, suffix array construction requires $40n$ to $48n$ bits of memory, whereas FM-indexes can be stored in only $2n$ bits. Despite this, lightweight suffix array construction algorithms [197] are used by Burrows-Wheeler-based read mapping tools, such as *BWA*. Direct and lightweight construction of compressed index structures is therefore an important issue. A gap between theory and practice existed for several years, but several practical results have been reported recently. For example, a lightweight Burrows-Wheeler construction algorithm by Kärkkäinen [119] requires only $8n$ bits of working space for DNA sequences (which is equal to the size of a normal text string) and was implemented in the short read mapping tool *Bowtie*. Other direct construction algorithms include the parallel algorithm of Sirén [232] and the lightweight construction algorithms in both internal and external memory settings of Ferragina *et al.* [65]. The former has the added value of being able to merge existing compressed suffix arrays, and the latter have very low working spaces. Moreover, a parallel BWT(S) construction algorithm [183] based on the Google MapReduce [50] framework has recently indexed the human genome in

[2] `http://code.google.com/p/libdivsufsort/`(last accessed September 2014)

about 10 minutes on the Amazon Elastic Compute Cloud. Finally, a lightweight construction algorithm for Lempel-Ziv indexes [14] has been reported that is competitive with construction algorithms for other compressed full-text indexes.

2.4.4 External memory suffix tree construction

Most work on external memory index structures has been done on construction algorithms, which have been extensively reviewed by Barsky *et al.* [19]. To summarize their results, external memory allows for larger sequences to be indexed, but the scalability of the algorithms is limited by the number of random accesses to S and the suffix tree under construction. This means that the practical performance of many construction algorithms is limited to sequences which are smaller than the size of the available main memory. As an exception, the B2ST algorithm [21] was able to index DNA sequences of 12Gbp in less than 8 hours, making this algorithm the first to partially overcome the above-mentioned bottlenecks. Furthermore, the authors believe the algorithm will scale up to sequences of 60Gbp.

2.5 Summary

In this review, we have shown the importance of data structures for processing and searching in strings, known as index structures. Many current sequence analysis tools heavily rely upon index structures for handling large amounts of data, which is currently a major concern to bioinformaticians. In Section 2.1, details concerning the most commonly used index structures were presented. The details given in this review are often omitted in articles describing tools and applications. However, we believe that these details are important to fully grasp the possibilities and limitations of these sequence analysis tools.

2.5.1 Prospects

We have made a basic classification of existing index structures and explained the memory-time trade-offs related to these data structures. Because the number of available index structures is vast, we were only able to skim over the technical details involved in the design of these data structures. However, the interested reader was guided to more in-depth work in the literature. Note that the index structures discussed in this review are mainly all-purpose full-text index structures, although some focused on exact pattern matching. There are, however,

other index structures specially designed for specific applications. *Affix index structures*, for example, allow bidirectional string searching. As a result, they can be used for searching RNA structure patterns [185] and for short read mapping [140]. *Weighted suffix trees* [114] can be used to find patterns in biological sequences that contain weights such as base probabilities, but are also applied in error correction [227]. *Geometric suffix trees* [228] have been used to index 3D protein structures. *Property suffix trees* have additional data structures to efficiently answer property matching queries. This can be useful, for example, in retrieving all occurrences of patterns that appear in a repetitive genomic structure [105].

Furthermore, both general purpose full-text index structures and specialized index structures will always be hampered with space-time trade-offs. Several index structures allow tuning this trade-off by setting a sparsification parameter. This optimization of the available main memory is required because of the large difference in speed between internal and external memory. In some cases, the available main memory does not suffice and external memory index structures have to be used. Moreover, we saw that the performance of external memory index structures highly depends on the application for which the index structure is used. There is still a lot of work to be done on increasing the performance of disk-based index structures.

Construction of index structures in external memory has seen more investigation and clearly shows that the use of current index structures is limited to sequences that fit in main memory. Main memory construction algorithms are limited by the available work space for which the demand is several times higher than the memory required for the final index structure.

In the future, algorithms and data structures will have to be improved further to keep up with the rapidly evolving sequencing technology and the growing amount of data in general. To tackle the bottlenecks related to index structures mentioned here, new directions for their design have to be investigated [63]. As a final note, we give some prospects for research on index structures for bioinformatics applications. Currently, the biggest issue in index structure research is closing the gap between theory and practice, which is illustrated by the fact that many theoretically superior index structures do not outperform simpler designs in practice. More engineering work has to be done to improve the practical performance of these index structures. These implementations should be grouped under a common interface in libraries and benchmarked using different types of (bio-

logical) sequences. One such library-project is the *Pizza&Chili website*[3], which bundles full-text compressed index structures for use in exact pattern matching. Another library containing several index structures, but also focusing on biological applications, is the *SeqAn* library [54].

Another significant topic for further research is the adaptation of index structures to modern hardware, such as multi-core CPUs [165, 241] and Solid State Disks. Recently, even more specialized hardware has been considered, including Graphical Processing Units (GPUs) [225] and FPGAs [62]. Alternatively, large computer clusters, local or on the cloud, could allow for massive parallelization of index structures. Some applications have already been ported to these new platforms, including read mapping and SNP finding [142] using cloud computing, sequence alignment [225] on GPUs and suffix array construction [183] using Google's MapReduce [50]. However, these techniques and implementations are very novel and further research will have to indicate their scope and potential.

For applications which require maintenance of the index structure, such as sequence databases or updating an existing index of the human genome, dynamic index structures are required. Historically, this is challenging due to the intrinsic interrelationship of suffixes, where insertion of a single character in a string can change the lexicographical order of many suffixes. However, some index structures that allow addition and removal of whole strings [67, 213] and single characters [221,222] can be found in the literature. Moreover, several index structures were recently proposed for processing a set of very similar strings [138,173], where the size of the index structure only depends on a single reference genome in the collection, rather than the combined size of all sequences in it.

Given these developments, index structures will continue to increase the performance of bioinformatics applications while coping with the continuous growth in sequence sizes.

2.5.2 Related work

Since the publication of the review article [247] on which this chapter is based, many new variants of the full-text index structures presented here have been developed, as well as new construction algorithms, string matching applications and implementations in bioinformatics tools. The new algorithms and data structures present multiple improvements, including improvements in construction and traversal time in main or secondary memory, lower memory requirements and

[3]Two mirrors at `http://pizzachili.di.unipi.it` and `http://pizzachili.dcc.uchile.cl` (last accessed September 2014)

lower peak memory, parameterized indexes that allow users to set the trade-offs, *etc.* A more in-depth literature review falls beyond the scope of this dissertation and as such we would do injustice to the important work of colleagues whose work was not cited. Furthermore, the impact of some contributions cannot yet be correctly estimated at this point. We will therefore limit the overview of related work to some advances related to the prospects made above and other developments that have an impact on the research in this work.

The development of several new index structures has been driven by applications requiring complex string searches. Affix index structures, for example, have seen several developments. As discussed at the beginning of this section, affix index structures expand beyond the expressiveness of suffix trees by allowing bidirectional search algorithms. Schnattinger *et al.* designed a memory efficient affix index structure, called *bidirectional wavelet index* [226]. This index structure consists of wavelet trees for BWT(S) and BWT(S_{rev}), with S_{rev} being the reverse of string S. Compared to the bidirectional BWT of Lam *et al.* [140], the bidirectional wavelet index is somewhat larger, but traversal is only logarithmically dependent on $|\Sigma|$ instead of linear. A bidirectional FM-index, called *FMD-index*, was proposed by Li [147]. The FMD-index consists of a single FM-index for the string $S\bar{S}$. This index was designed specifically for indexing double stranded DNA, but is not applicable to generic texts. The advantage of this approach, however, is an increase in speed for exact matching due to the use of a single index.

The last few years have also seen some new time-memory trade-offs for both classical and compressed index structures. In Chapter 3, we show how a small hash index containing suffix array intervals for fixed-length k-mers can be used to speed up top-down search in (enhanced) suffix arrays. Conceptually, this hash index contains pointers to all nodes in the LCP interval tree that have a fixed string depth. This technique was independently proposed by Grabowski and Raniszewski [92]. In the same chapter, we present an improvement to sparse suffix arrays by augmenting the index with a sparse version of the child array from enhanced suffix arrays. Together with a sparse LCP array and support for suffix links [126], an *enhanced sparse suffix array* index structure is formed. This conceptually simple index allows easy tuning of the time-memory trade-off by setting the sparseness. Fernandes and Freitas propose another improvement to non-compressed index structures [61]. They propose a new scheme for sampling the LCP array and sampling auxiliary data structures used for computing the parent interval of an LCP interval. Their index has low memory requirements

compared to classical full expressive index structures and is very fast in finding maximal exact matches.

As discussed in Section 2.2.3, improvements in auxiliary data structures have a direct effect on the performance of compressed index structures. Gog and Petri proposed several new bit vector representations and implemented them in FM-indexes, resulting in indexes that are either smaller or faster than previous state-of-the-art indexes [88]. This result clearly shows the importance of all building blocks of a compressed index structures. Wavelet trees, another building block, have received attention in a dedicated review [191]. Recent proposals for complete compressed index structures include, among others, CSTs taking advantage of repetitiveness in sequences [2] and improvements in the speed of certain operations for the currently smallest available CSTs [193].

Novel construction algorithms focused mainly on reducing work space requirements of main memory algorithms or combining main and secondary memory, resulting in *semi-external* construction algorithms. Lightweight construction algorithms for the Burrows-Wheeler transform are especially important, as $\mathrm{BWT}(S)$ eventually requires much less memory than $\mathrm{SA}(S)$, but the fastest main memory BWT construction algorithms first construct $\mathrm{SA}(S)$. Recently, Crochemore *et al.* developed a $\mathcal{O}(n^2)$ construction algorithm that only requires constant space [45] and Beller *et al.* designed a space-efficient semi-external BWT construction algorithm [26]. The latter algorithm was used to build $\mathrm{BWT}(S)$ for a 6GB file in memory with only 8GB of workspace and has no upper limit on file size due to its use of external memory. In addition, algorithms for constructing and querying the BWT for a collection of strings were also proposed [22]. One of the fastest and most used suffix array construction algorithms today is the SA-IS algorithm [197]. Recently, an $\mathcal{O}(1)$ workspace variant of this algorithm, called SACA-K, was developed [196]. Finally, in light of closing the gap between computer science and bioinformatics, an exposition of the SA-IS algorithm was given intended for a bioinformatics audience [229].

In the original review article, we argued that one of the biggest issues with index structure research was the gap between theory and practice. This gap is especially prominent in compressed index structures. A step towards closing this gap was made by Gog *et al.* whose *SDSL-lite* library [86] contains the highlights of 40 research publications[4]. This allows users to plug and play parts of a compressed index structure until the desired time-memory trade-off is reached. Other practical implementations can be found in recent genome analysis tools, such as

[4] `https://github.com/simongog/sdsl-lite` (last accessed September 2014)

read mappers. In addition, index structures continue to be used in hardware accelerated (*i.e.* using GPUs, FPGAs, *etc.*) applications [8].

Due to the recent availability of many genomes of the same species, compression and indexing of similar genomes has received a lot of attention [52]. Indexes for a collection of sequences have many applications, such as read mapping against multiple reference genomes, split-read alignment using an exon map (see Chapter 5) and others [233]. Since the pioneering work of Mäkinen *et al.* [173] several practical advances have been made in this field [46]. Most techniques focus on indexing either multiple sequence alignments [173, 233] or indexing a reference genome extended with a file of known variants between the additional genomes and reference genome [46, 109].

Chapter 3

essaMEM

In this chapter we present *essaMEM*, a tool for finding maximal exact matches that can be used in genome comparison and read mapping. *essaMEM* utilizes an *enhanced sparse suffix array* (ESSA), which integrates a child array into an existing sparse suffix array implementation. The chapter is based on the article "*essaMEM: finding maximal exact matches using enhanced sparse suffix arrays*" [248].

3.1 Introduction

Maximal exact matches are exact matches between two sequences that cannot be extended to the left or right without introducing a mismatch (Definition 1.2). MEMs are widely used in genome comparison tools as anchor points for alignment [32, 38, 137], but can also be used without alignment to define a genomic distance measure [51]. MEMs can also be used as seeds for alignment of high-throughput sequencing reads [154], especially for alignment of very long reads.

A naive algorithm for finding all MEMs of a given minimum length between two sequences S and P is to perform exact string matching between all pairs of suffixes $S[i..]$ and $P[j..]$ that meet the left-maximality requirement, *i.e.* $S[i-1] \neq P[j-1]$. The matching process continues until a mismatch is found, or until the end of either suffix is reached. The time complexity of this approach is $\mathcal{O}(|P|^2|S|)$, which is too high for practical applications.

Efficient MEM-finding algorithms can be subdivided into *online* and *indexed* methods. Algorithms of the former type construct a (compressed) index structure

for the concatenation of both sequences and iterate over the index to find the MEMs [97, 106]. Algorithms in the latter method match one sequence against an index of the other sequence. The advantage of indexed MEM-finding over online MEM-finding algorithms is the reusability of the constructed index and the lower memory requirements. We therefore focus on the indexed MEM-finding algorithms.

The indexed MEM-finding algorithm proceeds in a similar way to the naive algorithm presented above, but makes use of an index structure to speed up the matching process. A suffix of P is matched to all suffixes of S simultaneously using an index for S. In addition, some index structures allow to skip parts of the matching process for suffix $P[i..]$ using the result for suffix $P[i-1..]$. As such, the optimal time complexity $\mathcal{O}(|P|+|S|)$ can be reached at the cost of additional memory requirements.

Originally, suffix trees [137] or enhanced suffix arrays [3] were used to find MEMs. However, the size of these types of indexes is several times larger than the size of the indexed sequence. Khan *et al.* suggest the use of sparse suffix arrays [126]. Their SSA-based algorithm, *sparseMEM*, is able to find MEMs faster than previous methods, while using less memory. As a result of this lower memory footprint, SSAs can also index larger genomes than previous methods. For large sparseness values, however, the runtime increases dramatically. Ohlebusch *et al.* use (enhanced) compressed suffix arrays [201]. It was shown how the CSA-based MEM-finding algorithm, *backwardMEM*, outperforms *sparseMEM*, except when memory is abundant.

Our contribution, *essaMEM*, optimizes the method by Khan *et al.* by supplementing SSAs with a sparse child array for large sparseness factors. We show that the new index structure outperforms the previous design, while maintaining the same memory footprint. Furthermore, when combining both the suffix link and child arrays, we achieve a complete enhanced sparse suffix array (ESSA), which has the same expressiveness as suffix trees for substrings larger than the sparseness of the index. We show that ESSAs are competitive for MEM-finding with the CSA-based method by Ohlebusch *et al.* and outperform commonly used methods like *MUMmer* [137] and *Vmatch* [136]. This indicates that, although compressed index structures have recently become very popular [192, 247], the use of ESSA-based algorithms can be a viable option for further research.

Section 3.2 briefly describes the SSA implementation of Khan *et al.* and introduces enhanced sparse suffix arrays. In Section 3.3, we present the MEM-finding algorithm and discuss all optimizations we have made. In Section 3.4, we compare

essaMEM to previous MEM-finding methods. The final section also discusses new developments that were made since the initial publication of *essaMEM* [248].

3.2 Enhanced sparse suffix arrays

Suffix trees and suffix arrays are the basic data structures to allow for fast sequence analysis. However, the size of the indexed data sets have fueled the need for the development of other index structures with a lower memory footprint, as discussed in Section 2.2. One of the possible options to alleviate the memory requirements of suffix arrays is indexing only a sampled number of suffixes. This type of suffix arrays are called *sparse suffix arrays* (SSA), which were briefly discussed in Section 2.2.2.

Definition 3.1. *The sparse suffix array SSA(S) with sparseness value s stores the lexicographical ordering of every s'th suffix of string S as a permutation of its index positions:*

$$S[SA[(i-1)\cdot s]..] < S[SA[i\cdot s]..], \quad 0 < i < \left\lceil \frac{n}{s} \right\rceil .$$

Definition 3.1 is very intuitive and nearly identical to the classical definition of suffix arrays (Definition 2.2) except for the introduction of the sparseness value s. Although the memory footprint of an SSA is s times smaller than that of a SA, we will see that it has the same expressiveness as long as the queries used in search algorithms are at least s characters long. The cost for a decreased memory footprint is paid in additional computational resources.

3.2.1 Data structure

Similar to suffix arrays, sparse suffix arrays can be enhanced with additional information, stored in separate arrays, to increase the expressiveness of the index structure. In addition to an SSA, *enhanced sparse suffix arrays* (ESSA) consist of a sparse LCP array, sparse child array and facilities for retrieval of sparse suffix links. Most of the definitions used for enhanced suffix arrays (Section 2.1.3) still apply to sparse suffix arrays because they only rely on the presence of an array of lexicographically sorted strings. For example, the sparse LCP array contains the length of the longest common prefixes of two consecutive suffixes in the sparse suffix array. Likewise, an interval of SSA values with LCP of length ℓ can be grouped in a sparse ℓ interval, which serves as a node in a virtual sparse suffix

Table 3.1: Arrays used by enhanced sparse suffix arrays for string $S =$ `ACATACAGATG$` and sparseness value 2. From left to right: index position, sparse suffix array, sparse LCP array, sparse child array, sparse inverse suffix array and sampled suffixes of string S.

i	SSA	LCP	*child*	SA^{-1}	$S[\text{SA}[2i]..]$
0	4	-1		1	`ACAGATG$`
1	0	3	1	3	`ACATACAGATG$`
2	6	1	3	0	`AGATG$`
3	2	1	4	2	`ATACAGATG$`
4	8	2	2	4	`ATG$`
5	10	0	4	5	`G$`

tree. The sparse child array then stores the parent-child relationship in this virtual sparse suffix tree. Finally, a node with label $S[i..j]$ has a sparse suffix link that point to the node with label $S[i+s..j]$, or the root if $i+s>j$.

An example of an ESSA for sequence `ACATACAGATG$`, using sparseness value 2, is given in Table 3.1. This example can be compared to Table 2.1, which shows the normal enhanced suffix array. All suffixes with even index positions are added to the index, as can be seen in the SSA column. The values of the LCP array are based exclusively on the indexed suffixes, which are also printed in the last column. Values of the child array and the inverse suffix array now range in $[0..\frac{|S|-1}{2}]$ instead of $[0..|S|-1]$, which could be used to further reduce the memory footprint of the index if integers are stored succinctly. The definition of the child array is identical to that of the non-sparse version, but the sparse inverse suffix array is defined as $ISA[SSA[\frac{i}{s}]] = i$. Table 3.1 does not contain a sparse suffix link array, as we adopt the idea of Khan *et al.* to calculate suffix links online using a combination of the sparse LCP array and sparse ISA [51].

The virtual suffix tree that can traversed using the information stored in the arrays of Table 3.1 can be seen in Figure 3.1.

Enhanced sparse suffix arrays have the advantage of being modular, similar to ESAs. The basic index structure requires n bytes of storage for the indexed sequence and $\frac{4n}{s}$ bytes for the SSA. The sparse LCP array requires an additional $\frac{n}{s}$ bytes and both the sparse ISA and child arrays require $\frac{4n}{s}$ bytes of memory. In total, the ESSA thus consumes $n+\frac{13n}{s}$ bytes. For MEM-finding, however, we found that we can do without either the child array or the ISA depending on the sparseness value, resulting in a practical memory footprint of $n+\frac{13n}{s}$ bytes.

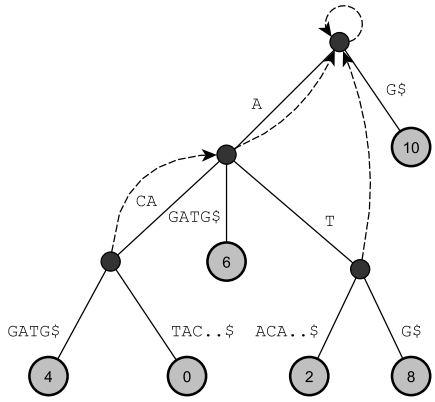


Figure 3.1: Sparse suffix tree for string $S =$ `ACATACAGATG$`, with sparseness value 2. Each number i inside a leaf represents suffix $S[i..]$ of the string S. Dashed arrows correspond to suffix links. For the sake of brevity, only the first characters followed by two dots and the special end-character \$ are shown for edge labels that spell out the rest of the suffix corresponding to the leaf the edge is connected with.

3.2.2 String matching

String matching with an SSA or ESSA is identical to using their classical counterparts. However, the patterns will only be matched to a fraction of the suffixes of the indexed sequence. To obtain the complete set of occurrence of a pattern P in an indexed sequence S, the index will have to be traversed multiple times for consecutive suffixes of P. The full string matching algorithm is presented in Algorithm 3.1. To find $occ(P, S)$ in an index with sparseness value s, the algorithm matches suffixes $P[0..]$ through $P[s-1..]$ to obtain all occurrences in the indexed suffixes of S. For suffixes different from P, the algorithm then checks if the unmatched prefix of P also matches the substring of S at these positions.

Algorithm 3.1 is able to find all occurrences of P in S ($|S| = n$) if $|P| = m \geq s$. To see this, let P occur at offset j in S. If $S[j..]$ is indexed, the match will be found by matching P to the index. If $S[j..]$ is not indexed, there will be an offset $j' > j$ with $j' \mod (s) = 0$. For j', it holds that $P[j'-j..] = S[j'..j' + (|P| - (j'-j) + 1)]$, and because j' is indexed, a match of $P[j'-j..]$ will be found, which then translates into a match of P at position j.

Algorithm 3.1 String matching on SSA and ESSA

Input: pattern P and sequence S, (E)SSA index I for S and sparseness value s
Output: $occ(P, S)$

```
1: for i in [0..s − 1] do
2:     occ(P[i..], I) ← match P[i..] to I
3:     for j in occ(P[i..], I) do
4:         if P[0..i] = S[j − i..j] then
5:             add j to occ(P, S)
6:         end if
7:     end for
8: end for
```

Note that the sparse index structures are unable to retrieve all matches of a pattern that is smaller than the sparseness value. In practice, however, the length of patterns is much higher than the sparseness value because sparse index structures are mainly used for very large sequences in which small patterns are very frequent and less useful. In addition, large sparseness values are rarely used as the memory footprint of the ESSA index becomes dominated by the footprint of the reference genome itself.

The runtime of Algorithm 3.1 depends on the type of matching algorithm used, which is $\mathcal{O}(m \log(\frac{n}{s}) + c)$ for the SSA, $\mathcal{O}(m + \log(\frac{n}{s}) + c)$ for the SSA augmented with the sparse LCP array and $\mathcal{O}(m + c)$ for the SSA augmented with both the sparse LCP array and sparse child array. In these formulas, c equals the total number of occurrences of all prefixes of P that are matched to the index, *i.e.* $\sum_{i=0}^{s-1} occ(P[i..], I)$. The values also need to be multiplied by s to obtain the full time complexity of the algorithm.

3.2.3 Construction

For the construction of the arrays making up the ESSA index structure, slight variants of the algorithms for the classical arrays can be used. The construction algorithms of all arrays, except the sparse child array, were taken from the implementation by Khan *et al.* They use the algorithm of Ferragina and Fischer [64] for constructing SSAs, which is based on radix sort. The ISA and LCP arrays can be easily calculated in a linear pass of the SSA [64]. For the construction of the sparse child table, we use the algorithm described in [3]. This algorithm can be used without major modifications because the child array only requires

knowledge of specific intervals within LCP arrays (LCP intervals [3]), for which the definition remains unchanged when introducing sparseness.

We did modify the algorithm, however, to make it work for sequences that do not contain a terminal character that is lexicographically larger than all characters in the alphabet of the sequence. Our modifications relate to the computation of *up* and *down* values, as described in Algorithm 6.2 in [3] and can be found in Algorithm 3.2.

Abouelhoda *et al.* [3] prove that the assigned child array values are correct. However, it is possible that not all *down* values are assigned. The down values of the last increasing sequence of LCP values will still be on the stack at the end of the algorithm if the sequence did not contain a terminal character appended to the sequence, or a value that is low enough to flush the stack. We modify the existing algorithm by clearing the stack at the end of the algorithm and executing the lines within the `while` loop to check whether additional *down* values can be found.

3.3 MEM-finding

Most MEM-finding algorithms roughly share the basic approach described in the introduction of this chapter. The implementation of common algorithmic stages can vary greatly, however, due to the specific design of the index structure used. We will first discuss the MEM-finding algorithm based on classical suffix tree and enhanced suffix array index structures [3, 137]. Subsequently, we will cover the main alterations made to the algorithm by the SSA implementation of Khan *et al.* Finally, we discuss the changes we proposed in our ESSA implementation. Our main contributions are the introduction of the sparse child array, an option to use suffix links in combination with the child array, and the introduction of sampling in the suffixes of the pattern sequence (parameter p).

3.3.1 Outline

The *essaMEM* MEM-finding algorithm is given in Algorithm 3.3. The base algorithm using classical index structures can be obtained by setting $s = p = 1$. The algorithmic effects of those sparseness parameters is described further down this section. All index-based MEM-finding algorithms using suffix tree variants follow a three-step approach of *i)* constructing an index structure for S, *ii)* matching all

Algorithm 3.2 Part of the construction algorithm for the sparse child array. The algorithm describes how *up* and *down* values are calculated.

Input: the LCP array LCP and a stack ST
Output: all *up* and *down* values calculated for $CHILD$

```
1:  lastIndex = −1
2:  ST.push(0)
3:  for i in [1..n/(s−1)] do
4:      while LCP[i] < LCP[ST.top] do
5:          lastIndex = ST.pop
6:          if LCP[i] ≤ LCP[ST.top] and LCP[ST.top] ≠ LCP[lastIndex] then
7:              CHILD[ST.top].down = lastIndex
8:          end if
9:      end while
10:     if lastIndex ≠ −1 then
11:         CHILD[i].up = lastIndex
12:         lastIndex = −1
13:     end if
14:     ST.push(i)
15: end for
    // Additional loop to fill in the remaining down values on the stack.
16: while 0 < LCP[ST.top] do
17:     lastIndex = ST.pop
18:     if 0 ≤ LCP[ST.top] and LCP[ST.top] ≠ LCP[lastIndex] then
19:         CHILD[ST.top].down = lastIndex
20:     end if
21: end while
```

suffixes of P against this index until a mismatch occurs or the minimum length ℓ is reached and *iii)* checking possible matches for maximality constraints.

Index construction has been covered in the previous section and Section 2.4 for general index structures. While left-maximality of potential MEMs has to be verified explicitly, right-maximality immediately follows from the properties of the index structure. At each (virtual) suffix tree node during the matching process, all positions contained in the leafs of all subtrees except for the subtree that continues to match the pattern contain right-maximal matches between the matched suffix and the reference genome. The length of the right-maximal match is equal to the length of the currently matched substring, *i.e.* the string depth of the node. The efficiency of the matching algorithm in line 6 of Algorithm 3.3 depends on the data structure, *e.g.* suffix tree, SA or ESA.

For every right-maximal exact match found this way, the algorithm explicitly compares characters to the left of the match to verify left-maximality. The character comparisons required for checking left-maximality can be done in constant time, as both S and the SA are stored uncompressed. Matches that are both right and left-maximal and whose length is at least ℓ are subsequently reported as MEMs.

In addition to the steps described thus far, a suffix link-based sliding window scan is used in line 18 to recover computations made for the previous suffix. As a result, MEMs can be computed with a linear, rather than a quadratic dependency on m. The fact that not all implementations contain suffix links by default is underpinned by the conditional statement above line 18. If the implementation does not contain suffix links, the matching process starts from the root of the (virtual) suffix tree.

3.3.2 Sparse suffix arrays

The MEM-finding algorithm by Khan *et al.* can be obtained from Algorithm 3.3 by using an SSA index structure and setting $s > 1$, while keeping $p = 1$. Similar to the exact string matching Algorithm 3.1, the introduction of a sparse index structure requires matching multiple suffixes of a sequence against the index to obtain all occurrences of that sequence. However, since the MEM-finding algorithm already iterates over all suffixes of P, the use of an SSA does not require additional traversals of the index. Furthermore, the matching phase in line 6 traverses a smaller index. This is noticeable as the SSA algorithm uses the binary search algorithm during the matching phase, which has a complexity of $\mathcal{O}(m + \log(\frac{n}{s}))$.

Algorithm 3.3 the *essaMEM* MEM-finding algorithm

Input: sequence S of length n, pattern P of length m, sparseness value s and sampling factor p

Output: all MEMs between S and P of minimum length ℓ

```
1: construct the enhanced sparse suffix array I for S with sparseness value s
   // ℓ_s is the minimum length of right-maximal exact matches
2: ℓ_s ← ℓ − (p · s − 1)
3: for i in [0..s − 1] do
4:     j ← i
5:     while j < m − ℓ_s do
6:         match P[j..] to I until the longest match is found
7:         store the suffix array interval of matches longer than ℓ_s
           // all stored matched are right-maximal
8:         if at least one match of length ℓ_s exists then
9:             for all right-maximal exact matches do
10:                check left-maximality by matching up to s · p characters
11:                if the length of the match is at least ℓ then
12:                    report a maximal exact match
13:                end if
14:            end for
15:        end if
16:        j ← j + s · p
17:        if suffix links are available then
18:            recover part of the matching process using sparse suffix links
19:        end if
20:    end while
21: end for
```

The price paid for the reduced memory footprint can be found in the loop on line 9 in which left-maximality is checked. This is because the set of right-maximal exact matches has expanded to matches of length $\ell - s$ and because the maximum number of bases that can be added to the match is now $s - 1$. MEM-finding using SSA-based algorithms becomes infeasible if $\ell - s$ becomes close to zero, as will be shown in Section 3.4.

In addition to the use of SSA in the matching phase of the algorithm, Khan *et al.* also support the sliding window scan using sparse suffix links. Because sparse suffix links target nodes whose label is s characters shorter, the special double loop in lines 3 and 5 are used and the inner loop has a step size of s.

3.3.3 Sparse child arrays

The first major improvement we made to the existing MEM-finding algorithm is the incorporation of sparse child arrays. For this, we use the child array traversal Algorithm 6.8 in [3]. However, minor adjustments are needed to incorporate the input and output specifications of the MEM-finding algorithm because the matching algorithm on line 6 might not start with a proper LCP interval. The algorithm has to check whether the provided depth ℓ of the starting interval $[i..j]$ equals the LCP value ℓ' of the LCP interval $\ell' - [i..j]$. This can be done in constant time using the child and LCP arrays. If $\ell = \ell'$, the algorithm starts matching again with finding the correct child interval of $[i..j]$. On the other hand, if $\ell < \ell'$, the algorithm starts the matching process at the edge connecting $[i..j]$ with its parent interval. By design of the original SSA method, the case $\ell > \ell'$ does not occur. As a result, the matching stage of the ESSA-based MEM-finding algorithm can be run in $\mathcal{O}(m)$ time.

3.3.4 Sparse suffix links

Khan *et al.* simulate suffix links using the LCP array and a sparse inverse suffix array (ISA) to speed up the matching stage of the MEM-finding algorithm. As the produced intervals match LCP intervals as defined in [3], the existing algorithm for simulating suffix links can be used in combination with a sparse child array. However, sparse suffix arrays decrease the LCP value of the suffix link by s instead of just one. As a result, the LCP value produced by the original SSA-based algorithm might be smaller than the LCP value of the corresponding LCP interval. Therefore, the matching algorithm described in Section 3.3.3 needs to check for possible discrepancies between the two definitions.

Due to the fact that suffix links are simulated by extending a subinterval until the right boundaries are found, the suffix link simulation can fail. As a result, the final MEM-finding algorithm still has worst case time complexity of $\mathcal{O}(m^2 + o)$, where o is dependent on the number of $\ell - s$ right-maximal exact matches.

Although suffix links and (sparse) child arrays can be combined in enhanced sparse suffix arrays, this combination does not lead to further improvements in execution time for the MEM-finding algorithm, especially if the parameter p is used (see Section 3.3.5). The combination of child arrays and suffix links might still be of interest for designing other algorithms.

3.3.5 Pattern suffix sampling

The final addition to *essaMEM* is the introduction of *sampling* in the pattern sequence P. This feature is enabled in Algorithm 3.3 by setting parameter $p > 1$. This parameter allows to skip the matching phase of several suffixes of P, but increases the number of right-maximal exact matches by decreasing the guaranteed minimum length of those matches. While SSA-based methods already introduce sparseness in the indexed reference genome, this parameter introduces sparseness in the pattern.

In contrast to the sampling of indexed suffixes, suffixes in the pattern are not sampled equidistantly because such a combined sampling strategy would not result in all MEMs being found. Instead, s consecutive suffixes are always sampled, separated by gaps of $s(p-1)$ suffixes that are skipped.

Lemma 3.1. *Algorithm 3.3 correctly identifies all MEMs of minimum length* $\ell \geq p \cdot s$.

Proof. Let (i, j, ℓ) be a MEM of length ℓ. It is enough to show that the algorithm identifies a suffix of this MEM of at least $\ell_s = \ell-(p\cdot s-1)$ bases long, as this suffix will be a right-maximal exact match. The argumentation used is similar to the proof of the correctness of the original SSA-based MEM-finding algorithm [126].

First consider the case $p = 1$. If suffix $S[i..]$ is not sampled, the distance d to the next sampled suffix of S is lower than or equal to $s-1$. Therefore, the length of $S[i + d..i + \ell - 1]$ is at least $\ell - s + 1$.

Next, consider the case $p > 1$. Again, the distance d to the first sampled suffix in S will be lower than or equal to $s - 1$. Due to the equidistant sampling in S, suffixes $S[i + d + s..], \ldots, S[i + d + s(p - 1)..]$ are also sampled. Furthermore, at least on of the suffixes $P[j + d..], P[j + d + s..], \ldots, P[j + d + s(p - 1)..]$ will be sampled in the pattern. This is because $(j+d+s(p-1))-(j+d)+1 > s(p-1)$,

which equals the maximum gap between two sampled suffixes in the pattern. As a result, the shortest suffix of (i, j, ℓ) sampled in both sequences has length $\ell - (d + s(p-1)) \geq \ell_s$. □

To illustrate the correctness of MEM-finding using a sparse index structure in combination with sampling suffixes in the pattern, Figure 3.2 shows all possible situations that can occur for $s = p = 2$ and a specific MEM.

Suffix sampling does not work well in combination with simulated sparse suffix links. This combination would require p suffix link simulation steps to be performed and simultaneously decreases the number of matched characters that can be recovered using sparse suffix links. Therefore, we set $p = 1$ when suffix links are enabled in *essaMEM*.

3.3.6 Implementation

essaMEM extends the SSA-based MEM-finding method by Khan *et al.* The updated source code is available and open source[1]. We have also included extra command line parameters to switch the use of suffix links or a sparse child array on or off. We also included the parameter p and support all native *MUMmer* v3.23 parameters. In addition, all features of the original *sparseMEM* method still function properly, including multithreading support. Because *essaMEM* supports all *MUMmer* functions, it can replace the native MEM-finding algorithm in the *MUMmer* alignment program[2].

3.4 Results

The correctness of the *essaMEM* MEM-finding algorithm was experimentally verified using *sparseMEM* as a benchmark tool. Both tools found the same set of MEMs on various data sets. We evaluated the performance and memory requirements of *essaMEM* against *Vmatch*, *MUMmer*, *sparseMEM* and *backwardMEM* using all relevant data sets provided in [126], which include six pairs of megabase-sized genomes, two pairs of gigabase-sized sequences and two sequencing read data sets. We also tested the impact of the different improvements we implemented.

[1] `https://github.com/readmapping/essaMEM`(last accessed September 2014)
[2] `http://mummer.sourceforge.net/`(last accessed September 2014)

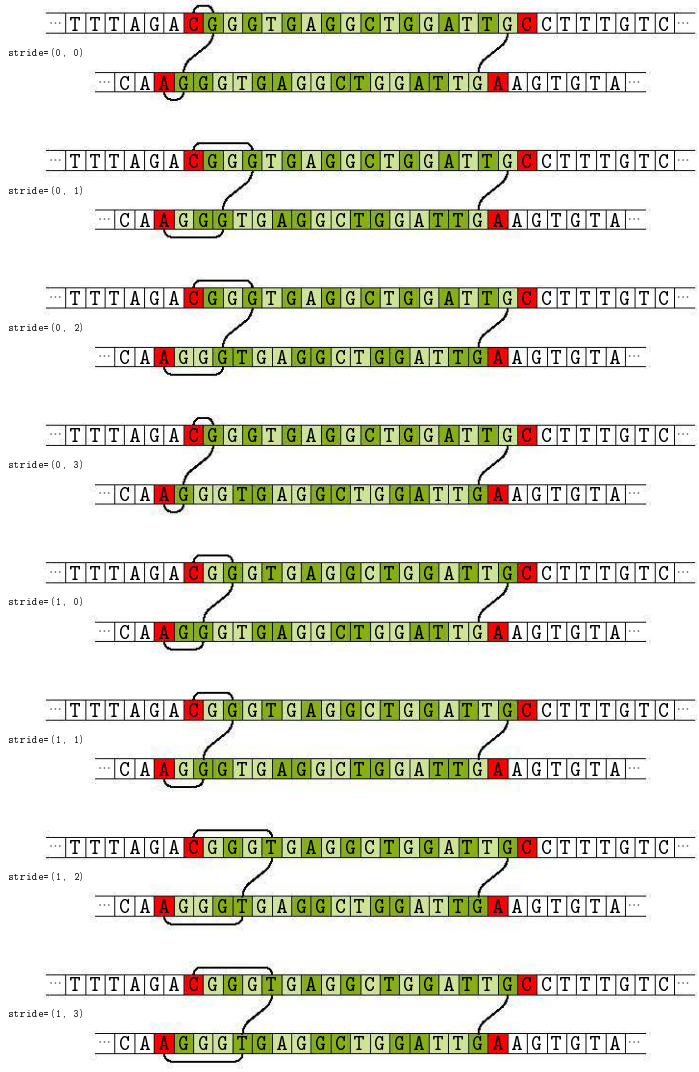

stride=(0, 0)
stride=(0, 1)
stride=(0, 2)
stride=(0, 3)
stride=(1, 0)
stride=(1, 1)
stride=(1, 2)
stride=(1, 3)

Figure 3.2: *(previous page)* All 8 possible cases that might occur during MEM-finding between a reference genome (top) indexed with sparseness $s = 2$, and a read sequence (bottom) with suffix sampling $p = 2$. The matching subsequence `GGGTGAGGCTGGATTG` (green) is maximal as it is flanked by mismatching base pairs `C` and `A` (red). Initial matches can only be found between two sampled suffixes, whose starting positions are indicated in dark green. Light green bases indicate starting positions of suffixes that are not sampled. With $s = 2$ and $p = 2$, every second suffix is indexed and $s \cdot (p - 1)$ consecutive blocks of suffixes are skipped for every successive block of s suffixes that are matched in the read. The right-maximal exact match (vertical connection lines) that is found by the algorithm is a suffix of the complete MEM, whose starting position depends on the stride of the sampled suffix blocks. Horizontal connection lines show the prefix of the right-maximal exact match that needs to be additionally inspected in order to find the left border of the complete MEM. The length of this prefix is at most $s \cdot p$.

3.4.1 Test Setup

A summary of all the different test data sets can be found in Table 3.2. The tests include six megabase-sized genome comparisons, two gigabase-sized genome comparisons and two data sets with NGS sequencing reads. Column 1 in Table 3.2 introduces an identifier for the data set and column 6 gives the minimum length ℓ that was used for the data set.

For all data sets, we compare *essaMEM*, *MUMmer* v3.23, *Vmatch* v2.1.7, *sparseMEM* and *backwardMEM*. For *essaMEM*, we used five settings:

- *essaMEM-1*: uses a sparse child array, no suffix link support and $p = 1$;
- *essaMEM-2*: uses a sparse child array, no suffix link support and p is optimal (default setting of *essaMEM*);
- *essaMEM-3*: uses both a sparse child array and suffix link support and sets $p = 1$;
- *essaMEM-4*: uses the *sparseMEM* binary search algorithm, no suffix link support and sets $p = 1$;
- *essaMEM-5*: uses the *sparseMEM* binary search algorithm, no suffix link support and p is optimal.

Table 3.2: Summary of all data sets used for benchmarking *essaMEM*. The first eight data sets are whole genome comparisons, of which the first six are megabase genomes and the last two are gigabase genomes. The last two data sets consist of a reference genome and a sequencing read data set.

index	reference	size	query	size	ℓ
1	*A. fumigatus*	29Mbp	*A. nidulans*	29Mbp	20
2	*M. musculus16*	95Mbp	*H. sapiens21*	35Mbp	50
3	*H. sapiens21*	35Mbp	*M. musculus16*	95Mbp	50
4	*D. simulans*	138Mbp	*D. sechellia*	167Mbp	50
5	*D. melanogaster*	169Mbp	*D. sechellia*	167Mbp	50
6	*D. melanogaster*	169Mbp	*D. yakuba*	166Mbp	50
7	Mouse (mm10)	2.7Gbp	Human (hg19)	3.1Gbp	100
8	Human (hg19)	3.1Gbp	Chimp (panTro3)	3.4Gbp	100
9	Chicken (galGal4)	1.1Gbp	SRR107602	447Mbp	100
10	*D. melanogaster*	169Mbp	SRR034674	214Mbp	50

Note that *sparseMEM* is equal to *essaMEM* without sparse child array, $p = 1$ and suffix link support enabled. For *Vmatch*, we also used two different performance optimization settings:

- *Vmatch-0*: *Vmatch* with parameter `-qspeedup 0`;
- *Vmatch-2*: *Vmatch* with parameter `-qspeedup 2`.

Other program parameters are the same as used in [126] and [201]. However, we explore a much larger interval of sparseness and compression values than previously reported by other authors. We test all powers of 2 that are smaller than the minimum length ℓ, except for the Mouse-Human and Human-Chimp data set, were the set of s-values equals $\{3, 4, 6, 8, 16, 32, 64\}$. The sampling value p in *essaMEM-2* and *essaMEM-5* for a given sparseness value s was obtained by testing a maximum of five successive values of p, with the largest value of p equal to the largest value for which $\ell_s = \ell - s \cdot p + 1 \geq 10$ holds. The final choice of p was set to:

$$p = \begin{cases} \left\lceil \frac{\ell-10}{s} \right\rceil, & \text{if } s \geq 4 \\ \left\lceil \frac{\ell-12}{s} \right\rceil, & \text{if } s < 4 \end{cases}$$

The runtimes do not include the time for the index construction phase, if the program contains such as phase. The resident set size was measured for the

Table 3.3: The real size of the index structure built by *sparseMEM*, *essaMEM* and *backwardMEM* for the *D. melanogaster* (169Mbp) genome. *sparseMEM* and *essaMEM* share the same memory footprint.

s	*sparseMEM*	*essaMEM-2*	*backwardMEM*
1	1861MB	1861MB	1031MB
2	997MB	997MB	709MB
4	576MB	576MB	548MB
8	370MB	370MB	468MB
16	268MB	268MB	427MB
32	218MB	218MB	407MB

memory results in this section. Additional details on the experimental results, including hardware configurations, can be found in Appendix C.

