Algorithms and Tools for the Analysis of High-Throughput DNA Sequencing Data

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Marcel Martin

Dortmund

2013

Tag der mündlichen Prüfung: 26. November 2013

Dekan: Prof. Dr.-Ing. Gernot A. Fink

1. Gutachter: Prof. Dr. Sven Rahmann
2. Gutachter: Prof. Dr. Jens Stoye

Abstract

High-throughput DNA sequencing technologies make it possible to determine the order of the nucleotides adenine, cytosine, guanine and thymine in DNA samples, resulting in millions of short strings (*reads*) over the alphabet (A, C, G, T). Advances in biological and biomedical research rely on the ability of bioinformatics to make sense out of that data with novel algorithms and tools. In this thesis, we contribute on four levels to the typical data processing pipeline in sequencing experiments and provide software tools that implement the described algorithms.

When sequenced DNA fragments are short, reads can contain *adapter sequences*. These artifacts are a technical requirement of the sequencing process. We describe how to remove them with a modified semiglobal alignment algorithm that finds overlapping regions between read and adapter. The algorithm is designed to only find alignments below a given error rate threshold, where the error rate is defined as the number of errors divided by the number of aligned adapter characters. We show how to use only linear space while still keeping track of all information necessary to correctly locate and remove adapter sequences. The algorithm can remove adapters also from colorspace reads, which come from a sequencing technology that queries two adjacent nucleotides (colors) of DNA at the same time. We show how to modify the trimming procedure to get correct results. The easy-to-use *cutadapt* tool is introduced. It contains additional features that make pre-processing of adapter-contaminated reads simple, and is in use by many other researchers.

The next step in the pipeline is *read mapping*, where the likely origin of reads is found on a given reference DNA. We concentrate on mapping reads from bisulfite sequencing experiments, in which sodium bisulfite is used to determine which cytosines have a methyl group attached to them. Methylation changes gene expression and is therefore biologically interesting. Bisulfite converts unmethylated cytosines into thymines. By comparing modified reads to the reference, methylation patterns can be determined. To map reads while allowing sequencing errors and also differences from bisulfite conversion, we introduce the bisulfite q-gram index, an extension of regular q-gram indices. For a given q-gram (string of length q), the index returns all positions in the reference where that bisulfite-converted q-gram may have originated. By efficiently simulating bisulfite conversion of the reference, the index can be constructed in time proportional to its memory usage. Simulation theoretically leads to an exponential increase in index size, but size is only triple that of a regular index on realistic references. We describe how to map reads with the index with the seed-and-extend paradigm, first finding short matches with the help of the index, and extending them to longer maximal error-free matches (seeds) with either a deterministic finite automaton (DFA) or an efficient bitparallel algorithm. Seeds are then extended to an alignment that covers the full read, and parts that were not bisulfite converted are detected. We show that the number of bisulfite strings of a given length n is approximately $1.19 \cdot 3.3^n$, and we show how to compress the index by up to 25% while retaining efficient access. We finally apply the full read mapping algorithm to a dataset of 454 bisulfite sequencing data using the *Verjinxer* tool.

Following read mapping, we continue with the *Exomate* tool, which is used in exome sequencing studies. The tool assists medical researchers in finding potentially disease-causing mutations in syndrome and tumor patients and consists of three parts. The first is an automated pipeline that directs all steps from read mapping up to finding of mutations (variant calling). The second part is a relational database that stores variants and metadata about patients, samples, etc. The third part is a web interface designed for highly interactive analysis of the variant data by medical researchers. We discuss how to filter variants by discarding those that are likely not disease causing, for example, by discarding low-quality variants, those also found in public databases (such as dbSNP), and those also in an unaffected control sample. Filtering options can be set within the web interface and result in dynamically generated SQL queries that typically finish within a few seconds. Extensive quality control is implemented through various queries that check for known chromosome loss, patient gender and dbSNP re-discovery rate. We describe how Exomate has been used successfully in multiple exome sequencing studies.

As last topic, we return to the task of aligning reads to a reference, but in this case, we study reads obtained as flowgrams from a specific sequencing technology. Flowgrams associate each sequenced nucleotide with a fractional length (an intensity). To get a regular sequence which can then be aligned, intensities are rounded, but measurement errors of intensities occur and are the dominant cause of sequencing errors (over- and undercalls). We propose to directly align a flowgram to a reference and to let the conversion to a regular sequence be guided by the reference, avoiding information loss from rounding. Variants of this idea have been suggested before, but our method, which we call flowgram-string alignment, is the first to use a well-founded statistical model. Our score function has two components that model the processes that take place: 1) editing of the sample relative to the reference and 2) measurement errors during sequencing. We give a recurrence equation that leads to a dynamic programming algorithm on the alignment matrix between reference string and flowgram such that the optimal flowgram alignment can be computed. For the first score component, we give a closed formula for alignment scores between a DNA string and a homopolymer (a single nucleotide repeated a number of times), which is then used to show that the score can be evaluated in constant time with some pre-processing. For the score component that assesses measurement errors, we use correctly aligned reads to estimate empirical frequencies for intensity measurements vs. sequence length, from which we derive log-odds scores and then approximate them by a function with five parameters. To evaluate the method, we compare flowgram-string alignment to regular alignment by simulating the two-stage sequencing process (editing events, sequencing errors) and find that our method considerably reduces the number of spurious editing events introduced by measurement errors.

Danksagungen

Diese Arbeit würde es nicht geben ohne viele andere Menschen, die mir geholfen haben und denen ich sehr dankbar bin.