3.4.2 Memory requirements

First, we compare all tools exclusively on memory consumption. The memory footprint of the classical index structures used in *MUMmer* and *Vmatch* is much higher than those of the tools using compressed or sparse index structures. Furthermore, they do not allow to set a memory-time trade-off. An example of the exact memory footprint of the other tools is given in Table 3.3.

sparseMEM and *essaMEM* share the same memory footprint as the sparse ISA array used in *sparseMEM* and sparse child array used in *essaMEM* have the same size. In addition, the index of both of these tools consists of the reference genome, SSA and sparse LCP array. In terms of memory consumption, *backwardMEM* starts with a lower memory footprint at no compression, but the memory footprint of *essaMEM* decreases faster. For the data set in Table 3.3, the index size of the MEM-finding tools is equal for s between 4 and 8.

In addition to the experimentally observed memory usage, we also report the theoretical memory requirements of the index structures. *MUMmer* requires approximately $15n$ bytes [137] and the index used in *Vmatch* requires $6n$ to $7n$ bytes[3]. For a reference genome of size n, the default setting of *essaMEM* requires $n + \frac{9n}{s}$ bytes of memory. *BackwardMEM* requires $\frac{4n}{s} + 1.375n$ bytes [85], where s is used for sampling suffix array values used in the CSA. As a result, *essaMEM* and *backwardMEM* theoretically have the same memory requirements for $s \approx 13$.

[3] `http://www.vmatch.de/virtman.pdf`(last accessed September 2014)

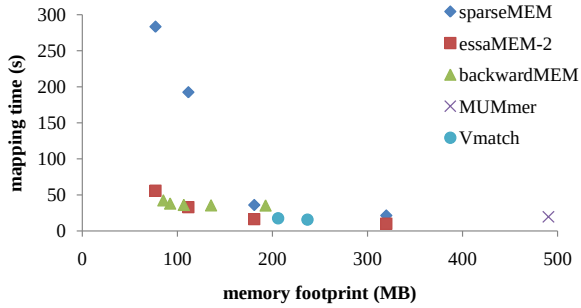
3.4.3 Time-memory trade-offs

The time-memory trade-offs achieved by all tools on the megabase-sized genome data sets and sequence read data sets can be found in Figures 3.3, 3.4 and 3.5. The tests for the gigabase-sized genome data sets only include *sparseMEM* and *essaMEM*, as the other programs did not meet the memory restrictions of our hardware. Results for these data sets are therefore given in the next section. The figures contain only the results for one *essaMEM* setting, namely *essaMEM-2*. This setting provides the best performance of all *essaMEM* settings. The exact time and memory statistics can be found in Appendix C.

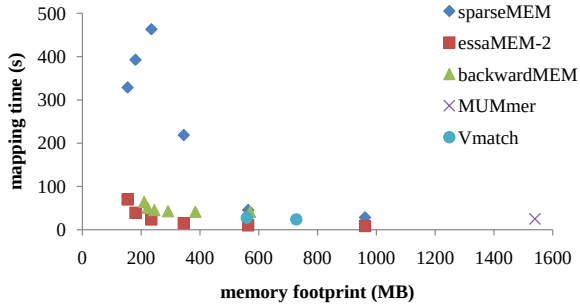
From the results in Figures 3.3 and 3.4, it is clear that *MUMmer* is one of the fastest MEM-finding algorithms, but its memory requirements are significantly higher than those of the other programs. Furthermore, its memory footprint cannot be altered.

Vmatch features a similar mapping time to *MUMmer*, but benefits from smaller memory requirements. In addition, *Vmatch* features two memory settings using the `-qspeedup` parameter. This memory footprint is, however, still higher than the memory footprint of *backwardMEM* (in all settings) and both *sparseMEM* and *essaMEM* (for $s \leq 2$). As a result, *Vmatch* lacks the flexibility of the programs using sparse or compressed index structures. Moreover, *Vmatch* is completely outperformed by *essaMEM*, both in mapping time and memory consumption.

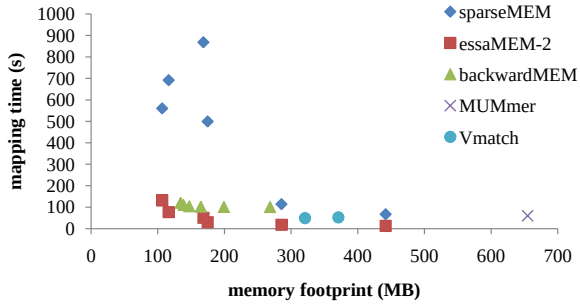
In all tests, *sparseMEM* is clearly outperformed by both *backwardMEM* and *essaMEM*, except for some cases when memory is abundant ($s = 1$). *sparseMEM* is very fast for small values of s, but *essaMEM* is still 2 to 10 times faster than *sparseMEM*. This gap further increases for intermediate values of s as the performance of *sparseMEM* quickly degrades. For the *D. melanogaster - D. yakuba* data set (Figure 3.4c), for example, *essaMEM* becomes up to 25 times faster than *sparseMEM*. The decrease in performance could be explained by the diminishing use of suffix links. For the highest values of s, however, the runtime of *sparseMEM* decreases again, whereas the performance of *essaMEM* slowly degrades. As a result, *sparseMEM* is only 4 to 6 times slower than *essaMEM* for low memory settings. The increase in performance for high sparseness values could be explained by the low performance of the suffix link simulation algorithm for intermediate sparseness values.



(a) *A. fumigatus* vs *A. nidulans*

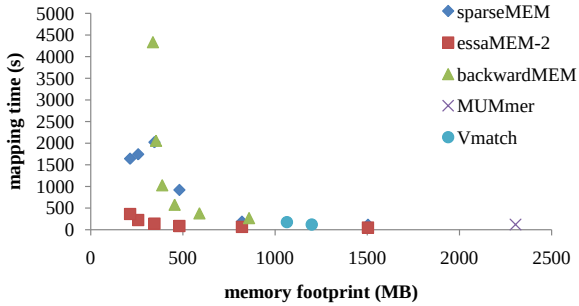


(b) *M. musculus16* vs *H. sapiens21*

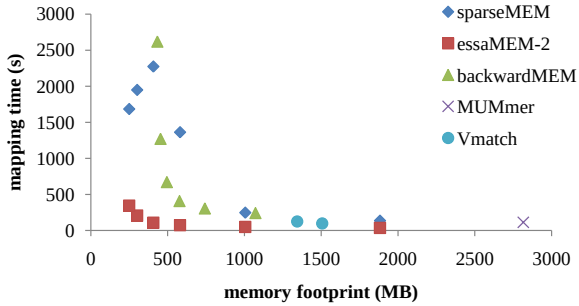


(c) *H. sapiens21* vs *M. musculus16*

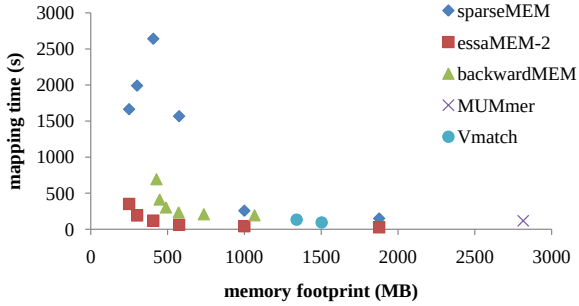
Figure 3.3: Scatterplot showing the memory-time trade-offs of MEM-finding between megabase-sized genome data sets 1 to 3.



(a) *D. simulans* vs *D. sechellia*

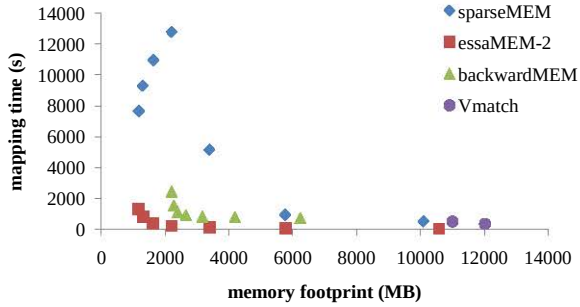


(b) *D. melanogaster* vs *D. sechellia*

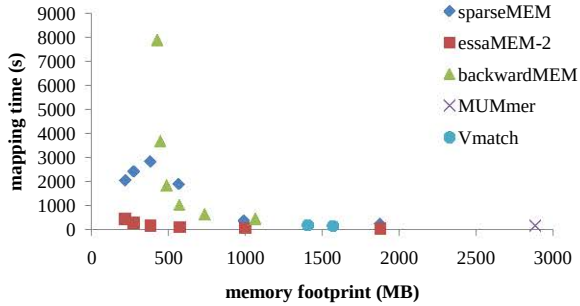


(c) *D. melanogaster* vs *D. yakuba*

Figure 3.4: Scatterplot showing the memory-time trade-offs of MEM-finding between megabase-sized genome data sets 4 to 6.



(a) *G. gallus* vs SRR107602 (454 read data set)



(b) *D. melanogaster* vs SRR034674 (454 read data set)

Figure 3.5: Scatterplot showing the memory-time trade-offs of MEM-finding between a reference genome and a read data set (data sets 9 to 10). For **(a)**, memory requirements for *MUMmer* were too high for data set 9 and are thus not shown.

A similar decrease in runtime for the mapping phase with high sparseness s can be observed using the sparse child array in *essaMEM*. This is, however, countered by the diminishing effect of the sampling parameter p and can therefore not be clearly seen in Figures 3.3 to 3.5. The lower running time for the mapping phase might be explained by a combination of the use of a sparse child array that can match more than one character at the same, smaller minimum lengths in the matching phase and improved I/O performance.

The results also indicate that *essaMEM* is in general somewhat faster than *backwardMEM* for comparable memory settings. If $p = 1$ due to a small difference between ℓ and s, however, *backwardMEM* is faster for intermediate values of s, whereas *essaMEM* is faster for large values of s. The exact values at which either method becomes faster than the other is dependent on the data set. In Figure 3.3c, for example, the runtime of *backwardMEM* is almost constant, whereas in Figure 3.4a, runtime steeply increases with increasing values of s. Note that, except for the *Aspergillus* data set in Figure 3.3a, the *essaMEM* data points are located below the *backwardMEM* data points. For most data sets, *essaMEM* with $s = 16$ even outperforms the fastest setting of *backwardMEM*. In contrast to SSA-based methods, however, *backwardMEM* puts no restriction on the maximum value of s and could thus be used for small ℓ settings in combination with a high compression factor.

The performance of *sparseMEM*, *essaMEM* and *backwardMEM* is dependent on the product of the sparseness value and the number of right-maximal matches. This dependency is more apparent for *backwardMEM* in our tests, but can also be detected in *essaMEM* as $p \cdot s$ approximates ℓ. To illustrate this, Table 3.4 shows the relationship between the number of right-maximal matches for the first six data sets and the mapping time for varying s. In addition, column 3 of Table 3.4 contains the mapping time of a modified version of *backwardMEM* that does not calculate the offset of the MEMs in the reference genome. We find that by disabling the expensive calculation of the offset, the overall runtime of *backwardMEM* becomes independent of s. The overhead caused by calculating the offset can be found by comparing the value in column 3 of Table 3.4 with columns 4 to 9. Note that by comparing the overhead of computing the offset for different values of s, a linear dependency on s can be found. Furthermore, by comparing the overhead for a fixed value of s with the different number of right-maximal matches in column 2 of Table 3.4, a linear dependency on the number of right-maximal matches can be found. These results confirm the theoretical complexity of the algorithm that is given in [201]. According to Gog [85], the

Table 3.4: Relationship between number of right-maximal matches (column 2) and runtime in seconds for different values of s (columns 4 to 9) on the megabase-sized genome data sets (column 1). Column 3 shows the runtime in seconds of *backwardMEM* if only the number of right-maximal exact matches is calculated without searching for the positions of the MEMs in the reference genome. As this runtime is independent of s, only one value is given.

data set	#MEMs	runtime for #MEMs	s					
			1	2	4	8	16	32
1	1.7M	34	35	36	36	38	43	
2	2.6M	38	41	42	43	45	51	64
3	2.6M	97	101	101	103	105	111	121
4	49.6M	167	269	379	579	1028	2057	4334
5	226.9M	167	242	306	408	671	1271	2620
6	395.4M	180	193	208	233	302	412	694

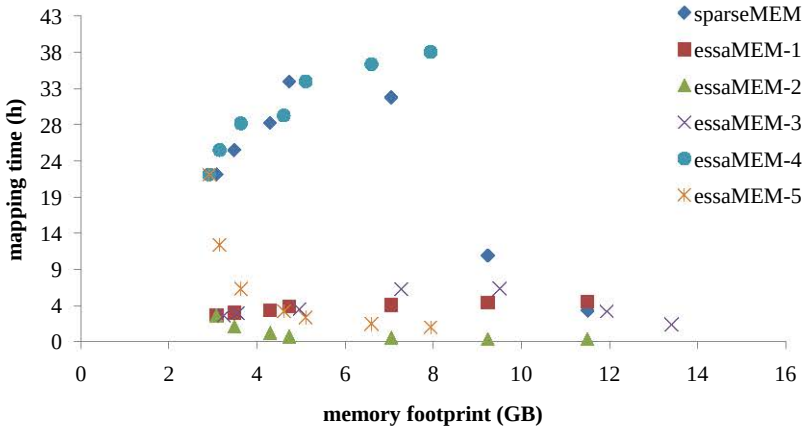
linear dependency on s could be removed by storing all suffix array indexes to disk, sorting them with an external memory algorithm, and finally streaming the uncompressed suffix array to get all required values.

3.4.4 Impact of optimizations

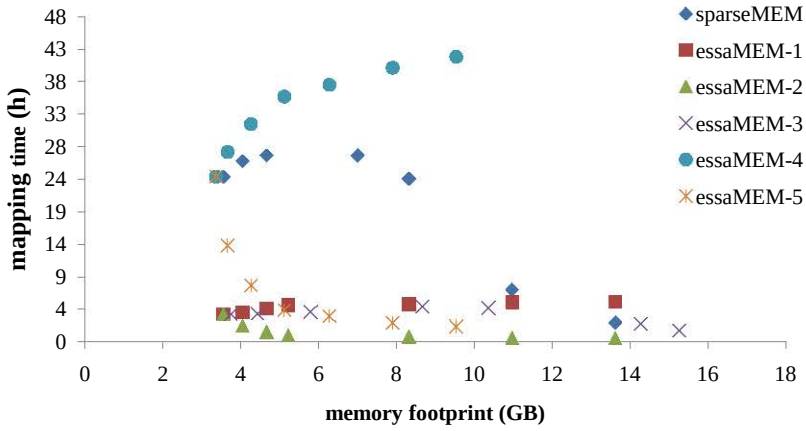
This section contains experimental results that can be used to assess the impact of the settings available in *essaMEM* (*essaMEM-1* to *essaMEM-5*). These results were obtained for the gigabase-sized genome data sets (data sets 7 and 8), and are shown in Figure 3.6. As a reference, Figure 3.6 also shows the results for *sparseMEM*. The difference in performance between *sparseMEM* and *essaMEM* is similar to that for the previous data sets. For the Mouse-Human data set, for example, *sparseMEM* becomes up to 50 times slower than *essaMEM-2*.

A first observation on the effect of our optimization is that the influence of parameter p is significant, especially for small values of s. For $s = 1$, optimizing p leads to an increase of performance of a factor 10 to 20. An extreme case is measured in the sequencing read data set for the Chicken genome (Figure 3.5a), where optimizing s results in a mapping time that is 40 times lower than the mapping time for $s = 1$.

The impact of suffix sampling can be seen by comparing the performance



(a) *M. musculus* vs *H. sapiens*



(b) *H. sapiens* vs *P. troglodytes*

Figure 3.6: Scatterplot showing the memory-time trade-offs of MEM-finding between a gigabe-sized genome data sets 7 and 8.

of *essaMEM-1* and *essaMEM-2* or *essaMEM-4* and *essaMEM-5*. The relative performance of parameter p and sparse suffix links can be found by comparing *essaMEM-2* and *essaMEM-3* or *sparseMEM* and *essaMEM-5*. These results show that for the given application and data sets, parameter p serves the same purpose as suffix links, as the effect is more significant for small sparseness factors. Suffix sampling in the pattern results, however, in a higher increase of performance in mapping time and does not require storage of additional data structures. Note, however, that suffix sampling increases the time required for checking the left-maximality of right-maximal exact matches.

In Figure 3.6, a decrease in mapping time of *sparseMEM* can be seen for large values of s. A possible reason for this could be the decrease in minimum length $(\ell - s + 1)$ during the matching phase, which results in less binary searches. In addition, better I/O performance and cache effects could positively affect the runtime of the program. A similar behavior is observed for *essaMEM* when the parameter p is not taken into account. For example, Figure 3.6 shows a decrease in runtime of *essaMEM* with $p = 1$ for increasing values of s. In addition to the effects that affect the runtime of *sparseMEM*, a higher sparseness value could increase the number of characters on the branches of the virtual sparse suffix tree, thus increasing the number of characters a sparse child array can match in one go.

The performance gain due to the use of the sparse child array can be seen by comparing *essaMEM-1* and *essaMEM-4* or *essaMEM-2* and *essaMEM-5*. The test results show that the performance gain is significant. This holds especially true for higher values of s, where the effect of either suffix sampling or suffix links diminishes. As a result, *essaMEM* remains competitive with *backwardMEM*, even for smaller memory settings.

If the enhanced sparse suffix array of *essaMEM* consists of both a sparse child array and sparse suffix links, the algorithm could benefit from both data structures. However, the tests do not show a significant improvement in mapping time of *essaMEM-3* compared to the minimum of the runtimes of *sparseMEM* and *essaMEM-1*. Furthermore, this setting has a higher memory requirement than other settings. Moreover, the suffix links and suffix sampling counteract each other. In some cases (see for example Figure 3.6a), the suffix links simulation even decreases the performance of this setting compared to *essaMEM-1*.

3.5 Related work

The algorithm and experimental results as described in this chapter are based on the initial release of *essaMEM* [248]. Since then, two new MEM-finding tool have been developed, called *slaMEM* [61] and *GPUMEM* [4]. *slaMEM* utilizes an FM-index augmented with both a new sampled LCP array representation and a new sampled data structure to obtain parent intervals in the LCP-interval tree. *GPUMEM* utilizes the massively parallel GPU threads together with a lightweight indexing structure.

Although *slaMEM* employs the *backwardMEM* MEM-finding algorithm, the authors did not implement a tunable memory-time trade-off. However, the single configuration of *slaMEM* is reported to be both fast and lightweight. For benchmarking their tool, the authors used data set 6 and data set 7 (the order of reference and query sequence was reversed for the latter data set). For the megabase-sized data set 6 (*D. melanogaster* versus *D. yakuba*), *slaMEM* is slightly faster than *essaMEM* using $s = 16$ (7% faster), but requires more memory (40% more). For higher sparseness settings, *slaMEM* is up to 70% faster than *essaMEM*. For the gigabe-sized genome data set, *slaMEM* is roughly 25% faster than the fastest setting of *essaMEM*, which was the setting with $s = 32$. For this setting, memory requirements of *slaMEM* were more than twice as high as those of *essaMEM*. For lower sparseness values, the difference in runtime between the tools becomes higher, and the memory footprint of *essaMEM* increases. As a result, *slaMEM* exhibits a very interesting time-memory trade-off that does not require optimizing parameter setttings.

Because *GPUMEM* is the only recent MEM-finding tool for the GPU, Abu-Doleh *et al.* compared the performance of their tool against CPU-based MEM-finding algorithms. *sparseMEM* and *essaMEM* are run using a maximum of 8 threads, but *slaMEM* and *MUMmer* do not support shared-memory parallelism. From the evaluations, *GPUMEM* outperforms all other tools on the tested data sets, with *essaMEM* having the highest performance among the CPU-based tools.

Since the initial release of *essaMEM*, we added a few improvements to the algorithm and implementation. The major change in the index structure and algorithm is the introduction of a k-mer table that connects sequences of fixed length k to the suffix intervals in the ESSA index structure. This table is designed to partially mitigate the loss of suffix links when using the suffix sampling by providing pointers to positions in the LCP interval tree at string depth k.

In detail, the new data structure consists of an array of length Σ^k, with one entry for every sequence of length k. An entry for sequence P stores the left and

right boundary of the maximal suffix array interval $[i..j]$, such that

$$S[SA[i]..SA[i]+k-1] = \ldots = S[SA[j]..SA[j]+k-1] = P,$$

but $S[SA[i-1]..SA[i-1]+k-1] \neq P$ and $S[SA[j+1]..SA[j+1]+k-1] \neq P$. If P is the LCP of the suffixes in the suffix array interval $[i..j]$, the entry stores the location of an LCP interval; if not the table entry points to an edge between two nodes in the LCP interval tree.

The new data structure is used on line 6 in Algorithm 3.3. Instead of starting at the root, the algorithm skips the first k bases and moves in constant time to string depth k in the virtual suffix tree, if possible. If the entry for the k-length prefix of the current query is empty, the algorithm breaks and continues with the next step of the loop.

The value of k is automatically decided as being equal to the minimum of 10 and $\ell - s \cdot p + 1$, which is equal to the minimum length of right-maximal exact matches in Algorithm 3.3. The maximum value 10 fixes the maximum memory requirements for this data structure to $8|\Sigma_{\text{DNA}}|^{10} = 8 \cdot 4^{10}$, which is less than 10MB. In this calculation, we used 2 four-byte integers to store the suffix array interval boundaries. In practice, the additional memory requirements are negligible compared to the rest of the index.

Table 3.5 shows the increase in performance by using the k-mer table. The values in this table represent the gain in performance, expressed in relative speed-up, by using the k-mer table in the *essaMEM-2* setting. The impact of this improvement is significant, as runtime is up to three times lower than the previous optimal runtime of *essaMEM*.

In addition to the introduction of the k-mer lookup table, we also added the option to store the ESSA index to disk for later usage.

Table 3.5: The effect of the introduction of a k-mer table in the ESSA index structure. The values in this table represent the gain in performance, expressed in the relative speed-up, by using the k-mer table in the *essaMEM-2* setting. The rows represent the different data sets in Table 3.2 and the columns represent different sparseness values.

data set	s					
	1	2	4	8	16	32
1	2.50	2.73	2.96	3.21	2.00	
2	1.24	1.37	1.33	1.51	2.43	2.68
3	1.43	1.65	1.61	1.82	2.70	2.97
4	1.24	1.38	1.39	1.59	2.32	2.61
5	1.32	1.50	1.39	1.59	2.53	2.75
6	1.48	1.71	1.44	1.67	2.78	2.94
10	1.17	1.27	1.27	1.42	2.06	2.34

Chapter 4

ALFALFA

This chapter covers a novel read mapper called *ALFALFA*. Rapid evolutions in sequencing technology force read mappers into flexible adaptation to longer reads, changing error models, memory barriers and novel applications. *ALFALFA* achieves high performance in accurately mapping long single-end and paired-end reads to gigabase-scale reference genomes, while remaining competitive for mapping shorter reads. Its seed-and-extend workflow is underpinned by fast retrieval of super-maximal exact matches from an enhanced sparse suffix array, with flexible parameter tuning to balance performance, memory footprint and accuracy. The chapter is based on the article "*A long fragment aligner called ALFALFA*" [249].

4.1 Introduction

Mapping sequencing reads to reference genomes plays a key role in many genomics analysis pipelines. For a single read, it consists of finding optimal substring alignments among all alignments between the read and reference genome. A more technical description of the read mapping problem is given in Section 1.3.2. Due to the size of the data sets, read mappers rely on a combination of efficient index structures, search algorithms and a multitude of heuristics. In addition, read mappers and their underlying index structures are under constant development to handle specific applications or data models and to further improve implementations [153,247]. An overview of the similarities and differences between existing read mappers is given in Section 1.3.3.

Recent advances in next-generation sequencing technologies have led to increased read lengths, higher error rates and error models showing more and longer indels. This general trend is likely to continue with third-generation sequencing technologies like Oxford Nanopore and Pacific Biosciences [181]. Most of the current read mappers target short reads and allow for no or low numbers of mismatches and/or indels. This makes them vulnerable to the ongoing technological advances. It has inspired a second generation of novel read mappers (*GEM* [179]), while authors of short read mappers present new versions equipped for aligning longer reads with higher error rates (*Bowtie* 2 [141], *BWA-SW* [151], *BWA-MEM* [148] and *CUSHAW3* [162]). Recurring strategies include increasing the seed lengths, clustering neighboring seeds into candidate regions and optimizing the implementations of global and local alignment algorithms using banded and bit-parallel versions. However, except for *BWA-SW* and *BWA-MEM*, none of the existing mappers scales well for read lengths up to several kilobases.

The read mapper *ALFALFA* is extremely fast and accurate at mapping long reads ($>$ 500bp), while still being competitive for moderately sized reads ($>$ 100bp). Its implementation of the canonical seed-and-extend approach is empowered by a novel index structure, combined with several new optimizations and heuristics. Both end-to-end and local read alignment are supported, and several strategies for paired-end mapping can efficiently handle large variations in insert size. *ALFALFA* is unique in using enhanced sparse suffix arrays to index reference genomes. This data structure facilitates fast calculation of maximal and super-maximal exact matches [248] and supports the important design goal of balancing between processing speed, memory consumption and mapping accuracy. The speed-memory trade-off is tuned by setting the sparseness value of the index. The techniques and heuristics used to filter and combine seeds and candidate regions are designed to handle longer reads. Furthermore, *ALFALFA* uses a chaining algorithm to speed up dynamic programming extension of candidate regions.

The name of the read mapper *ALFALFA* is an acronym for "A Long Fragment Aligner/A Long Fragment Aligner". It is repeated twice as a pun on repetitive and overlapping fragments observed in genome sequences that heavily distort read mapping and genome assembly. The reference to the flowering plant *Medicago sativa* in the pea family *Fabaceae* bearing the same vernacular name underscores the ability of enhanced sparse suffix arrays to index entire plant genomes (that are among the largest genomes known), without giving up expressiveness for fast string matching with respect to the memory-intensive suffix trees and suffix

arrays.

The algorithms and heuristics used in *ALFALFA* are presented in Section 4.2. This section first covers our implementation of the seed-and-extend alignment strategy on a high level. Afterwards, each individual step and heuristic is discussed in more detail. The design choices made in the creation of *ALFALFA* are motivated by discussing their effect on performance or using case studies that were encountered during the development of the algorithm. In Section 4.3, *ALFALFA* is evaluated on a wide variety of real and simulated read data and compared against other long read mappers.

4.2 Algorithms & heuristics

The ultimate goal set by *ALFALFA* is to report a number of *different feasible* alignments (see Definition 1.5) between the reads and the reference genome. The minimum score d required for an alignment to be deemed feasible is calculated using a parameter expressing the maximum percentage of differences based on the read length. This parameter is also used to determine when two alignments are considered different, namely when the difference between their mapping positions is less than the maximal allowed edit distance.

Due to huge differences in size between reads and reference genomes, most read mappers share a high-level strategy of *i*) finding matching segments that are used to *ii*) prune the search space to genomic regions in which *iii*) alignments are found that meet a particular scoring threshold. These steps are usually preceded by a step in which either the reference genome or read data set is indexed. The *ALFALFA* algorithm is outlined in Figure 4.1. *ALFALFA* is the first read mapper that is underpinned by an ESSA index structure ((Figure 4.1 step *a*)). *ALFALFA* takes advantage of the technological evolution by using MEMs and SMEMs as seeds (Figure 4.1 step *b*). These seeds are then extensively filtered and triaged to allow for more accurate prioritization of candidate regions (Figure 4.1 step *c*). To further limit the number of expensive dynamic programming computations needed, *ALFALFA* chains seeds together to form a gapped alignment. As a result, the extension phase is limited to filling gaps in between chains while evaluating alignment quality (Figure 4.1 step *d*).

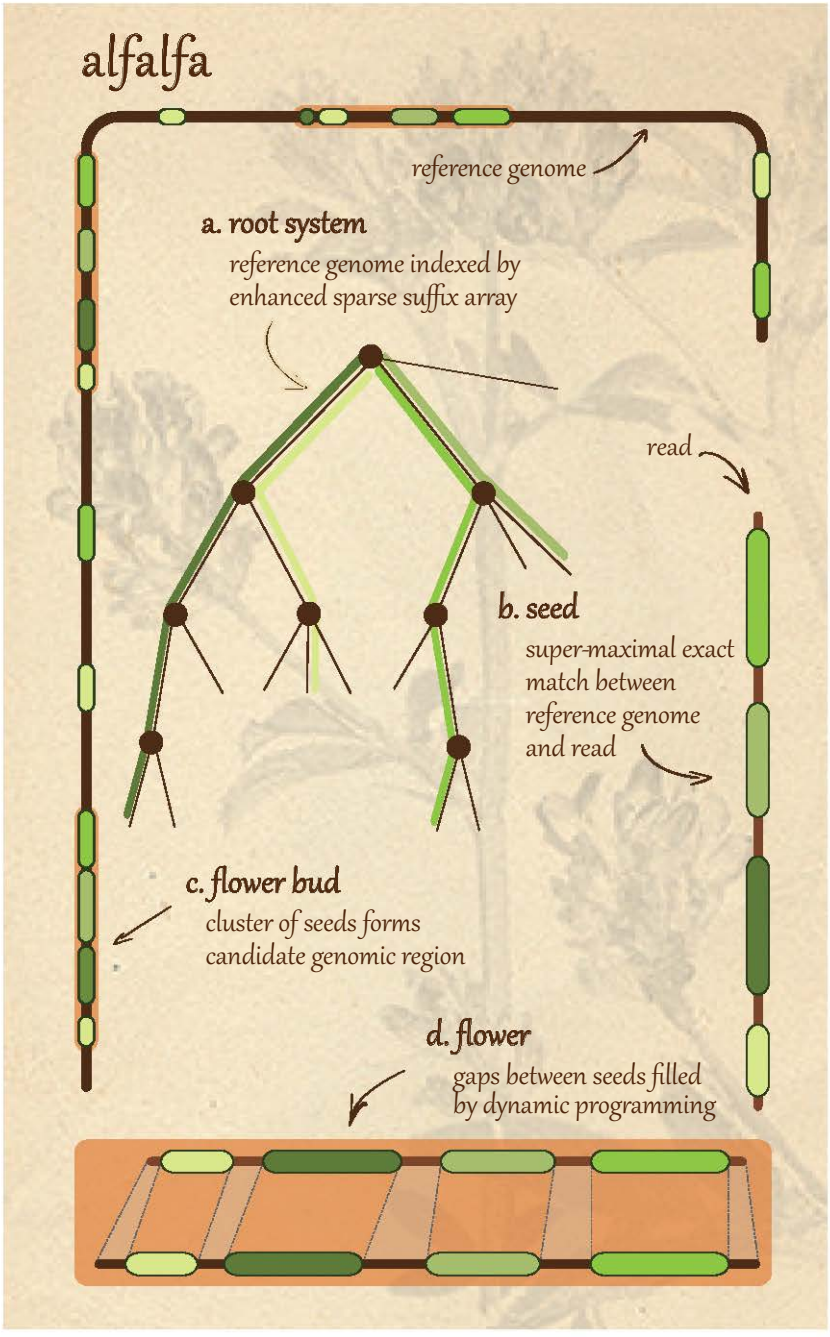

alfalfa
reference genome
a. root system
reference genome indexed by
enhanced sparse suffix array
read
b. seed
super-maximal exact
match between
reference genome
and read
c. flower bud
cluster of seeds forms
candidate genomic region
d. flower
gaps between seeds filled
by dynamic programming

Figure 4.1: *(previous page) ALFALFA* follows a canonical seed-and-extend workflow for mapping reads onto a reference genome. The reference genome is indexed by an ESSA (a) to enable quick retrieval of MEMs and SMEMs between a read and the reference genome (b). Seeds are then grouped into non-overlapping clusters that mark candidate genomic regions for read alignment (c). Handling of candidate regions is prioritized by agglomerate base pair coverage of the seeds. The final extend phase samples seeds from candidate regions to form collinear chains that are bridged using banded dynamic programming (d). All of these steps strive to make optimal reuse of seeds in order to avoid superfluous computations. Background image used with permission from Walter Obermayer.

Another outline of the high-level single-end alignment strategy of *ALFALFA* is given in Algorithm 4.1. The seeds calculated in line 4 are filtered and grouped together into candidate regions in line 9. After being sorted, these candidate regions are further examined in line 13 to see if they may contain a good alignment. The seeds in the regions that meet the criteria are chained together in line 16, after which the chains are extended in line 17 using dynamic programming. The result of the extension procedure is an alignment score, which is converted into a full alignment in line 28 for the best alignments found.

The various sub procedures in Algorithm 4.1 are discussed in detail in the next sections. These sections contain a description and the pseudocode of the algorithms and heuristics used, supplemented with examples to illustrate design choices and a discussion on advantages and disadvantages of the methods and viable options for parameter settings, when applicable.

Algorithm 4.1 outline of single-end mapping algorithm used by *ALFALFA*

Input: reference genome G, list of reads L_R
Output: SAM file containing alignments of reads onto G

```
1: read input and parse parameter options
2: construct enhanced sparse suffix array I for G
3: for all reads R in L_R do
4:     seeds ← CALCULATESEEDS(G, R, I)
5:     if no seeds were found then
6:         lower minimum seed length
7:         seeds ← CALCULATESEEDS(G, R, I)
8:     end if
9:     regions ← IDENTIFYCANDIDATEREGIONS(seeds)
10:    sort regions according to seed coverage of the read sequence
11:    while regions remaining and not enough feasible alignments found do
12:        C ← current region
13:        if EXTENDREGION(C) then
14:            sort seeds in C according to length
15:            for all seed s ∈ C do
16:                chain ← BUILDCHAIN(s, C)
17:                alignment ← EXTENDCHAIN(chain)
18:                if FEASIBLEALIGNMENT(alignment) then
19:                    add alignment to R
20:                end if
21:            end for
22:        end if
23:    end while
24:    if no feasible alignments found then
25:        RESCUEPROCEDURE(G, R, I, regions)
26:    end if
27:    sort feasible alignments
28:    POSTPROCESS(R)
29: end for
```

As an example, Figures 4.2 to 4.4 depict the flow of the algorithm for one read. Figure 4.2 shows the reference genome as separate segments (chromosomes or contigs) organized in a circle around a read sequence (light green) in the center. Note that the read sequence is not drawn in scale with the reference genome. The figure also shows the location of all seeds found as black bars around the read (in this case all MEMs with $\ell \geq 30$ were used as seeds). The seeds are also connected to their position in the reference genome. Bars above (below) the read represent seeds located on the forward (reverse) strand. The dots on the reference genome represent regions containing MEMs and are thus candidate mapping locations. In addition, dots in which more lines converge will have a higher score as more MEMs are found in there. In this example, most regions will contain only a single small seed and will thus be filtered out in later phases. Furthermore, the seeds span the entire read sequence only in a single region. This region, identified by the black dot outside the circle, is also the region from which the read was simulated.

In Figure 4.3, we have zoomed in to the area around the highest scoring candidate region. The read and reference genome are shown in the same way as in Figure 4.2, but now the displayed region of the reference genome is small enough to show the exact location of the seeds on the reference genome. The black arc outside the reference genome shows the location of the candidate region as delineated by *ALFALFA*. This candidate region extends somewhat beyond all seeds to take into account parts of the read that are not covered by seeds and additional insertions. The black segment on the reference genome corresponds to the location from which the read was simulated, thus corresponding to the location of an optimal alignment. The simulated region does not include the first two seeds, but does extend beyond the last seed. As a result, the candidate region identified by *ALFALFA* completely contains the simulated region.

Finally, Figure 4.4 illustrates the results of the chaining and alignment phase of the algorithm. The candidate region found by *ALFALFA* is shown at the top and the read at the bottom. Most read mappers extend candidate regions by performing full dynamic programming within the boundaries of the region. *ALFALFA*, however, combines seeds in the candidate region into a chain, covering large sections of the matching bases of the alignment. The seeds in the example are depicted as connected bars between the read and reference genome. Seeds making up the chain are shown in black, whereas those not included in the chain are shown in red.

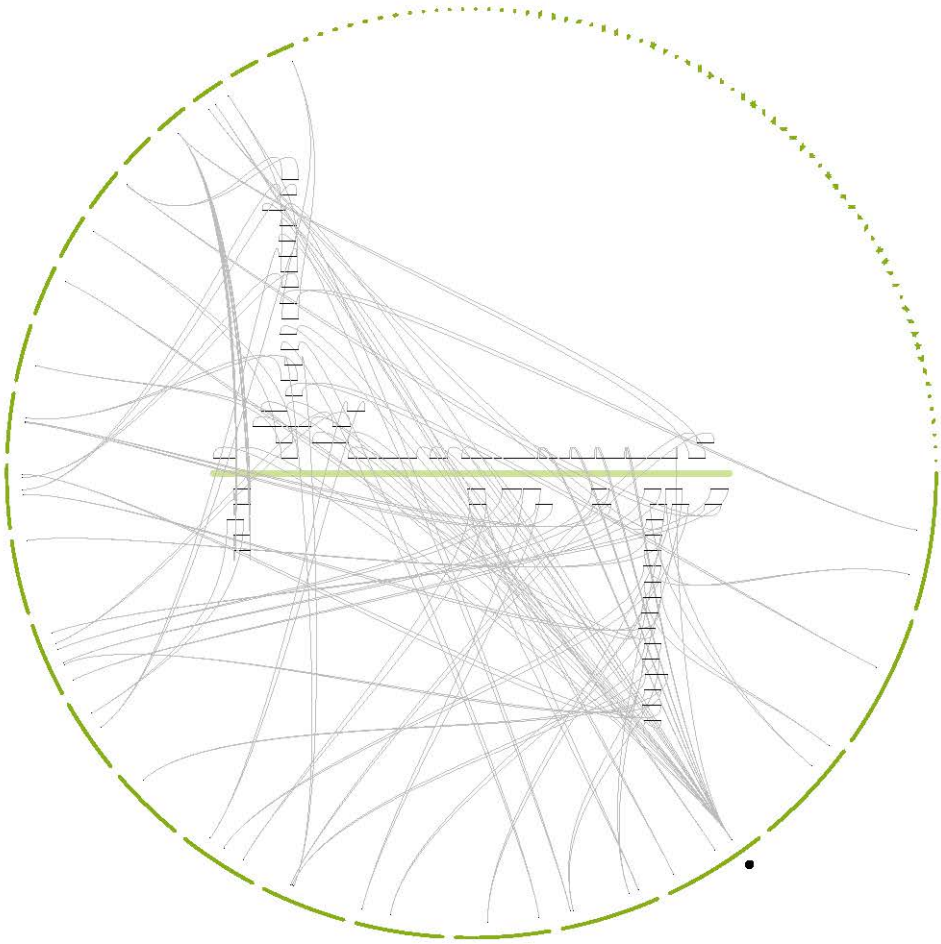


Figure 4.2: Example illustrating the general workflow of *ALFALFA*, showing the collection of seeds (MEMs) that are found between a read (light green bar in the center) and a reference genome (green circle, each segment is a chromosome or contig). Seeds are depicted as black bars around the read, on top of the read for seeds found on the forward strand and below the read for seeds found on the reverse strand. The seeds are connected to their location on the reference genome. As a reference, the black dot outside the circle represents the simulated location of the read.

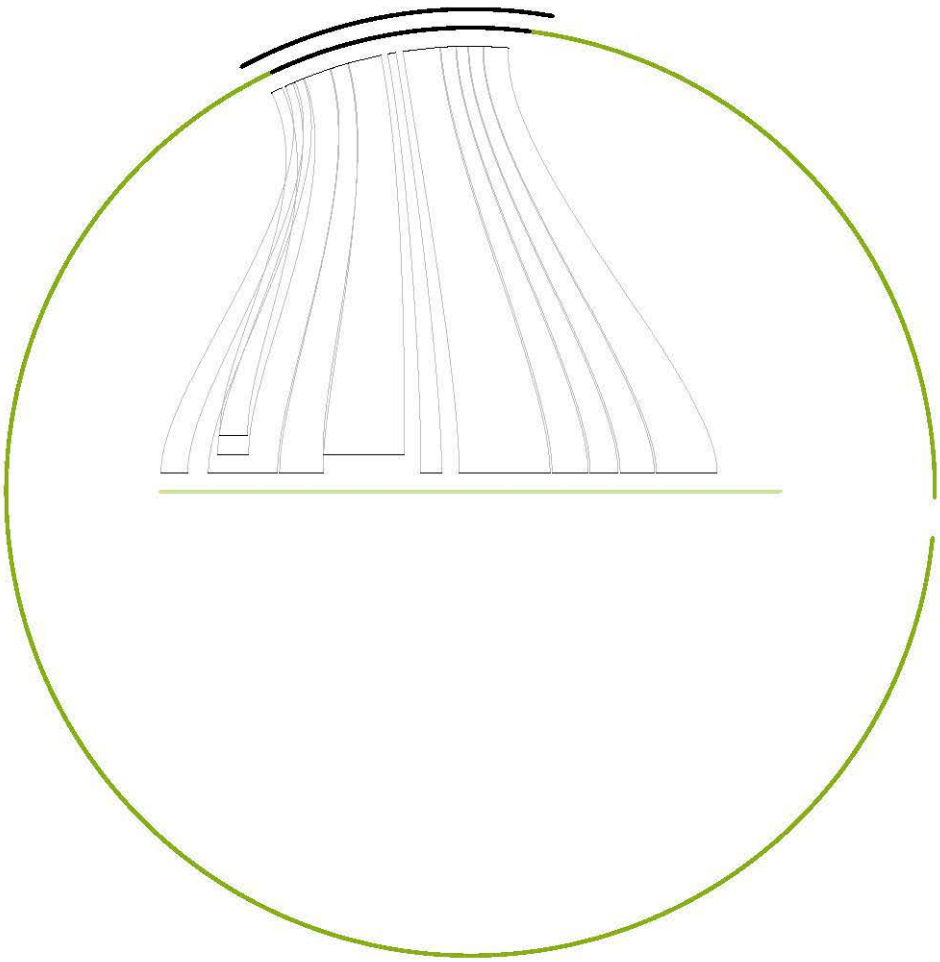


Figure 4.3: Example illustrating the general workflow of *ALFALFA*, showing the collection of seeds (MEMs) in a single candidate region. Seeds between the read (light green bar in the center) and the area around the candidate region (green circle) are depicted as connected black lines. The black segment on the reference genome indicates the region of an optimal (simulated) alignment and the black segment outside the circle indicates the part of the reference genome defined by *ALFALFA* as the candidate alignment region.

Dynamic programming is only performed in between two seeds and at the edges of the candidate region. As a result, large sections of the alignment are already fixed before costly dynamic programming routines are applied. The figure also clearly illustrates the importance of finding a good set of seeds. Ideally, the chain covers as much of the read sequence as possible. To illustrate the quality of the seeds for this example, grey areas in Figure 4.4 represent parts of an optimal alignment that consists of three or more consecutive matching bases. Among those, light grey areas are covered by seeds, whereas darker ones are not. In this example, all large matching areas are covered, except some at the end of the read. Most matches that were not found as seeds are due to the lower limit imposed on seeds by *ALFALFA* during seed-finding.

4.2.1 Seed-finding

Seed-finding is the first major phase in the mapping process. Depending on the data and parameter settings, it usually takes about a quarter to half of the total runtime. Ideally, seed-finding produces a limited number of long seeds that cover as much of the mapping location as possible. Finding too many seeds results in an exponential increase of candidate mapping locations and usually favors highly repetitive regions in the genome. As a result, mapping locations may be missed. Finding too few seeds results in a possible loss of good mapping locations and shorter chains. The latter may increase the computational cost of the extension phase.

One of the strongholds of *ALFALFA* is its use of the *essaMEM* algorithm described in Chapter 3 as a way to identify seeds. As a result, *ALFALFA* inherits several parameters from *essaMEM* to tune the performance of the seed-finding process such as the sparseness of the index, suffix sampling in the read and the minimum seed length. In addition, *ALFALFA* can limit the maximum number of seeds per read offset to further reduce the number of seeds. Moreover, *ALFALFA* implements two new MEM-finding algorithms that focus on finding SMEMs (see Definition 1.3). The choice of seed-finding algorithm also impacts the performance of the read mapping algorithm.

By default, *ALFALFA* uses a fixed sparseness value of 12 for reference genomes that have a similar size to the human genome. An automated choice of the suffix sampling in the read is made using the function described in Section 3.4.1. *ALFALFA* finds variable-length seeds with a fixed minimum length, which is 40 by default. The default was chosen, based on our experiments for the human reference genome and varying read lengths.

Figure 4.4: Example illustrating the general workflow of *ALFALFA*, showing the candidate alignment region found by *ALFALFA* (top) and the read (bottom), together with the seeds in the candidate region (connected bars). Seeds used in the chain are depicted in black, and seeds that are filtered out are depicted in red. Grey areas show regions in an optimal alignment of three or more consecutive matching bases. Light gray areas are covered by seeds, whereas dark gray areas are found using dynamic programming.

For shorter genomes, this minimum seed length could be lowered to increase sensitivity of the algorithm. On the other hand, *ALFALFA* automatically tunes the minimum seed length for reads longer than 1000bp. For those reads, the minimum seed length ℓ is incremented by 20 for every 500bp above 1kbp, with the total increment being divided by the maximum percentage of errors allowed in accepting alignments.

Super-maximal exact matches

For long read mapping, the performance of the *essaMEM* MEM-finding algorithm described in Chapter 3 does not suffice. Furthermore, the number of MEMs between a short read sequence and a large reference genome can be very high, hindering fast identification of a small set of candidate mapping locations. As a result, *ALFALFA* implements two other seed-finding algorithms that focus on finding SMEMs in addition to a small subset of rare or very large MEMs. The use of SMEMs as seeds is motivated by the fact that the MEM with the highest similarity to a part of the reference genome, *i.e.* the longest, will most likely be found in a region containing a good alignment between read and reference genome.

We have designed Algorithm 4.2 to find a smaller set of MEMs, of which most are SMEMs. Furthermore, it is guaranteed to find all SMEMs if there is no sparseness in the read or index of the reference genome. Technically, the algorithm is highly similar to the MEM-finding algorithm given in Algorithm 3.3. The main difference is that Algorithm 4.2 does not keep track of *all* right-maximal exact matches longer than the minimum length. Given an offset in the read, left-maximality is only checked for the longest right-maximal exact matches.

Lemma 4.1. *If the sparseness s of the ESSA and the sparseness p in the read are equal to one (no sparseness), the set of MEMs returned by Algorithm 4.2 is exactly the set of all SMEMs.*

Proof. To see this, suppose that $m = (i, j, \ell)$ is an SMEM that corresponds to the substring $S[i..i+\ell-1]$ in the reference genome S and $P[j..j+\ell-1]$ in the read P. If there is no sparseness, the algorithm loops over all offsets of P. For offset j, the algorithm will match exactly ℓ characters because *i)* at least ℓ characters can be matched due to $P[j..j+\ell-1] = S[i..i+\ell-1]$ and *ii)* no more than ℓ characters can be matched due to m being an SMEM. Similarly, the algorithm will not find any right-maximal match with offset j and length ℓ that is not left-maximal,

Algorithm 4.2 SMEM-finding algorithm

Input: reference genome G, ESSA index I and read sequence R
Output: a set of SMEMs between G and R

$\ell \leftarrow$ MINIMUMLENGTH($|P|$,e) // *minimum similarity* $= 100 - e$
$s \leftarrow$ sparseness of I
$p \leftarrow$ SPARSENESS(ℓ,s) // *p is the sparseness in the read*
// *ℓ_s is the minimum length of right-maximal exact matches*
$\ell_s \leftarrow \ell - (p \cdot s - 1)$
for all i **in range** $[0..s-1]$ **do**
    for j **from** i **to** $|R| - \ell_s$ **do**
        $xmi \leftarrow$ match $R[j..]$ to I
        if # *characters matched* $\geq \ell_s$ **then**
            // *xmi contains the longest right-maximal exact matches*
            for all right-maximal exact matches in *xmi* **do**
                compare up to $s \cdot p$ characters to check for left-maximality
            end for
            if $\max(s, p) = 1$ **and** all matches in *xmi* are MEMs **then**
                report all matches in *xmi* as SMEMs
            else if $\max(s, p) > 1$ **then**
                $\ell_{min} = \ell - s$
                report MEMs in *xmi* whose length is at least ℓ_{min}
            end if
        end if
        $j \leftarrow j + s \cdot p$
    end for
end for

because otherwise a longer MEM could be found that contains m. As a result, the condition in line 12 has been met and m is reported.

Conversely, if the algorithm reports a MEM $m = (i, j, \ell)$, it is an SMEM. No other MEM matched $P[j..j + \ell]$ because it did not match any substring of S (line 7). As a result, there is no other MEM that covers $P[j..j+\ell-1]$ and extends to the right of it. Similarly, if there would be a MEM covering at least $P[j..j+\ell-1]$ and extending to the left of it, this match would be found as a right-maximal exact match together with m, but it would not be left-maximal. In that case, the condition in line 12 would be false and m would not be reported. Therefore, no other MEM can be found that is longer than ℓ characters and in which m is fully contained. □

By default, *ALFALFA* induces sparseness in both the read and reference genome. In this case, Algorithm 4.2 is not guaranteed to produce the set of all SMEMs. An example of an SMEM that is not reported and a reported MEM that is not an SMEM are shown in Figure 4.5. The SMEM $m = (2, 2, 5)$ covering $S[2..6]$ and $P[2..6]$ is not found. The suffix $S[2..]$ is not sampled and therefore m cannot be found if the offset in P is 2. For offset 3, m would be reported by Algorithm 3.3 because it is a MEM. For offset 3, however, Algorithm 4.2 finds a longer right-maximal exact match $(7, 3, 7)$. It is therefore unable to find m. Although the MEM $m' = (19, 19, 8)$ is contained in the longer MEM $m'' = (30, 19, 11)$, and is thus not an SMEM, it will be reported. This is because suffix $S[30..]$ is not sampled and thus m'' is not found at offset 19 in P but in offset 20.

Although Algorithm 4.2 does not have the nice theoretical property of guaranteeing to find all SMEMs, we have found that the set of seeds produced by the algorithm is sufficient for our purposes. In a post-processing step, it is easy to filter out all SMEMs from the data set returned by Algorithm 4.2. In addition, the larger data set allows *ALFALFA* to build larger chains for use in the extension phase of the mapping algorithm. For this reason and to somewhat counter the effect of sparseness, we have decreased the minimum seed length requirements in line 15.

From our experiments, we found that the reduced set of SMEMs is ideal for identifying a small set of candidate regions, but was sometimes too small to build large chains to be used in the extension phase of the algorithm. Algorithm 4.3 is inspired by the seed-finding procedure in *BWA-MEM* and is a combination of previous algorithms in that it will produce more MEMs in regions in which few SMEMs are found. The algorithm first proceeds as in Algorithm 3.3 by finding all right-maximal exact matches of a given minimal seed length.

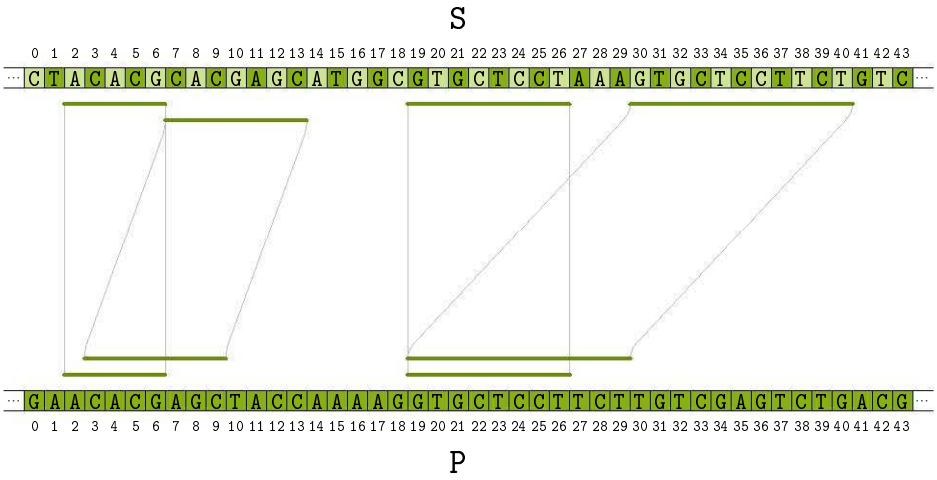


Figure 4.5: MEMs of minimum length 5 between reference genome S and read P. Among the MEMs, only the one of length 8 with offset 19 in both S and P (third from the left) is not an SMEM, as it is contained in a larger MEM. In addition to the MEMs, the figure shows the sampling of S in an ESSA index structure with sparseness 2. Only suffixes that start at dark green characters are sampled in the index.

It then sorts the intervals of right-maximal exact matches in descending order of rightmost matched position in the read. This sort procedure is used to filter out intervals of right-maximal exact matches that will not be SMEMs in line 14. Finally, the algorithm either finds all MEMs as in Algorithm 3.3 or only the longest MEMs for an offset in P as in Algorithm 4.2 based on the size of the latter data set (condition in line 15).

Comparison of seed-finding settings

The parameter settings not only have an impact on the performance of the seed-finding phase, but also affect the rest of the mapping algorithm. Experimental results have shown that the use of Algorithm 3.3 results in the highest accuracy, but it can also lead to a runtime that is three to four times higher than the runtime achieved by using the other seed-finding algorithms. Moreover, the difference in accuracy is in most cases less than 0.01%.

Algorithm 4.3 MEM and SMEM-finding algorithm

Input: reference genome G, ESSA index I and read sequence R
Output: a set of MEMs between G and R

```
    // define variables similar to Algorithm 3.3
 1: ℓ ← MINIMUMLENGTH(|R|,e)
 2: s ← sparseness of I
 3: p ← SPARSENESS(ℓ,s) // p is the sparseness in the read
 4: ℓ_s ← ℓ − (p · s − 1)
 5: intervals ← empty list
 6: for all sampled positions i in R do
 7:     calculate mli and xmi in a similar way to Algorithm 3.3
 8:     intervals[i] ← (i, mli, xmi)
 9: end for
10: sort intervals in descending order of rightmost matched position in R
11: smems ← empty dictionary
12: for all (i, mli, xmi) in intervals do
13:     d ← length of right-maximal match in xmi
14:     if [i..i + d] is not contained in smems then
15:         if |xmi| ≤ EXT then // EXT is a parameter, default 10
16:             report all mems up to mli
17:         else
18:             report all mems from xmi
19:         end if
20:         update smems dictionary
21:     end if
22: end for
```

Overall, Algorithm 4.2 provides the best trade-off, with Algorithm 4.3 resulting in a slightly higher accuracy, at the cost of a small increase in runtime.

In addition to the choice of seed-finding algorithm, a higher minimum seed length results in a lower runtime, but misses some alignments due to the lower number of seeds. In contrast, a lower minimum seed length results in more seeds, which in turns results in a higher chance of finding a good alignment at the cost of increased runtime.

As an example of the impact of the different seed-finding strategies and parameter settings on the progress of the algorithm, Figures 4.6 and 4.7 show possible changes in the example shown Figures 4.2 to 4.4. Figure 4.6 depicts the situation of a high number of seeds resulting from a low minimum seed length setting. Compared to Figure 4.2, a lot more candidate regions will be identified and analyzed, resulting in a higher runtime. In general, finding more seeds will increase the runtime of all phases of the algorithm, while increasing accuracy. In some cases, however, an abundance of seeds may lead to some alignments being missed, as the algorithm might not be able to distinguish good candidate regions or may be unable to form a good chain from the seeds in a region. In contrast, Figure 4.7 shows the most common effect of a low number of seeds. The algorithm will usually still be able to recover all candidate regions containing optimal alignments, but the chain will be made up of fewer seeds. This will increase the runtime of the extension phase of the algorithm.

Overall, the trade-offs obtained by the different settings do not vary greatly. This is probably due to the many heuristics employed in the remainder of the algorithm that are designed to cope with a wide array of (badly chosen) parameter settings. One of these heuristics is a rescue procedure which is used when no seeds were found. This rescue procedure will lower the minimum seed length twice in an attempt to find seeds. If no seeds were found after this procedure, the algorithm skips to the next read.

4.2.2 Candidate regions

ALFALFA groups the seeds found in the previous phase of the algorithm into *candidate regions* of roughly the size of the read length. These candidate regions are subsequently sorted and explored further if they meet certain criteria. This part of the algorithm corresponds to lines 9-13 and 25 in Algorithm 4.1.

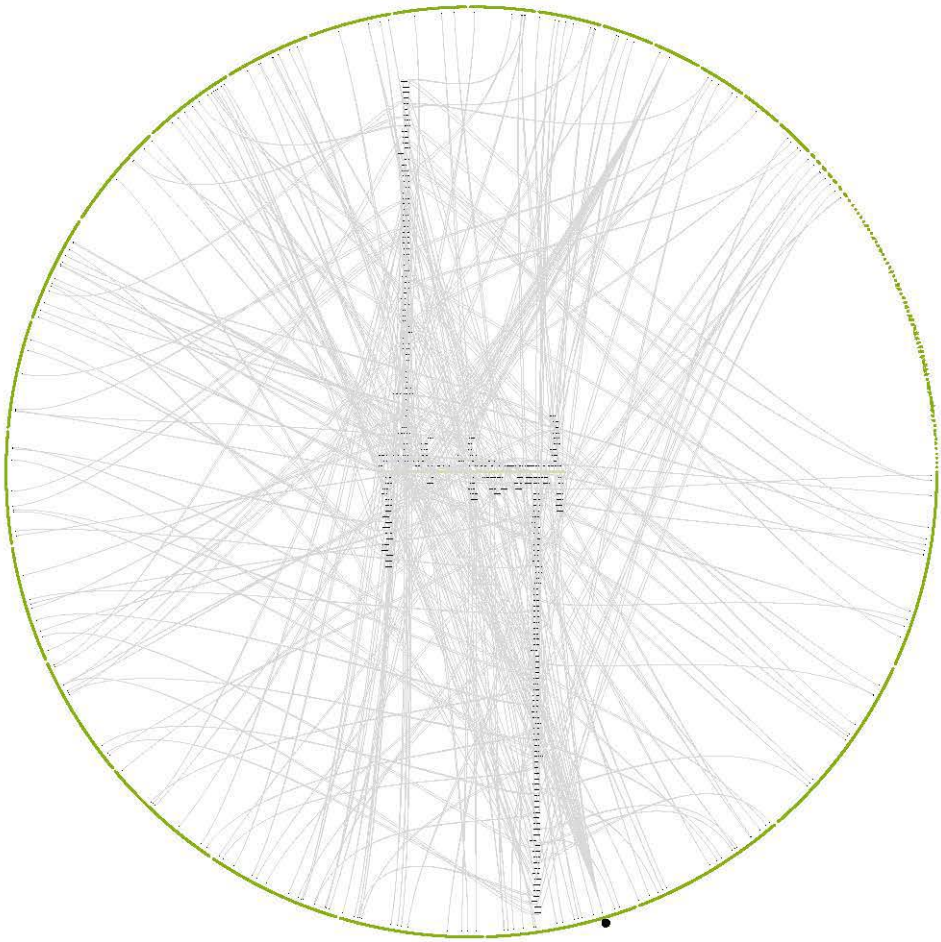


Figure 4.6: Example illustrating the effect of too many seeds found in the read in Figure 4.2, showing the collection of seeds (MEMs) that are found between a read (light green bar in the center) and a reference genome (green circle, each segment is a chromosome or contig). The seeds are depicted as black bars around the read, on top of the read for seeds found on the forward strand and below the read for seeds found on the reverse strand. The seeds are connected to their location on the reference genome. The black dot outside the circle represents the simulated location of this read.

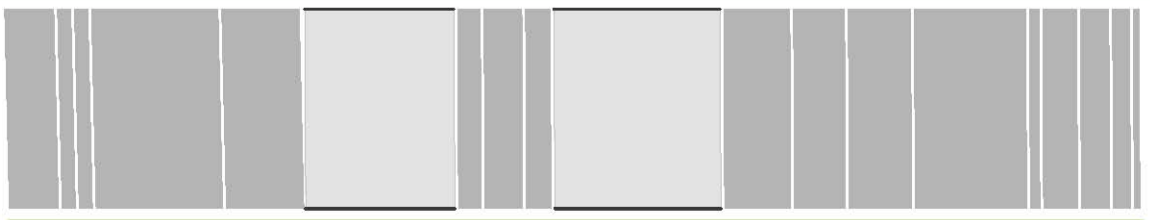


Figure 4.7: Example illustrating the effect of too few seeds in the read in Figure 4.4, showing the candidate alignment region found by *ALFALFA* (top) and the read (bottom), together with the seeds in the candidate region (connected bars). Seeds used in the chain are depicted in black, and seeds that are filtered out are depicted in red. Grey areas show regions in an optimal alignment of three or more consecutive matching bases. Only few areas are covered by seeds (light grey areas), whereas many areas are found using dynamic programming (dark grey areas).

Candidate region identification

Seed-finding in *ALFALFA* serves two goals, namely *i)* identifying few candidate regions that have a high likelihood of containing high-scoring alignments and *ii)* creating collinear chains that cover the entire read. Algorithm 4.4 takes both goals into consideration when identifying candidate regions. The algorithm first filters the set of seeds to only the longest and rarest seeds, which serves the first goal. Candidate regions are then identified using the filtered list. Afterwards, however, seeds that were filtered out are merged with existing candidate regions and reused during chain building.

In detail, Algorithm 4.4 filters out all SMEMs appearing more than 10 000 times in the reference genome, and all MEMs that are not SMEMs if they are either shorter than 50bp or appear more than 20 times in the reference genome. The parameters employed are similar to those used in the seed-finding Algorithm 4.3 and are not hard coded. Filtering the MEMs this way significantly reduces the number of candidate regions and thus the runtime of the algorithm. For example, on a data set containing half a million 600bp reads simulated from the human genome, the filter decreased the number of candidate regions ten-fold and lowered the runtime five-fold.

The filtered list of seeds is then sorted by offset in the reference genome and traversed left-to-right to find candidate regions. Candidate region boundaries are formed by extrapolating seed boundaries to cover the entire read, supplemented with additional bases to take into account gaps in the alignment. *ALFALFA* adds seeds to a candidate region as long as they fit the extrapolated boundaries.

Candidate region boundaries are recalculated each time a seed is added that is longer than the longest seed already included in the region. This is based on the idea that longer seeds are more reliable in identifying a candidate region. After recalibration of the candidate region boundaries, *ALFALFA* checks if the region would overlap the previous one. If this is the case, both regions are merged.

Merging of overlapping regions is performed to avoid possible loss of information in repeated regions. As an example, Figure 4.8 shows a situation in which parts of the read contain tandem repeats. The black arc on the reference genome (green circle) represents the location of an optimal alignment. Most seeds fall within this region, but several seeds were also found preceding this region. Due to the left-to-right nature of the candidate region identification, a first candidate region is started from the seeds preceding the optimal alignment region. This region extends to halfway the optimal region, after which a second candidate region is started.

Algorithm 4.4 Identify candidate regions

Input: set of MEMs and SMEMs *seeds* between reference genome G and read R
Output: list of candidate regions made up from a subset of the seeds

```
1: topSeeds, reserve ← empty list
2: diff ← maximum allowed differences
3: for all s in seeds do  // split the set of seeds
4:     if (s is an SMEM and occ(s, G) < 10 000)
5:                 or (occ(s, G) < 20 and |s| ≥ 50) then
6:         add s to topSeeds
7:     else
8:         add s to reserve
9:     end if
10: end for
11: sort topSeeds according to offset in the reference genome
12: for all s in topSeeds do  // identify candidate regions
13:     i ← offset of s in G
14:     j ← offset of s in R
15:     ℓ ← length of s
16:     C ← current candidate region
17:     if G[i..i + ℓ − 1] falls within C then
18:         if ℓ > longest seed in C then // recalculate boundaries of C
19:             left bound of C ← i − j − diff
20:             right bound of C ← i − j + |R| + diff
21:         end if
22:         add s to C
23:         if C now overlaps other regions then
24:             merge C with other overlapping regions
25:         end if
26:     else
27:         start new candidate region with s
28:     end if
29: end for
30: for all s in reserve do  // merge reserve with candidate regions
31:     if s can be fit in existing region then
32:         add s to existing region
33:     end if
34: end for
```

Both candidate regions cover only part of the optimal alignment region and would lead to sub-optimal alignments. Using the merging algorithm, however, a single candidate region is created in which the optimal alignment is found.

After candidate region identification, seeds filtered out at the beginning of Algorithm 4.4 are merged back into the candidate regions. The set of all seeds making up a region will be used to rank the region and to eventually build the chain. The extra seeds help to avoid situations such as that of Figure 4.7 in which the extension procedure is almost as costly as full dynamic programming.

Candidate region selection

After candidate region identification, the resulting list of regions is sorted according to the percentage of bases in the read that are covered by at least one seed. The coverage can be easily computed by traversing the list of seeds that is sorted by start position in the read and calculating the overlap between two consecutive seeds in the list. This second sort does not require additional computational resources as it is mandatory for the chaining algorithm in the extension phase of the algorithm.

In previous experiments, we also tested sorting candidate regions according to the sum of the lengths of the seeds making up the candidate region, but this advantaged adversely repetitive regions in the reference genome.

The list of candidate regions is then traversed in sorted order and candidate regions are extended until a maximum number of feasible alignments have been found (line 4.1.11). By default, *ALFALFA* searches for a maximum of 5000 feasible alignments meeting a minimum score threshold. This tunable value can be much higher than the number of reported alignments, as the optimal alignment might not be the first feasible alignment found.

For a candidate region to be extended (line 4.1.13), the following two criteria must be satisfied:

- the region contains at least 2 seeds or the read coverage is higher than the minimum required coverage,
- fewer than f candidate regions have been extended that did not produce a feasible alignment or the region contains a unique seed.

The first extension criterion increases performance of the algorithm by limiting the extension of candidate regions to those regions that are supported by at least two seeds or one long seed. The minimum coverage is 25% by default.

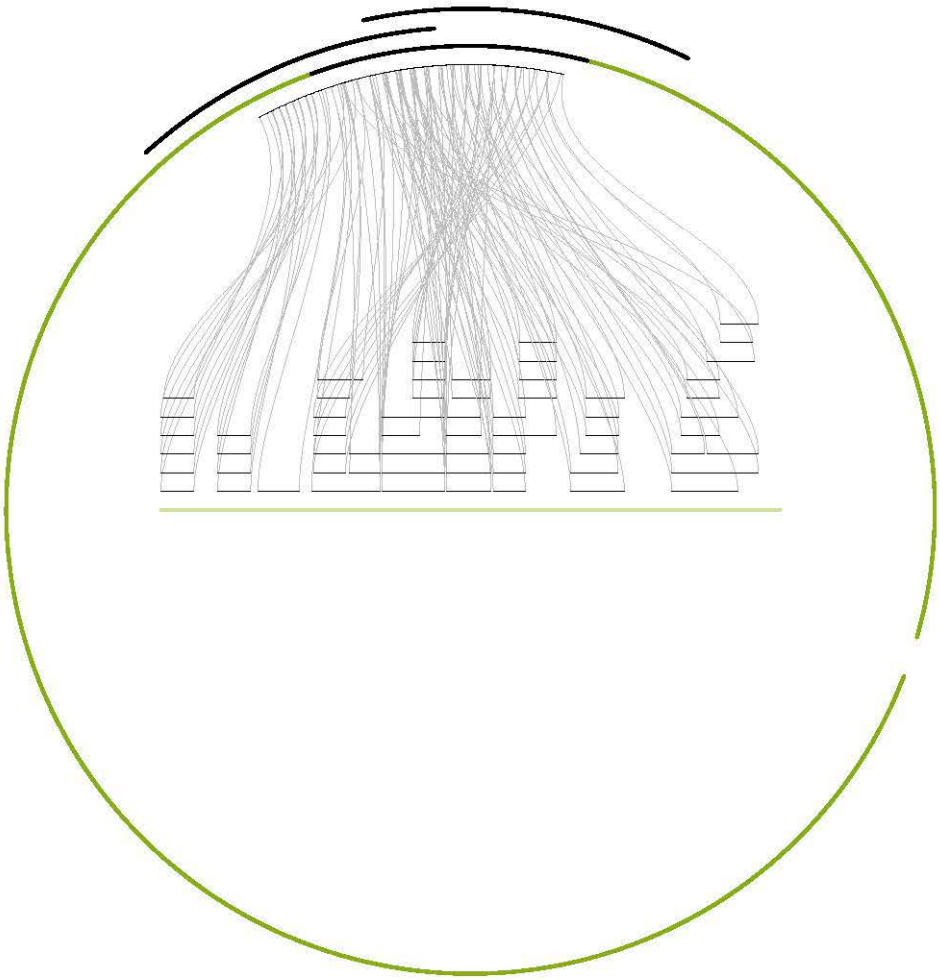


Figure 4.8: Example of a problem that could arise if overlapping candidate regions would be allowed. The figure represents the read in the center, a region of the reference genome around two candidate regions as the green circle and seeds between the reference genome and the read as connected black bars. The black line on the reference genome represents the region in which an optimal alignment of the read can be found and the two arcs outside the circle represent candidate regions found by *ALFALFA*. Both candidate regions overlap the region containing an optimal alignment, but not fully contain it. As both regions overlap, *ALFALFA* will merge them.

The first part of the second extension criterion halts the current stage of the algorithm if regions showing good coverage do not produce feasible alignments. The value of f is 10 by default. In contrast, if every candidate region produces at least one feasible alignment, the extension of regions is continued. Furthermore, the bad candidate region counter is reset if a region produces an alignment with a new best alignment score. In addition to continued analysis of successive candidate regions that were successful, a region is always extended if it contains a seed that covers an area of the read that was not previously covered by higher ranked candidate regions.

Rescue procedures

If *ALFALFA* fails to produce at least one alignment meeting the minimum score set by the user, it can start a two-stage rescue procedure (line 4.1.25). The first stage reiterates over the list of candidate regions using a significantly lower minimum score for the threshold and minimum coverage requirements. As a compensation, the maximum number of alignments calculated per read is lowered. Failure of this rescue procedure is most likely due to bad seed-finding or candidate region identification. Therefore, a second rescue procedure first invokes the seed-finding rescue procedure using less stringent parameter settings, followed by another round of candidate region identification along the same lines as the first rescue procedure.

Rescue procedures are turned on by default. However, it is possible to disallow rescuing to boost performance. As expected, the rescue procedures negatively affect the runtime of *ALFALFA*, but the effect was never significant in all of our experiments. Furthermore, sensitivity of *ALFALFA* significantly increases using the rescue procedures at the cost of some false positives.

4.2.3 Candidate region extension

This section provides details about the extension phase of the seed-and-extend algorithm (lines 14 to 21 in Algorithm 4.1). In *ALFALFA*, candidate mapping regions are linked to the seeds that were used to delimit the region. These seeds are reused in collinear chains, which serve as partial alignments of the read in the candidate region. Full alignments are obtained by applying dynamic programming procedures in between the gaps of the chains.

Chaining

The greedy procedure used by *ALFALFA* to form collinear chains of seeds is given in Algorithm 4.5. The algorithm starts from the list of seeds found in the candidate region, sorted by offset in the read, and one seed designated as the *anchor* for the alignment. Starting from the anchor seed, the algorithm extends the chain to the left and right by adding seeds that do not exhibit a large *skew* to the previous seed in the chain and do not overlap with it more than 10 bases.

The *skew* between two seeds is defined as the absolute difference of the distances between the substrings in the read and the reference genome. For example, if the distance between the substrings in the read is 50bp and the distance between the substrings in the reference genome is 70bp, the skew is 20. In this example, a chain consisting of those two seeds would lead to a large deletion in the alignment, which is not common in practice.

An example motivating the use of restricted skew between two seeds is given in Figure 4.9. Grey areas in the figure represent sequences of more than three matches in an optimal alignment between the reference genome (top) and the read (bottom). Connected bars represent seeds found between the read and the reference genome. Although it is collinear with the other seeds, the pair of connected red lines is not used as part of the chain because it does not meet the skew condition. Note that inclusion of the red seed in the chain would have led to a sub-optimal alignment, as is indicated by the location of the dark grey areas.

The maximal amount of skew allowed between two consecutive seeds in a chain is dependent on the outer distance between the matching substrings and the maximal allowed percentage of differences in a feasible alignment. The value of this parameter can be increased if large insertions and deletions are expected to be present in the chain. If optimal alignments would contain long gaps, the algorithm will produce much shorter chains, resulting in a higher dynamic programming cost. It will, however, still be able to retrieve the alignment.

Number of chains per candidate region

Because Algorithm 4.5 depends on a good choice for the anchor seed, it is likely that the first chain does not result in an optimal alignment within that region. This is the case when candidate regions contain many seeds, which often occurs in repetitive regions of the reference genome or in the cases where candidate regions were merged during identification. *ALFALFA* therefore constructs multiple chains per candidate region.

Algorithm 4.5 Collinear chaining

Input: reference genome G, read R, list of *seeds* in a candidate region and one seed a used as anchor
Output: *chain* of seeds collinear to a

function DISTANCE(G,*s1*,*s2*)
    return outer distance between intervals *s1* and *s2* in sequence G
end function

$e \leftarrow$ preset maximum allowed percentage of differences // *parameter* `-e`
add a to *chain*
sort *seeds* according to offset in R
$p \leftarrow a$
for all s in *seeds* to the right of a **do**
    $sDist \leftarrow$ DISTANCE(G,p,a)
    $pDist \leftarrow$ DISTANCE(R,p,a)
    $skew \leftarrow |sDist - pDist|$
    if overlap between s and $p < 10$ **and** $skew \leq e \cdot \max(sDist, pDist)$ **then**
        add s to *chain*
        $prevSeed \leftarrow s$
    end if
end for
$prevSeed \leftarrow a$
for all s in *seeds* to the left of a **do**
    ... // *same procedure as for seeds to the right of a*
end for

Figure 4.9: Example motivating the limitation of *skew* in between two seeds in a chain. Grey areas in the figure represent sequences of more than three matches in an optimal alignment between the reference genome (top) and read (bottom). Connected bars represent seeds found between read and reference genome. Black lines correspond to seeds in the chain. Red lines are not included in the chain.

Initially, *ALFALFA* considers the entire list of seeds, ordered by length, as possible anchors for a chain. After each call of Algorithm 4.5, seeds that were used in the chain are removed from the list of potential anchors. In most cases, we have found that this procedure eliminates most seeds as anchors after the first two chains have been constructed. Moreover, the remaining anchors usually cannot be combined with other seeds due to a high skew. *ALFALFA* will not consider these single-seed chains if their coverage of the read is below the same threshold set for candidate regions.

A situation in which constructing multiple chains is mandatory is depicted in Figure 4.10, which uses the same candidate region representation as before. Seeds depicted in black are the only seeds that are part of an optimal alignment, whereas seeds depicted in red lead to suboptimal alignments. Furthermore, there is a tie for the choice of the initial anchor (longest seed), which shows that the longest seed does not always lead to the best possible chain.

Alignment

The chains obtained by Algorithm 4.5 represent gapped alignments of the read against the reference genome, not containing the unaligned sections between each pair of consecutive seeds and possibly at the beginning and end of the read. Algorithm 4.6 describes how chains are extended to full alignments that also cover the previously unaligned sections.

Before starting the alignment procedure, Algorithm 4.6 first checks whether the chain meets the same requirements imposed on candidate regions. Similar to the coverage requirements for candidate regions, single-seed chains that cover few bases of the read are omitted from alignment.

If the chain meets the minimum coverage requirements, the algorithm proceeds by filling the gaps in the alignment between each pair of consecutive seeds. For normal gaps, we use a standard banded dynamic programming routine. In some cases, however, dynamic programming can be avoided. For example, if the gap in both reference genome and read is a single base, the algorithm just inserts a single mutation. Likewise, if there is a positive distance between two seeds in one sequence, but no gap in the other, the algorithm adds an insertion or deletion of appropriate size. Finally, banded semi-global alignment is performed in the region between the ends of the read and the ends of the chain.

For dynamic programming, we use an affine gap model with default match bonus of 1, mismatch penalty of 4, gap open penalty of 6 and gap extension penalty of 1.

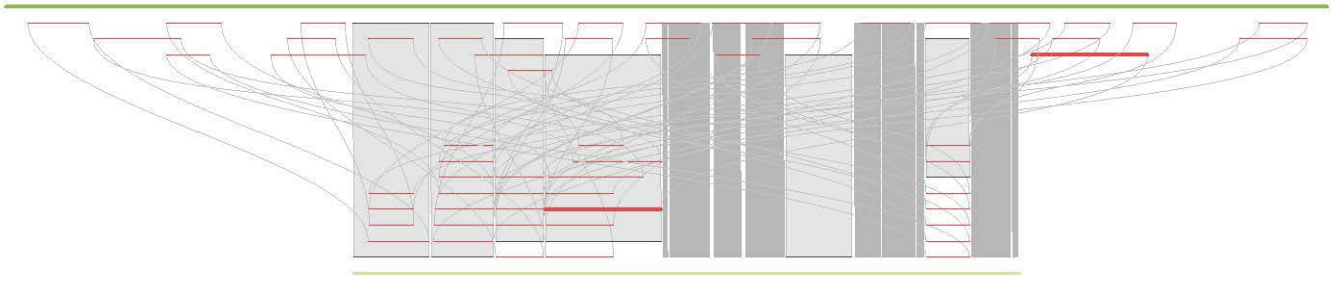


Figure 4.10: Example motivating the use of multiple chains per candidate region. Connected black and red lines depict seeds between a candidate region in the reference genome (top) and the read (bottom). Only the seeds depicted in black are part of an optimal alignment, of which the matches are shown by the grey areas. The large number of seeds makes finding a good chain challenging. This can be alleviated by the construction of multiple chains.

Algorithm 4.6 Extend chain to alignment

Input: reference genome G, read R, collinear *chain* of seeds between G and R
Output: Alignment between R and the region of G bounded by *chain*

$c \leftarrow$ minimum coverage set by parameter `-c`
if *chain* consists of 1 seed **and** coverage of $R < c$ **then**
    return
end if
$p \leftarrow$ **pop** first seed from chain
add p matches to alignment
for all seeds s in *chain* **do**
    calculate *gap* between p and s
    if *gap* can be filled with single mutation **then**
        add mutation to alignment
    else if *gap* can be filled with single (long) indel **then**
        add insertion or deletion of length *gap* to alignment
    else
        perform banded alignment in *gap*
    end if
    add s to alignment
    $p \leftarrow s$
end for
if first seed of chain did not start at first base of R **then**
    perform semi-global alignment between first seed and start of R
    $l, g \leftarrow$ local,global alignment score
    if *acceptLocal* **and** $g < l + 3$ mutation differences **then**
        use local alignment
    else
        use global alignment
    end if
end if
if last seed of chain did not end at last base of R **then**
    ... // *same procedure as above*
end if
if alignment score $\geq$ *minscore* **and** no previous alignment found within *minDistance* bases **then**
    add alignment to R
end if

The band size is estimated using the maximum allowed percentage of differences in a feasible alignment. This value is decreased after each dynamic programming routine to more correctly represent the current situation after several differences have already been added.

The implementation of the dynamic programming algorithms was obtained from the *klib*[1] library. We made some minor changes to the library, in addition to the changes made in *BWA-MEM*, to facilitate global and semi-global alignments. These changes allow the semi-global alignment algorithm to report both local and global alignment scores. Similar to *BWA-MEM*, *ALFALFA* uses both scores to automatically decide between both alignment types. Local alignment is only preferred if the global alignment score is smaller than the adjusted local alignment score for which three mutations were added.

The obtained alignment score is compared to a minimum score to decide whether the alignment is considered feasible. The minimum alignment score is obtained by starting from the maximum obtainable score and subtracting the maximum allowed number of differences e multiplied by the mismatch penalty. On the command line, the maximum number of differences is expressed as a fraction that is multiplied by the read length to obtain e. In addition to being feasible, the alignment should also be e bases separated from another feasible alignment with a higher alignment score.

The use of chain-guided alignment can significantly lower the computational cost of the extension phase of the algorithm, as illustrated in Figure 1.15 (see Section 1.4). In practice, we have found that chain-guided alignment is up to twice as fast as alignments using standard banded dynamic programming. Moreover, chain guided alignment does not seem to lead to a significant drop in accuracy, thanks to the conservative collinear chaining algorithm.

4.2.4 Alignment post-processing

During the alignment process, *ALFALFA* stores all alignments whose alignment score is higher than a minimum value. The number of feasible alignments is usually much higher than the requested number of alignments a. In a final post-processing step (line 28), *ALFALFA* selects the a highest scoring alignments and computes mapping qualities and CIGAR strings for these alignments.

To obtain the CIGAR string for an alignment, *ALFALFA* needs to perform another extension of the chain/candidate region that led to the alignment. This

[1] `https://github.com/attractivechaos/klib`(last accessed September 2014)

is because the dynamic programming routines in Algorithm 4.6 do not trace back the alignment after finding the alignment score. By default, the algorithm will again perform the chain-guided alignment, but users can also request the usage of a banded dynamic programming routine. This latter option might improve the quality of the alignment in some cases at the cost of increasing runtimes. In our benchmarks, the gain in accuracy was small, whereas the runtime sometimes increased significantly.

Mapping quality is estimated by a function that is widely used among read mappers. If s_1 is the best alignment score of the best alignment and s_2 is the alignment score of the second best alignment, the mapping quality for the best alignment is given by

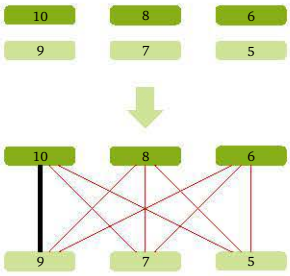
$$250 \times \frac{s_1 - s_2}{s_1}.$$

This means that the mapping quality is zero if the two best alignments have an equal alignment score. In addition, mapping quality uses the sum of the alignment scores of concordantly mapped reads when mapping paired-end reads. For ease of comparison, we also set the maximum attainable mapping quality to 60, which is the same as used by *BWA-MEM*.

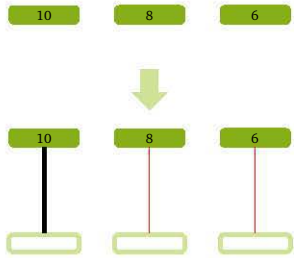
4.2.5 Paired-end read mapping

Similar to single-end read mapping, *ALFALFA* is designed to report a maximum number of different feasible paired-end alignments between reads and the reference genomes. The added requirement is that pairs need to be aligned concordantly with respect to a preset orientation and insert size (see Section 1.2.2). Parameter settings for mapping paired-end reads have been adopted from *Bowtie* 2 and include options to (dis)allow mates to overlap, contain or dovetail one another and options to (dis)allow discordant or unpaired alignments. A graphical representation of the six different paired-end mapping algorithms implemented in *ALFALFA* can be found in Figure 4.11.

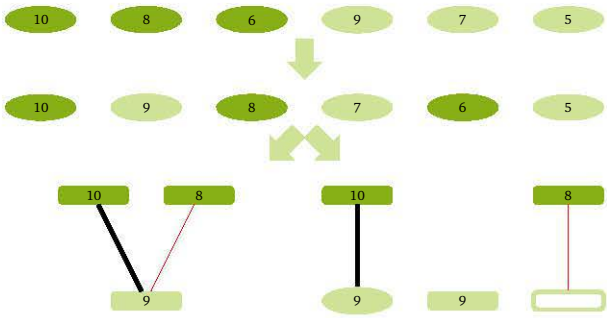
The first approach we tested is widely used among read mappers. The algorithm independently maps both mates of the pair and checks concordancy for every combination of feasible alignments found for the two mates. This method is also described in Algorithm 4.7, as it is currently the default paired-end mapping algorithm.



paired-end method 1

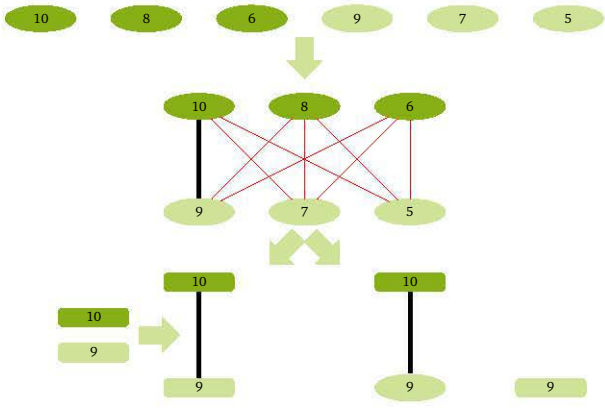


paired-end method 2



paired-end method 3

paired-end method 4



paired-end method 5

paired-end method 6

Figure 4.11: *(previous page)* Graphical representation of the various methods for paired-end alignment implemented by *ALFALFA*. The schematic represents alignments as colored rectangles, candidate regions as ellipses and other regions in the reference genome (containing no seeds) as empty rectangles. Dark green is used for mate 1 and light green for mate 2. Numbers within shapes represent scores used to sort candidate regions and alignments, *i.e.* alignment scores or coverage of bases in the read. Lines between shapes indicate pairing between structures to check orientation and insert size. Structures that can be paired are indicated by thick black lines, whereas failed comparisons are shown in red.

Algorithm 4.7 Default paired-end read mapping algorithm

Input: reference genome G, list of paired-end reads L_R
Output: SAM file containing alignments of reads onto G

```
1: read input and parse parameter options
2: construct the enhanced sparse suffix array I for G
3: for all mate_1, mate_2 in R do
4:     calculate alignments for mate_1
5:     calculate alignments for mate_2
6:     for all pairs of alignments (a_1,a_2) of respectively mate_1 and mate_2 do
7:         if orientation and insert size between a_1 and a_2 are correct then
8:             report concordant alignment pair (a_1,a_2)
9:         end if
10:    end for
11:    if not enough concordant pairs found then
12:        RESCUE(mate_1,mate_2)  // try other paired-end alignment method
13:    end if
14:    if report discordant or unpaired alignments then
15:        report all discordant and unpaired alignments found
16:    end ifPOSTPROCESS(mate_1,mate_2)
17: end for
```

The second method for paired-end mapping is also used frequently among read mappers. For this method, a single mate is mapped, after which alignments for the second mate are found using dynamic programming in a window defined by the concordancy constraints (orientation and insert size). If the alignments of the first mate did not result in concordant alignments, the second mate is aligned and the process of finding a concordant alignment is repeated.

For the next methods, we tried to limit the number of times the single-end extension phase is called, as we found this to be the most costly subroutine in the mapping algorithm. This is done by using information from the candidate regions of both mates. In the third and fourth methods, we first calculated candidate regions for both mates. We subsequently merged the lists of candidate regions into a single list, which is then sorted by coverage of the reads. This procedure effectively prioritizes extension of the best candidate regions among those of both mates.

Using the union of candidate regions from both mates, method 3 behaves like a single-end mapping algorithm and reports alignments for both mates. The alignments reported by the algorithm are then compared to find concordant pairs. In contrast to method 3, method 4 does not postpone the pairing of alignments until all candidate regions have been processed. Immediately after finding a feasible alignment, method 4 searches for a concordant alignment of the other mate in a window defined by the concordancy constraints. In contrast to method 2, however, this method can make use of a chain-guided alignment using the candidate regions for the other mate.

The last two methods also use candidate region information to speed up alignment of paired-end reads. Instead of merging the lists of candidate regions of both mates, however, methods 5 and 6 compare all pairs of candidate regions for the concordancy constraints and filter those candidate regions that do not pair with any other region of the other mate. Using the filtered list of candidate regions, method 5 then behaves exactly like methods 1 and 3, whereas method 6 uses the same strategy employed by method 4. A recap of the relationship between all methods for paired-end alignment can be found in Figure 4.12.

If the algorithms did not find enough concordant alignments, rescue procedures can be invoked. These rescue procedures will call method 1, except for method 1 itself, which calls method 2. *ALFALFA* will also combine the best alignments for both mates into discordant pairs or select unpaired alignments, if required by the user. Finally, the post-processing procedure calculates mapping quality and CIGAR strings for the alignments that will be reported.

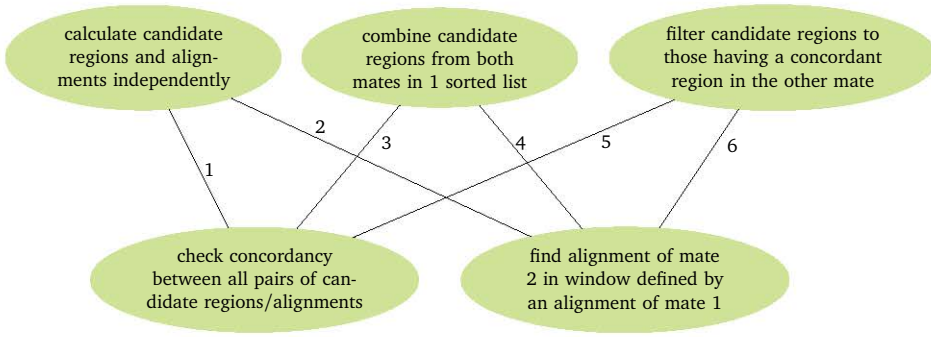


Figure 4.12: Schematic showing the relationship between all implemented methods for paired-end mapping. The top diagrams represent the different ways candidate regions are used. The bottom diagrams represent the two methods for finding concordant alignments, *i.e.* pairing two existing alignments or calculating an alignment for one mate given an alignment of the other.

Tests have shown that all methods have similar performance in both time and accuracy. Overall, we chose method 1 as the default, because of its simple design and stable performance over all data sets. In comparison to the first method, the second is somewhat more accurate in some situations, but it is not well suited for very long reads or for data sets with a broad variation in insert sizes. In contrast, methods 5 and 6 were somewhat faster than method 1 due to the shorter list of candidate regions. However, the lower runtime comes at the cost of a drop in accuracy. The trade-off obtained by methods 3 and 4 is almost identical and lies in between that of method 1 on the one hand and methods 5 and 6 on the other hand. Further testing might reveal more information on which methods perform best for certain data sets.

4.3 Results

ALFALFA accepts FASTA/FASTQ input and outputs to SAM format. The read mapper is open source and distributed under the very permissive MIT license. Appendix E provides details on the command line options that can be used to tweak *ALFALFA*, as well as default parameter settings and general usage tips. *ALFALFA* is available for download as `C++` source code at `http://alfalfa.ugent.be`.

Execution speed, memory footprint and accuracy of *ALFALFA* have been

scrutinized in a benchmark study that includes five other state-of-the-art long read mappers. Executables for *ALFALFA* v0.8, *Bowtie* v2 – 2.2.3, *BWA* v0.7.9*a* and *CUSHAW* v3.0.3 were built from source. Build 1.376 (beta) of *GEM* was obtained from its website, as source code was not available at the time of writing.

Two configurations of all read mappers were tested. First, read mappers were configured to produce a maximum of 4 alignments per read, if possible. Second, read mappers were configured to produce a single best alignment per read. Other parameters were kept to their default settings, unless the authors suggested specific settings for certain types of data. Appendix D contains information on the chosen parameter settings, as well as other technical details of the experimental setup, including hardware configurations.

The human genome is used as a reference genome to map a large array of moderately sized reads generated by current sequencing platforms and artificial reads generated by two simulators covering lengths expected to become commonplace in the near future. These simulated data sets are also crucial in evaluating mapping accuracy, which otherwise could not be evaluated objectively on true data. Care was taken to cover a broad range of error models observed in read sets generated by current sequencing technologies.

The *wgsim* simulator v0.3.1-r13 [149] — developed for SAMtools, but now a standalone project — was used to generate a series of single-end reads with lengths of one, five and ten thousand base pairs. Errors were introduced at rates between 2% and 10% of the total read length, with varying indel/mutation frequencies. Reads ranging from 100bp to 10kbp and abiding to specific error models induced by Illumina and 454 technologies were generated using the *Mason* simulator v0.1.1 [103]. Default parameter settings were used to generate reads of length 100bp and 200bp and parameter settings from the literature [141, 253] were used for longer read data sets.

4.3.1 Memory footprint

An index of the human genome assembly GRCh37 was constructed by all read mappers using their default parameter settings, except for *GEM*. A pre-built *GEM* index was downloaded from the *GEM* website as the indexer of this mapper ran into a fatal error on our test environment. Memory requirements of read mappers are mainly dependent on the memory footprint of the index structures they use. An overview of the index structure memory requirements can be found in Table 4.1.

Table 4.1: Comparison of the used index structures and memory requirements of evaluated mappers in benchmarking *ALFALFA*. The ESSA index is evaluated for two different sparseness values. The read mappers *BWA-SW* and *BWA-MEM* both use the same index structure, jointly reported as BWA. The third column indicates whether memory footprint can be tuned via user-specified options. The fourth column reports the memory footprint of the index structure when stored on disk. The fifth column provides peak memory of the read mapper observed during alignment of 1kbp reads. The time needed to construct the index structure is given in the last column. [§]A prebuilt *GEM* index was downloaded from the *GEM* website as the indexer of this mapper ran into a fatal error in our test environment.

mapper	index type	fixed	disk (GB)	peak (GB)	constr. (h:mm)
ALFALFA ($s = 4$)	ESSA	no	10.7	11.0	0:19
ALFALFA ($s = 12$)	ESSA	no	5.6	5.7	0:10
Bowtie 2	FM-index	no	3.9	5.3	1:58
BWA	FM-index	yes	5.2	5.4	1:18
CUSHAW3	FM-index	yes	3.7	3.5	0:36
GEM	FM-index	no	4.8	5.0	N.A.[§]

In this table, *BWA-SW* and *BWA-MEM* are reported as BWA, as they both use the same index structure. From the table, it can be seen that most tools require $3 - 5$GB of memory, both for storing the index on disk and for the peak memory during mapping a data set of 1kbp reads. Among the tested read mappers, *CUSHAW3* seems to be the most memory efficient one. In contrast, *ALFALFA* requires twice as much memory as the other tools when configured with a lower sparseness setting. The default setting (sparseness value 12) is competitive in terms of memory requirements with the other tools. The last column of Table 4.1 also shows index construction time, which is the lowest for *ALFALFA*.

4.3.2 Performance and accuracy on simulated data

Performance and accuracy results for most simulated and real data sets can be found in Figures 4.13 to 4.17. Performance is shown in the top bars, expressed in milliseconds per read. Accuracy is expressed as the percentage of reads that were not mapped within 10 bases of the simulated origin. Absolute values of the mapping time and more accuracy results can be found in Appendix D.

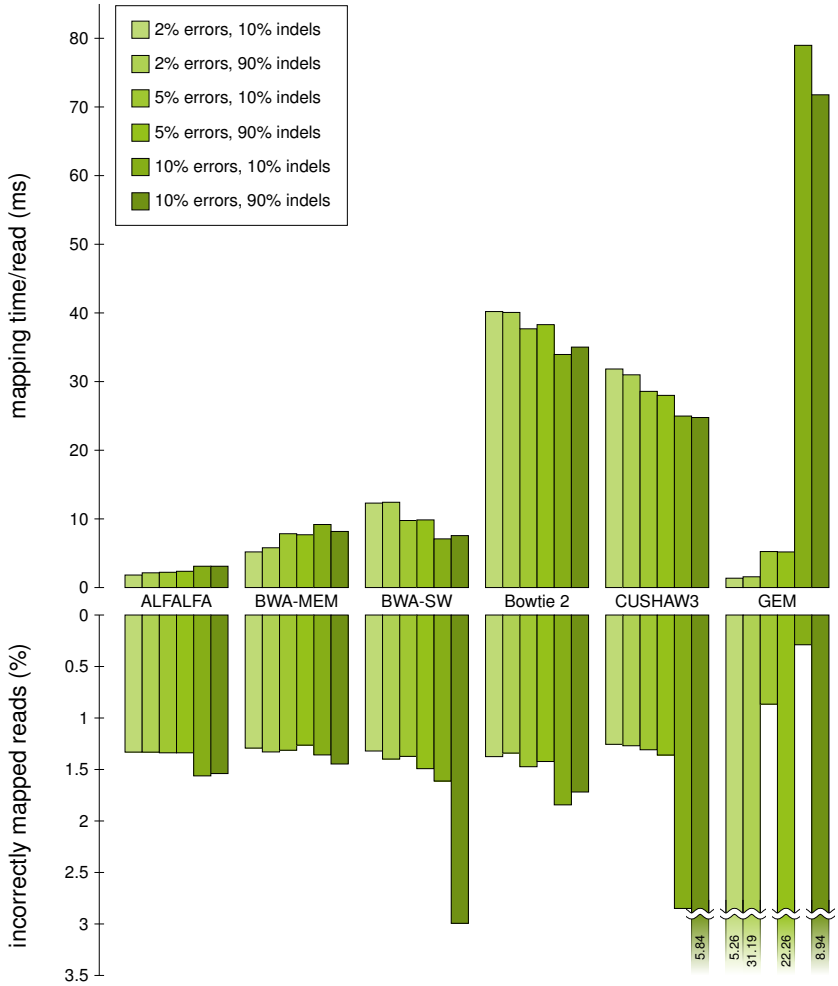


Figure 4.13: Accuracy and performance comparison of several long read mappers on 200 000 simulated 1kbp single-end reads. The six different data sets vary in error rate and percentage of errors that are indels as specified in the legend. Upper bars show the average mapping time per read (in milliseconds), whereas lower bars show the percentage of reads for which no alignment was found within 10bp of the simulated origin. All tools were configured to produce a maximum of one alignment per read.

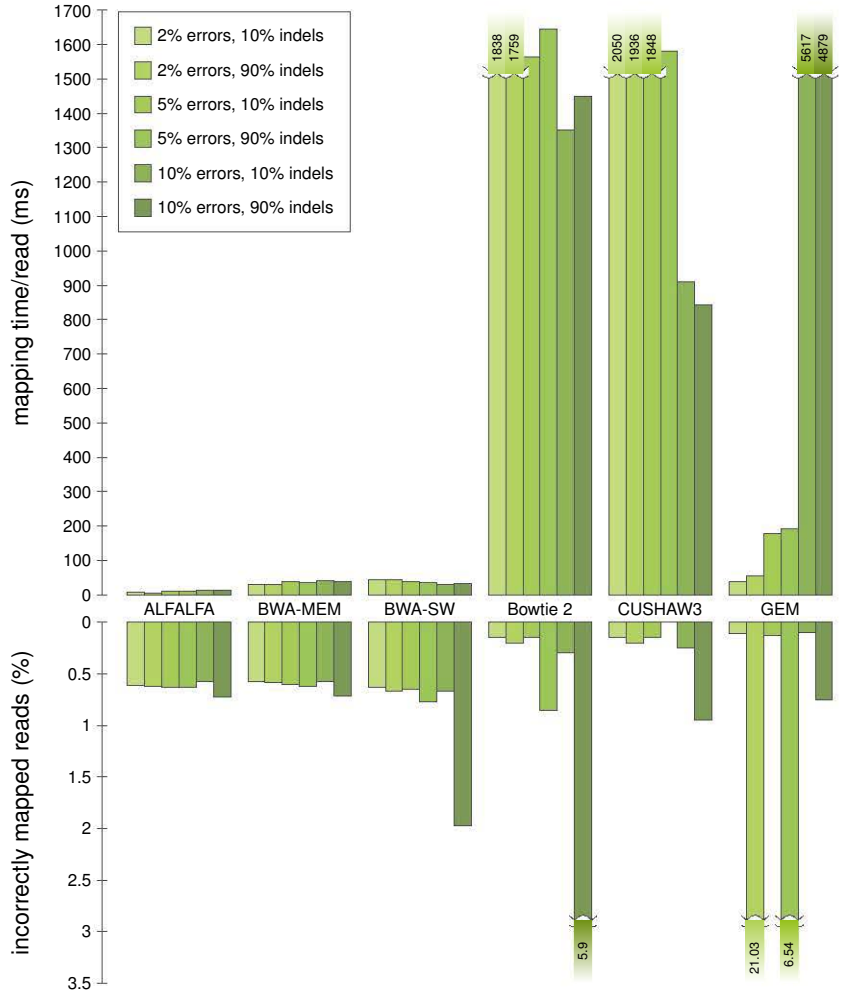


Figure 4.14: Accuracy and performance comparison of several long read mappers on 200 000 simulated 5kbp single-end reads. The six different data sets vary in error rate and percentage of errors that are indels as specified in the legend. A data set that was one hundred times smaller was used for *Bowtie* 2 and *CUSHAW3* due to low performance. For *GEM*, the first three results were obtained using the full data sets and the last three using the reduced data set. Upper bars show the average mapping time per read (in milliseconds), whereas lower bars show the percentage of reads for which no alignment was found within 10bp of the simulated origin. All tools were configured to produce a maximum of one alignment per read.

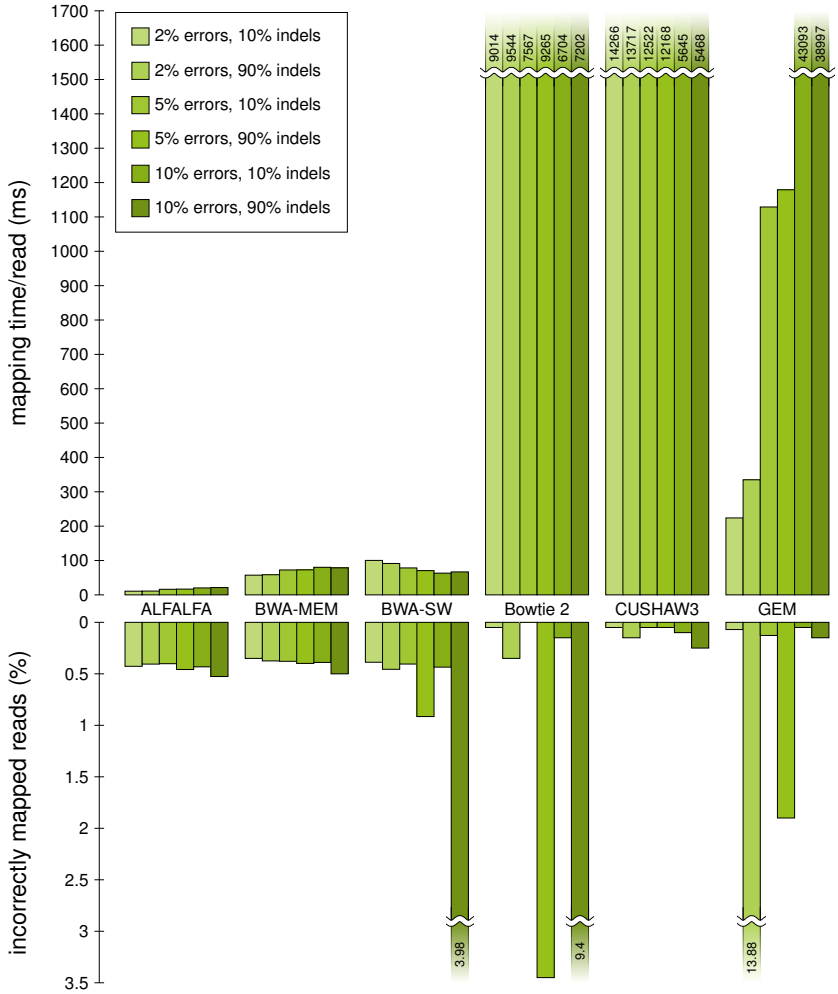


Figure 4.15: Accuracy and performance comparison of several long read mappers on 200 000 simulated 10kbp single-end reads. The six different data sets vary in error rate and percentage of errors that are indels as specified in the legend. A data set that was one hundred times smaller was used for *Bowtie* 2 and *CUSHAW3* due to low performance. For *GEM*, the first three results were obtained using the full data sets and the last three using the reduced data set. Upper bars show the average mapping time per read (in milliseconds), whereas lower bars show the percentage of reads for which no alignment was found within 10bp of the simulated origin. All tools were configured to produce a maximum of one alignment per read.

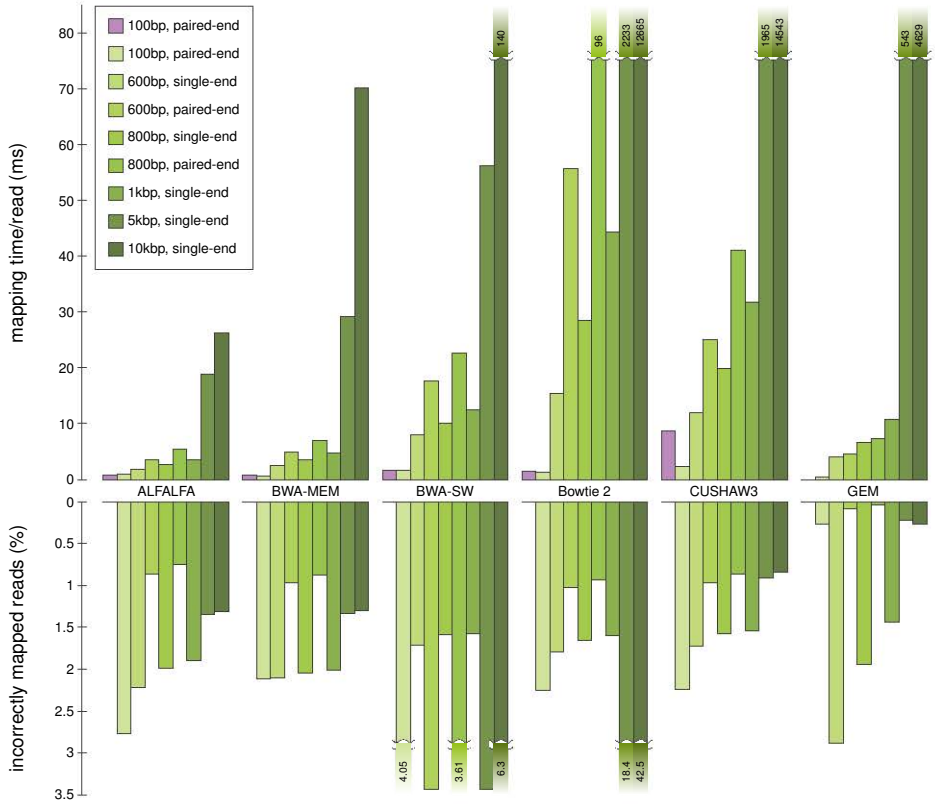


Figure 4.16: Accuracy and performance comparison of several long read mappers on reads following Illumina error model and have varying read lengths as specified in the legend. For the two longest read data sets, results for *Bowtie* 2 and *CUSHAW3* were obtained using a data set that was ten times smaller due to low performance. Upper bars show the average mapping time per read (in milliseconds), whereas lower bars show the percentage of reads for which no alignment was found within 10bp of the simulated origin. All tools were configured to produce a maximum of one alignment per read.

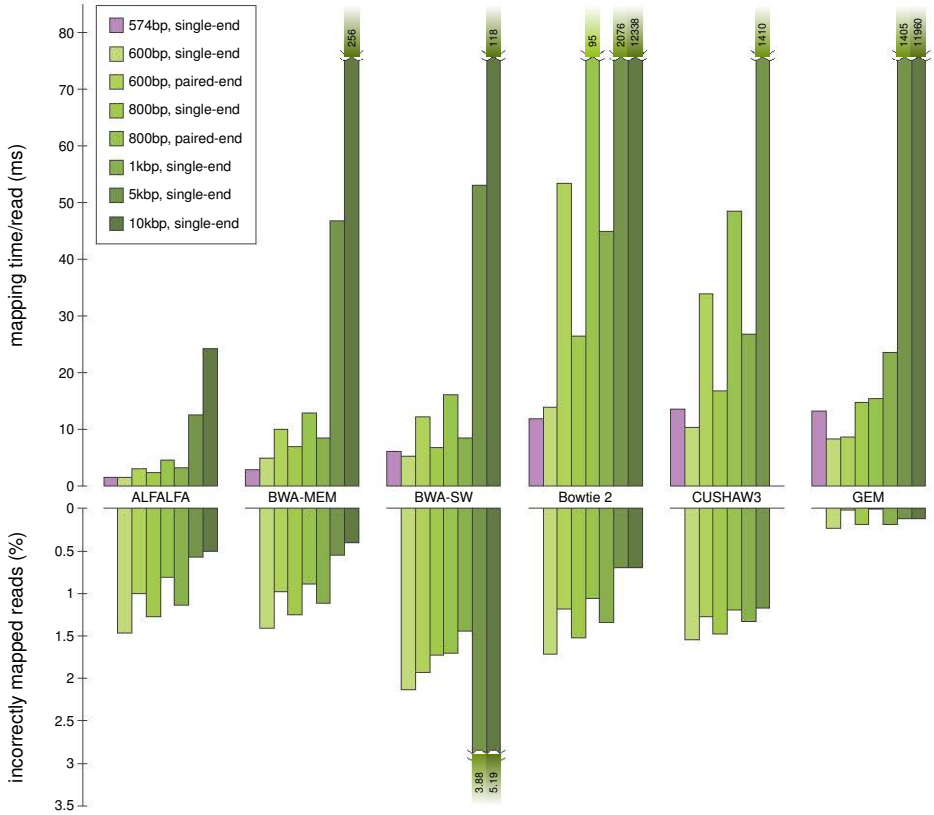


Figure 4.17: Accuracy and performance comparison of several long read mappers on reads following 454 error model and have varying read lengths as specified in the legend. For the two longest read data sets, results for *Bowtie* 2 were obtained using a data set that was ten times smaller due to low performance. Results for *CUSHAW3* on 5kbp reads were also obtained using the reduced data set and no results were obtained for the 10kbp data set due to a fatal error. Upper bars show the average mapping time per read (in milliseconds), whereas lower bars show the percentage of reads for which no alignment was found within 10bp of the simulated origin. All tools were configured to produce a maximum of one alignment per read.

Performance

ALFALFA is the fastest read mapper. It is only outperformed by *GEM* and *BWA-MEM* for the shortest reads in Figure 4.16 and by *GEM* for a single data set of 1kbp reads in Figure 4.13. The difference in runtime between *ALFALFA* and the other mappers increases with read length. For reads longer than 1kbp, *ALFALFA*, *BWA-MEM* and *BWA-SW* become orders of magnitude faster than the other mappers. This can be clearly seen in Figures 4.14 and 4.15. Even compared to *BWA-MEM*, the second fastest mapper, *ALFALFA* is on average three times faster and up to five times faster for reads of at least 1kbp long.

The performance of *GEM* is mostly affected by the user-set error rate. For Illumina reads with low error rates and low error reads simulated with wgsim, *GEM* is among the fastest algorithms. For high error rate data sets, however, *GEM* becomes much slower. Performance of *CUSHAW3* could be higher on hardware that supports SSE4 operations, which *CUSHAW3* uses by default. However, *CUSHAW3* is known to focus more on accuracy than speed [162]. In addition, runtimes of *CUSHAW3* dramatically increase when multiple alignments per read are requested (see Appendix D). The performance of *Bowtie* 2 and *BWA-MEM* is also influenced by the number of alignments per read requested, but the increase in runtime is less significant.

The performance of paired-end read mapping can be seen in Figures 4.16 and 4.17. For most mappers, there is no loss in performance when mapping single and paired-end reads. The exceptions are *CUSHAW3* and *Bowtie* 2, whose performance is much lower when mapping paired-end reads due to the high read length and large insert size window and the fact that these mappers perform full dynamic programming to find an alignment for the mate of a mapped read.

Figures 4.13, 4.14 and 4.15 also show that the performance of *ALFALFA*, *BWA-MEM* and *GEM* decreases for reads containing more errors, whereas the performance of *Bowtie* 2, *BWA-SW* and *CUSHAW3* increases. This could be explained by the fact that the latter mappers stop the alignment procedure more rapidly for reads that are more difficult to map, whereas the former increase the effort in finding an alignment for these reads. The type of errors, *i.e.* mutations versus indels, does not seem to have an effect on runtime.

If memory is abundant, the runtime of *ALFALFA* can be further improved by lowering the sparseness of the index. *ALFALFA* is up to twice as fast when the sparseness is lowered from the default value of 12 to 4, the lowest setting tested (see Appendix D). The effect of different sparseness values on performance and accuracy is also shown in Figure 4.18.

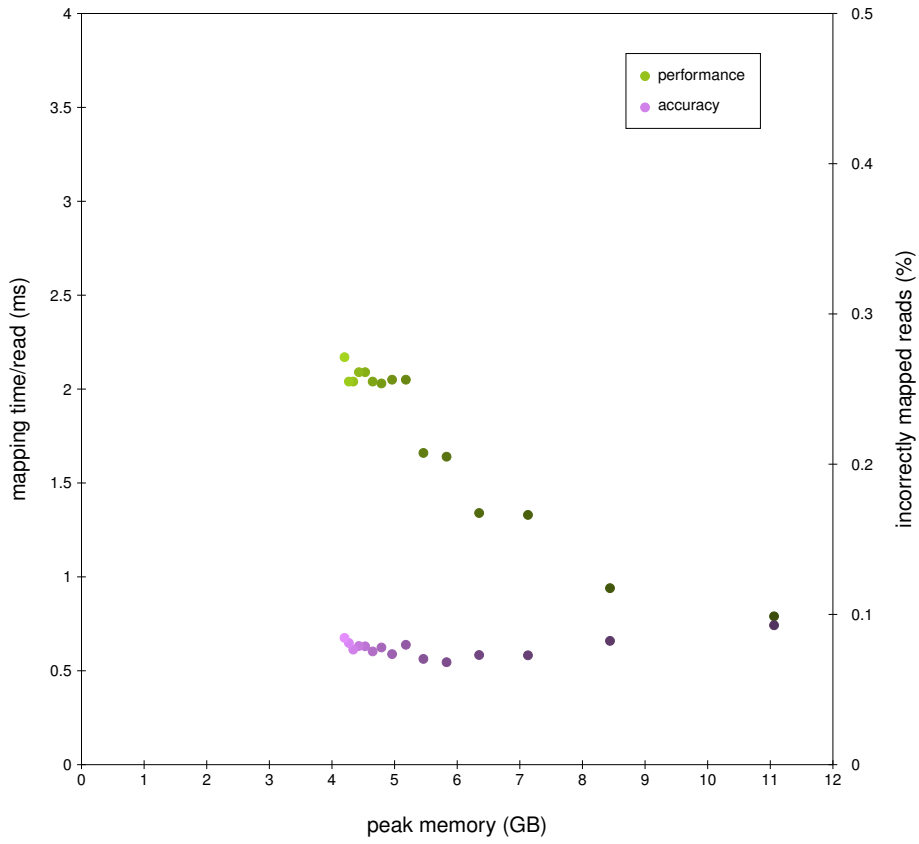


Figure 4.18: Influence of sparseness when indexing the reference genome with an enhanced sparse suffix array on the memory footprint, execution speed and accuracy of the read mapper *ALFALFA*. The scatterplot displays the peak memory measured during read mapping for all even sparseness values between 4 and 32, against the corresponding performance and accuracy of the read mapper. *ALFALFA* reported a maximum of 4 alignments per read. Green dots (left axis) plot performance as the average time needed to map one read (in milliseconds). Purple dots (right axis) plot accuracy as the percentage of reads for which no alignment was found within 10bp of the simulated origin or with less errors than were simulated. Darker shades of green and purple correspond to lower sparseness values. Evaluation was done on 600bp reads simulated with wgsim. The benchmark study evaluates *ALFALFA* with a sparseness value of 12, corresponding to the fifth dot from the right.

Accuracy

On simulated reads, accuracy was measured using the *recall rate* and our own definition of accuracy. Recall rate is defined as the number of reads for which an alignment is found within 10bp of the simulated origin. Our *accuracy* measure is less stringent and considers a read to be mapped correctly if an alignment either fulfills the recall rate requirements or has an edit distance that is not higher than the number of simulated differences to the reference genome. The lower bars in Figures 4.13 to 4.17 represent the recall rate in case each mapper reported a single alignment per read. When returning multiple alignments per read, a read is considered to be mapped correctly if at least one of the returned alignments fulfills the requirements imposed by our own accuracy measure. These additional results, together with the results using our own definition of accuracy, can be found in Appendix D.

In contrast to the performance results, the difference in accuracy between the evaluated read mappers is small. All tested mappers exhibit both a high recall rate and accuracy when reporting either a single or up to four alignments per read. We will therefore refer to accuracy for both measures, unless we want to stress the difference between the two accuracy measures used. In general, *CUSHAW3*, *BWA-MEM* and *ALFALFA* are the most accurate mappers, with *BWA-SW* and *Bowtie* 2 having a somewhat lower accuracy. In most cases, either *CUSHAW3*, *BWA-MEM* or *GEM* is the most accurate mapper, by a small margin.

The accuracy of *GEM* is highly dependent on the command line parameter settings. We have tried several parameter settings to optimize the time-accuracy trade-off, but it is possible that *GEM* reaches a more optimal trade-off for untested parameter settings. As a result, the accuracy of *GEM* can vary greatly, being the highest for some data sets, but the lowest for other data sets. This effect can be seen in Figure 4.13. On the 1kbp data sets with 2% errors, setting the parameters to this maximum error value results in a very low accuracy. In contrast, on 5% and 10% error rates, *GEM* has the highest recall rate for the data sets with low numbers of indel errors. The effect of the sparseness of the ESSA index on the accuracy of *ALFALFA* is depicted in Figure 4.18, but is rather small in general.

From the results of the wgsim data sets in Figures 4.13, 4.14 and 4.15 it can be seen that the accuracy of all mappers drops with increasing error rate. A noticeable exception is *GEM*, whose accuracy depends on the chosen parameter settings. The effect of increasing error rate seems smallest for *BWA-MEM*, whereas *CUSHAW3* does not perform well for reads with 10% errors. It is,

however, possible to increase the accuracy of *CUSHAW3* using command line parameters, as by default *CUSHAW3* allows only 10% errors. In addition to the raw error rate, an increase in the number of indel type errors has also a detrimental effect on accuracy. This effect seems smallest for *Bowtie* 2, whereas it has the highest effect on *GEM*.

In contrast to the above, an increase in read length has a predominantly positive effect on accuracy. For the longest reads, accuracy is almost 100% for most mappers. Note, however, that several of the results for *Bowtie* 2, *CUSHAW3* and *GEM* were obtained on a smaller data set due to a forced timeout in our testing environment of 72 hours. Nonetheless, a few samples on a different machine indicated that these mappers indeed have a high accuracy at the cost of performance.

The type of errors also has an impact on accuracy. Wgsim simulated reads have a uniformly distributed error model, which differs from the Illumina and 454 error models. For equal read length, the accuracy on simulated reads with an Illumina error profile is lower than the accuracy on reads with a 454 error profile. For Illumina reads, *CUSHAW3* is more accurate than *BWA-MEM* and *ALFALFA*, whereas the reverse is true on 454 reads.

The effect of paired-end read mapping on accuracy can be seen in Figures 4.16 and 4.17. As expected, paired-end read data sets exhibit a higher accuracy than single end read data sets. *GEM* hugely benefits from paired-end data sets for Illumina type reads. The only exception is *BWA-SW*, which performs worse for paired-end reads. This might be explained by the fact that *BWA-SW* automatically tries to estimate insert size, whereas other mappers trust on users to present insert size boundaries.

From the Additional tables in Appendix D, we have found that the difference between the accuracy and recall rate measures is noticeable for most mappers. The biggest effect was present in *BWA-SW*, *Bowtie* 2 and *CUSHAW3*, whereas the lowest effect was measured for *GEM*.

If multiple alignments per read are reported (see Appendix D), the accuracy of several mappers increases significantly. The effect is the greatest for *CUSHAW3* and *ALFALFA*, whereas it is lower for *BWA-MEM* and *GEM*. As a result, *ALFALFA* becomes the most accurate mapper for some data sets in this setting. In contrast, the accuracy of *Bowtie* 2 drops frequently, as the `-k` mode that is required to return multiple alignments works differently from the regular mode. Finally, *BWA-SW* does not offer an option to return multiple alignments per read.

4.3.3 Mapping quality

In addition to accuracy and recall rate, we compared the sensitivity and specificity of *ALFALFA* against that of other mappers. These evaluated measures are represented in receiver operating characteristic (ROC) curves in which the true positive rate is plotted against the false positive rate in terms of mapping quality values (MAPQ field in SAM files). For these plots, we limited ourselves to the wgsim simulated reads and used the evaluation script in the wgsim package to generate the data points. In addition to the wgsim simulated read data sets presented in Figure 4.13, we used a data set of single-end reads of length 600bp with a small (1%) error rate. The ROC curves are presented in Figure 4.19 and 4.20.

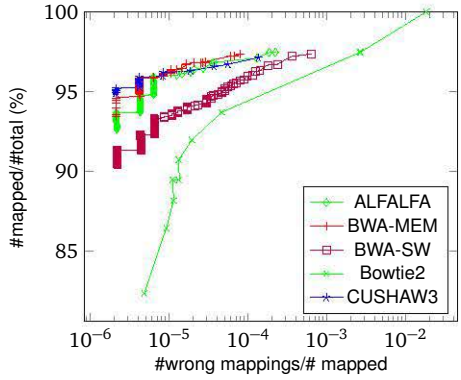
Overall, *Bowtie* 2 has the highest sensitivity, which reaches 100%. However, *Bowtie* 2 is also less able to distinguish between good and bad alignments. *CUSHAW3*, *BWA-MEM* and *ALFALFA* exhibit the best trade-off between true positives and false positives. For the 600bp data set presented in Figure 4.19a, *CUSHAW3* is most sensitive for high mapping quality, whereas *BWA-MEM* becomes more sensitive for lower mapping quality values. *ALFALFA* obtains a trade-off that fluctuates between that of *CUSHAW3* and *BWA-MEM*. For the 1kbp data sets with higher error rate, *BWA-MEM* is best able to distinguish between true and false positive hits, with *ALFALFA* a close second.

4.3.4 Performance and accuracy on real data

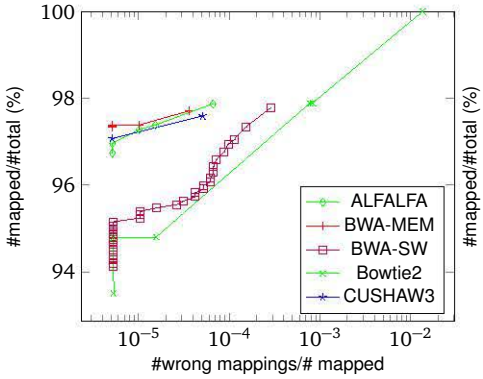
To validate our findings on simulated data, we also compared the performance of *ALFALFA* on one real Illumina read data set and one real 454 data set. The results can be found in Table 4.2. Because the real origin of the reads cannot be indisputably determined, we use the sensitivity, *i.e.* the number of mapped reads as an accuracy measure.

As our focus was on long reads, the Illumina read data set that consists of 2×100bp paired-end reads falls out of the scope of *ALFALFA*. As a result, it is outperformed by *BWA-MEM* and *GEM* in terms of mapping time and *BWA-MEM*, *BWA-SW* and *CUSHAW3* in terms of sensitivity. The default parameter settings for *ALFALFA* were optimized for longer reads. Therefore, a higher sensitivity for shorter reads can be gained by changing the default parameter settings.

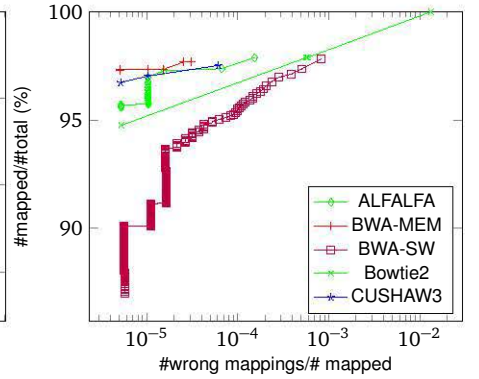
The single-end 454 reads have an average length of 574, which is well within the scope of our mapper. For this data set, *ALFALFA* is by far the fastest mapper. In addition, it also has the highest sensitivity. This is consistent with the good accuracy of *ALFALFA* for the simulated 454 reads.



(a) 600bp, 1% errors

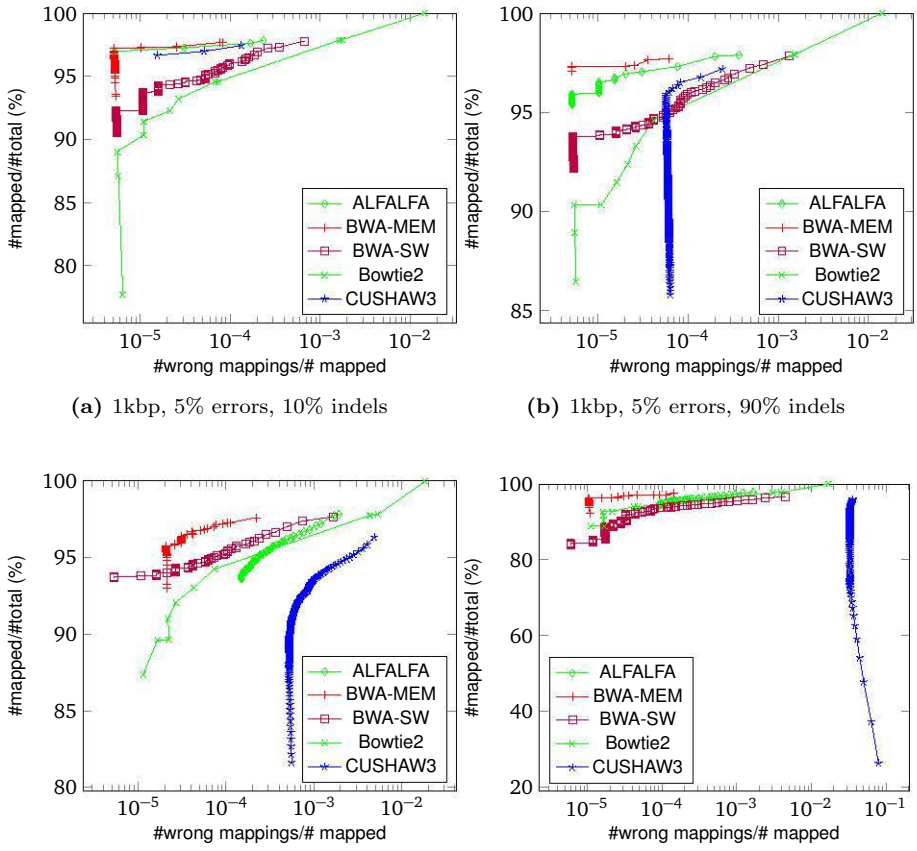


(b) 1kbp, 2% errors, 10% indels



(c) 1kbp, 2% errors, 90% indels

Figure 4.19: ROC curves plotting the sensitivity (vertical axis) against the false positive rate (horizontal axis) using different mapping quality cut-offs (MAPQ$>$ 0). Data set (a) consist of half a million single-end reads and data sets (b-c) consist of 200 000 single-end reads. Read length, number of simulated errors and the percentage of errors that are indel type errors are shown below each figure.



(a) 1kbp, 5% errors, 10% indels

(b) 1kbp, 5% errors, 90% indels

(c) 1kbp, 10% errors, 10% indels

(d) 1kbp, 10% errors, 90% indels

Figure 4.20: ROC curves plotting the sensitivity (vertical axis) against the false positive rate (horizontal axis) using different mapping quality cut-offs (MAPQ> 0). All data sets consist of 200 000 single-end reads. The data sets differ in the number of errors and the percentage of those errors that are indels.

Table 4.2: Benchmark comparison of long read mappers on two real data sets. The Illumina paired-end read data set (SRA:ERR024139) consists of 2×100bp reads and the reads of the 454 single-end data set (SRA:SRR003161) are on average 574bp long. The performance measures are runtime in h:mm and percentage of mapped reads (sensitivity).

	Illumina reads		454 reads	
	runtime	sensitivity	runtime	sensitivity
ALFALFA	5:48	99.09	0:33	99.75
BWA-MEM	5:19	99.71	1:04	99.60
BWA-SW	12:18	99.34	2:20	97.54
Bowtie 2	11:04	97.98	4:33	99.02
CUSHAW3	64:30	99.67	5:10	91.31
GEM	3:09	97.65	5:02	93.29

Chapter 5

Mesalina

Analysis of biological sequences encompasses many different types of sequences, including DNA, RNA and proteins. In comparison to DNA sequence mapping, alignment of cDNA sequences and RNA-seq reads to a eukaryotic reference genome poses additional algorithmic challenges due to the presence of large gaps in the alignment caused by splicing. Moreover, RNA-seq mappers or spliced alignment programs also have to adapt in response to the changing landscape in sequencing technology. In this chapter, we present *Mesalina*, a prototype of a spliced alignment algorithm based on *essaMEM* and *ALFALFA*. Preliminary results indicate that *mesalina* is competitive in terms of accuracy and has a high performance that is more robust with respect to increasing read length. *Mesalina* has been developed in collaboration with Dieter De Smedt, Dr. Yao-Cheng Lin and Dr. Lieven Sterck and the results in this chapter were presented in the conference paper "*Fast and accurate cDNA mapping and splice site identification*" [250].

5.1 Introduction

Transcriptomics, the study of the RNA content of a cell, reveals information that can not be obtained from the DNA content of the cell using genomics. Similar to genome analysis, however, the study of the transcriptome requires mapping and alignment of short sequences against a reference genome. These short sequences include expressed sequence tags (ESTs), full-length complementary DNA (cDNA) and high-throughput sequencing reads produced by RNA-seq [252]. In comparison to the mapping problem discussed thus far in this dissertation, alignment

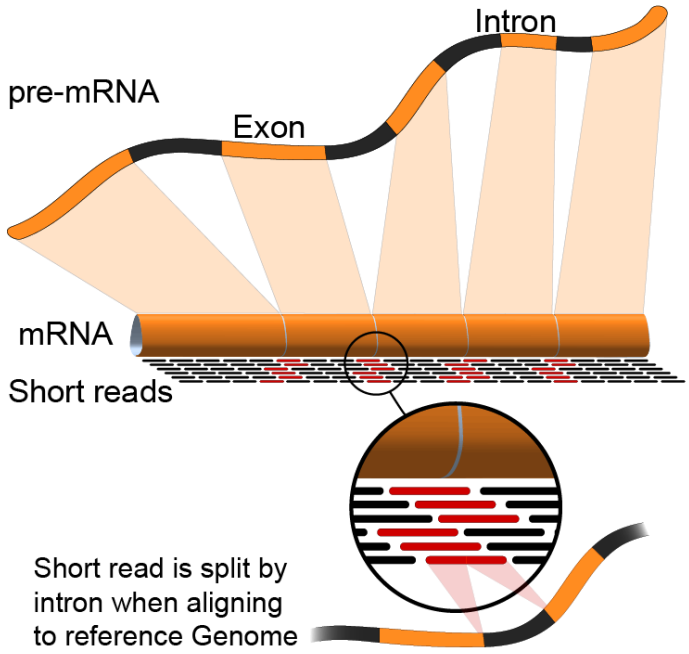


Figure 5.1: Graphical representation of the spliced alignment problem. The top sequence contains an alternating sequence of exons and introns, with the latter not being present in the sequenced mRNA. When aligning reads to the original pre-mRNA, the alignment can contain gaps spanning introns, which is shown in the magnified detail at the bottom of the image. Image by Rgocs (Transferred from en.wikipedia to Wikimedia Commons).

of RNA-seq reads against a reference genome poses additional challenges due to splicing in eukaryotic cells.

Figure 5.1 illustrates the spliced alignment problem. Genes in eukaryotic genomes are made up of an alternating sequence of *exons* and *introns*. The size of exons and introns is variable and can range from a few bases up to hundreds of thousands of nucleotides. After transcription of the genomic DNA, introns are removed and exons are joined together by a process called *RNA splicing*. In accordance to this term, the boundaries between exons and introns are called splice sites. In most cases, splice sites are either surrounded by GT-AG dinucleotides (canonical splice sites) or less frequently by GC-AG or AT-AC dinucleotides (semi-canonical splice-sites). In rare cases, however, splice sites are non-canonical,

meaning that they are not surrounded by any of the previous boundaries. Splicing does not only pose a greater challenge to mapping algorithms, but the identification of splice sites is of independent interest, as splicing errors are related to diseases [159].

Because introns are usually much larger than the deletions detected by traditional alignment algorithms, standard DNA read mappers fail to align reads spanning multiple exons, as illustrated at the bottom of Figure 5.1. However, many strategies already exist to address or circumvent this computational challenge. For example, it is possible to map reads to a reference *transcriptome* instead of a reference genome. However, an assembled transcriptome is not always available, relies heavily on the underlying gene model and does not allow to identify novel genes. Similar to transcriptome mapping, it is possible to use maps of known splice sites to guide spliced alignment.

For unguided spliced alignment and *de novo* splice site detection, two main mapping strategies are employed in practice [81]. These two strategies are illustrated in Figure 5.2 and Figure 1.8 contains a list of recently developed mapping tools. *Exon-first* mappers first align reads without taking possible splice sites into account. This stage can be done using traditional read mappers. Reads mapped this way provide a rough map of all the exons of the reference genome. The unmapped reads are split into shorter segments, which are mapped independently. Finally, connections between the mapped segments are searched to identify the exact splice site locations. Examples of exon-first mappers are *TopHat* [240], *TopHat2* [128], *MapSplice* [251], *SpliceMap* [17] and *SOAPsplice* [110].

The second major strategy for spliced alignment is called *seed-and-extend* or *seed-extend*. It uses the same approach taken by many standard DNA mappers, including *ALFALFA* (see Chapter 4). The high-level strategy remains unchanged, but the implementation of the seed and extend phases in spliced aligners differs from that in DNA mappers. This strategy is used, among others, by *GMAP* [258], *QPALMA* [47], *GSNAP* [257], and *STAR* [53].

In addition to the previous dichotomy, spliced aligners are usually also optimized for a specific type of input data. Most recent aligners focus on short RNA-seq reads, whereas *GMAP*, for example, focuses more on longer cDNA and EST sequences. In general, spliced aligners using the seed-and-extend approach are several times slower than their competitors using the exon-first strategy. However, exon-first approaches are known to miss spliced alignments that map to the genome contiguously [81]. Many aligners are designed for mapping very short reads that contain few sequence errors between read and genome. Moreover,

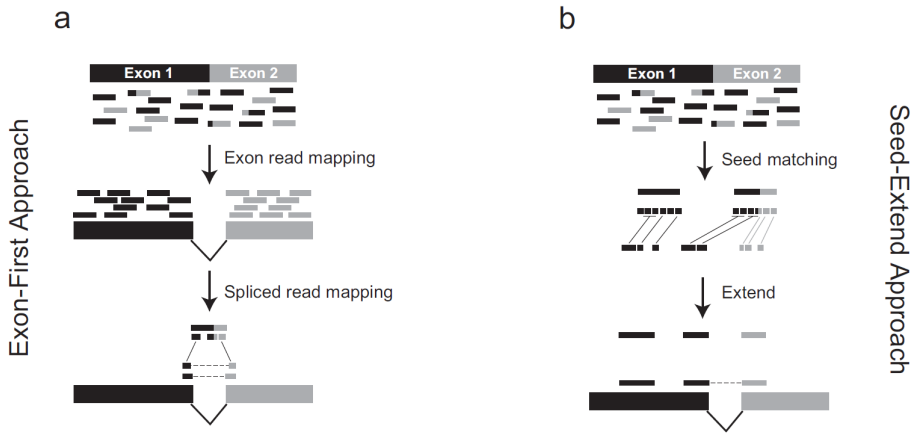


Figure 5.2: Two major strategies employed for spliced alignment. Black and gray colors indicate sequences originating from respectively exon 1 and exon 2. The exon-first approach (a) first maps unspliced reads and often assembles these mappings into an exon-map. The unmapped reads are divided into smaller pieces, which are mapped independently. The alignments of each piece are finally joined into an aligned of the read. The seed-and-extend approach (b) works similar to the seed-and-extend approach for DNA mappers (see Chapter 4), with the added feature of closing gaps caused by splicing in the extension phase. Figure by Garber *et al.* [81].

many novel spliced aligners are not able to detect rare non-canonical splice sites. In contrast, mappers that overcome these shortcomings tend to be much slower than current short read spliced aligners.

ALFALFA is designed for mapping long reads and remains accurate in the presence of a high number of sequence errors between read and genome. Although *ALFALFA* uses the seed-and-extend approach, it could also be used in conjunction with the exon-first strategy, as this strategy relies on standard DNA mappers for aligning (pieces) of reads. However, *ALFALFA* mainly excels at mapping long reads and the exon-first strategy relies on splitting reads into smaller pieces. Furthermore, long reads can potentially contain more splice sites than shorter reads, requiring the read to be split into more pieces to obtain alignments that fall within a single exon.

In order to align reads across intron boundaries, *ALFALFA* requires adjustments to several phases of the seed-and-extend strategy. To illustrate the need

for changes to the original algorithm, a chain-guided alignment on a spliced read is shown in Figure 5.3. In comparison to Figure 4.4, the candidate region (top) contains two large gaps resulting from spliced out introns. This has an impact on several key steps in the algorithm.

First, candidate regions need to be much larger encompassing a fragment/segment of the reference genome that is several times larger than the maximum expected intron size. As a result, candidate regions will usually contain a higher number of seeds. Second, more chains will need to be considered due to the increased number of seeds per candidate region. Third, chains should include seeds from multiple exons, notwithstanding the fact that these exhibit a large *skew*. This can clearly be seen in Figure 5.3, in which the skew between seeds on different exons is large. Fourth, the standard dynamic programming algorithms will perform poorly if used to close intronic gaps. Finally, the alignment algorithm needs to be able to identify the exact splice site locations, taking into account the existence of canonical and semi-canonical splice sites.

In this chapter, we investigate several solutions to the problems addressed above. The mapping algorithm that is modified to be able to map cDNA and RNA-seq reads has been dubbed *mesalina*. This seed-and-extend spliced aligner is designed to achieve a high performance on long spliced reads, while maintaining a high accuracy. The algorithm combines *ALFALFA* with powerful dynamic programming algorithms introduced by *GMAP*. Unlike many other novel spliced aligners, *mesalina* is also able to detect non-canonical splice sites. The details of the changes made to the *ALFALFA* algorithm are documented in Section 5.2. Preliminary testing on a proof-of-principle implementation indicates that *mesalina* achieves a high performance for long reads, while maintaining an accuracy that is comparable to that of *GMAP*. These results are discussed in Section 5.3.

5.2 Spliced alignment

mesalina is a cDNA and RNA-seq mapper that combines the long read DNA mapper *ALFALFA* with additional techniques to compute alignments that bridge large gaps and identify splice sites. Among the spliced alignment algorithms, it is classified as a seed-and-extend mapper. Because the *mesalina* algorithm shares both its general outline and most of the implementation details with *ALFALFA*, this section will focus on the differences between the *ALFALFA* and *mesalina* algorithms. For ease of reading, however, we provide a short summary of the techniques shared by both algorithms.

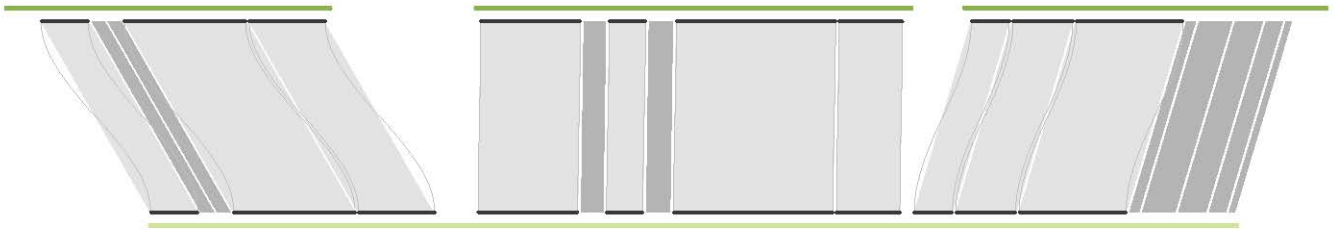


Figure 5.3: Illustration of a chain-guided alignment when mapping cDNA and spliced reads. This figure shows the candidate region on the reference genome (top) and the read (bottom), together with the seeds in the candidate region (connected lines). Grey areas show regions in an optimal alignment of three or more consecutive matching bases. Light gray areas are covered by seeds, whereas dark gray areas need to be found using dynamic programming. There are obvious gaps in the candidate region that correspond to the positions of large deletions caused by splicing out introns. The size of these gaps can range from a few bases to several hundreds of thousands bases.

The seed-and-extend strategy consists of first finding short matches between a read and the reference genome using an efficient index structure. *Mesalina* maintains the use of ESSA index structures and MEMs and SMEMs as seeds. The seeds are utilized to prune the alignment search space to regions that have around the same size of the final alignment. In case of spliced alignments, the length of an alignment is several times larger than the length of the read. For this reason, *mesalina* employs a different strategy for identifying and selecting candidate regions. Full alignments between the read and the candidate regions are calculated in a final extension stage. This is the stage that has seen most changes to the *ALFALFA* algorithm. The extension stage continues to rely on chain-guided alignment. However, gaps spanning an intron are detected using a special form of dynamic programming, called *sandwich dynamic programming*, which was introduced in *GMAP* [258].

The following sections provide a detailed description of the identification and selection of candidate regions and chain-guided alignment. For more details on the first phases of the algorithm, we refer to Chapter 4.

5.2.1 Candidate regions

Similar to *ALFALFA*, candidate regions consist of a set of seeds that are located relatively close in the reference genome. In the current proof-of-principle implementation, however, *mesalina* uses a less complex candidate region identification algorithm than the one described in Section 4.2.2. Instead of splitting the set of seeds in two, *mesalina* utilizes all seeds found in the previous phase of the algorithm to define new candidate regions. In practice, however, the increase in the number of candidate regions is small due to an increase in the length of candidate regions. Second, *mesalina* does not recalibrate candidate regions boundaries using the currently longest seed in the region (see line 18 in Algorithm 4.4). Instead, *mesalina* combines consecutive seeds in the reference genome if they are no further apart than a user-set maximum intron size.

In detail, MEMs are first sorted by their offset in the reference genome. This sorted list of MEMs is then processed from left to right. Candidate regions are formed by consecutive seeds in the sorted list that *i)* are not separated more than the user-set maximum intron size in the reference genome, *ii)* do not overlap in the reference genome and *iii)* have a certain user-set maximum overlap in the read.

Candidate region selection is handled similarly to *ALFALFA*. Candidate regions are sorted according to the percentage of bases of the read covered by at

least one seed and only clusters with a high enough coverage are extended. Due to a higher number of seeds per candidate region, minimum coverage requirements are higher (40% by default) than those required by candidate regions in *ALFALFA*. To compensate for this more strict parameter setting, no other restrictions are imposed on candidate regions in order to qualify for extension in the next phase of the algorithm.

5.2.2 Candidate region extension

To find alignments within candidate regions, *mesalina* uses a chain-guided alignment, similarly to *ALFALFA*. The chain-guided alignment strategy is more common in spliced aligners as bridging gaps caused by introns usually requires anchors at both sides of the gap. In contrast to *ALFALFA*, *mesalina* currently uses a simpler heuristical chaining strategy that does not take into account the skew between two consecutive seeds. The reason for this is again the presence of large deletions caused by splicing, as can be clearly seen in Figure 5.3. Furthermore, the proof-of-principle implementation of *mesalina* currently constructs a single chain per candidate region. This will likely be changed in future versions of the tool.

The alignment algorithm used by *ALFALFA* in Section 4.2.3 has also been modified. This section covered the procedures used to fill gaps in between consecutive seeds of a chain. In this gapped alignment of seeds, gaps can either result from differences within exonic sequences or span an intron. All gaps are resolved using specific dynamic programming routines, similar to the ones used in *GMAP* [258]. Which dynamic programming algorithm gets chosen depends on the *skew* between two consecutive seeds in the chain. To reiterate the definition, the skew between two seeds is the difference between *i)* the gap gap_G between the seeds in the reference genome, and *ii)* the gap gap_R between the seeds in the read sequence.

In addition to the skew, the choice of the dynamic programming algorithm also depends on a user-set minimum intron length. If the skew is smaller than the minimum intron length, the algorithm acts similarly to *ALFALFA* and performs a basic global banded alignment over the region defined by the gap between the seeds. Spliced alignment is performed only in case the skew is larger than the minimum intron size. This case is handled using *sandwich dynamic programming*, which was introduced in *GMAP* [258] and is discussed in detail below. Only the case in which gap_G is far greater than gap_R occurs in practice as this corresponds to a splice event. The converse case, in which gap_R is far greater than gap_G, is

also handled by sandwich dynamic programming but only rarely occurs. The extra distance in the read is then covered by a single long insertion. Finally, gaps between the seeds at both ends of the chain and both ends of the read are handled using standard semi-global alignment. As a result, no introns can be found that are not surrounded by seeds on both sides of the intron, which is a known limitation of the seed-and-extend strategy. *GMAP* solves this problem by reseeding the candidate region using shorter seeds.

To identify intron boundaries, *mesalina* uses sandwich dynamic programming to close the gap between two seeds in a candidate region. Performing a standard variant of dynamic programming becomes infeasible in case this region spans an intron, as the intron itself can span several thousand nucleotides. This situation is illustrated in Figure 5.4a in which the width of the dynamic programming matrix DP is much higher than its height. Furthermore, standard dynamic programming routines do not take into account the presence of a single large gap and do not allow to take into account additional information such as the presence of pairs of canonical dinucleotides.

In contrast to the standard dynamic programming routines, sandwich dynamic programming consists of filling two smaller dynamic programming matrices and retrieving splice site locations using a combination of the scores in both matrices.

The sandwich dynamic programming algorithm is illustrated in Figure 5.4b. The algorithm first performs standard banded dynamic programming between the gap_R region in the read and two regions of similar size $gap_{G'}$ on the left and right end of the gap in the reference genome. To allow for indels and some flexibility in the alignment, $gap_{G'}$ is a few bases longer than gap_R. Both gaps also include a few bases of the seeds, as depicted by the small overlap of the gap regions and the seeds in Figure 5.4b. Also note that the matrices DP_l and DP_r are filled from opposite corners due to opposite alignment anchor points.

To find the exact location of the splice site, each position in gap_R is tested and receives a score. An intron is inserted at the position with the highest score. The score for a position is the sum of three terms: *i)* the maximum score of that position (row) in DP_l, *ii)* the maximum score of the next position (row) in DP_r and *iii)* a bonus if the position would result in a canonical or semi-canonical splice site.

Although this method promotes canonical and semi-canonical splice sites, it is also able to detect non-canonical splice sites if no canonical splice sites are located within the region where dynamic programming is performed or if the score for

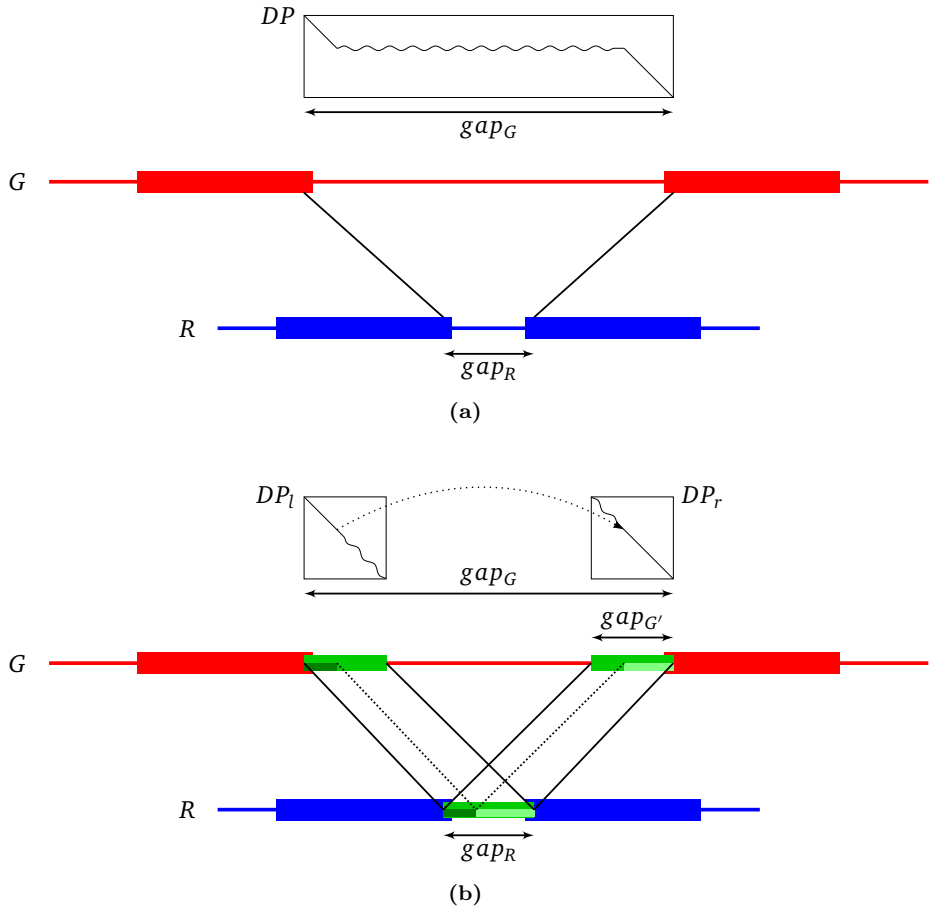


Figure 5.4: Dynamic programming routines used to fill the gap between two seeds in case of a large skew. Seeds on the reference genome G are separated by an intron, whereas the distance gap_R in the read R is much smaller. Standard dynamic programming (a) would result in low performance due to the large dimension of the matrix DP. Sandwich dynamic programming (b) uses two smaller dynamic programming matrices DP_l and DP_r that have dimension $gap_{G'} \times gap_R$. Location of the exon-intron boundaries is decided using a combination of the alignment scores in both matrices. The full alignment consists of traces in DP_l, DP_r and the intron gap indicated by the dotted line between the two matrices. Figures adapted from De Smedt [49].

a possible non-canonical splice site is much higher than possible canonical splice sites within the same region.

5.2.3 Implementation

Currently, a proof-of-principle implementation of *mesalina* is available for download as a stand-alone mapper[1]. The source code extends version 0.6 of *ALFALFA*, which is several steps behind the version presented in Chapter 4. As such, *mesalina* does not include all features of the current *ALFALFA* implementation. The most important algorithmic differences were outlined in this section, but differences in the available parameters, as described in Appendix E, should be checked from the command line. The implementation of sandwich dynamic programming as first described in *GMAP* was implemented by Dieter De Smedt.

5.3 Results

To validate the potential of the approach taken in *mesalina*, we ran the current implementation on several simulated read sets in addition to a data set of expressed sequence tags (ESTs). We compared the performance and accuracy of *mesalina* against *GMAP* [258] (v2013-08-19), GNSAP [257] (v2013-08-19) and *TopHat2* [128] (v2.0.9).

The *RNASeqReadSimulator* program [158] was used to simulate reads from *Arabidopsis thaliana* (TAIR10[2]). Three data sets were produced with varying read lengths of 75bp, 200bp and 500bp. Each data set contained 100 000 reads with uniform expression profile and simulated substitution errors. An error rate of 5% was used, which is consistent with PacBio CCS (consensus sequence) data [210]. A data set of 48 438 ESTs used in the original *GMAP* article were obtained from the *GMAP* homepage[3] and mapped against the human genome assembly GRCh37, obtained from the UCSC genome browser website.

All tests with simulated reads were run on a single core of a Dell PowerEdge R610 server with Intel Xeon processor at clock speed 3.07GHz and 48GB RAM running Debian 7.2. Tests with ESTs were run on a single core of a PowerEdge R520 TPM server with Intel Xeon processor at clock speed 1.90GHZ and 8GB RAM/core running Debian 3.2.41. The tools were run using a single thread and with default parameter settings. The performance was measured as the

[1] `https://github.com/readmapping/mesalina`

[2] TAIR10_exon_20101028 `http://arabidopsis.org/index.jsp`(last accessed August 2013

[3] `http://research-pub.gene.com/gmap/`(last accessed February 2014)

runtime of the programs using the GNU/Linux `time` command, excluding index construction time, as this is independent of the size of the read data set. For the ESTs, the sensitivity was used as accuracy measure. Accuracy results for simulated data show the percentage of correctly mapped reads. A read is mapped correctly if the mapper returns an alignment that maps the read to the correct simulated mapping position and whose CIGAR string correctly identifies the intron boundaries set by the gene annotation data.

5.3.1 Memory footprint

Memory requirements of *mesalina* are equal to those of *ALFALFA* as both tools share the same index structure. For the *A. thaliana* reference genome, the memory requirements of *mesalina* are 1165MB, 378MB and 205MB for sparseness values of 1, 4 and 12, respectively. In comparison, *GMAP* and *GSNAP* share the same index and thus the same memory footprint, which is 493MB for *A. thaliana*. *TopHat2* requires 171MB for the *A. thaliana* reference genome, which is the lowest memory footprint of the evaluated tools. Overall, *mesalina* can be configured to have a memory footprint similar to that of other read mappers, but has the advantage of being able to boost performance at the cost of memory by adjusting the sparseness of the index structure.

5.3.2 Performance and accuracy trade-offs

The performance and accuracy achieved on the simulated read data sets are summarized in Table 5.1. In addition to *GMAP*, *GSNAP* and *TopHat2*, three different settings of *mesalina* were tested that differed in the sparseness s of the ESSA index.

Table 5.1 clearly shows the detrimental impact of read length on both the accuracy and performance of all tested mappers. This can be explained by an increased number of reads containing (multiple) introns, especially when reads are longer than the average exon length (250bp for *A. thaliana*).

A comparison between *mesalina* and *GMAP* is interesting as both seed-and-extend mappers share several algorithmic techniques. *GMAP* is the most accurate among all tested spliced aligners, but its runtime is much higher than that of the other mappers. Although *mesalina* is generally less accurate than *GMAP*, the difference in accuracy is relatively small. For reads of length 200bp, we even report a slightly higher accuracy, although the absolute difference in mapped reads is small due to the size of the data sets.

Table 5.1: Performance and accuracy of spliced aligners on three sets of 100 000 reads, simulated from the *A. thaliana* genome. The data sets are identified by the length of the reads, which are respectively 75bp, 200bp and 500bp.

	runtime (seconds)			accuracy (% correct)		
	75bp	200bp	500bp	75bp	200bp	500bp
mesalina ($s = 1$)	35	41	52	84.4	76.9	62.1
mesalina ($s = 4$)	59	92	164	82.4	74.4	59.7
mesalina ($s = 12$)	54	81	136	82.0	73.9	59.2
GMAP	459	849	1532	85.5	76.1	63.6
GSNAP	101	363	1785	88.2	79.2	65.2
TopHat2	23	76	240	83.6	70.1	52.4

TopHat2 is known to be very fast and accurate for short reads, which is also illustrated by the results of the 75bp data set in Table 5.1. Compared to the other read mappers, however, its accuracy drops significantly for longer reads and its performance drops tenfold. Although *mesalina* is slower than *TopHat2* for shorter reads, it becomes more than four times faster than *TopHat2* for longer reads, while maintaining a much higher accuracy.

By default, the sparseness setting of the ESSA index in *mesalina* is 12, as for this value the memory footprint of *mesalina* is comparable to that of the other mappers. If memory is abundant, the sparseness can be lowered and *mesalina* can become more than twice as fast. Lower sparseness settings also positively affect the accuracy of *mesalina*. This effect is caused by the fact that the seed-finding algorithm produces more seeds when sparseness is low. The effect could, however, be mitigated by more complex seed-finding and candidate region identification algorithms, such as those that are currently used by *ALFALFA*.

In addition to the results for simulated reads presented in Table 5.1, we evaluated the performance and the sensitivity of *mesalina* and *GMAP* on a data set of roughly 50 thousand EST sequences. Due to the size of these sequences, *TopHat2* and *GSNAP* were unable to produce meaningful results. *Mesalina* (with $s = 12$) was more than 25 times as fast as *GMAP*. In contrast, *GMAP* was able to map 97.1% of the sequences, whereas *mesalina* was only able to map 91.7% of all sequences.

These preliminary experimental results indicate that our approach achieves a new and interesting performance-accuracy trade-off, especially for longer reads.

5.3.3 Discussion

Many novel spliced aligners are very fast and accurate for mapping short RNA-seq reads. They are, however, not designed to handle longer reads and few are able to detect non-canonical splice sites. In contrast, mappers designed to map ESTs and longer cDNA sequences have a much lower throughput than current short read mappers. Our goal was to bridge this gap by combining techniques from *ALFALFA* and sensitive alignment procedures from *GMAP* in a novel seed-and-extend spliced aligner called *mesalina*.

From an algorithmic perspective, *mesalina* demonstrates a promising combination of tried-and-tested techniques. As a result, the algorithm can either be seen as a speed-boost for seed-and-extend algorithms, such as *GMAP*, or as technique to provide spliced alignment support to long read mappers. To the best of our knowledge, the only algorithm containing a similar combination of techniques is part of recent versions of the *segemehl* read mapper [102]. This algorithm uses a combination of an enhanced suffix array for near-exact matching, seed chaining and *split alignment*, which is similar to sandwich dynamic programming.

Preliminary experimental results indicate that *mesalina* attains the goals that were set and achieves a new and interesting trade-off between performance and accuracy. It is much faster than *GMAP* in all test cases, while being only slightly less accurate. It is, however, much more accurate than *TopHat2*. Although *TopHat2* remains faster for shorter reads, *mesalina* performs better for longer reads. We should remark that these tests are still preliminary and performed on a small data set. Furthermore, the low accuracy of *TopHat2* could probably be alleviated by tuning command line parameters.

Although the current version of *mesalina* already shows promising results, the algorithm still has much room for improvement to obtain a higher accuracy for reads that are more difficult to map and the implementation could still be improved to obtain higher overall performance and a lower memory footprint.

The greedy chaining algorithm is currently the major source of misalignments and could be replaced by an optimal collinear chaining algorithm that allows restrictions on gap size [174]. Other causes of misalignments include failure to detect splice sites at the end of reads and failure to distinguish between two consecutive introns separated by an exon smaller than the minimum seed length. An example of a misalignment in shown in Figure 5.5. The small exon of length 14, shown in red, is not detected by *mesalina* due to a lack of seeds at both sides of the intron. The latter is caused by the restriction on the minimum seed length.

The runtime could be further decreased by selecting good parameter settings,

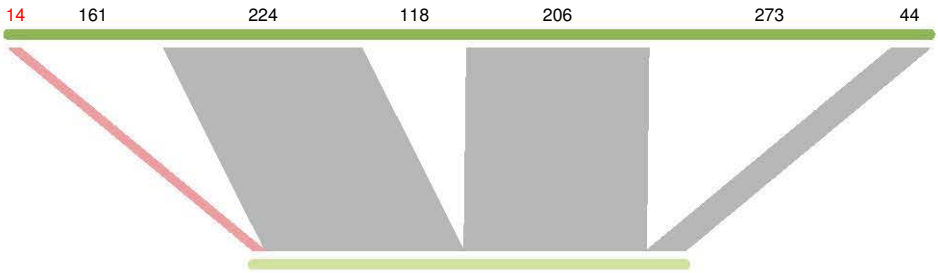


Figure 5.5: Alignment showing one of the main causes for misalignments using the current version of *mesalina*. The candidate alignment region found by *mesalina* (top) and the read (bottom) are shown, together with the exonic areas (shaded areas) and the size of the introns and exons. The red area represents a small exon at the beginning of the alignment. *Mesalina* was unable to find this gap due to the lack of an anchor seed at the left of the gap.

such as minimum seed length, but also by, for example, using a bit-parallel dynamic programming implementation in the extension stage. The memory footprint of the index could further be reduced by bit-encoding the reference genome.

In addition to algorithmic improvements, more rigorous tests need to be performed on large and varied data sets and experimental results need to be compared to a larger set of spliced aligners, using different parameter settings.

Finally, the current implementation of *mesalina* still lacks some of the features supported by other spliced aligners, including specific algorithms for the detection of micro-exons and alternative splicing, and paired-end read mapping.

Concluding remarks

In this dissertation, novel algorithms were presented for fast and accurate mapping of long next generation sequencing reads. Development of these algorithms is motivated by the (r)evolution in sequencing technology that continues to produce data at higher speed and lower cost. As a result, computational analysis and interpretation become true bottlenecks in life science research. In Section 1.2, we covered the advances made in sequencing technology and discussed the wide variety of sequencing reads with different features these platforms produce.

Read mapping plays a key role in many genomics analysis pipelines and therefore need to adapt to the changing landscape in sequencing technology. In Section 1.3.3, we showed that past decade gave rise to many novel mapping algorithms and updates of existing tools (see Figure 1.8). Furthermore, read mapping algorithms closely follow the advances in technology and the demand for new applications. However, few read mappers outright outperform older mappers. Instead, many have unique features, or present a unique trade-off between accuracy, performance and memory requirements. For example, the read mappers *ALFALFA* and *mesalina* are very fast in accurately mapping sets of long reads that can contain relatively large numbers of mismatches and/or indels. Furthermore, the enhanced sparse suffix array index structure utilized by both mappers can be tuned to further balance processing speed, memory consumption and mapping accuracy.

Although sequencing technology evolves towards longer reads, many recently proposed still target short reads and allow for no or low numbers of mismatches and/or indels. Furthermore, the evaluation of *ALFALFA* and *mesalina* shows that the performance of most current long read mappers still rapidly degrades with increasing read length. The benchmark results in Chapter 4 also show that the performance and accuracy of read mappers can depend on other factors. For example, several mappers are susceptible to the presence of many indel type

errors or a large variation on the insert size of paired-end reads. There are also other, more application-specific factors that influence performance, such as the number of alignments reported per read. As a result, the algorithms presented in this dissertation make a substantial contribution to the field of read mapping.

The most important component of the algorithms developed in this dissertation is the enhanced sparse suffix array index structure that facilitates fast string searches in the reference genome. Although index structures are already widely used to speed up bioinformatics applications, we have found that there is a gap between the field of index structure research and its application domains. As a result, most tools are designed using basic implementations of index structures, without taking full advantage of the latest advances in indexing technology. We tried to bridge the gap between these two fields of research in Chapter 2 with a comprehensive review of the basic ideas behind classical full-text index structures and an overview of the limitations of these data structures as well as the research done in the last decade to overcome these limitations.

Since the review in Chapter 2 was written [247], several new index structures have been proposed, some of which are described in Section 2.5. Some of these recent developments work towards eliminating some limitations pointed out by us in the review, such as lightweight construction algorithms [22, 26, 45, 196] and practical implementations of compressed index structures [86].

Although every review will at some point be outdated, our survey of full-text index structures has already partially achieved its goal of introducing index structure research to the bioinformatics community. The review has been cited in research articles to provide a theoretical background [79, 168], but also in light of sequence alignment tools [61, 248], index structure construction [43, 229], and compression of biological data [46, 52, 83].

We also contributed to the field of index structure research through the development of the enhanced sparse suffix array, which was introduced in Chapter 3. We applied our index structure to the application of finding all maximal exact matches between two sequences in the tool *essaMEM*. The ESSA enhances an existing SSA design [126] with a sparse variant of the child array present in the ESA index structure [3]. Sparse child arrays can be seamlessly incorporated in sparse suffix arrays, with minor modifications to the construction algorithm. In addition to the sparse child array, we modified the existing MEM-finding algorithm of Khan *et al.* [126] by introducing sparseness in the query sequence (sequence that is not indexed) as well. Both modifications improved the performance of MEM-finding considerably, without, increasing the memory footprint of the in-

dex. Later, we improved the performance of *essaMEM* further by introducing a k-mer table that connects sequences of fixed length k to the suffix intervals in the ESSA index structure. This added data structure improves the performance up to three times.

The evaluation in Section 3.4 not only demonstrate that *essaMEM* is the fastest MEM-finding algorithm at that time, but also that the use of ESSA-based algorithms is a viable option for further research. For example, memory requirements can be further improved by compressing the reference genome, whose size currently dominates the memory footprint of the index. In addition, algorithms could be developed to efficiently find specific types of MEMs, such as maximal unique matches and super-maximal exact matches.

Since the initial release of *essaMEM*, two new MEM-finding tool have been developed, called *slaMEM* [61] and *GPUMEM* [4]. Both tools report to be faster than *essaMEM* (which might not yet utilize the k-mer table optimization) on most of tested data sets. It is, however, shown that *essaMEM* is the fastest CPU-based tool available [4], as *slaMEM* does not implement shared-memory parallelism and *GPUMEM* is a GPU-based MEM-finding tool. Moreover, *GPUMEM* is sometimes outperformed by *essaMEM* if no load balancing is performed. This suggests a possible path for future research as *essaMEM* also currently lacks methods for load balancing.

The *essaMEM* algorithm is not only an efficient MEM-finding tool, but is also forms an important component of the long read mapper *ALFALFA*, which is presented in Chapter 4. *ALFALFA* is extremely fast for accurately mapping long reads ($>$ 500bp) due to its implementation of the canonical seed-and-extend approach that is empowered by *essaMEM*, combined with several new optimizations and heuristics.

Ideally, seed-finding produces a limited number of long seeds that cover as much of the mapping location as possible. To this end, *ALFALFA* finds maximal and super-maximal exact matches, using either the MEM-finding algorithm from *essaMEM* or two new algorithms that more rapidly find a smaller set of interesting MEMs. *ALFALFA* implements several heuristics to boost the performance of seed-finding, such as automatic calculation of the minimum seed length and restrictions on the size of the output. We also compared the effect of the implemented algorithms on the accuracy of the read mapper.

SMEMs and rare MEMs are combined into candidate alignment regions using the locally longest seeds to define boundaries for the candidate regions. Merging of overlapping regions is performed to avoid possible loss of information in

repeated regions. Furthermore, seeds that are not used for candidate regions identification are recovered later in the algorithm to help prioritize the extension of candidate regions and during the chain-guided alignment.

The number of candidate regions that are examined depends on the quality of the seeds within that region. We use various extension criteria that, for example, take into account read coverage and presence of unique seeds. Candidate region extension is performed using multiple chain-guided alignments, which potentially require only a fraction of the computational cost of the typically used banded alignment routines. Although we employ a greedy chaining algorithm, few sub-optimal alignments are obtained due to several heuristics, such as restrictions on the skew between two consecutive seeds in the chain.

ALFALFA also includes several paired-end mapping strategies that follow the outline of the single-end alignment strategy, but also test the paired-end alignment restrictions at different stages of the algorithm. Furthermore, both single-end and paired-end alignment strategies contain procedures that automatically overcome to stringent parameter settings and increase the effort made to find an alignment, if necessary.

The overview of existing read mappers in Section 1.3.3 clearly shows that no single read mapper evaluation method can take into account all aspects and features of the tested mappers, especially because there is currently still a lack of generally accepted benchmarking methods. Nevertheless, we have tried to perform a fair evaluation of *ALFALFA* against other read mappers by using a large number of simulated read data sets with different features and by using multiple methods for measuring the accuracy of the tested read mappers.

Similar to *essaMEM*, the ESSA allows balancing the performance and accuracy trade-off. The default sparseness value used for all tests in Section 4.3 results in a memory footprint that is comparable to that of the other mappers in the benchmark. When memory is abundant, however, *ALFALFA* can be up to twice as fast by lowering the sparseness value.

The high performance of *ALFALFA* on long reads, but also on moderately sized reads, immediately stands out from the benchmark results. *ALFALFA* is several times faster than its closest competitor, and is only outperformed by other read mappers on the shortest read sets. Furthermore, the performance of *ALFALFA* remains high when reporting multiple alignments per read.

ALFALFA is among the most accurate read mappers that were tested, who are highly accurate in general. Some read mappers are slightly more accurate than *ALFALFA*, but the opposite is also true on some data sets. *ALFALFA* is

also very robust to different error rates, error models and various measures of accuracy. Furthermore, *ALFALFA* has a high sensitivity for mapping real 454 reads and presents a nice trade-off between sensitivity and specificity.

Future versions of *ALFALFA* might benefit from lower memory requirements and higher performance through the earlier listed improvements to *essaMEM*. Currently, the bottleneck in runtime is situated in the extension of candidate regions. Further optimizations to the dynamic programming routines might help alleviate this bottleneck, as well as or heuristics that ensure fewer extensions are necessary. Because *ALFALFA* currently supports basic multi-threading shared-memory parallelism, an important route towards increasing the performance is implementation of more advanced parallelization techniques.

Another direction for improving *ALFALFA* is manual or automatic tuning of parameter settings by gathering alignment statistics during the alignment of the first set of reads. Likewise, optimal default parameter configurations could be found for each sequencing technology. Because accuracy is already high, however, extensive testing will be required to achieve even higher accuracy and solutions might severely harm the current performance.

In addition to improvements to the mapping and alignment algorithm, an important step for future work is an evaluation of the performance of *ALFALFA* using different types of reads and different reference genomes. For example, reads produced by Pacific Biosciences and Oxford Nanopore contain a higher number of sequencing errors than the reads used in the current evaluation. Similarly, mapping reads to prokaryotic genomes and polyploid plant genomes imposes additional challenges. Finally, it would be interesting to evaluate the accuracy of read mappers in specific difficult-to-map regions of a genome. To tackle these challenges, it might be required to search for inexact seeds in addition to MEMs and SMEMs.

In Chapter 5, we investigated the possibility of using *ALFALFA* for spliced alignment of cDNA sequences and long RNA-seq reads. Spliced alignment poses additional algorithmic challenges due to the presence of large gaps in the alignment at splice sites. In contrast to DNA mapping, accurately mapping RNA-seq reads also becomes more challenging if read length increases. Although *ALFALFA* is accurate in mapping DNA reads, modifications are required to several stages of the algorithm to also work well for spliced alignment. We therefore introduced *mesalina*, a prototype of a spliced alignment algorithm based on *essaMEM* and *ALFALFA*. The prototype modifies the candidate region identification and extension stages of the *ALFALFA* algorithm. Candidate regions are much larger to

span multiple exons, chain-guided alignment does not restrict the skew of consecutive seeds and the sandwich dynamic programming technique, introduced in GMAP, is used to span intronic gaps and identify splice sites.

Preliminary evaluation of a proof-of-principle implementation indicate that *mesalina* achieves a new and interesting trade-off between performance and accuracy. It is much faster than GMAP in all test cases, while being only slightly less accurate. It is, however, much more accurate than TopHat2. Although TopHat2 remains faster for shorter reads, *mesalina* performs better for longer reads.

Although the current version of *mesalina* already shows promising results, the algorithm still has much room for improvement to obtain a higher accuracy for reads that are more difficult to map and the implementation could still be improved to obtain higher overall performance and a lower memory footprint. The greedy chaining algorithm is currently the major source of misalignments and could be replaced by an optimal collinear chaining algorithm that allows restrictions on gap size [174]. Other causes of misalignments include failure to detect splice sites at the ends of reads and failure to differentiate between two consecutive introns separated by an exon smaller than the minimum seed length. A possible solution might be to perform another seed-finding routine that uses a lower minimum seed length and is limited to the candidate region. In addition, multiple alignments will need to be performed when short seeds give rise to multiple possible exon locations.

The algorithms developed in this dissertation have shown to be fast and accurate in mapping various types of long sequencing reads. These results show that it is possible for read mappers to keep up with the continuous technological advances in sequencing technology and data growth by using a good combination of advanced index structures, search algorithms and heuristics. However, as technology evolves, all read mappers discussed in this dissertation might once be outdated. We therefore look forward to others building on our ideas, similarly to how we incorporated previous results into our algorithms.

Appendix A

List of abbreviations

ALFALFA	:	a long fragment aligner (2×)
ASCII	:	american standard code for information interchange
BAM	:	binary alignment/map format
BLOSUM	:	block substitution matrix
bp	:	base pairs
BWT	:	Burrows-Wheeler transform
cDNA	:	complementary DNA
CPU	:	central processing unit
CSA	:	compressed suffix array
CST	:	compressed suffix tree
DNA	:	deoxyribonucleic acid
EBI	:	european bioinformatics institute
ESA	:	enhanced suffix array
ESSA	:	enhanced sparse suffix array
EST	:	expressed sequence tag
FPGA	:	field-programmable gate array
Gbp	:	Giga (billions) base pairs

GNU	:	GNU's Not Unix!
GPU	:	graphics processing unit
GST	:	generalized suffix tree
indel	:	insertion or deletion
I/O	:	input/output
IUPAC	:	international union of pure and applied chemistry
kbp	:	kilo (thousands) base pairs
LCA	:	lowest common ancestor
LCP	:	longest common prefix
LF	:	laft-to-first
LZ	:	Lempel-Ziv
MAM	:	maximal almost-unique match
MAPQ	:	mapping quality
Mbp	:	Mega (millions) base pairs
MEM	:	maximal exact match
miRNA	:	microRNA
MPI	:	Message Passing Interface
MUM	:	maximal unique match
N.A.	:	not available
NCBI	:	national center for biotechnology information
NGS	:	next generation sequencing
nt	:	nucleotide
PAM	:	point accepted mutation (matrix)
RAM	:	random-access memory
RNA	:	ribonucleic acid
ROC	:	receiver operating characteristic
SA	:	suffix array
SA^{-1}	:	inverse suffix array

SAM	:	sequence alignment/map format
SIMD	:	single instruction, multiple data
sl	:	suffix link
SMEM	:	super-maximal exact match
SNP	:	single-nucleotide polymorphism
SRA	:	short read archive
SSA	:	sparse suffix array succinct suffix array
SSE2	:	streaming SIMD extensions 2
ST	:	suffix tree
XML	:	extensible markup language

Appendix B

Sequence alignment and mapping format

The standard output format for read mappers is the *Sequence Alignment/Map* (SAM) format [152], which is a TAB-delimited text file. This format stores reference genome, read, mapper and alignment information. The alignment information consists of mapping position, the alignment itself, paired-end mapping information, mapping quality mapping and other supplementary information. In addition, the SAM format allows developers to add new fields to an alignment line to store extra information. SAM files can be compressed in binary form, resulting in a *Binary Alignment/MAP* (BAM) file using the *SAMtools* software package, which was introduced together with the file format itself [152]. The *SAMtools* package also contains various postprocessing tools for indexing, sorting, variant calling and viewing of the alignment information. This section contains an overview of all required fields in a SAM file, as well as the optional fields that are used by *ALFALFA* (see Chapter 4) and *mesalina* (see Chapter 5).

B.1 Example

In-depth information on the fields of the SAM format is given in the next sections. To illustrate the meaning of the SAM lines and fields, an example SAM file is given below, showing a possible alignment outcome of an alignment for paired-end reads given in Figure B.1. This figure shows the situation in which a pair of reads `r00/1` with sequence `ATAACTCCAGC` and `r00/2` with sequence `ATTCTGTTC` are

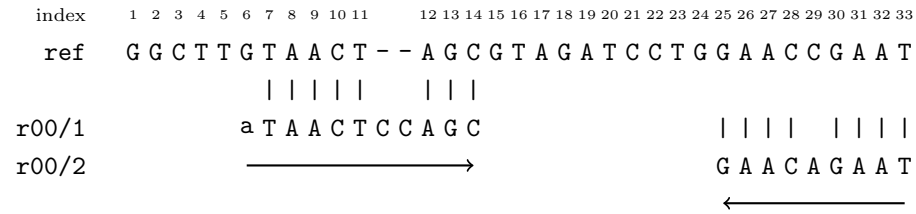


Figure B.1: Example of paired-end alignment. The top string is the reference genome with name `ref`. Below are two reads `r00/1` and `r00/2` that are respectively the first and second mate of a pair. The first mate is aligned on the forward strand and the second in the opposite direction. The positions of the reference genome are given 1-based, which is in correspondence with the SAM standard.

mapped against the reference genome `ref`. The sequence of `r00/2` in Figure B.1 is the reverse complement of the actual sequence, as this is the orientation of the alignment, which is also indicated by the arrow underlining the sequence. The insert size of the fragment is 28bp according to the alignment, as the first base of `r00/1` is not aligned, but *clipped*.

```
@HD VN:1.5  SO:queryname
@SQ SN:ref  LN:33
@PG ID:alfalfa  CL:...  VN:0.8
r00 99  ref 7   60  1S5M2I3M    =   25  28
   ATAACTCCAGC *   AS:i:0  NM:i:2  X0:Z:1S5=2I3=
r00 147 ref 25  60  9M  =   7   -28 GAACAGAAT   *
   AS:i:4  NM:i:2  X0:Z:4=1X4=
```

SAM lines are TAB-delimited and positions or coordinates in the file format are 1-based, which differs from the 0-based system used throughout this dissertation.

The header in the above example provides some global information about the SAM file. The SAM version used was 1.5, alignment lines are ordered by the `QNAME` field, alignment was performed by *ALFALFA* version 0.8 and reads were mapped to a single reference genome `ref` of size 33.

The example contains two alignment lines, *i.e.* one for each read. Although the read names differ, they share the same value for the `QNAME` field. This value is obtained by removing the `/1` and `/2` artifacts from the read names. The order of the reads (first and second mate) can be retrieved from the `FLAG` field.

The value of the `FLAG` field for the first alignment line is 99, with binary representation $99 = 2^6 + 2^5 + 2^1 + 2^0 = 1100011$, meaning that the read is paired (2^0), both mates are properly aligned and fulfill orientation and insert size criteria (2^1), the mate of this read is mapped reverse complemented (2^5) and this read is the first mate in the pair (2^6). The value of the `FLAG` field of the second alignment can be written as $147 = 2^7 + 2^4 + 2^1 + 2^0 = 10010011$. From this, we learn again that this read is part of a pair that is properly aligned (2^0 and 2^1) and that this read is mapped reverse complemented (2^4) and is the second mate in the pair (2^7).

The next two fields after the `FLAG` give the reference sequence (chromosome) of the alignment and the mapping position. Note that `POS` is 1-based and that the mapping position of the first read refers to its second base, as the first is not aligned, but *clipped* from the alignment. The three fields before the `SEQ` field provide more paired-end alignment information, respectively the mate's alignment reference sequence in `RNEXT`, mapping position in `PNEXT` and insert size in `TLEN`. As both mates align to the same reference sequence, the value of `RNEXT` is `=`. In addition, the `PNEXT` value of one mate equals the `POS` value of the other. `TLEN` is the number of bases from the left-most aligned based to the right-most aligned base. Because the first mate is aligned to the left of the second mate, its `TLEN` value is positive, whereas that of the second mate is negative.

Some fields, such as `MAPQ`, `SEQ` and `QUAL`, are self-explanatory. `MAPQ` is at the maximum allowed value for the mapper as no other alignment was found. `SEQ` shows the aligned sequence, which is the reverse complement of the read sequence in the case of the second mate. `QUAL` is marked as `*` because the quality values of the read sequences were unknown (which can happen if the reads are given in FASTA format).

The CIGAR string of the first alignment is `1S5M2I3M`, which means that the alignment consists of a clipped base (not included in the alignment due to local alignment), followed 5 matches/mismatches, an insertion gap of 2 bases and another 3 matches/mismatches. In this case, all matches/mismatches are indeed matches, as indicated by the `XO` optional field. This is not the case for the alignment of the second mate, which contains one mismatch and whose CIGAR string is `9M`. The position of the mismatch is given in the `XO` field, whose value is `4=1X4=`.

The presence of a mismatch in the alignment of the second mate can also be seen from the edit distance and alignment score of the alignment, given by respectively the `NM` and `AS` fields. The position of the mismatch could also be

retrieved in linear time from the alignment and both read and reference sequences. The alignment score was calculated in this example using a match score of 1, a mismatch penalty of −4, gap opening penalty of −6 and gap extension penalty of −1. Note that due to local alignment, the `NM` and `AS` fields do not take into account the first base of the first read.

Header lines

Most SAM files contain the optional header section which contains information on the SAM file format version used, reference sequences (chromosomes), read groups and mapping tool information. Header lines start with an '`@`' symbol followed by a two-letter line identifier, while alignment lines do not. Each line contains multiple fields that have a two-letter identifier, followed by a colon. The header consists of the following lines:

- `@HD` is the header line containing the fields:
  - `VN:` is required if this line is present and contains the SAM version number that is supported
  - `SO:` contains the sorting order of the alignments in the file; the possible values for this field are `unknown` (default), `unsorted`, `queryname` and `coordinate`; the first three are self-explanatory, and the last means that the alignment lines are primary sorted by their `RNAME` field, with the order defined by the `@SQ` lines in the header and secondary sorted by their `POS` field
- `@SQ` lines contain a reference sequence dictionary and have two mandatory fields and optional fields, which contain more sequence information, but which we do not utilize; the mandatory fields are:
  - `SN:` reference sequence name
  - `LN:` reference sequence length
- `@RG` has read group information with information on the sequencing process; these lines are not reported by our read mappers
- `@PG` contains program information; these lines contain information on programs used for alignment; our mapping tools add this line to the output file with the required `ID`-tag and the following optional tags:
  - `ID:` unique program identifier, which is a required field

  - `CL:` command line used for mapping
  - `VN:` version of the mapper

- `@CO` are comment lines of which more than one are allowed; these lines are not reported by our mappers.

Alignment lines

The alignment lines in SAM files represent a single alignment of a read against the reference genome, allowing multiple lines for additional alignments or other mates of paired-end reads. Each line consists of 11 mandatory fields, which need to appear in an exact order and several optional fields, which have no particular order. If information about a field is unavailable, its value can be set to `0` for integer fields or `*` for fields containing text strings.

Some fields were generally designed for sets of reads originating from the same DNA template, such as is the case for paired-end reads. We will give the description of these fields for paired-end reads, as these are mostly used in practice, but note that the original definition is more general and allows for more than two segments per template.

A brief description of all mandatory fields and the optional fields used in this work are given below and a more detailed description of two fields, `FLAG` and `CIGAR` follow. A highly detailed description of the fields can be found in the SAM manual[1].

The mandatory fields are:

i) `QNAME` is the field that contains the read name. Multiple lines and alignments can be present with the same value of this field. Both mates of pair should have the same value for this field.

ii) `FLAG` is a bitwise flag that stores information on the alignment that can be represented as a boolean value. The meaning of the flags is given in Table B.1.

iii) `RNAME` contains reference sequence name of the alignment, corresponding to a name on the `@SQ` lines in the header, if present. If the fragment is unmapped, the value of this field is `*`, and no assumptions can be made about `POS` and`CIGAR`.

[1] `https://github.com/samtools/hts-specs`

iv) `POS` is the 1-based leftmost mapping position of the alignment in the sequence specified by `RNAME`. The value of `POS` refers to the first matched base in the read, excluding bases that are clipped in preprocessing or unaligned when local alignment is used. If the fragment is unmapped, `POS` is 0.

v) `MAPQ` contains the mapping quality as an integer value in the interval [0..255], with 255 representing an unknown mapping quality.

vi) `CIGAR` is is a run-length encoded form of the alignment representation. More information about this field is given below. `*` indicates an unknown value.

vii) `RNEXT` is the reference sequence name of the mate of this read in the pair. If the value of this field would equal that of `RNAME`, `=` is used instead. For unmapped mates or single-end reads this field has value `*`.

viii) `PNEXT` stores the `POS` value of the mate's primary alignment for paired-end reads and is zero otherwise.

ix) `TLEN` is a signed value representing the insert size of a paired-end read alignment. From the aligner's perspective, `TLEN` is the number of bases between the leftmost mapped base and the rightmost mapped base. The value is positive for the leftmost mate and negative for the rightmost mate (according to the alignment). It is set to zero for single-end reads or if the information is unknown.

x) `SEQ` contains the read sequence. It can be set to `*` to save space.

xi) `QUAL` contains the read quality values, as given in the FASTQ file. It can be set to `*` to save space.

The optional fields are of the form `TAG:TYPE:VALUE`, where `TAG` is a two-letter identifier, `TYPE` indicates whether the field is an integer `i`, a string `Z`, or another value. Only integer and string are used in the *ALFALFA* [249] custom SAM format.

- `AS:i:` is the alignment score of the alignment.
- `NM:i:` is the edit distance of the alignment.
- `XO:Z:` is the CIGAR string containing match/mismatch information (see below).

Table B.1: Description of the bitwise flags making up the `FLAG` field in an alignment line of the SAM format. The positions start from the least significant bit (corresponding to the number 1).

bit index	description
1	set for paired-end reads
2	set if both mates are properly aligned according to the aligner
3	set if the read is unmapped
4	set if the mate in this pair is unmapped
5	set if the read is mapped reverse complemented
6	set if the paired mate is mapped reverse complemented
7	set if this read is the first mate in the pair
8	set if this read is the second mate in the pair
9	set if the read is a secondary alignment; this alignment is not considered the best one by the mapper
10	is platform/mapper unique, does not have a unique definition and not used in *ALFALFA* and *mesalina*
11	set for PCR duplicates and not used in *ALFALFA* and *mesalina*
12	set for supplementary alignments indicating chimeric alignments and not used in *ALFALFA* and *mesalina*

Flag field

The value of the `FLAG` field is an integer consisting of 12 bitwise flags. The flags are given in Table B.1, ordered by bit index from least significant bit to most significant bit.

CIGAR field

The CIGAR string is a run-length encoded representation of the alignment, where consecutive sequences of matches, mismatches, gaps and other alignment artifacts are represented by characters preceded by their length. The description of the symbols in a CIGAR string are given in Table B.2.

As shown in Table B.2, the symbol `M` represents both matches and mismatches. Although this decreases the level of information, the exact state of the base can be derived from the alignment position, CIGAR string and both read and reference genome. The use of one symbol further decreases the size of the CIGAR string. Most mappers, including *ALFALFA* and *mesalina*, use the `M` symbol instead of

Table B.2: Description of CIGAR string in SAM alignment lines. The first column shows the symbols used in the CIGAR string and the second column provides a brief description of those symbols.

symbol	description
`=`	sequence match
`X`	sequence mismatch
`M`	sequence match or mismatch
`I`	insertion to the reference genome
`D`	deletion from the reference genome
`N`	skipped region in the reference genome
`S`	soft clipping (bases present in `SEQ` field)
`H`	hard clipping (bases not present in `SEQ` field)
`P`	silent deletion of bases (padding), not used.

the combination of `=` and `X`. The optional `XO` field in our custom SAM format contains the alternative CIGAR string using the distinct symbols for matches and mismatches.

The symbol `N` is used to indicate a region skipped in the reference genome. This can be used by spliced aligners to indicate the location of introns (see Chapter 5).

Soft and hard clipping are used at the ends of an alignment and can be used by read mappers producing local alignments or read mappers that preprocess reads by removing low quality bases from the ends of the read.

An example of the CIGAR string is given in Figure B.2. The alignment starts at base 4 in the read, followed by 8 matches, an insertion-type gap of length 2, another 10 matches, a mismatch and 5 more matches. The CIGAR string is either `3S8M2I16M` using the `M` symbol or `3S8=2I10=1X5=` using the separate symbols for matches and mismatches.

```
    C A T G A A C A T G A - - A G A C G T G T G C T C T C C A T
(a)     | | | | | | | |   | | | | | | | | | | |  | | | | |
    a c a G A A C A T G A A A A G A C G T G T G C C C T C C A

(b) S S S = = = = = = = = I I = = = = = = = = = = X = = = = =
(c)  3S       8=      2I       10=       1X   5=
(d)  3S       8M      2I            16M
```

Figure B.2: Example of (a) an alignment between a reference genome (top) and a read sequence (bottom), (b) the CIGAR characters for the alignment, (c) the CIGAR string using matches and mismatch characters, and (d) the CIGAR string with a single character for matches and mismatches.

Appendix C

Details of the essaMEM experimental results

This appendix contains additional details on the benchmark method and experimental environment used for the experimental tests performed on *essaMEM* in Chapter 3. In addition, this appendix also contains tables with the exact experimental results that are depicted in the figures of Section 3.4.

C.1 Testing environment and experimental measurements

Two different machines were used for testing. Machine 1 is a cluster consisting of Intel Xeon L5420 CPUs with 16GB RAM/node running scientific Linux 5. Machine 2 is a cluster with dual-socket quad-core Intel Xeon Nehalem (L5520) processors at clock speed 2.27GHz and 12GB RAM/node running Scientific Linux 6.1. All tests with $\ell = 100$ are run on machine 1 and all tests with $\ell \leq 50$ are run on machine 2.

Programs used in testing were compiled locally using gcc v4.1.2, unless the source code was not freely available. All programs were run single-threaded, unless state otherwise, on a full node of the cluster and were the single active process on that node. Due to limitations of the cluster, all tests were limited to a maximum wall time of 72 hours.

All programs except *Vmatch* are executed with parameters `-maxmatch -l`

`LENGTH -n reference.fasta query.fasta`. The first parameter indicates that MEMs are calculated, the second sets the minimum length to `LENGTH`, the third sets the program to only match the nucleotide characters and the final parameters are the reference and query sequences. In addition, *sparseMEM* and *essaMEM* are run with parameter `-k SPARSE` to set the sparseness value s. The compression factor of *backwardMEM* was set using the appropriate binary `backwardMEMK`, where `K` is the sparseness factor. Finally, the parameters `-suflink 0/1` and `-child 0/1` are used to switch suffix links and sparse child array support on or off in *essaMEM* and `-skip p` sets parameter p to `p`.

Vmatch first constructs an index using `mkvtree -db reference.fasta -dna -indexname INDEX -pl -allout`. Afterwards, *Vmatch* is run with parameters `vmatch -q query.fasta -qspeedup 0/2 -l LENGTH INDEX`. We report results for *Vmatch* with `-qspeedup 0` and `-qspeedup 2`. The 32 bit version of *Vmatch* was used for the megabase-sized genomes (1-6) because it has a smaller memory footprint and the 64 bit version was used for data set 9, due to memory limitations in constructing the index.

Wall times of test runs were measured using the GNU/Linux `time` command. In addition, the runtime does not include the time for the index construction phase, if the program contains such as phase. The resident set size (`rss`) value of the Unix `ps` command is used for memory measurements. Both peak memory usage and mean memory usage are reported.

C.2 Additional tables

Table C.1: Test results of MEM-finding between megabase-sized genomes (data sets 1-3 in Table 3.2) for all settings of *essaMEM*. The parameter settings for the different *essaMEM* results can be found in Section 3.4.1. *essaMEM-1* and *essaMEM-2* share the same memory requirements, as well as *essaMEM-4* and *essaMEM-5*. All time results are in seconds and all memory results are in Megabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem).

data		*essaMEM-1*				*essaMEM-2*	*essaMEM-3*				*essaMEM-4*				*essaMEM-5*
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Tmem	Pmem	Emem	time	time
1	1	294	320	320	68	10	411	435	435	21	176	315	205	411	47
	2	162	181	181	69	16	220	238	238	31	103	172	123	365	73
	4	96	112	112	60	33	125	140	140	65	66	101	101	315	161
	8	62	77	77	56	56	77	91	91	71	48	70	70	263	263
	16	46	59	59	3970	3970	53	66	66	4423	39	56	56	4097	4097
2	1	953	982	961	110	9	1334	1354	1334	28	572	964	589	669	25
	2	524	592	564	104	10	715	778	750	40	333	539	378	567	39
	4	310	373	345	96	15	405	466	438	94	214	311	252	539	67
	8	202	263	235	85	24	250	309	281	116	155	216	188	465	115
	16	149	209	181	76	39	173	232	204	76	125	193	165	380	201
	32	122	182	154	70	70	134	193	165	70	110	174	146	328	328
3	1	355	442	442	243	13	496	580	580	65	213	348	303	1340	46
	2	195	286	286	210	18	266	355	355	96	124	216	216	1165	75
	4	115	175	175	195	30	151	210	210	200	80	140	140	1038	124
	8	75	169	169	183	51	93	186	186	214	58	152	152	852	198
	16	55	116	116	157	77	64	125	125	156	47	141	141	730	374
	32	45	107	107	132	132	50	111	111	125	41	135	135	590	590

Table C.2: Test results of MEM-finding between megabase-sized genomes (data sets 1-3 in Table 3.2). All time results are in seconds and all memory results are in Megabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem). The memory measurements of *essaMEM* are the same as for *sparseMEM*. Note that *MUMmer* does not contain an option to set the index size and *Vmatch* only allows to set the `-qspeedup` parameter.

data		*sparseMEM*				*essaMEM-2*	*backwardMEM*				*MUMmer*			*Vmatch*		
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Pmem	Emem	time	Pmem	Emem	time
1	1	294	320	320	22	10	158	485	193	35	490	490	20	(*Vmatch-0*)		
	2	162	181	181	36	16	99	427	135	36				220	206	18
	4	96	112	112	192	33	70	398	107	36				(*Vmatch-2*)		
	8	62	77	77	283	56	55	384	92	38				237	237	16
	16	46	59	59	3599	3970	48	383	85	43						
2	1	953	982	961	29	9	512	1508	571	41	1539	1539	25			
	2	524	592	564	46	10	322	1355	384	42				(*Vmatch-0*)		
	4	310	373	345	219	15	226	1230	291	43				748	559	27
	8	202	263	235	463	24	179	1183	245	45				(*Vmatch-2*)		
	16	149	209	181	393	39	155	1181	222	51				748	727	24
	32	122	182	154	329	70	143	1147	210	64						
3	1	355	442	442	67	13	191	561	268	101	655	655	60			
	2	195	286	286	114	18	120	492	199	101				(*Vmatch-0*)		
	4	115	175	175	500	30	84	457	165	103				321	321	49
	8	75	169	169	869	51	66	440	147	105				(*Vmatch-2*)		
	16	55	116	116	692	77	58	448	139	111				371	371	53
	32	45	107	107	560	132	53	427	134	121						

Table C.3: Test results of MEM-finding between megabase-sized genomes (data sets 3-6 in Table 3.2) for all settings of *essaMEM*. The parameter settings for the different *essaMEM* results can be found in Section 3.4.1. *essaMEM-1* and *essaMEM-2* share the same memory requirements, as well as *essaMEM-4* and *essaMEM-5*. All time results are in seconds and all memory results are in Megabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem).

data		*essaMEM-1*				*essaMEM-2*	*essaMEM-3*				*essaMEM-4*				*essaMEM-5*
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Tmem	Pmem	Emem	time	time
4	1	1378	1526	1504	656	50	1930	2065	2042	120	827	1608	965	3446	141
	2	758	843	820	592	66	1034	1113	1089	167	482	812	551	3139	205
	4	448	504	481	452	86	586	638	616	359	310	471	346	2616	338
	8	293	366	346	461	141	362	433	413	457	224	320	278	2334	570
	16	215	278	257	411	224	250	312	291	370	181	244	224	1857	1067
	32	177	234	213	363	363	194	251	230	351	159	220	199	1527	1527
5	1	1687	1905	1883	600	35	2362	2564	2542	140	1012	1875	1224	3102	116
	2	928	1028	1005	549	49	1266	1358	1335	217	591	998	676	3015	192
	4	548	603	581	499	73	717	768	746	589	380	573	416	2703	335
	8	359	428	407	410	107	443	510	489	645	274	382	325	2280	543
	16	264	322	301	358	207	306	363	343	368	221	281	260	2054	1042
	32	216	270	250	344	344	237	291	270	334	195	256	235	1547	1547
6	1	1687	1912	1877	597	27	2362	2566	2536	148	1012	1875	1224	3400	117
	2	928	1026	1000	549	41	1266	1356	1329	239	591	994	670	3195	195
	4	548	601	575	502	62	717	766	740	659	380	573	410	2814	339
	8	359	435	407	453	120	443	518	489	726	274	382	325	2333	579
	16	264	330	301	402	191	306	371	343	399	221	288	260	2132	1049
	32	216	278	250	351	351	237	298	270	337	195	263	235	1728	1728

Table C.4: Test results of MEM-finding between megabase-sized genomes (data sets 3-6 in Table 3.2). All time results are in seconds and all memory results are in Megabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem). The memory measurements of *essaMEM* are the same as for *sparseMEM*. Note that *MUMmer* does not contain an option to set the index size and *Vmatch* only allows to set the `-qspeedup` parameter.

data		*sparseMEM*				*essaMEM-2*	*backwardMEM*				*MUMmer*			*Vmatch*		
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Pmem	Emem	time	Pmem	Emem	time
4	1	1378	1526	1504	115	50	741	2327	859	269	2304	2304	119	(*Vmatch-0*) 1133	1064	176
	2	758	843	820	186	66	465	2025	590	379						
	4	448	504	481	920	86	327	1953	455	579						
	8	293	366	346	2023	141	258	1823	388	1028				(*Vmatch-2*)		
	16	215	278	257	1744	224	224	1876	354	2057				1199	1199	119
	32	177	234	213	1642	363	207	1805	337	4334						
5	1	1687	1905	1883	136	35	907	2912	1072	242	2816	2816	112	(*Vmatch-0*) 1407	1343	124
	2	928	1028	1005	248	49	569	2582	742	306						
	4	548	603	581	1363	73	401	2338	578	408						
	8	359	428	407	2274	107	316	2274	495	671				(*Vmatch-2*)		
	16	264	322	301	1949	207	274	2294	454	1271				1506	1506	97
	32	216	270	250	1685	344	253	2273	433	2620						
6	1	1687	1912	1877	149	27	907	2912	1066	193	2815	2815	117	(*Vmatch-0*) 1439	1340	133
	2	928	1026	1000	257	41	569	2503	736	208						
	4	548	601	575	1568	62	401	2418	572	233						
	8	359	435	407	2642	120	316	2264	489	302				(*Vmatch-2*)		
	16	264	330	301	1991	191	274	2266	448	412				1503	1503	94
	32	216	278	250	1664	351	253	2194	428	694						

Table C.5: Test results of MEM-finding between gigabase-sized genomes (data sets 7-8 in Table 3.2) for all settings of *essaMEM*. The parameter settings for the different *essaMEM* results can be found in Section 3.4.1. *essaMEM-1* and *essaMEM-2* share the same memory requirements, as well as *essaMEM-4* and *essaMEM-5*. All time results are in hours:minutes:seconds and all memory results are in Gigabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem).

data		*essaMEM-1*				*essaMEM-2*	*essaMEM-3*				*essaMEM-4*				*essaMEM-5*
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Tmem	Pmem	Emem	time	time
	3	10.923	11.726	11.495	5:20:15	0:20:12	14.565	15.271	13.399	2:18:57	7.282	11.482	7.940	38:26:47	1:55:07
	4	8.875	9.625	9.235	5:11:39	0:23:24	11.606	12.160	11.929	4:03:26	6.144	9.249	6.595	36:50:35	2:21:25
	6	6.827	7.283	7.046	4:53:01	0:32:47	8.648	9.054	9.500	7:03:19	5.007	7.033	5.108	34:33:04	3:14:08
7	8	5.803	6.185	4.733	4:43:59	0:40:41	7.169	7.506	7.276	6:59:13	4.438	5.929	4.609	30:02:05	4:01:20
	16	4.267	4.530	4.300	4:12:31	1:09:51	4.950	5.197	4.967	4:19:25	3.584	4.287	3.633	28:59:30	7:02:32
	32	3.499	3.716	3.486	3:52:12	2:01:24	3.840	4.050	3.565	3:50:19	3.158	3.816	3.153	25:26:25	12:51:10
	64	3.115	4.005	3.082	3:27:50	3:27:50	3.286	4.146	3.248	3:28:13	2.944	3.303	2.915	22:12:00	22:12:00
	3	12.549	13.856	13.618	5:55:34	0:32:06	16.732	15.483	15.258	1:41:16	8.366	14.274	9.533	42:03:27	2:17:38
	4	10.196	11.225	10.967	5:52:11	0:35:25	13.333	14.960	14.269	2:41:41	7.059	10.963	7.903	40:26:27	2:49:08
	6	7.843	8.566	8.322	5:35:13	0:46:32	9.934	10.604	10.365	5:01:02	5.751	8.319	6.280	37:55:06	3:49:30
8	8	6.666	7.242	7.003	5:25:16	0:56:10	8.235	8.774	8.667	5:14:46	5.098	7.000	5.120	36:11:45	4:41:02
	16	4.902	5.270	4.668	4:56:23	1:29:08	5.686	6.055	5.798	4:25:17	4.118	5.028	4.266	32:07:16	8:19:38
	32	4.019	4.298	4.049	4:21:26	2:22:49	4.412	4.670	4.432	4:11:22	3.627	4.145	3.666	28:00:25	14:14:21
	64	3.578	4.011	3.558	4:06:09	4:06:09	3.774	4.162	3.750	4:03:03	3.382	4.080	3.367	24:23:39	24:23:39

Table C.6: Test results of MEM-finding between gigabase-sized genomes (data sets 7-8 in Table 3.2) for *sparseMEM* and *essaMEM*. Note that *essaMEM* and *sparseMEM* share the same memory requirements. All time results are in hours:minutes:seconds and all memory results are in Gigabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem).

data		*sparseMEM*				*essaMEM-2*
set	s	Tmem	Pmem	Emem	time	time
	3	10.923	11.726	11.495	4:08:10	0:20:12
	4	8.875	9.625	9.235	11:26:34	0:23:24
	6	6.827	7.283	7.046	32:24:58	0:32:47
7	8	5.803	6.185	4.733	34:32:00	0:40:41
	16	4.267	4.530	4.300	29:02:37	1:09:51
	32	3.499	3.716	3.486	25:25:46	2:01:24
	64	3.115	4.005	3.082	22:12:53	3:27:50
	3	12.549	13.856	13.618	2:51:27	0:32:06
	4	10.196	11.225	10.967	7:41:35	0:35:25
	6	7.843	8.566	8.322	24:04:47	0:46:32
8	8	6.666	7.242	7.003	27:32:00	0:56:10
	16	4.902	5.270	4.668	27:31:21	1:29:08
	32	4.019	4.298	4.049	26:40:03	2:22:49
	64	3.578	4.011	3.558	24:23:54	4:06:09

Table C.7: Test results of MEM-finding between genomes and NGS read data sets (data sets 9-10 in Table 3.2) for all settings of *essaMEM*. The parameter settings for the different *essaMEM* results can be found in Section 3.4.1. *essaMEM-1* and *essaMEM-2* share the same memory requirements, as well as *essaMEM-4* and *essaMEM-5*. All time results are in seconds and all memory results are in Megabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem).

data		*essaMEM-1*				*essaMEM-2*	*essaMEM-3*				*essaMEM-4*				*essaMEM-5*
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Tmem	Pmem	Emem	time	time
9	1	10 469	10 575	10 092	2569	62	14 657	14 668	14 122	463	6282	10 575	6485	17 718	276
	2	5758	5764	5764	2431	93	7852	7812	6665	663	3664	5764	3668	16 518	460
	4	3403	3384	3384	2220	150	4450	4410	4410	1529	2356	3384	2362	14 922	820
	8	2225	2208	2202	2001	252	2748	2715	2715	2406	1701	2208	1691	13 032	1450
	16	1636	1629	1629	1787	416	1898	1885	1885	1765	1374	1629	1374	11 213	2461
	32	1341	1327	1302	1562	835	1472	1455	1455	1545	1211	1327	1208	9369	4895
	64	1194	1176	1176	1341	1341	1260	1240	1035	1353	1129	1175	1112	7656	7656
10	1	1687	1875	1875	780	72	2362	2537	2537	240	1012	1875	1216	3886	176
	2	928	998	989	733	86	1266	1329	1329	319	591	998	668	4040	276
	4	548	573	565	615	119	717	739	739	804	380	571	408	3171	468
	8	359	382	381	555	177	443	465	465	856	274	381	300	2891	774
	16	264	273	273	525	294	306	314	314	502	221	271	232	2371	1317
	32	216	219	219	461	461	237	239	239	430	195	219	219	2154	2154

Table C.8: Test results of MEM-finding between genomes and NGS read data sets (data sets 9-10 in Table 3.2). All time results are in seconds and all memory results are in Megabytes. Memory measurements are expressed as theoretical memory requirements of the index structure (Tmem), peak experimental memory measurements (Pmem) and mean experimental memory measurements (Emem). The memory measurements of *essaMEM* are the same as for *sparseMEM*. Note that *MUMmer* does not contain an option to set the index size and *Vmatch* only allows to set the `-qspeedup` parameter. [§]For data set 9, no results were obtained for *MUMmer* and the 64-bit version of *Vmatch* was used.

data		*sparseMEM*				*essaMEM-2*	*backwardMEM*				*MUMmer*			*Vmatch*		
set	s	Tmem	Pmem	Emem	time	time	Tmem	Pmem	Emem	time	Pmem	Emem	time	Pmem	Emem	time
9	1	10 469	10 575	10 092	536	62	5627	15 510	6232	752	*	*	*	(*Vmatch-0*)		
	2	5758	5764	5764	960	93	3533	15 151	4187	800						
	4	3403	3384	3384	5163	150	2487	13 922	3165	849				12 565	10 999	520
	8	2225	2208	2202	12 773	252	1963	13 513	2654	936				(*Vmatch-2*)		
	16	1636	1629	1629	10 940	416	1701	12 672	2398	1138				12 565	12 016	372
	32	1341	1327	1302	9293	835	1570	13 062	2270	1555						
	64	1194	1176	1176	7654	1341	1505	12 680	2206	2431						
10	1	1687	1875	1875	253	72	907	2406	1064	454	2885	2885	187	(*Vmatch-0*)		
	2	928	998	989	387	86	569	2592	735	657						
	4	548	573	565	1909	119	401	2338	570	1035				1477	1406	198
	8	359	382	381	2848	177	316	2373	488	1848				(*Vmatch-2*)		
	16	264	273	273	2439	294	274	2294	446	3684				1569	1569	167
	32	216	219	219	2066	461	253	2194	426	7886						

Appendix D

Details of the ALFALFA experimental results

This appendix contains additional details on the benchmark method and experimental environment used for the experimental tests performed on *ALFALFA* in Chapter 4. In addition, Section D.2 contains tables with the exact experimental results that are depicted in the figures of Section 4.3.

D.1 Testing environment and experimental measurements

All tests were run on a cluster with dual-socket quad-core Intel Xeon Nehalem (L5520) processors at clock speed 2.27GHz and 12GB RAM/node running Scientific Linux 6.5. This machine was also used for benchmarking *essaMEM* in Chapter 3, but differs in the version of the operating system.

Programs used in testing were compiled locally using gcc v4.4.7, except for *GEM*. All programs were run single-threaded on a full node of the cluster and were the single active process on that node. Due to limitations of the cluster, all tests were limited to a maximum wall time of 72 hours.

The human genome assembly GRCh37, obtained from the UCSC genome browser website, was used as a reference genome for both simulation and mapping processes.

D.1.1 Data sets

ALFALFA achieves a high performance in mapping long sequencing reads that are expected to become commonplace in the near future. As mapping of this type of read data has not yet been benchmarked and sequencing platforms show various error rates, we examined a broad range of different read lengths, error models and error rates using two existing read simulators.

The *wgsim* package is not designed to simulate reads for particular sequencing technologies, but allows to specify a uniform error rate and a relative fraction of indels over mutations in generating simulated reads. In order to evaluate the robustness of read mappers against different error models, we explored error rates ranging from 2% to 10% with either low (10%) or high (90%) indel rates.

The *Mason* read simulator generates reads with errors modeled after the ones produced by Illumina and 454 sequencing technologies. Although these technologies do not (yet) produce reads having lengths as the ones covered in our benchmark study, future sequencing technologies attaining such read lengths could produce errors resembling those of current technologies. The *Mason* simulated read sets also test the robustness of mappers against different error models.

To prove the competitiveness of *ALFALFA* on current moderately sized sequencing reads, artificial data sets were simulated using *Mason* to resemble reads generated by current sequencing platforms. This was done using the same parameter settings as used in the *Bowtie 2* and *RazerS3* benchmarks [141,253]. We also used a read data set of shorter reads, simulated using *wgsim*. In addition, we also included several real read sets in the benchmark study. Note that simulated data allow for an evaluation of the accuracy of read mappers in addition to measuring their speed of execution, whereas real data sets only allow for evaluation of performance.

Real reads

The real Illumina data set included in the benchmark study consists of sequencing run ERR024139 of the ERX009608 experiment (EBI Short Read Archive). It contains over 26 million paired-end 100bp reads with an average insert size of 300bp. This data set was also used in the benchmark study of *CUSHAW2* [163]. The 454 read set that is part of our benchmark has NCBI Short Read Archive accession number SRR003161 and consists of approximately 1.3 million reads with an average length of 574bp. This data set is also part of the benchmark undertaken for several other mappers [141,151,179].

Wgsim simulated reads

A total of 18 read sets of 200 000 reads were simulated with *wgsim* v0.3.1-*r*13 [149] using all possible combinations of 3 read lengths, 3 error rates and 2 indel rates. To simulate different indel models, both the mutation rate and indel fraction were set, but base error rate was set to zero. The following command line and arguments were used to generate the read sets:

```
wgsim -e 0.00 -1 <read length> -N 200000 -R <indel rate>
    <reference.fa> <reads.fq>
```

- read length $= \{1000, 5000, 10\,000\}$
- error rate $= \{0.02, 0.05, 0.1\}$
- indel rate $= \{0.1, 0.9\}$

A final read set of 600bp single-end reads was used to compare the specificity of read mappers and was obtained using the following command line:

```
wgsim -h -R 0.3 -1 600 -N 500000 -r 0.01 -S 11 <reference.fa>
    <reads.fq>
```

Mason simulated reads

Read sets with Illumina and 454 error models were simulated with *Mason* v0.1.1 using either default parameters (100bp and 200bp data sets), parameters used in *Bowtie 2* [141] (longer 454 data sets) or *RazerS3* [253] (longer Illumina data sets). Due to the long execution time and memory requirements of the *Mason* read simulator, some data sets were generated in parts (using different seeds for the random generator) and concatenated afterwards. The following command line arguments were used:

1 million 2×100bp and 2×200bp read pairs following the Illumina error model.

```
mason illumina -N 1000000 -o <output_name> -sq -ll 300 -le 30 -mp
    -rn 1 -n 100 <reference.fa>
mason illumina -N 1000000 -o <output_name> -sq -ll 3000 -le 300 -mp
    -rn 1 -n 200 <reference.fa>
```

1 million 600bp and 800bp reads and 2×600bp and 2×800bp read pairs following Illumina and 454 error models. The simulation was performed in two parts of each 0.5 million reads with seeds 0 and 2 000 000.

```
mason illumina -N 500000 -o <output_name> -s <seed> -sq -hs 0.006
    -hM 32 -n 600 -pi 0.0001 -pd 0.0001 <reference.fa>
mason illumina -N 500000 -o <output_name> -s <seed> -sq -hs 0.006
    -hM 32 -n 800 -pi 0.0001 -pd 0.0001 <reference.fa>
mason illumina -N 500000 -o <output_name> -s <seed> -sq -ll 3000
    -le 300 -mp -rn 1 -hs 0.006 -hM 32 -n 600 -pi 0.0001
    -pd 0.0001 <reference.fa>
mason illumina -N 500000 -o <output_name> -s <seed> -sq -ll 3000
    -le 300 -mp -rn 1 -hs 0.006 -hM 32 -n 800 -pi 0.0001
    -pd 0.0001 <reference.fa>
mason 454 -N 500000 -o <output_name> -s <seed> -sq -hn 2 -nm 600
    -ne 60 -k 0.3 -bm 0.4 -bs 0.2 <reference.fa>
mason 454 -N 500000 -o <output_name> -s <seed> -sq -hn 2 -nm 800
    -ne 80 -k 0.3 -bm 0.4 -bs 0.2 <reference.fa>
mason 454 -N 500000 -o <output_name> -s <seed> -sq -ll 3000
    -le 300 -mp -rn 1 -hn 2 -nm 600 -ne 60 -k 0.3 -bm 0.4 -bs 0.2
    <reference.fa>
mason 454 -N 500000 -o <output_name> -s <seed> -sq -ll 3000
    -le 300 -mp -rn 1 -hn 2 -nm 800 -ne 80 -k 0.3 -bm 0.4 -bs 0.2
    <reference.fa>
```

100 000 1kbp reads following Illumina and 454 error models.

```
mason illumina -N 100000 -o <output_name> -sq -hs 0.006 -hM 32
    -n 1000 -pi 0.0001 -pd 0.0001 <reference.fa>
mason 454 -N 100000 -o <output_name> -sq -hn 2 -nm 1000 -ne 100
    -k 0.3 -bm 0.4 -bs 0.2 <reference.fa>
```

50 000 5kbp reads following Illumina and 454 error models. The simulation was performed in five parts of each 10 000 reads by increasing the seed by 5 million for each part, starting from zero.

```
mason illumina -N 10000 -o <output_name> -s <seed> -sq -hs 0.006
    -hM 32 -n 5000 -pi 0.0001 -pd 0.0001 <reference.fa>
mason 454 -N 10000 -o <output_name> -s <seed> -sq -hn 2 -nm 5000
    -ne 500 -k 0.3 -bm 0.4 -bs 0.2 <reference.fa>
```

10 000 10kbp reads following Illumina and 454 error models. The simulation was performed in ten parts of each 1000 reads by increasing the seed by 10 million for each part, starting from zero.

```
mason illumina -N 1000 -o <output_name> -s <seed> -sq -hs 0.006 -hM
    32 -n 10000 -pi 0.0001 -pd 0.0001 <reference.fa>
mason 454 -N 1000 -o <output_name> -s <seed> -sq -hn 2 -nm 10000
    -ne 1000 -k 0.3 -bm 0.4 -bs 0.2 <reference.fa>
```

D.1.2 Performance and accuracy measurements

Wall times of test runs were measured using the GNU/Linux `time` command. Index construction time was not taken into account, but can be found in Table 4.1. Peak memory measurements in Table 4.1 were obtained using the Python script `ps_mem.py` [31]. Tools for measuring recall rate and accuracy are custom developed for this benchmark study and are available using the *ALFALFA* `evaluate` command. The true positive rate and false positive rate were obtained using the `wgsim_eval.pl` script, which is part of the *wgsim* tool.

For the *accuracy measure*, a read is considered to be mapped correctly if at least one of the reported alignments maps within 10bp of the original simulated position, or if it has at least one alignment with an edit distance that is not higher than that of the simulated read. The *recall rate* only takes the distance to the simulated origin into consideration. Quality values reported by read mappers were used for the ROC curves in Section 4.3.3.

The commands that were used to evaluate accuracy using the *ALFALFA* `evaluate` command require all input files to be sorted by read name, which can be done using *SAMtools* [152].

For the real read data sets, *ALFALFA* `evaluate summary` is used to provide information on the number of mapped reads for which the actual mapping locations are unknown.

```
alfalfa evaluate summary [--paired] -i <results.sam> -o <output>
    --reads <number of reads in the data set>
```

ALFALFA `evaluate sam` is used to evaluate the accuracy for reads simulated by *Mason*. It requires a SAM file containing the original mapping positions as produced by the *Mason* simulator. The reference genome is required to compute the edit distance from the CIGAR string.

```
alfalfa evaluate sam -r <reference.fa> -i <results.sam>
    -o <output> --reference-sam <reference sam.sam>
```

ALFALFA `evaluate wgsim` is used to evaluate the accuracy for reads simulated by *wgsim*. Because *wgsim* stores information about the original location and the number of differences in the name of a read, no additional SAM file is required. The reference genome is required to compute the edit distance from the CIGAR string.

```
alfalfa evaluate wgsim -r <reference.fa> -i <results.sam>
    -o <output>
```

D.1.3 Read mappers

Many long read mappers have been developed over the last couple of years. As it is practically infeasible to evaluate all read mappers in a single benchmark study, we limited ourselves to five state-of-the-art read mappers whose properties resemble those of *ALFALFA*, namely *Bowtie* 2, *BWA-MEM*, *BWA-SW*, *CUSHAW3* and *GEM*. All these mappers are reported to have a high performance in mapping next generation sequencing data, including currently available long reads, but are not specifically designed for a single sequencing technology. They are accurate in the presence of many indels, have a low memory footprint and are freely available. If possible, the tested read mappers were built from source code and their index structures were generated locally. Read mappers were configured with default parameter settings, unless their authors suggested specific settings for certain types of data.

ALFALFA (v0.8) commands

The sparseness of the index structure has to be specified during index construction. The results in this paper use sparseness 12, unless stated otherwise. We also explored the effect of the sparseness in Figure 4.18. Parameter `-a` was used the set the maximum number of reported alignments. Results in Chapter 4 were obtained using `-a 1`. Tables in Section D.2 also contains results for `-a 4`. For the *wgsim* data sets, edit distance was specified as the percentage of differences that were generated using the parameter `-e`. For all remaining data sets, the default edit distance was used. Global alignment was specified for data sets with read length < 500bp, whereas local alignment was specified for longer reads.

Index construction:

```
alfalfa index -r <reference.fa> -p <index_prefix> -s <sparseness>
```

Single-end reads:

```
alfalfa align -i <index_prefix> -0 <reads.fq> -o <output.sam>
     -e <max edit distance> -a <alignment count> [--local]
```

Paired-end reads:

```
alfalfa align -i <index_prefix> -1 <mate1.fq> -2 <mate2.fq> -I
    <min_insert_size> -X <max_insert_size> -o <output.sam> -e <max
    edit distance> -a <alignment count> [--local]
```

BWA-MEM (v0.7.9a) commands

Both *BWA-MEM* and *BWA-SW* use the same index construction command from the *BWA*-suite. The parameter `-a` was used to report multiple alignments and the parameter `-I` was used to set the mean insert size. The default insert size standard deviation was used.

Index construction:

```
bwa index -p <index_prefix> <reference.fa>
```

Single-end reads:

```
bwa mem [-a] <index_prefix> <reads.fq> > <output.sam>
```

Paired-end reads:

```
bwa mem [-a] -I <mean_insert_size> <index_prefix> <mate1.fq>
    <mate2.fq> > <output.sam>
```

BWA-SW (v0.7.9a) commands

Both *BWA-MEM* and *BWA-SW* use the same index construction command from the *BWA*-suite. *BWA-SW* does not allow the user to specify a maximum percentage of errors nor does it allow to specify bounds on the insert size.

Index construction:

```
bwa index -p <index_prefix> <reference.fa>
```

Single-end reads:

```
bwa bwasw -f <output.sam> <index_prefix> <reads.fq>
```

Paired-end reads:

```
bwa bwasw -f <output.sam> <index_prefix> <mate1.fq> <mate2.fq>
```

Bowtie (v2-2.2.3) commands

We mainly used the default parameter settings of *Bowtie* 2. To report multiple alignments, we used parameter `-k 4`. For the results requiring a single alignment per read, we omitted parameter `-k`. Global alignment was specified for data sets with read length < 500bp, whereas local alignment was specified for longer reads. For paired-end reads, we also specified minimum and maximum insert size.

Index construction:

```
Bowtie 2-build <reference.fa> <index_prefix>
```

Single-end reads:

```
Bowtie 2-align -x <index_prefix> -U <reads.fq> -S <output.sam>
    [--bwa-sw-like] [-k 4]
```

Paired-end reads:

```
Bowtie 2-align -x <index_prefix> -1 <mate1.fq> -2 <mate2.fq>
    -S <output.sam> -I <min_insert_size> -X <max_insert_size>
    [--bwa-sw-like] [-k 4]
```

CUSHAW (v3.0.3) commands

We mainly used the default parameter settings of *CUSHAW3*, but used parameter `-multi` to specify the maximum number of alignments to be reported. We specified average insert size and standard deviation for paired-end reads.

Index construction:

```
cushaw3 index -p <index_prefix> <reference.fa>
```

Single-end reads:

```
cushaw3 align -r <index_prefix> -f <reads.fq> -o <output.sam>
    [-multi <alignments>]
```

Paired-end reads:

```
cushaw3 align -r <index_prefix> -q <mate1.fq> <mate2.fq> -o
    <output.sam> -avg_ins <avg_insert_size> -ins_std
    <std_insert_size> [-multi <alignments>]
```

GEM (build 1.376 beta) commands

A pre-built *GEM* index was downloaded from the *GEM* website as the indexer of this mapper ran into a fatal error on our test environment. By default, *GEM* does not report its results into SAM format but provides a tool to convert its output to SAM format. For FASTQ input files, *GEM* requires setting the quality value offset. For real and simulated data generated by the *Mason* read simulator, we set this quality value to `offset-33`. Because *wgsim* does not produce meaningful quality values, we set the quality offset to `ignore` for data sets generated by this read simulator.

Parameter settings were mainly taken from the original *GEM* publication [179], including `--fast-mapping` option. We used parameter `-d` to set the maximum number of reported alignments per read. For paired-end reads, minimum and maximum insert size were set using the definition in the *GEM* paper. These parameter settings deviate from the definition used by other read mappers. For *wgsim* data, error percentages for the `-m` and `-e` options were set to the values used by the simulator (2%, 5% or 10%). For 454 data, we used `-m 0.08 -e 0.08`. For Illumina data we used `-m 0.06 -e 0.06`. Lower values were tried as well for Illumina data sets, but these resulted in a significant drop in accuracy.

Single-end reads:

```
gem-mapper -q <quality-offset> -m <error_percentage>
    -e <error_percentage> --fast-mapping -I <index_name>
    -i <reads.fq> -o <output_prefix>
gem-2-sam -q <quality-offset> --expect-single-end-reads
    -i <output_prefix.map> -o <output.sam>
```

Paired-end reads:

```
gem-mapper -q <quality-offset> -m <error_percentage>
    -e <error_percentage> --fast-mapping -p -E 0.30
    --min-insert-size <min_insert_size>
    --max-insert-size <max_insert_size> -I <index_name>
    -1 <mate1.fq> -2 <mate2.fq> -o <output_prefix>
gem-2-sam -q <quality-offset> --expect-paired-end-reads
    -i <output_prefix.map> -o <output.sam>
```

D.2 Additional tables

The following tables contain accuracy and performance results for several read mappers on an extensive benchmark of long and moderately sized read sets. In addition to the read mappers shown in Section 4.3, results for *ALFALFA* with an index built using sparseness value 4 are also included. The difference in memory requirements between both indexes of *ALFALFA* can be found in Table 4.1.

For most mappers, results for two parameter settings are reported. The first setting triggers read mappers to return 4 alignments per read, or all for *BWA-MEM*. The second setting allows mappers to report only a single alignment per read. As an exception, *BWA-SW* does not provide the option to report multiple alignments.

Wall times for read mapping are reported excluding the time needed to build the index, but including time needed to generate the SAM output file. Some of the read mappers were run on a reduced set of reads due to long runtimes for certain data sets. These cases are reported with the description of the individual data sets.

For all simulated data sets, the accuracy and recall rates are given. The recall rate is measured as the percentage of reads for which an alignment was found within 10bp of the simulated origin. For accuracy, a read is considered to be successfully mapped if an alignment was found within 10bp of the simulated origin or if an alignment was found containing fewer differences than the number of simulated errors.

Table D.1: Data set with 499 998 simulated single-end reads, having a length of 600bp and containing 1% errors.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:06	99.91	99.76	0:06	99.27	98.36
ALFALFA ($s = 12$)	0:14	99.93	99.78	0:14	99.31	98.41
BWA-MEM	0:45	99.95	99.80	0:35	99.33	98.44
BWA-SW				0:50	99.19	98.31
Bowtie 2	2:09	98.20	97.71	1:58	99.07	98.16
CUSHAW3	4:07	99.95	99.79	1:33	99.36	98.44
GEM	1:08	99.99	99.98	1:09	99.90	99.89

Table D.2: Data set with 199 980 simulated single-end reads, having a length of 1kbp and containing 2% errors of which 10% are indels.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:03	99.98	99.88	0:03	99.67	98.70
ALFALFA ($s = 12$)	0:06	99.98	99.87	0:06	99.66	98.67
BWA-MEM	0:20	99.98	99.87	0:17	99.68	98.71
BWA-SW				0:41	99.56	98.68
Bowtie 2	2:30	99.16	98.59	2:13	99.60	98.62
CUSHAW3	5:21	99.98	99.87	1:46	99.69	98.74
GEM	0:04	94.79	94.78	0:04	94.77	94.74

Table D.3: Data set with 199 980 simulated single-end reads, having a length of 1kbp and containing 2% errors of which 90% are indels.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:03	99.88	99.86	0:03	98.99	98.65
ALFALFA ($s = 12$)	0:06	99.90	99.87	0:07	99.00	98.67
BWA-MEM	0:22	99.89	99.86	0:19	99.00	98.67
BWA-SW				0:41	98.95	98.60
Bowtie 2	2:35	99.13	98.72	2:13	99.02	98.66
CUSHAW3	5:23	99.90	99.87	1:43	99.10	98.73
GEM	0:05	68.95	68.94	0:05	68.92	68.81

Table D.4: Data set with 199 980 simulated single-end reads, having a length of 1kbp and containing 5% errors of which 10% are indels.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:05	99.93	99.85	0:04	99.42	98.61
ALFALFA ($s = 12$)	0:08	99.95	99.86	0:07	99.45	98.66
BWA-MEM	0:31	99.96	99.86	0:26	99.48	98.69
BWA-SW				0:32	99.40	98.63
Bowtie 2	2:18	99.13	98.07	2:05	99.38	98.53
CUSHAW3	4:46	99.95	99.86	1:35	99.54	98.69
GEM	0:17	99.27	99.26	0:17	99.21	99.13

Table D.5: Data set with 199 980 simulated single-end reads, having a length of 1kbp and containing 5% errors of which 90% are indels.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:05	99.85	99.85	0:05	98.82	98.67
ALFALFA ($s = 12$)	0:07	99.85	99.84	0:07	98.81	98.66
BWA-MEM	0:30	99.88	99.86	0:25	98.92	98.74
BWA-SW				0:32	98.80	98.51
Bowtie 2	2:21	99.18	98.28	2:07	98.80	98.58
CUSHAW3	4:47	99.85	99.83	1:33	98.88	98.64
GEM	0:17	78.01	77.98	0:17	77.97	77.74

Table D.6: Data set with 199 980 simulated single-end reads, having a length of 1kbp and containing 10% errors of which 10% are indels.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:07	99.47	98.95	0:06	98.76	97.75
ALFALFA ($s = 12$)	0:10	99.81	99.67	0:10	99.11	98.44
BWA-MEM	0:36	99.92	99.84	0:30	99.31	98.64
BWA-SW				0:23	99.16	98.39
Bowtie 2	1:59	99.09	96.99	1:53	98.98	98.16
CUSHAW3	2:43	98.42	98.32	1:23	97.88	97.15
GEM	4:24	99.97	99.94	4:23	99.88	99.71

Table D.7: Data set with 199 980 simulated single-end reads, having a length of 1kbp and containing 10% errors of which 90% are indels.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:07	99.53	99.28	0:07	98.34	98.03
ALFALFA ($s = 12$)	0:10	99.72	99.67	0:10	98.59	98.46
BWA-MEM	0:32	99.83	99.75	0:27	98.81	98.55
BWA-SW				0:25	97.83	97.01
Bowtie 2	2:06	99.14	97.56	1:56	98.59	98.28
CUSHAW3	2:51	95.52	95.37	1:22	94.51	94.16
GEM	4:01	91.60	91.50	3:59	91.53	91.06

Table D.8: Data set with 199 976 simulated single-end reads, having a length of 5kbp and containing 2% errors of which 10% are indels. [§]Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:15	99.99	99.98	0:15	99.61	99.36
ALFALFA ($s = 12$)	0:20	99.99	99.98	0:20	99.63	99.39
BWA-MEM	1:40	99.99	99.98	1:36	99.66	99.43
BWA-SW				2:30	99.61	99.37
Bowtie 2[§]	2:28	99.90	99.90	1:01	99.90	99.85
CUSHAW3[§]	5:44	100.00	100.00	1:08	99.90	99.85
GEM	2:06	99.95	99.95	2:06	99.93	99.89

Table D.9: Data set with 199 976 simulated single-end reads, having a length of 5kbp and containing 2% errors of which 90% are indels. §Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:15	99.98	99.98	0:15	99.39	99.37
ALFALFA ($s = 12$)	0:20	99.98	99.98	0:19	99.40	99.38
BWA-MEM	1:40	99.98	99.98	1:36	99.44	99.42
BWA-SW				2:25	99.36	99.33
Bowtie 2§	2:43	99.90	99.85	0:58	99.80	99.80
CUSHAW3§	5:40	100.00	100.00	1:04	99.80	99.80
GEM	3:04	79.10	79.10	3:05	79.08	78.97

Table D.10: Data set with 199 976 simulated single-end reads, having a length of 5kbp and containing 5% errors of which 10% are indels. §Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:25	99.98	99.98	0:24	99.48	99.35
ALFALFA ($s = 12$)	0:36	99.98	99.98	0:35	99.48	99.37
BWA-MEM	2:08	99.99	99.98	2:03	99.52	99.39
BWA-SW				2:03	99.48	99.35
Bowtie 2§	2:19	99.90	99.90	0:52	99.90	99.85
CUSHAW3§	5:11	100.00	100.00	1:01	99.90	99.85
GEM	9:57	99.99	99.99	9:52	99.95	99.88

Table D.11: Data set with 199 976 simulated single-end reads, having a length of 5kbp and containing 5% errors of which 90% are indels. §Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:24	99.98	99.98	0:24	99.38	99.38
ALFALFA ($s = 12$)	0:34	99.98	99.98	0:33	99.38	99.37
BWA-MEM	2:05	99.98	99.98	1:57	99.39	99.38
BWA-SW				1:58	99.27	99.23
Bowtie 2§	2:18	99.20	99.20	0:54	99.20	99.15
CUSHAW3§	5:10	100.00	100.00	0:52	100.00	100.00
GEM	10:37	93.72	93.70	10:42	93.69	93.46

Table D.12: Data set with 199 976 simulated single-end reads, having a length of 5kbp and containing 10% errors of which 10% are indels. §Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:29	99.95	99.92	0:29	99.43	99.35
ALFALFA ($s = 12$)	0:42	99.97	99.97	0:42	99.47	99.42
BWA-MEM	2:20	99.98	99.98	2:14	99.51	99.42
BWA-SW				1:42	99.43	99.33
Bowtie 2§	2:10	99.95	99.95	0:45	99.70	99.70
CUSHAW3§	2:00	100.00	100.00	0:30	99.75	99.75
GEM§	3:07	100.00	100.00	3:07	99.95	99.90

Table D.13: Data set with 199 976 simulated single-end reads, having a length of 5kbp and containing 10% errors of which 90% are indels. §Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:30	99.90	99.89	0:31	99.23	99.22
ALFALFA ($s = 12$)	0:42	99.91	99.91	0:41	99.28	99.27
BWA-MEM	2:15	99.91	99.91	2:07	99.30	99.28
BWA-SW				1:47	98.53	98.03
Bowtie 2§	2:00	94.25	94.25	0:48	94.10	94.10
CUSHAW3§	2:02	99.20	99.20	0:28	99.05	99.05
GEM§	2:43	99.60	99.45	2:42	99.55	99.25

Table D.14: Data set with 199 922 simulated single-end reads, having a length of 10kbp and containing 2% errors of which 10% are indels. §Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:32	99.99	99.99	0:32	99.64	99.57
ALFALFA ($s = 12$)	0:36	100.00	99.99	0:35	99.64	99.57
BWA-MEM	3:19	99.99	99.99	3:11	99.71	99.65
BWA-SW				5:34	99.69	99.61
Bowtie 2§	12:17	99.90	99.90	5:00	99.95	99.95
CUSHAW3§	43:51	100.00	100.00	7:55	99.95	99.95
GEM	12:19	99.99	99.99	12:27	99.97	99.93

Table D.15: Data set with 199 922 simulated single-end reads, having a length of 10kbp and containing 2% errors of which 90% are indels. [§]Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:33	99.99	99.99	0:33	99.59	99.59
ALFALFA ($s = 12$)	0:36	99.99	99.99	0:36	99.60	99.59
BWA-MEM	3:13	99.99	99.99	3:15	99.63	99.63
BWA-SW				5:04	99.56	99.54
Bowtie 2[§]	12:15	99.70	99.70	5:18	99.65	99.65
CUSHAW3[§]	43:33	100.00	100.00	7:37	99.85	99.85
GEM	18:23	86.24	86.24	18:36	86.23	86.12

Table D.16: Data set with 199 922 simulated single-end reads, having a length of 10kbp and containing 5% errors of which 10% are indels. [§]Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:45	99.99	99.99	0:45	99.62	99.59
ALFALFA ($s = 12$)	0:54	99.99	99.99	0:54	99.63	99.60
BWA-MEM	4:15	99.99	99.99	4:01	99.66	99.62
BWA-SW				4:21	99.63	99.59
Bowtie 2[§]	10:57	99.95	99.95	4:12	100.00	100.00
CUSHAW3[§]	39:43	100.00	100.00	6:57	99.95	99.95
GEM	64:15	99.99	99.98	62:43	99.95	99.87

Table D.17: Data set with 199 922 simulated single-end reads, having a length of 10kbp and containing 5% errors of which 90% are indels. [§]Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:49	99.98	99.98	0:48	99.54	99.54
ALFALFA ($s = 12$)	0:56	99.98	99.98	0:56	99.54	99.54
BWA-MEM	4:05	99.99	99.99	4:02	99.60	99.60
BWA-SW				3:55	99.24	99.09
Bowtie 2[§]	12:00	96.60	96.60	5:08	96.55	96.55
CUSHAW3[§]	40:04	100.00	100.00	6:45	99.95	99.95
GEM[§]	0:39	98.10	98.10	0:39	98.10	98.10

Table D.18: Data set with 199 922 simulated single-end reads, having a length of 10kbp and containing 10% errors of which 10% are indels. [§]Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	1:03	99.96	99.87	1:01	99.55	99.46
ALFALFA ($s = 12$)	1:09	99.98	99.97	1:07	99.59	99.57
BWA-MEM	4:29	99.99	99.99	4:27	99.64	99.61
BWA-SW				3:30	99.60	99.56
Bowtie 2[§]	10:26	99.85	99.85	3:43	99.85	99.85
CUSHAW3[§]	15:11	100.00	100.00	3:08	99.95	99.90
GEM[§]	25:22	100.00	100.00	23:56	100.00	99.95

Table D.19: Data set with 199 922 simulated single-end reads, having a length of 10kbp and containing 10% errors of which 90% are indels. [§]Measurements restricted to the first 2000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	1:10	99.91	99.88	1:08	99.44	99.41
ALFALFA ($s = 12$)	1:14	99.93	99.92	1:10	99.48	99.47
BWA-MEM	4:30	99.92	99.92	4:22	99.50	99.50
BWA-SW				3:43	97.43	96.02
Bowtie 2[§]	9:50	90.60	90.60	3:59	90.60	90.60
CUSHAW3[§]	15:49	99.80	99.80	3:02	99.75	99.75
GEM[§]	21:48	99.95	99.85	21:39	99.95	99.85

Table D.20: Data set with 2×1 million simulated paired-end reads, having a length of 100bp and generated with Illumina error model. Insert size is 300bp with a standard deviation of 10%.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:23	99.53	98.30	0:21	99.48	97.02
ALFALFA ($s = 12$)	0:14	99.56	98.51	0:14	99.51	97.23
BWA-MEM	0:10	99.95	97.88	0:09	99.95	97.88
BWA-SW				0:27	99.91	95.95
Bowtie 2	0:27	99.85	98.90	0:21	99.89	97.75
CUSHAW3	0:41	99.95	99.40	0:38	99.92	97.76
GEM	0:08	99.92	99.73	0:08	99.92	99.73

Table D.21: Data set with 2 × 1 million simulated paired-end reads, having a length of 200bp and generated with Illumina error model. Insert size is 3kbp with a standard deviation of 10%.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:11	99.89	99.52	0:10	99.80	98.51
ALFALFA ($s = 12$)	0:21	99.89	99.56	0:20	99.80	98.57
BWA-MEM	0:18	99.88	98.57	0:19	99.88	98.57
BWA-SW				1:17	99.83	97.58
Bowtie 2	3:10	99.80	99.33	2:06	99.85	98.49
CUSHAW3	1:35	99.98	99.76	1:05	99.89	98.54
GEM	0:15	99.99	99.97	0:15	99.99	99.97

Table D.22: Data set with 1 million simulated single-end reads, having a length of 600bp and generated with Illumina error model.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:14	98.97	98.96	0:13	97.82	97.77
ALFALFA ($s = 12$)	0:30	98.99	98.97	0:31	97.83	97.78
BWA-MEM	0:51	99.05	99.04	0:41	97.94	97.90
BWA-SW				2:12	98.47	98.28
Bowtie 2	4:47	98.46	98.28	4:17	98.26	98.20
CUSHAW3	7:26	99.46	99.42	3:20	98.35	98.27
GEM	1:09	97.21	97.21	1:08	97.11	97.11

Table D.23: Data set with 2 × 1 million simulated paired-end reads, having a length of 600bp and generated with Illumina error model. Insert size is 3kbp with a standard deviation of 10%.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:26	99.84	99.83	0:25	99.17	99.13
ALFALFA ($s = 12$)	0:57	99.85	99.83	0:59	99.18	99.14
BWA-MEM	1:21	99.08	99.03	1:21	99.08	99.03
BWA-SW				4:53	98.20	96.56
Bowtie 2	23:01	99.07	99.00	15:29	99.03	98.97
CUSHAW3	37:33	99.91	99.90	6:56	99.10	99.04
GEM	1:16	99.92	99.92	1:16	99.92	99.92

Table D.24: Data set with 1 million simulated single-end reads, having a length of 800bp and generated with Illumina error model.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:28	99.06	99.06	0:28	98.02	98.00
ALFALFA ($s = 12$)	0:43	99.08	99.08	0:45	98.03	98.02
BWA-MEM	1:14	99.01	99.00	0:58	97.97	97.95
BWA-SW				2:48	98.53	98.42
Bowtie 2	8:47	98.57	98.40	7:54	98.36	98.35
CUSHAW3	13:31	99.48	99.47	5:30	98.45	98.42
GEM	1:53	98.17	98.17	1:51	98.06	98.06

Table D.25: Data set with 2×1 million simulated paired-end reads, having a length of 800bp and generated with Illumina error model. Insert size is 3kbp with a standard deviation of 10%.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:55	99.89	99.89	0:57	99.25	99.24
ALFALFA ($s = 12$)	1:30	99.90	99.90	1:30	99.26	99.25
BWA-MEM	1:55	99.14	99.12	1:55	99.14	99.12
BWA-SW				6:16	97.83	96.39
Bowtie 2	40:10	99.01	98.94	26:42	99.09	99.07
CUSHAW3	71:50	99.93	99.93	11:25	99.16	99.14
GEM	2:02	99.97	99.97	2:03	99.97	99.97

Table D.26: Data set with 100 000 simulated single-end reads, having a length of 1kbp and generated with Illumina error model.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:03	99.09	99.09	0:03	98.13	98.13
ALFALFA ($s = 12$)	0:06	99.11	99.11	0:05	98.11	98.11
BWA-MEM	0:09	99.01	99.01	0:07	97.99	97.99
BWA-SW				0:20	98.53	98.43
Bowtie 2	1:28	98.61	98.44	1:14	98.41	98.40
CUSHAW3	2:28	99.44	99.43	0:53	98.47	98.46
GEM	0:18	98.71	98.71	0:17	98.56	98.56

Table D.27: Data set with 50 000 simulated single-end reads, having a length of 5kbp and generated with Illumina error model. §Measurements for *single alignment* restricted to the first 5000 reads. †Measurements for *multiple alignments* restricted to the first 5000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:09	99.10	99.10	0:09	98.63	98.63
ALFALFA ($s = 12$)	0:15	99.12	99.12	0:15	98.65	98.65
BWA-MEM	0:25	99.12	99.12	0:24	98.66	98.66
BWA-SW				0:46	96.85	96.56
Bowtie 2§	20:05	98.93	98.47	3:06	81.60	81.60
CUSHAW3†	13:03	99.80	99.80	27:18	99.09	99.09
GEM	7:32	99.88	99.88	7:32	99.78	99.78

Table D.28: Data set with 10 000 simulated single-end reads, having a length of 10kbp and generated with Illumina error model. §Measurements for *single alignment* restricted to the first 1000 reads. †Measurements for *multiple alignments* restricted to the first 1000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:03	98.97	98.97	0:03	98.62	98.62
ALFALFA ($s = 12$)	0:04	98.98	98.98	0:04	98.69	98.69
BWA-MEM	0:12	98.98	98.98	0:11	98.70	98.70
BWA-SW				0:23	94.63	93.70
Bowtie 2§	21:21	99.23	98.47	3:31	57.5	57.5
CUSHAW3†	19:43	99.95	99.95	40:23	99.16	99.16
GEM	14:56	99.86	99.86	12:51	99.73	99.73

Table D.29: Data set with 1 million simulated single-end reads, having an average length of 600bp and generated with the 454 error model.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:15	99.64	99.54	0:15	98.73	98.36
ALFALFA ($s = 12$)	0:26	99.75	99.69	0:25	98.86	98.54
BWA-MEM	1:44	99.84	99.77	1:21	99.02	98.59
BWA-SW				1:26	98.63	97.87
Bowtie 2	4:08	98.39	97.43	3:51	98.71	98.28
CUSHAW3	6:46	99.71	99.63	2:51	99.00	98.45
GEM	2:19	99.93	99.91	2:16	99.88	99.76

Table D.30: Data set with 2×1 million simulated paired-end reads, having an average length of 600bp and generated with the 454 error model. Insert size is 3kbp with a standard deviation of 10%.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:30	99.64	99.39	0:28	99.18	98.65
ALFALFA ($s = 12$)	0:51	99.81	99.70	0:51	99.38	99.00
BWA-MEM	2:44	99.53	99.02	2:46	99.53	99.02
BWA-SW				3:22	99.32	98.07
Bowtie 2	21:35	99.22	98.55	14:50	99.36	98.81
CUSHAW3	19:29	99.53	98.94	9:24	99.48	98.73
GEM	2:05	99.99	99.99	2:23	99.99	99.99

Table D.31: Data set with 1 million simulated single-end reads, having an average length of 800bp and generated with the 454 error model.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:21	99.77	99.74	0:21	98.86	98.65
ALFALFA ($s = 12$)	0:37	99.83	99.80	0:38	98.93	98.72
BWA-MEM	2:13	99.87	99.83	1:54	99.03	98.74
BWA-SW				1:53	98.83	98.28
Bowtie 2	7:50	98.64	97.68	7:20	98.78	98.48
CUSHAW3	10:27	99.69	99.64	4:40	98.91	98.52
GEM	4:08	99.97	99.96	4:06	99.92	99.81

Table D.32: Data set with 2×1 million simulated paired-end reads, having an average length of 800bp and generated with the 454 error model. Insert size is 3kbp with a standard deviation of 10%.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:41	99.79	99.70	0:40	99.32	99.05
ALFALFA ($s = 12$)	1:16	99.88	99.83	1:15	99.43	99.19
BWA-MEM	3:37	99.48	99.11	3:35	99.48	99.11
BWA-SW				4:27	99.20	98.29
Bowtie 2	36:40	99.24	98.50	26:27	99.32	98.94
CUSHAW3	26:05	99.41	98.96	13:26	99.35	98.80
GEM	4:16	100.00	99.99	4:17	100.00	99.99

Table D.33: Data set with 100 000 simulated single-end reads, having an average length of 1kbp and generated with the 454 error model.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:03	99.81	99.78	0:03	98.96	98.81
ALFALFA ($s = 12$)	0:05	99.86	99.84	0:05	99.03	98.86
BWA-MEM	0:16	99.89	99.87	0:14	99.10	98.89
BWA-SW				0:14	98.94	98.56
Bowtie 2	1:20	98.86	97.88	1:14	98.89	98.66
CUSHAW3	1:52	99.68	99.64	0:44	98.96	98.67
GEM	0:39	99.98	99.96	0:39	99.92	99.81

Table D.34: Data set with 50 000 simulated single-end reads, having an average length of 5kbp and generated with the 454 error model. [§]Measurements for *single alignment* restricted to the first 5000 reads. [†]Measurements for *multiple alignments* restricted to the first 5000 reads.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:08	99.97	99.97	0:08	99.46	99.46
ALFALFA ($s = 12$)	0:10	99.97	99.97	0:10	99.43	99.43
BWA-MEM	0:42	99.98	99.98	0:38	99.45	99.45
BWA-SW				0:44	99.24	96.12
Bowtie 2[§]	28:08	99.72	98.00	2:53	99.30	99.30
CUSHAW3[†]	8:20	99.5	99.5	19:35	98.97	98.83
GEM	19:35	100.00	100.00	19:31	99.93	99.88

Table D.35: Data set with 10 000 simulated single-end reads, having an average length of 10kbp and generated with the 454 error model. [§]Measurements for *single alignment* restricted to the first 1000 reads. [†]Fatal error encountered during mapping process.

	multiple alignments			single alignment		
mapper	runtime	accuracy	recall	runtime	accuracy	recall
ALFALFA ($s = 4$)	0:04	99.46	98.45	0:04	99.23	98.22
ALFALFA ($s = 12$)	0:03	99.89	99.79	0:04	99.60	99.50
BWA-MEM	0:44	99.96	99.96	0:42	99.60	99.60
BWA-SW				0:19	99.98	94.81
Bowtie 2[§]	35:23	99.95	97.92	3:25	99.30	99.30
CUSHAW3[†]	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
GEM	33:27	100.00	100.00	33:13	99.94	99.88

Appendix E

ALFALFA command line structure

This section provides details on the command line options that can be used to tweak *ALFALFA*, default settings and general usage tips. The software package offers separate commands for index construction, read mapping and evaluating mapping accuracy. Index construction can also be combined with read mapping during a single run of the package. The `alfalfa` command has the following general anatomy

```
alfalfa <command> [<subcommand>] [options]
```

where `command` is either `index`, `align` or `evaluate`. Only the `evaluate` command requires an additional subcommand. The command line atonomy of *ALFALFA* is graphically represented in Figure E.1.

In what follows we describe the options associated with each of the commands. Options can have a single-letter (preceded by a single hyphen) or multi-letter (preceded by a double hyphen) name, or both. In the latter case, both names of the option can be used interchangeably. Each description of an option starts with its name or names (separated by a forward slash), followed by a tuple (between round brackets) indicating the data type and default value of the argument that has to be passed to the option. If no default value is given, it is mandatory to pass an argument to the option. Options for which no tuple is given are used as toggles to enable/disable certain features and need no extra argument.

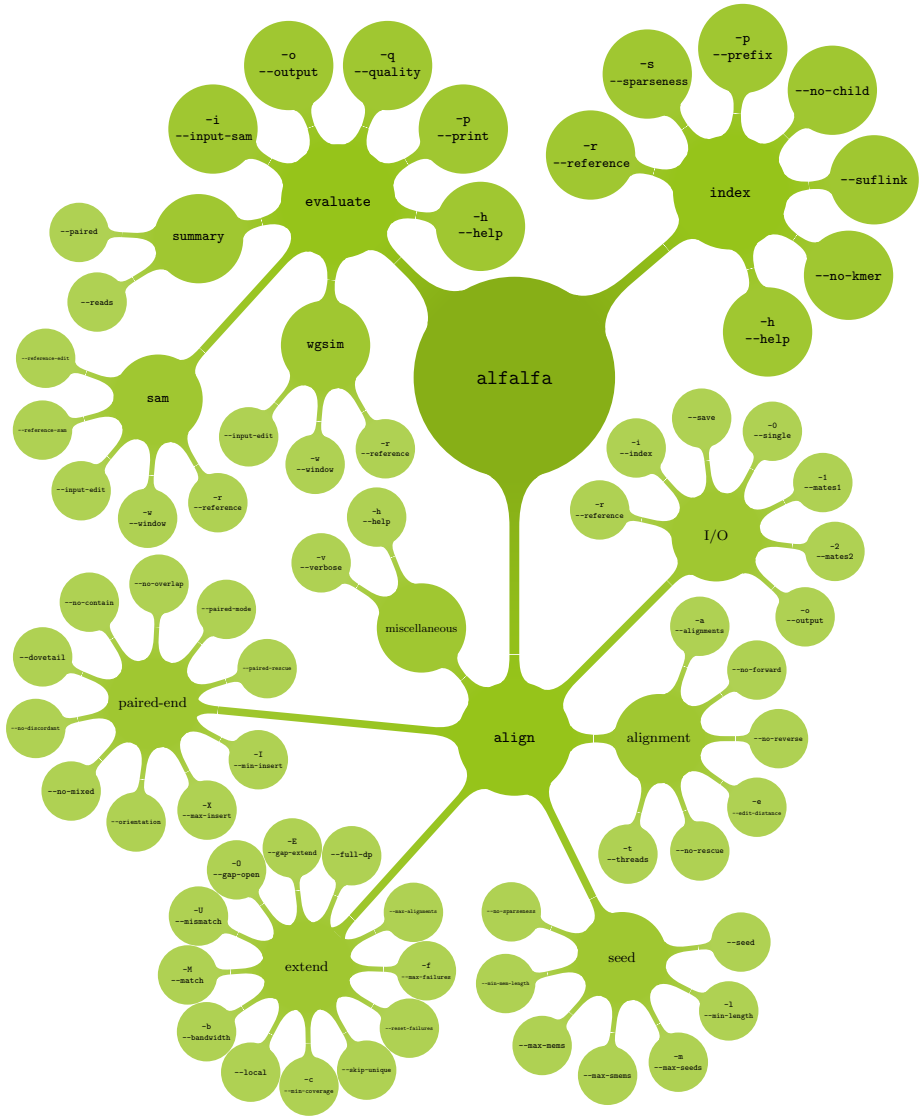


Figure E.1: Command line anatomy of *ALFALFA*. Options can be used to customize the read mapper and are grouped per command, subcommand and category. See Supplemental Data for further details on the options used by *ALFALFA*.

E.1 Indexing a reference genome

The `index` command is used to construct the data structures for indexing a given reference genome. The constructed index is stored to disk over multiple files that are contained in the same directory and that have names sharing the same prefix. Index files contain bookkeeping information (extension `.aux`) and individual arrays of an enhanced sparse suffix array: reference genome (extension `.ref`), suffix array (extension `.sa`), longest common prefix array (extension `.lcp`), inverse suffix array (extension `.isa`; optional), child array (extension `.child`; optional) and 10-mer lookup array (extension `.kmer`; optional). The inverse suffix array is the only array that is not constructed by default. The child array and 10-mer lookup array are not strictly necessary and may be omitted in order to save memory, but at the cost of a drop in performance. The `index` command can be skipped as the `align` command also provides the option to generate an index and store it to disk. All options for customizing the `index` command can therefore also be used in combination with the `align` command.

E.1.1 Options

`-r/--reference (file).` Specifies the location of a file that contains the reference genome in multi-FASTA format.

`-s/--sparseness (int, 12).` Specifies the sparseness of the index structure as a way to control part of the speed-memory trade-off.

`-p/--prefix (string, filename passed to the -r option).` Specifies the prefix that will be used to name all generated index files. The same prefix has to be passed to the `-i` option of the `align` command to load the index structure when mapping reads.

`--no-child.` By default, a sparse child array is constructed and stored in an index file with extension `.child`. The construction of this sparse child array is skipped when the `--no-child` option is set. This data structure speeds up seed-finding at the cost of (4/s) bytes per base in the reference genome. As the data structure provides a major speed-up, it is advised to have it constructed.

`--suflink.` Suffix link support is disabled by default. Suffix link support is enabled when the `--suflink` option is set, resulting in an index file with extension `.isa` to be generated. This data structure speeds up seed-finding

at the cost of $4/s$ bytes per base. It is only useful when sparseness is less than four and minimum seed length is very low (< 10), because it conflicts with skipping suffixes in matching the read. In practice, this is rarely the case.

`--no-kmer`. By default, a 10-mer lookup table is constructed that contains the suffix array interval positions to depth 10 in the virtual suffix tree. It is stored in an index file with extension `.kmer` and requires only 8MB of memory. The construction of this lookup table is skipped when the `--no-kmer` option is set. The lookup table stores intervals for sequences of length 10 that only contain {`A,C,G,T`}. This data structure speeds up seed-finding if the minimum seed length is greater than 10.

`-h/--help`. Prints to standard error the version number, usage description and an overview of the options that can be used to customize the software package.

E.2 Mapping and aligning a read set

The `align` command is used for mapping and aligning a read set onto a reference genome. As this process can be customized through a long list of options, we have grouped them into several categories.

E.2.1 I/O options

`-r/--reference (file, part of the index)`. Specifies the location of a file that contains the reference genome in multi-FASTA format.

`-i/--index (string)`. Specifies the prefix used to name all generated index files. If this option is not set explicitly, an index will be computed from the reference genome according to the settings of the options that also apply to the `index` command.

`--save`. Specifies that if an index is constructed by the `align` command itself, it will be stored to disk. This option is ignored if the index is loaded from disk (option `-i`).

`-0/--single (file)`. Specifies the location of a file that contains single-end reads. Both FASTA and FASTQ formats are accepted. If both single-end and paired-end reads are specified, single-end reads are processed first.

`-1/--mates1` (`file`). Specifies the location of a file that contains the first mates of paired-end reads. Both FASTA and FASTQ formats are accepted.

`-2/--mates2` (`file`). Specifies the location of a file that contains the second mates of paired-end reads. Both FASTA and FASTQ formats are accepted.

`-o/--output` (`file`, **filename passed to the** `-r` **option with additional** `.sam` **extension**). Specifies the location of the generated SAM output file containing the results of read mapping and alignment.

E.2.2 Alignment options

`-a/--alignments` (`int`, `1`). Specifies the maximum number of alignments reported per read.

`--no-forward`. Do not compute alignments on the forward strand.

`--no-reverse`. Do not compute alignments on the reverse complement strand.

`-e/--edit-distance` (`float`, `0.08`). Represents the maximum percentage of differences allowed in accepting alignments and used in combination with the dynamic programming score function to calculate the minimum alignment score.

`--no-rescue`. Disables rescue procedures that are normally initiated when no seeds and/or alignments are found with the current parameters.

`-t/--threads` (`int`, `1`). Number of threads used during read mapping. Using more than one thread results in reporting read alignments in a different order compared to the order in which they are read from the input file(s).

E.2.3 Seed options

`--seed` (`MEM` | `SMEM` | `PSMEM`, `SMEM`). Specifies the type of seeds used for read mapping. Possible values are `MEM` for maximal exact matches, `SMEM` for super-maximal exact matches, and `PSMEM` for SMEMs with additional rare MEMs. The use of SMEMs generally boosts performance without having a negative impact on accuracy compared to the use of MEMs. On the other hand, there are usually many more MEMs than SMEMs, in general resulting in a higher number of candidate genomic regions. Reporting all MEMs might be useful if reporting more candidate mapping locations is preferred.

`-l/--min-length (int, auto).` Specifies the minimum seed length. This value must be greater than the sparseness value used to build the index (option `-s`). By default, the value of this option is computed automatically using the following procedure. A value of 40 is used for reads shorter than 1kbp. The value is incremented by 20 for every 500bp above 1kbp, with the total increment being divided by the maximum percentage of errors allowed in accepting alignments (option `-e`).

`-m/--max-seeds (int,` 10 000`).` Specifies the maximum number of same-length seeds that will be selected per offset in the read sequence. The value passed to this option is multiplied by the automatically computed skip factor that determines sparse matching of sampled suffixes from the read sequence. As a result, the actual number of seeds per starting position in the read might still vary. Higher values of this option result in higher numbers of seeds, increasing in turn the number of candidate genomic regions.

`--max-smems (int, 10).` Specifies the maximum number of SMEMs per offset in the read sequence to allow MEM-finding. This only applies to PSMEM seeds.

`--max-mems (int, 20).` Specifies the maximum number of MEMs per offset in the read that can be used for candidate region identification.

`--min-mem-length (int, 50).` Specifies the minimum length MEMs need to have to be used for candidate region identification.

`--no-sparseness.` Disables the use of sparseness in the read sequence during seed-finding.

E.2.4 Extend options

`--max-alignments (int, 5000).` Specifies the maximum number of alignments calculated per read. This value should be higher than the number of reported alignments (option `-a`). Decreasing this value can increase performance of the algorithm, at the cost of a lower accuracy and worse mapping quality estimation.

`-f/--max-failures (int, 10).` Specifies the maximum number of successive candidate regions that are investigated without success before *ALFALFA* stops extending the candidate regions of a read. Extension can be restarted only if the remaining candidate regions contain unique seeds.

`--reset-failures`. If set, the counter of successive candidate regions that are investigated without success is reset if a feasible alignment is found. By default the counter is only reset if a new best alignment is found.

`--skip-unique`. By default, *ALFALFA* extends all candidate regions containing unique seeds. If this flag is set, the uniqueness criterion is not taken into account when deciding upon the extension of a candidate region.

`-c/--min-coverage (float, 0.25)`. Specifies the minimum percentage of the read length that candidate regions containing a single seed need to cover before extension of the candidate region is taken into consideration.

`--local`. By default, *ALFALFA* uses global alignment during the last phase of the mapping process. Global alignment in essence is end-to-end alignment, as it entirely covers the read but only covers the reference genome in part. Local alignment is used during the last phase of the mapping process if the `--local` option is set, which may result in soft clipping of the read.

`-b/--bandwidth (int, 100)`. Specifies the maximum bandwidth that is used by the banded alignment algorithm. The bandwidth used is automatically inferred from the specification of the maximum percentage of errors allowed in accepting alignments (option `-e`), but is bound by this parameter.

`-M/--match (int, 1)`. Specifies the positive score assigned to matches in the dynamic programming extension phase.

`-U/--mismatch (int, -4)`. Specifies the penalty assigned to mismatches in the dynamic programming extension phase.

`-O/--gap-open (int, -6)`. Specifies the penalty O for opening a gap (insertion or deletion) in the dynamic programming extension phase. The total penalty for a gap of length ℓ equals $O + \ell \cdot E$. The use of affine gap penalties can be disabled by setting this value to zero.

`-E/--gap-extend (int, -1)`. Specifies the penalty E for extending a gap (insertion or deletion) in the dynamic programming extension phase. The total penalty for a gap of length ℓ equals $O + \ell \cdot E$.

`--full-dp`. By default, *ALFALFA* uses chain-guided alignment to retrieve the CIGAR alignment. If this parameter is set, banded dynamic programming is performed instead. Activating this setting can greatly increase runtime, but can sometimes lead to more optimal alignments.

E.2.5 Paired-end mapping options

`-I/--min-insert (int, 0).` Specifies the minimum insert size.

`-X/--max-insert (int, 1000).` Specifies the maximum insert size.

`--orientation (fr | rf | ff, fr).` Specifies the orientation of mates. `fr` means a forward upstream first mate and reverse complemented downstream second mate or vice versa. `rf` means a reverse complemented upstream first mate and forward downstream second mate or vice versa. `ff` means a forward upstream first mate and forward downstream second mate or vice versa. Note that these definitions are literally taken over from Bowtie 2.

`--no-mixed.` Disables searching for unpaired alignments.

`--no-discordant.` Disables searching for discordant alignments.

`--dovetail.` Allows switching between upstream and downstream mates in the definition of their orientation (option `--orientation`).

`--no-contain.` Disallows concordant mates to be fully contained within each other.

`--no-overlap.` Disallows concordant mates to overlap each other.

`--paired-mode (1 | 2 | 3 | 4 | 5 | 6, 1).` Specifies the algorithm used to align paired-end reads. The possible algorithms are discussed in detail in the methods section. Algorithms `1` and `2` do not use information from candidate regions. Algorithms `3` and `4` prioritize extension of candidate regions over both reads. Algorithms `5` and `6` filter the list of candidate regions using the paired-end constraints. Algorithms with an odd number pair mapped reads after alignment. Algorithms with an even number perform dynamic programming across a window defined by the insert size restrictions to search for a bridging alignment reaching the other mate.

`--paired-rescue.` Enables a rescue procedure if no concordant alignment is found using the current parameter settings.

E.2.6 Miscellaneous options

`-v/--verbose (int, 0).` Turns on lots of progress reporting about the alignment process. Higher numbers give more verbose output. Information is printed

to standard error and is useful for debugging purposes. The default value `0` disables progress reporting. The maximum verbosity level currently supported is `7`.

`-h/--help`**.** Prints to standard error the version number, usage description and an overview of the options that can be used to customize the software package.

E.3 Evaluating mapping accuracy

The `evaluate` command is used for evaluating the accuracy of simulated reads and summarizing statistics from the SAM formatted alignments reported by a read mapper. It requires an additional subcommand that influences both the functionality and the input of the `evaluate` command. Currently supported subcommands are `summary`, `sam` and `wgsim`.

The `evaluate` command requires all input files to be sorted by read name. This can easily be done using SAMtools. Furthermore, read names of both mates should be identical for paired-end reads.

E.3.1 Options shared by all subcommands

`-i/--input-sam` **(`file`).** Specifies the location of a SAM file that contains the read mapping alignments that need to be evaluated.

`-o/--output` **(`file`, standard output).** Specifies the location of the file that will contain the generated output.

`-q/--quality` **(comma-separated list of `ints` between `0` and `255`, `0`).** The values in the list represent quality thresholds. For each specified quality threshold, output is produced that reports only on the subset of alignments with quality value greater than or equal to the threshold.

`-p/--print`**.** Triggers the generated output to contain a list of all reads from the input SAM file followed by a binary value. Zero indicates that the read is either unmapped or incorrectly mapped and one indicates that the read was mapped (`summary` subcommand) or mapped correctly (other subcommands).

`-h/--help`. Prints to standard error the version number, usage description and an overview of the options that can be used to customize the software package.

E.3.2 Summary subcommand options

The `evaluate summary` subcommand reports statistics about the number of mapped reads for which the actual mapping locations are unknown.

`--reads (int)`. Specifies the number of reads given as input to the read mapper that produced the input SAM file (option `--input-sam`). This number can be different from the number of reads contained in the input SAM file (option `--input-sam`) if for example unmapped reads are not reported.

`--paired`. By default, the input SAM file (option `--input-sam`) is supposed to contain single-end reads. If the `--paired` option is set, it is supposed to contain paired-end reads. The summary for paired-end reads contains information on the number of reads mapped as paired and unpaired, as indicated by the `flag` field of the SAM format.

E.3.3 Sam subcommand options

The `evaluate sam` subcommand is used to evaluate the accuracy for sequencing reads generated by the Mason simulator and other read simulators that produce a reference SAM file containing alignments for the simulated reads.

`-r/--reference (file)`. Specifies the location of a file that contains the reference genome in multi-FASTA format.

`-w/--window` **(comma-separated list of** `ints`**,** `10`**).** The values in the list represent window sizes around the position in the reference genome from which the simulated read was extracted. An alignment is considered to be mapped correctly if it is mapped within a given window around the simulated position. Output is generated for each individual value.

`--input-edit (string, NM)`. Specifies the field of the SAM format that contains the edit distance of the alignments in the input SAM file (option `-i`). If no such field exists, the edit distance is computed from the CIGAR string, the read sequence and the reference genome. An alignment that is not mapped within a certain window around the simulated position (option

`-w`) is considered accurately mapped if its edit distance is less than the edit distance of the alignment taken from the reference SAM file (option `--reference-sam`).

`--reference-sam` **(`file`).** Specifies the location of a reference SAM file containing alignments of the simulated reads as generated by the Mason simulator. Alignments contained in this file should be sorted by read name, which is easily done using SAMtools.

`--reference-edit` **(`string`, `XE`).** Specifies the field of the SAM format that contains the edit distance of the alignments in the reference SAM file (option `--reference-sam`). The default value is set to `XE` because this is the field used by the Mason simulator.

E.3.4 Wgsim subcommand options

The `evaluate wgsim` subcommand is used to evaluate the accuracy for reads simulated by wgsim.

`-r`/`--reference` **(`file`).** Specifies the location of a file that contains the reference genome in multi-FASTA format.

`-w`/`--window` **(comma-separated list of `ints`, `10`).** The values in the list represent window sizes around the position in the reference genome from which the simulated read was extracted. An alignment is considered to be mapped correctly if it is mapped within a given window around the simulated position. Output is generated for each individual value.

`--input-edit` **(`string`, `NM`).** Specifies the field of the SAM format that contains the edit distance of the alignments in the input SAM file (option `-i`). If no such field exists, the edit distance is computed from the CIGAR string, the read sequence and the reference genome. An alignment that is not mapped within a certain window around the simulated position (option `-w`) is considered accurately mapped if its edit distance is less than the edit distance of the alignment taken from the reference SAM file (option `--reference-sam`).

Bibliography

[1] 1000 Genomes Project Consortium and others. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[2] A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[3] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[4] A. Abu-Doleh, K. Kaya, M. Abouelhoda, and U.V. Catalyürek. Extracting maximal exact matches on GPU. In *Workshop on Multi-Threaded Architectures and Applications*. to appear in IEEE, 2014.

[5] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, New York, 2008.

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

[7] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[8] S. Aluru and N. Jammula. A review of hardware acceleration for computational genomics. *IEEE Design & Test*, 31(1):19–30, 2014.

[9] A. Andersson, N.J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[10] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194(11):487–488, 1970.

[11] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Workshop on Algorithm Engineering and Experiments*, pages 84–97. SIAM, 2010.

[12] D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Combinatorial Pattern Matching*, pages 83–94. Springer, 2007.

[13] D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv based compressed text indices. *Journal of Experimental Algorithmics*, 15(1.5), 2010.

[14] D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. *Information and Computation*, 209(7):1070–1102, 2011.

[15] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1-2):54–101, 2012.

[16] N. Askitis and R. Sinha. RepMaestro: scalable repeat detection on disk-based genome sequences. *Bioinformatics*, 26(19):2368–2374, 2010.

[17] K.F. Au, H. Jiang, L. Lin, Y. Xing, and W.H. Wong. Detection of splice junctions from paired-end RNA-Seq data by SpliceMap. *Nucleic Acids Research*, 38(14):4570–4578, 2010.

[18] R. Baeza-Yates, E.F. Barbosa, and N. Ziviani. Hierarchies of indices for text retrieval. *Journal of Information Systems*, 21(6):497–514, 1996.

[19] M. Barsky, U. Stege, and A. Thomo. A survey of practical algorithms for suffix tree construction in external memory. *Software: Practice and Experience*, 40(11):965–988, 2010.

[20] M. Barsky, U. Stege, A. Thomo, and C. Upton. A new method for indexing genomes using on-disk suffix trees. In *Information and knowledge management*, pages 649–658, 2008.

[21] M. Barsky, U. Stege, A. Thomo, and C. Upton. Suffix trees for inputs larger than main memory. *Information Systems*, 36(3):644–654, 2011.

[22] M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.

[23] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[24] S.J. Bedathur and J.R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Data Engineering*, pages 720–731. IEEE, 2004.

[25] S.J. Bedathur and J.R. Haritsa. Search-optimized suffix-tree storage for biological applications. In *High Performance Computing*, pages 29–39. Springer, 2005.

[26] T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch. Space-efficient construction of the Burrows-Wheeler transform. In *String Processing and Information Retrieval*, pages 5–16. Springer, 2013.

[27] G. Benson, Y. Hernandez, and J. Loving. A bit-parallel, general integer-scoring sequence alignment algorithm. In *Combinatorial Pattern Matching*, pages 50–61. Springer, 2013.

[28] O. Berkman and U. Vishkin. Recursive Star-Tree Parallel Data Structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[29] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[30] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[31] P. Brady. Memory measurement script. `http://www.pixelbeat.org/scripts/ps_mem.py`, May 2014.

[32] N. Bray and L. Patcher. MAVID: Constrained ancestral alignment of multiple sequences. *Genome Research*, 14(4):693–699, 2004.

[33] G.S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Discrete Algorithms*, pages 581–590. ACM, 2006.

[34] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC SRC, 1994.

[35] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Experimental Algorithms*, pages 94–105. Springer, 2010.

[36] M.J. Chaisson and G. Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics*, 13(1):238, 2012.

[37] Y.F. Chien, W.K. Hon, R. Shah, and J.S. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Data Compression Conference*, pages 252–261. IEEE, 2008.

[38] J. Choi, H. Cho, and S. Kim. GAME: a simple and efficient whole genome alignment method using maximal exact match filtering. *Computational Biology and Chemistry*, 29(3):244–253, 2005.

[39] D.R. Clark and J.I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *Discrete Algorithms*, pages 383–391, 1996.

[40] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *String Processing and Information Retrieval*, pages 176–187. Springer, 2008.

[41] R. Clifford. Distributed suffix trees. *Journal of Discrete Algorithms*, 3(2-4):176–197, 2005.

[42] S.J. Cokus, S. Feng, X. Zhang, Z. Chen, B. Merriman, C.D. Haudenschild, S. Pradhan, S.F. Nelson, M. Pellegrini, and S.E. Jacobsen. Shotgun bisulphite sequencing of the Arabidopsis genome reveals DNA methylation patterning. *Nature*, 452(7184):215–219, 2008.

[43] M. Comin and M. Farreras. Parallel continuous flow: A parallel suffix tree construction tool for whole genomes. *Journal of Computational Biology*, 21(4):330–344, 2014.

[44] T.C. Conway and A.J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[45] M. Crochemore, R. Grossi, J. Kärkkäinen, and G.M. Landau. A constant-space comparison-based algorithm for computing the Burrows–Wheeler transform. In *Combinatorial Pattern Matching*, pages 74–82. Springer, 2013.

[46] A. Danek, S. Deorowicz, and S. Grabowski. Indexing large genome collections on a PC. *arXiv preprint arXiv:1403.7481*, 2014.

[47] F. De Bona, S. Ossowski, K. Schneeberger, and G. Rätsch. Optimal spliced alignments of short sequence reads. *BMC Bioinformatics*, 9(Suppl 10):O7, 2008.

[48] J. De Schrijver. *Identification of New Molecular Species using Second-Generation Sequencing*. PhD thesis, Ghent University, 2012.

[49] D De Smedt. Snelle en Nauwkeurige cDNA Mapping en Splice Site Identificatie. Master's thesis, Ghent University, 2013.

[50] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[51] M. Deloger, M. El Karoui, and M. Petit. A genomic distance based on MUM indicates discontinuity between most bacterial species and genera. *Journal of Bacteriology*, 191(1):91–99, 2009.

[52] S. Deorowicz and S. Grabowski. Data compression for sequencing data. *Algorithms for Molecular Biology*, 8(1):25, 2013.

[53] A. Dobin, C.A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T.R. Gingeras. STAR: ultrafast universal RNA-Seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[54] A. Döring, D. Weese, T. Rausch, and K. Reinert. SeqAn: an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9(1):11, 2008.

[55] R. Drmanac, A.B. Sparks, M.J. Callow, A.L. Halpern, N.L. Burns, B.G. Kermani, P. Carnevali, I. Nazarenko, G.B. Nilsen, G. Yeung, et al. Human genome sequencing using unchained base reads on self-assembling DNA nanoarrays. *Science*, 327(5961):78–81, 2010.

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

[57] B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. II. error probabilities. *Genome Research*, 8(3):186–194, 1998.

[58] M. Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

[59] J. Felsenstein, J. Archie, W. Day, W. Maddison, C. Meacham, F. Rohlf, and D. Swofford. The Newick tree format. `http://evolution.genetics.washington.edu/phylip/newicktree.html`, 1986.

[60] F. Fernandes, P.G.S. da Fonseca, L.M.S. Russo, A.L. Oliveira, and A.T. Freitas. Efficient alignment of pyrosequencing reads for re-sequencing applications. *BMC bioinformatics*, 12(1):163, 2011.

[61] F. Fernandes and A.T. Freitas. slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array. *Bioinformatics*, 30(4):464–471, 2013.

[62] E. Fernandez, W. Najjar, and S. Lonardi. String matching in hardware using the FM-index. In *Field-Programmable Custom Computing Machines*, pages 218–225. IEEE, 2011.

[63] P. Ferragina. Data structures: time, I/Os, entropy, joules! In *European Symposium on Algorithms*, pages 1–16. Springer, 2010.

[64] P. Ferragina and J. Fischer. Suffix arrays on words. In *Combinatorial Pattern Matching*, pages 328–339. Springer, 2007.

[65] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[66] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:1.12–1.31, 2009.

[67] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[68] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. Scott Vitter. On searching compressed string collections cache-obliviously. In *Principles of Database Systems*, pages 181–190. ACM, 2008.

[69] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Discrete Algorithms*, pages 269–278. SIAM, 2000.

[70] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[71] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[72] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20, 2007.

[73] J.W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1Part1):175–179, 1984.

[74] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470. Springer, 2007.

[75] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[76] P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods*, 6(11s):S6–S12, 2009.

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

[78] L. Foschini, R. Grossi, A. Gupta, and J.S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.

[79] M.C. Frith and L. Noé. Improved search heuristics find 20 000 new alignments between human and mouse genomes. *Nucleic Acids Research*, 42(7):e59–e59, 2014.

[80] M. Hsi-Yang Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 21(5):734–740, 2011.

[81] M. Garber, M.G. Grabherr, M. Guttman, and C. Trapnell. Computational methods for transcriptome annotation and quantification using RNA-seq. *Nature Methods*, 8(6):469–477, 2011.

[82] A. Ghoting and K. Makarychev. I/O efficient algorithms for serial and parallel suffix tree construction. *ACM Transactions on Database Systems*, 35(4):25, 2010.

[83] R. Giancarlo, S.E. Rombo, and F. Utro. Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies. *Briefings in Bioinformatics*, 15(3):390–406, 2014.

[84] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software, Practice and Experience*, 33(11):1035–1049, 2003.

[85] S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, Universität Ulm, 2011.

[86] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Experimental Algorithms*, 2014. To appear.

[87] S. Gog and J. Fischer. Advantages of shared data structures for sequences of balanced parentheses. In *Data Compression Conference*, pages 406–415. IEEE, 2010.

[88] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 2013.

[89] R. González and G. Navarro. Compressed text indexes with fast locate. In *Combinatorial Pattern Matching*, pages 216–227. Springer, 2007.

[90] R. González and G. Navarro. A compressed text index on secondary memory. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 71:127–154, 2009.

[91] S. Grabowski, G. Navarro, R. Przywarski, A. Salinger, and V. Mäkinen. A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science*, 17(6):1365–1384, 2006.

[92] S. Grabowski and M. Raniszewski. Two simple full-text indexes based on the suffix array. *arXiv preprint arXiv:1405.5919*, 2014.

[93] N. Grimsmo. On performance and cache effects in substring indexes. Report IDI-TR-2007-04, Norwegian University of Science and Technology, Norway, 2007.

[94] R. Grossi. A quick tour on suffix arrays and compressed suffix arrays. *Theoretical Computer Science*, 412(27):2964–2973, 2011.

[95] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Discrete Algorithms*, pages 841–850. SIAM, 2003.

[96] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[97] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 11th edition, 1997.

[98] M. Halachev, N. Shiri, and A. Thamildurai. Efficient and scalable indexing techniques for biological sequence data. In *Bioinformatics Research and Development*, pages 464–479. Springer, 2007.

[99] E.C. Hayden. Technology: the $1,000 genome. *Nature*, 507(7492):294–295, 2014.

[100] D. Hernandez, P. François, L. Farinelli, M. Østerås, and J. Schrenzel. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, 2008.

[101] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[102] S. Hoffmann, C. Otto, S. Kurtz, C.M. Sharma, P. Khaitovich, J. Vogel, P.F. Stadler, and J. Hackermüller. Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS Computational Biology*, 5(9):e1000502, 2009.

[103] M. Holtgrewe. Mason – a read simulator for second generation sequencing data. `http://www.seqan.de/projects/mason.html`, May 2014.

[104] M. Holtgrewe, A.K. Emde, D. Weese, and K. Reinert. A novel and well-defined benchmarking method for second generation read mapping. *BMC Bioinformatics*, 12(1):210, 2011.

[105] W.K. Hon, M. Patil, R. Shah, and S.V. Thankachan. Compressed property suffix trees. *Information and Computation*, 232:10–18, 2013.

[106] W.K. Hon and K. Sadakane. Space-economical algorithms for finding maximal unique matches. In *Combinatorial Pattern Matching*, pages 144–152. Springer, 2002.

[107] W.K. Hon, K. Sadakane, and W.K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[108] W.K. Hon, R. Shah, and J.S. Vitter. Compression, indexing, and retrieval for massive string data. In *Combinatorial Pattern Matching*, pages 260–274. Springer, 2010.

[109] L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013.

[110] S. Huang, J. Zhang, R. Li, W. Zhang, Z. He, T.W. Lam, Z. Peng, and S.M. Yiu. SOAPsplice: genome-wide *ab initio* detection of splice junctions from RNA-seq data. *Frontiers in Genetics*, 2(46), 2011.

[111] E. Hunt, M.P. Atkinson, and R.W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11(3):256–271, 2002.

[112] H. Hyyrö. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Computing*, 10(1):29–39, 2003.

[113] H. Hyyrö. A note on bit-parallel alignment computation. In *Stringology*, pages 79–87, 2004.

[114] C.S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. The weighted suffix tree: an efficient data structure for handling molecular weighted sequences and its applications. *Fundamenta Informaticae*, 71(2):259–277, 2006.

[115] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Combinatorial Pattern Matching*, pages 60–71. Springer, 2006.

[116] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[117] A. Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[118] G. Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science*, pages 549–554. IEEE, 1989.

[119] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[120] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[121] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[122] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *South American Workshop on String Processing*, pages 141–155, 1996.

[123] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proceedings of the 2nd Conference on Computing and Combinatorics*, pages 219–230. Citeseer, 1996.

[124] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

[125] W.J. Kent. BLAT - the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[126] Z. Khan, J.S. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.

[127] M.O. Kiilekci, J.S. Vitter, and B. Xir. Time-and space-efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet trees. In *Bioinformatics and Biomedicine*, pages 622–625. IEEE, 2010.

[128] D. Kim, G. Pertea, C. Trapnell, H. Pimentel, R. Kelley, and S.L. Salzberg. TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, 14(4):R36, 2013.

[129] D.K. Kim, M. Kim, and H. Park. Linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays. *Algorithmica*, 52(3):350–377, 2008.

[130] P. Klus, S. Lam, D. Lyberg, M.S. Cheung, G. Pullan, I. McFarlane, G.S.H. Yeo, and B.Y.H. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27, 2012.

[131] D.E. Knuth, J.H. Morris, and V.B. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(1):323–350, 1977.

[132] S. Koren, M.C. Schatz, B.P. Walenz, J. Martin, J.T. Howard, G. Ganapathy, Z. Wang, D.A. Rasko, W.R. McCombie, E.D. Jarvis, and A.M. Phillippy. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature Biotechnology*, 30(7):693–700, 2012.

[133] M.O. Kulekci, W.K. Hon, R. Shah, J.S. Vitter, and B. Xu. PSI-RA: A parallel sparse index for read alignment on genomes. In *Bioinformatics and Biomedicine*, pages 663–668. IEEE, 2010.

[134] F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

[135] S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[136] S. Kurtz. The Vmatch large scale sequence analysis software. `http://www.vmatch.de/`, July 2011.

[137] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[138] S. Kuruppu, S.J. Puglisi, and J. Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Australasian Computer Science Conference*, pages 91–98. Australian Computer Society, Inc., 2011.

[139] N.C. Kyrpides, P. Hugenholtz, J.A. Eisen, T. Woyke, M. Göker, C.T. Parker, R. Amann, B.J. Beck, P.S.G. Chain, J. Chun, et al. Genomic encyclopedia of bacteria and archaea: Sequencing a myriad of type strains. *PLoS Biology*, 12(8):e1001920, 2014.

[140] 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*, pages 31–36. IEEE, 2009.

[141] B. Langmead and S.L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[142] B. Langmead, M.C. Schatz, J. Lin, M. Pop, and S.L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134, 2009.

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

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

[145] W. Lee, M.P. Stromberg, A. Ward, C. Stewart, E.P. Garrison, and G.T. Marth. MOSAIK: A hash-based algorithm for accurate next-generation sequencing short-read mapping. *PloS One*, 9(3):e90581, 2014.

[146] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[147] H. Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012.

[148] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2014.

[149] H. Li. The wgsim read simulator. `https://github.com/lh3/wgsim`, May 2014.

[150] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[151] H. Li and R. Durbin. Fast and accurate long read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

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

[153] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[154] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

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

[156] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

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

[158] W. Li. RNASeqReadSimulator: a simple RNA-seq read simulator. `http://alumni.cs.ucr.edu/~liw/rnaseqreadsimulator.html`, September 2014.

[159] K.H. Lim, L. Ferraris, M.E. Filloux, B.J. Raphael, and W.G. Fairbrother. Using positional distribution to identify splicing elements and predict pre-mRNA processing defects in human genes. *Proceedings of the National Academy of Sciences*, 108(27):11093–11098, 2011.

[160] C.M. Liu, T. Wong, E. Wu, R. Luo, S.M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, et al. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[161] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, and M. Law. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012:ID 251364, 2012.

[162] Y. Liu, B. Popp, and B. Schmidt. CUSHAW3: sensitive and accurate base-space and color-space short-read alignment with hybrid seeding. *PloS One*, 9(1):e86869, 2014.

[163] Y. Liu and B. Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, 2012.

[164] Y. Liu, B. Schmidt, and D.L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[165] W.K. Loh, Y.S. Moon, and W. Lee. A fast divide-and-conquer algorithm for indexing human genome sequences. *Transactions on Information and Systems*, 94(7):1369–1377, 2011.

[166] N.J. Loman, R.V. Misra, T.J. Dallman, C. Constantinidou, S.E. Gharbia, J. Wain, and M.J. Pallen. Performance comparison of benchtop high-throughput sequencing platforms. *Nature Biotechnology*, 30(5):434–439, 2012.

[167] M.G. Maaß. Computing suffix links for suffix trees and arrays. *Information Processing Letters*, 101(6):250–254, 2007.

[168] N. Maillet, C. Lemaitre, R. Chikhi, D. Lavenier, and P. Peterlongo. Compareads: comparing huge metagenomic experiments. *BMC Bioinformatics*, 13(Suppl 19):S10, 2012.

[169] V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003.

[170] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Combinatorial Pattern Matching*, pages 420–433. Springer, 2004.

[171] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[172] V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching - efficient secondary memory and distributed implementation of compressed suffix arrays. In *Algorithms and Computation*, pages 681–692. Springer, 2004.

[173] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[174] V. Mäkinen, L. Salmela, and J. Ylinen. Normalized N50 assembly metric using gap-restricted co-linear chaining. *BMC Bioinformatics*, 13(1):255, 2012.

[175] N. Malhis, Y.S.N. Butterfield, M. Ester, and S.J.M. Jones. Slider - maximum use of probability information for alignment of short sequence reads and SNP detection. *Bioinformatics*, 25(1):6–13, 2009.

[176] U. Manber and E.W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[177] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[178] M. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

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

[180] M. Marín and G. Navarro. Distributed query processing using suffix arrays. In *String Processing and Information Retrieval*, pages 311–325. Springer, 2003.

[181] C.E. Mason and O. Elemento. Faster sequencers, larger datasets, new challenges. *Genome Biology*, 13(3):314, 2012.

[182] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[183] R.K. Menon, G.P. Bhat, and M.C. Schatz. Rapid parallel genome indexing with MapReduce. In *Workshop on MapReduce and its Applications*, pages 51–58. ACM, 2011.

[184] M.L. Metzker. Sequencing technologies – the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2010.

[185] F. Meyer, S. Kurtz, R. Backofen, S. Will, and M. Beckstette. Structator: fast index-based search for RNA sequence-structure patterns. *BMC Bioinformatics*, 12(1):214, 2011.

[186] D.R. Morrison. PATRICIA-practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[187] J.C. Mu, H. Jiang, A. Kiani, M. Mohiyuddin, N. Bani Asadi, and W.H. Wong. Fast and accurate read alignment for resequencing. *Bioinformatics*, 28(18):2366–2373, 2012.

[188] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[189] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[190] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13:2, 2009.

[191] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[192] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2:1–2:61, 2007.

[193] G. Navarro and L. Russo. Fast fully-compressed suffix trees. In *Data Compression Conference*, pages 283–291. IEEE, 2014.

[194] 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*, 48(3):443–453, 1970.

[195] Z. Ning, A.J. Cox, and J.C. Mullikin. SSAHA: a fast search method for large DNA databases. *Genome Research*, 11(10):1725–1729, 2001.

[196] G. Nong. Practical linear-time o(1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):15, 2013.

[197] G. Nong, S. Zhang, and W. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[198] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *String Processing and Information Retrieval*, pages 322–333. Springer, 2010.

[199] E. Ohlebusch and S. Gog. A compressed enhanced suffix array supporting fast string matching. In *String Processing and Information Retrieval*, pages 51–62. Springer, 2009.

[200] E. Ohlebusch and S. Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Information Processing Letters*, 110(3):123–128, 2010.

[201] E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *String Processing and Information Retrieval*, pages 347–358. Springer, 2010.

[202] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Workshop on Algorithm Engineering and Experiments*, pages 60–70. SIAM, 2007.

[203] B.D. Ondov, A. Varadarajan, K.D. Passalacqua, and N.H. Bergman. Efficient mapping of applied biosystems SOLiD sequence data to a reference genome for functional genomic applications. *Bioinformatics*, 24(23):2776–2777, 2008.

[204] R.V. Pandey and C. Schlötterer. DistMap: a toolkit for distributed short read mapping on a Hadoop cluster. *PloS One*, 8(8):e72614, 2013.

[205] B. Phoophakdee and M.J. Zaki. Genome-scale disk-based suffix tree indexing. In *Management of Data*, pages 833–844. ACM, 2007.

[206] S.J. Puglisi, W.F. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *String Processing and Information Retrieval*, pages 122–133. Springer, 2006.

[207] S.J. Puglisi, WF. Smyth, and A.H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):1–31, 2007.

[208] M.A. Quail, M. Smith, P. Coupland, T.D. Otto, S.R. Harris, T.R. Connor, A. Bertoni, H.P. Swerdlow, and Y. Gu. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genomics*, 13(1):341, 2012.

[209] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Discrete Algorithms*, pages 233–242. ACM, 2002.

[210] R.J. Roberts, M.O. Carneiro, and M.C. Schatz. The advantages of SMRT sequencing. *Genome Biology*, 14(6):405, 2013.

[211] S.M. Rumble, P. Lacroute, A.V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: accurate mapping of short color-space reads. *PLoS Computational Biology*, 5(5):e1000386, 2009.

[212] L. Russo, G. Navarro, and A. Oliveira. Parallel and distributed compressed indexes. In *Combinatorial Pattern Matching*, pages 348–360. Springer, 2010.

[213] L.M.S. Russo, G. Navarro, and A.L. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):No. 53, 2011.

[214] L.M.S. Russo, G. Navarro, A.L. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.

[215] L.M.S. Russo and A.L. Oliveira. A compressed self-index using a Ziv-Lempel dictionnary. *Information Retrieval*, 11(4):359–388, 2008.

[216] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation*, pages 410–421. Springer, 2000.

[217] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Discrete Algorithms*, pages 225–232. ACM, 2002.

[218] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[219] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[220] K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.

[221] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. A four-stage algorithm for updating a Burrows-Wheeler transform. *Theoretical Computer Science*, 410(43):4350–4359, 2009.

[222] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8(2):241–257, 2010.

[223] F. Sanger, S. Nicklen, and A.R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[224] M.C. Schatz. Cloudburst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[225] M.C. Schatz, C. Trapnell, A.L. Delcher, and A. Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[226] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012.

[227] J. Schröder, H. Schröder, S.J. Puglisi, R. Sinha, and B. Schmidt. SHREC: a short-read error correction method. *Bioinformatics*, 25(17):2157–2163, 2009.

[228] T. Shibuya. Geometric suffix tree: Indexing protein 3-D structures. *Journal of the ACM*, 57(3):15, 2010.

[229] A.M.S. Shrestha, M.C. Frith, and P. Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in Bioinformatics*, 15(2):138–154, 2014.

[230] J.T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[231] R. Sinha, S.J. Puglisi, A. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *Management of Data*, pages 661–672. ACM, 2008.

[232] J. Sirén. Compressed suffix arrays for massive data. In *String Processing and Information Retrieval*, pages 63–74. Springer, 2009.

[233] J. Sirén, N. Valimaki, and V. Makinen. Indexing graphs for path queries withapplications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

[234] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[235] L. Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[236] H. Stranneheim and J. Lundeberg. Stepping stones in DNA sequencing. *Biotechnology Journal*, 7(9):1063–1073, 2012.

[237] The NIH HMP Working Group et al. The NIH human microbiome project. *Genome Research*, 19(12):2317–2323, 2009.

[238] Y. Tian, S. Tata, R.A. Hankins, and J.M. Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005.

[239] F. Transier and P. Sanders. Compressed inverted indexes for in-memory search engines. In *Workshop on Algorithm Engineering and Experimentation*, pages 3–12. SIAM, 2008.

[240] C. Trapnell, L. Pachter, and S.L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–1111, 2009.

[241] D. Tsirogiannis and N. Koudas. Suffix tree construction algorithms on modern hardware. In *Extending Database Technology*, pages 263–274. ACM, 2010.

[242] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[243] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *ACM Journal of Experimental Algorithmics*, 14:2, 2009.

[244] J.C. Venter, M.D. Adams, E.W. Myers, P.W. Li, R.J. Mural, G.G. Sutton, H.O. Smith, M. Yandell, C.A. Evans, E.A. Holt, et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[245] J.S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[246] A. Voskoboynik, N.F. Neff, D. Sahoo, A.M. Newman, D. Pushkarev, W. Koh, B. Passarelli, H.C. Fan, G.L. Mantalas, K.J. Palmeri, K.J. Ishizuka, C. Gissi, F. Griggio, R. Ben-Shlomo, D.M. Corey, L. Penland, R.A. White, I.L. Weissman, and S.R. Quake. The genome sequence of the colonial chordate, Botryllus schlosseri. *eLife*, 2:e00569, 2013.

[247] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Research*, 40(15):6993–7015, 2012.

[248] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

[249] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. A long fragment aligner called ALFALFA. *submitted*, 2014.

[250] M. Vyverman, D. De Smedt, Y. Lin, L. Sterck, B. De Baets, V. Fack, and P. Dawyndt. Fast and accurate cDNA mapping and splice site identification. In *Bioinformatics Models, Methods and Algorithms*, page Nr 64, 2014.

[251] K. Wang, D. Singh, Z. Zeng, S.J. Coleman, Y. Huang, G.L. Savich, X. He, P. Mieczkowski, S.A. Grimm, and C.M. Perou. MapSplice: accurate mapping of RNA-Seq reads for splice junction discovery. *Nucleic Acids Research*, 38(18), 2010.

[252] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. *Nature Review Genetics*, 10(1):57–63, 2009.

[253] D. Weese, M. Holtgrewe, and K. Reinert. RazerS 3: Faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599, 2012.

[254] P. Weiner. Linear pattern matching algorithm. In *Switching and Automata Theory*, pages 1–11. IEEE, 1973.

[255] K.A. Wetterstrand. DNA sequencing costs: Data from the NHGRI genome sequencing program (GSP). `www.genome.gov/sequencingcosts`, May 2014.

[256] B. Wold and R.M. Myers. Sequence census methods for functional genomics. *Nature Methods*, 5(1):19–21, 2008.

[257] T.D. Wu and S. Nacu. Fast and SNP-tolerant detection of complex variants and splicing in short reads. *Bioinformatics*, 26(7):873–881, 2010.

[258] T.D. Wu and C.K. Watanabe. GMAP: a genomic mapping and alignment program for mRNA and EST sequences. *Bioinformatics*, 21(9):1859–1875, 2005.

[259] M. Zhao, W.P. Lee, E.P. Garrison, and G.T. Marth. SSW library: An SIMD Smith-Waterman C/C++ library for use in genomic applications. *PloS One*, 8(12):e82138, 2013.

[260] Z. Zhao, J. Yin, Y. Zhan, W. Xiong, Y. Li, and F. Liu. PSAEC: An improved algorithm for short read error correction using partial suffix arrays. In *Frontiers in Algorithmics and Algorithmic Aspects in Information and Management*, pages 220–232. Springer, 2011.

[261] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[262] D. Zou, Y. Dou, and F. Xia. Optimization schemes and performance evaluation of Smith–Waterman algorithm on CPU, GPU and FPGA. *Concurrency and Computation: Practice and Experience*, 24(14):1625–1644, 2012.

Nederlandstalige samenvatting

Onderzoek in bioinformatica wordt momenteel gedomineerd door een (r)evolutie op het gebied van sequeneringstechnologie. Deze technologie produceert korte strings (*reads*) die de exacte volgorde van de basen in een DNA fragment bevatten. De continue evolutie in deze technologie heeft ervoor gezorgd dat biologische sequenties steeds sneller en goedkoper kunnen worden geproduceerd, wat op zich geleid heeft tot nieuwe toepassingsmogelijkheden en prestigieuze projecten. Echter, de huidige programma's voor het analyseren van deze data kunnen de groei moeilijk bijhouden. In huidige projecten wordt de *bottleneck* gevormd door onder andere opslagcapaciteit, computationele analyse en interpretatie van de data.

Eén van de belangrijkste onderdelen in het analyseren van biologische sequenties is het afbeelden van *reads* op een referentiegenoom. In dit proces wordt voor elke *read* de posities binnen het genoom gezocht waarbij de *read* het kleinste aantal verschillen vertoont met het overeenkomstig deel van het referentiegenoom. Zo een alignering van een *read* met een fragment van het referentiegenoom bevat ook een lijst met alle verschillen en gelijkenissen tussen beide sequenties. Het Olympische motto *citius*, *altius*, *fortius* is goed van toepassing in dit computationeel en algoritmisch uitdagend probleem, aangezien van de afbeeldingsprogramma's wordt verwacht een optimale balans te vinden tussen maximale snelheid, minimaal geheugenverbruik en maximale nauwkeurigheid. Verder moeten afbeeldingsprogramma's ook rekening houden met andere snel wijzigende eigenschappen van de *reads*, zoals de lengte en de soort en het aantal sequeneringsfouten. De huidige generatie afbeeldingsprogramma's is ontwikkeld voor het afbeelden van korte *reads* die geen of slechts een klein aantal mutaties en/of kleine inserties en deleties bevatten. Daartegenover staat dat, met de huidige trend in sequeneringstechnologie, *reads* steeds langer worden, een groter aantal fouten bevatten en meer en

lange inserties en deleties bevatten.

In deze doctoraatsthesis ontwikkelen we geavanceerde algoritmen en indexstructuren om snel en nauwkeurig lange *reads* af te beelden op een referentiegenoom. De ontwikkelde programma's steunen op een combinatie van efficiënte indexstructuren, zoekalgoritmen en groot aantal slimme heuristieken. De structuur van de thesis benadrukt het belang van elk van deze componenten, waarvan de meeste ook toepasbaar zijn op andere problemen binnen de analyse van biologische sequenties.

Het **eerste hoofdstuk** bevat inleidende begrippen en notaties rond de concepten *string*, biologische sequenties en soorten gemeenschappelijke sequenties. Veel van deze begrippen zijn vertrouwd voor iemand met een achtergrond in bioinformatica en kunnen dus veilig worden overgeslaan. Verder bevat het inleiddnde hoofdstuk achtergrondinformatie over sequeneringstechnologie, waaronder een historisch overzicht en een vergelijking van de huidige platformen voor sequenering. Verder bespreken we ook de eigenschappen van de *read* sequenties, waaronder de kwaliteitswaarden die aan afzonderlijke basen kunnen worden gekoppeld en bepaalde geavanceerde technieken die bij het sequeneren worden gebruikt. Deze eigenschappen hebben ook invloed op de keuzes die gemaakt zijn tijdens de ontwikkeling van onze afbeeldingsalgoritmen.

Naast de eigenschappen van de data, worden ook de algoritmische doelstellingen besproken. Er worden definities en notaties opgesteld voor het aligneren van sequenties en het afbeelden van *reads* op een referentiegenoom. Verder geven we ook een overzicht van de reeds beschikbare afbeeldingsprogramma's, met als doel het aantonen van de diversiteit binnen de beschikbare programma's en het kiezen van programma's die gebruikt kunnen worden om onze methoden te evalueren.

Tot slot bespreken we in dit hoofdstuk ook enkele vaak gebruikte algoritmen voor het aligneren van sequenties die gebruik maken van dynamisch programmeren. Niettegenstaande deze algoritmen algemeen bekend zijn binnen de bioinformatica, is een herhaling van deze concepten nuttig doordat geoptimaliseerde versies van deze algoritmen een belangrijke component vormen binnen de afbeeldingsalgoritmen.

Het **tweede hoofdstuk** bevat een overzicht van de klasse van indexstructuren die in deze thesis gebruikt worden. Indexstructuren zijn gespecialiseerde datastructuren die vaak worden gebruikt binnen de bioinformatica omdat ze snelle zoekopdrachten binnen grote sequenties mogelijk maken. Ze worden ook gebruikt als onderdeel van afbeeldingsalgoritmen om snel overeenkomsten te vinden tussen delen van een *read* en een referentiegenoom en zo de verdere zoekruimte te

beperken. Verder bepaalt de grootte van de indexstructuur ook de hoeveelheid geheugen dat nodig is voor het afbeeldingsalgoritme.

Alhoewel indexstructuren veelvuldig worden gebruikt binnen de bioinformatica, zijn hun structuur, eigenschappen en beperkingen minder gekend. Verder werden er de afgelopen jaren vele nieuwe indexstructuren ontwikkeld met nieuwe complexe structuren die een andere balans vertonen tussen geheugeneisen en snelheid. In dit hoofdstuk willen we een overzicht geven van de structuur van de meest gebruikte indexstructuren en recent ontwikkelde varianten. We bespreken verbanden tussen indexstructuren, hun eigenschappen en beperkingen en de balans die ze bereiken tussen snelheid en geheugen.

In het **derde hoofdstuk** introduceren we een nieuwe indexstructuur, *enhanced sparse suffix array* genaamd. Deze indexstructuur is gebaseerd op suffixtabellen die de lexicografische ordening van de suffixen van een string bevatten. Het woord *sparse* duidt op het feit dat slechts een beperkt aantal suffixen geïndexeerd wordt. Het woord *advanced* duidt op de extra informatie die wordt bijgehouden over de langste gemeenschappelijke prefixen van opeenvolgende waarden in de suffixtabel. Deze extra informatie zorgt voor een significante versnelling van zoekopdrachten.

Verder wordt ook een nieuw algoritme voorgesteld voor het vinden van maximale exacte matches tussen twee sequenties. Een gelijke substring of match tussen twee sequenties wordt maximaal genoemd indien de karakters links en rechts van de match niet overeenkomen in de twee sequenties. Deze maximale exacte matches worden vaak gebruikt in programma's voor het aligneren van sequenties als ankers waar de aligering kan worden gestart. In het bijzonder kunnen ze ook worden gebruikt in afbeeldingsprogramma's, vooral voor het afbeelden van lange *reads*.

Het programma *essaMEM* bevat het algoritme dat gebruik maakt van de nieuwe indexstructuur om snel maximale exacte matches te vinden tussen twee sequenties. Onze testresultaten tonen aan dat *essaMEM* sneller is dan andere veelgebruikte programma's voor het vinden van maximale exacte matches, waaronder *sparseMEM* [126]. Verder presteert *essaMEM* ook goed in vergelijking met *backwardMEM* [201], dat steunt op de complexe gecomprimeerde suffixtabel indexstructuur. Hiermee tonen we aan dat het gebruik van de categorie *sparse* indexstructuren een interessante piste is voor verder onderzoek.

In het **vierde hoofdstuk** komen we tot het hoofdresultaat van deze thesis. We introduceren *ALFALFA*: een nieuw algoritme dat specifiek werd ontwikkeld om snel en nauwkeurig lange DNA *reads* af te beelden op een referen-

tiegenoom. *ALFALFA* implementeert de veelvuldig toegepaste *seed-and-extend* techniek, waarbij voor het eerste deel gesteund wordt op de resultaten uit Hoofdstuk 3. De *enhanced sparse suffix arrays* worden gebruikt om gedeeltelijke matches te vinden tussen de *read* en het referentiegenoom. Deze *seeds* worden gefilterd en gegroepeerd om zo kandidaatregio's af te bakenen in het referentiegenoom. Hierbij worden algoritmen en heuristieken gebruikt die specifiek gekozen werden om te kunnen omgaan met lange *reads*. Een alignering tussen een afgebakende regio van het referentiegenoom en een *read* wordt bekomen door eerst een ketting te bouwen van *seeds* en daarna gebruik te maken dynamisch programmeren algoritmen om de gaten tussen opeenvolgende seeds in de ketting op te vullen (*extension* stap).

We hebben *ALFALFA* vergeleken met andere afbeeldingsprogramma's op basis van snelheid, nauwkeurigheid en geheugenverbruik. De tests omvatten vele datasets met een grote verscheidenheid aan eigenschappen en verschillende maatstaven voor het meten van nauwkeurigheid. De resultanten tonen aan dat *ALFALFA* heel snel en nauwkeurig is voor heet afbeelden van lange *reads* ($>$ 500bp) en nog steeds competitief is voor middelgrote *reads* ($>$ 100bp). De bekomen balans tussen snelheid, geheugengrootte en nauwkeurigheid kan ook worden aangepast naar de specifieke behoeften van gebruikers via het aanpassen van enkele parameters.

In het **vijfde hoofdstuk** worden de algoritmen uit het vorige hoofdstuk aangepast zodanig dat ook cDNA sequenties en *reads* bekomen uit RNA-seq experimenten kunnen worden afgebeeld. Genen van eukaryoten bestaan uit opeenvolgende regio's *exonen* en *intronen*. Na transcriptie verdwijnen de intronen echter door *splicing*, waardoor aligneringen van cDNA sequenties en RNA-seq *reads* met een referentiegenoom grote gaten kunnen bevatten. Traditionele afbeeldingsprogramma's kunnen moeilijk overweg met deze sequenties doordat intronen meestal groter zijn dan de deleties die kunnen worden gedetecteerd.

Voor deze toepassing hebben we *ALFALFA* gecombineerd met nieuwe algoritmen voor het combineren van *seeds* en krachtige dynamisch programmeren algoritmen die werden geïntroduceerd door *GMAP* [258]. Een prototype voor een nieuw afbeeldingsprogramma met deze combinatie van technieken werd *mesalina* gedoopt. Voorlopige rersultaten tonen aan dat *mesalina* heel snel is in vergelijking met gelijkaardige programma's, vooral voor langere *reads*. Verder blijkt het prototype ook competitief te zijn op het vlak van nauwkeurigheid.

Alle ontwikkelde algoritmen werden geïmplementeerd in `C++` en zijn vrij beschikbaar op `https://github.com/readmapping`.

List of Figures

List of Tables

Index