Mein Betreuer Sven Rahmann hat mich während meiner ganzen Arbeit unterstützt, indem er immer Zeit hatte, meine Fragen zu beantworten. Seine neuen und manchmal ungewöhnlichen Ideen, die er in unsere Diskussionen einfließen ließ, brachten mich in meinen Überlegungen oft voran. Jens Stoye weckte vor Jahren mein Interesse an der Bioinformatik und ermunterte mich in einer schwierigen Phase weiterzumachen. Heinrich Müller und Lars Hildebrand haben ohne zu zögern zugestimmt, die Prüfungskommission zu vervollständigen. Tobias Marschall ist der ideale Bürokollege. Durch unsere Diskussionen hatte er wohl einen größeren Einfluss auf mich als er es selbst vermuten würde. Meine Kollegen Marianna D'Addario und Dominik Kopczynski hatten immer Zeit und ein offenes Ohr für mich, wenn ich jemanden zum Reden brauchte – sowohl fachliche als auch mal andere Dinge betreffend. Johannes Köster hat das Snakemake-Programm geschrieben, welches ich nutzte, um meine Forschung reproduzierbar zu machen. Michael Zeschnigk half mir dabei, den biologischen Teil unserer Arbeit zu verstehen. Durch seine Erklärungen konnte meine Arbeit letztlich relevant für Forscher in der Genetik werden.

Neben Sven, Jens, Tobias, Marianna, Dominik, Johannes und Michael haben auch Christina Czeschik, Roland Wittler und Romina Martin Teile des Manuskripts gelesen und hilfreiche Rückmeldungen gegeben, die fast ausnahmslos eingeflossen sind.

Meine Eltern haben auf andere Art zum Gelingen beigetragen, denn sie haben es überhaupt erst möglich gemacht, dass ich soweit gekommen bin. Romina, die nicht nur meine Frau, sondern auch eine Forscherkollegin ist, bin ich ewig dankbar.

Marcel Martin
Dortmund, im November 2013

Contents

1 Introduction

DNA sequencing has revolutionized medical and biological research. The development was catalyzed by the invention of highly parallel *high-throughput sequencing* (HTS) technologies. With them, the order of the four nucleotides adenine, cytosine, guanine and thymine (A, C, G, T) in millions of DNA fragments can be determined at the same time, at rates and costs that had previously not been achievable. This enabled new applications in biological and biomedical research, but also introduced new challenges resulting from the need to process the data generated by the sequencers. This thesis contributes to the area of bioinformatics that is concerned with tackling those challenges. On the one hand, we describe algorithms which have been designed to work with high-throughput sequencing data, but we also focus on making those algorithms available in the form of usable software tools.

For many applications, data from a HTS instrument first undergoes pre-processing – such as removal of low-quality sequences –, is then aligned to a reference sequence, and then the experimental results are derived from the aligned sequences. In each of its main chapters, this thesis contributes to one of those respective stages: Chapter 2 discusses how to remove contaminating so-called adapter sequences from certain types of sequencing data. Chapter 3 describes an index structure that can be used to align reads to a reference that have undergone a chemical modification with sodium bisulfite. Chapter 4 focuses on the analysis steps after read mapping, proposing a computational pipeline for exome sequencing data. Chapter 5 introduces a novel alignment method for data from sequencers that produce their output in the so-called flowgram representation.

1.1 DNA sequencing

DNA sequencing is the process of determining the base sequence of a sample of DNA. It started being used for medical and biological research already around 1977, when the chain-termination sequencing method was invented by Sanger et al. (1977). *Sanger sequencing* was gradually improved over the years and culminated in the sequencing of the human genome with its 3 billion base pairs (Lander et al., 2001), which could only be achieved by implementing a high degree of automation. The Sanger method allows to sequence fragments of up to around 1000 nucleotides (nt), but only a few fragments can be sequenced in a single experiment. Even with automation, sequencing the human genome took years to complete and was a world-wide effort that cost billions of dollars. Sanger sequencing is still an economical method when only a particular, small stretch of DNA needs to be sequenced, but routine sequencing of mammalian – in particular human – genomes has only become possible through the invention of high-throughput DNA sequencing methods, sometimes called *second-generation* or *next-generation* sequencing (NGS) methods.

Modern sequencing methods have in common that they create enormous amounts of data. Depending on the specific technology, millions or even billions of DNA fragments are sequenced in parallel. Each obtained DNA sequence, also called a *read*, has a typical length of currently 35, 50, 100 or 250 nt, resulting in up to 600 Gbp (gigabasepairs) for a single run of a machine such as an Illumina HiSeq 2000.

1.2 Applications

High-throughput sequencing can be used to sequence the entire genome of an organism; this is called *whole-genome sequencing*. The approach is usually to break the DNA of the organism randomly and to sequence the resulting fragments (*shotgun sequencing*). The problem of computationally reconstructing the full, original DNA sequence from the reads is called *de-novo assembly* (Earl et al., 2011). The obtained DNA sequence is not useful in itself, but further computational methods on top of it can then be used to predict the locations of genes, splicing patterns, or to compare different species in order to determine phylogenetic relationships.

In many cases, a known *reference sequence* of the same or a highly related species is available for comparison. Sequencing such an organism is called *re-sequencing*. Its focus is typically on finding differences (such as mutations) between the reference and the sample of interest. Assembling reads is not necessary, but for each read, its likely origin on the reference needs to be determined while allowing differences caused by mutations and sequencing errors; this is called *read mapping*. Read mapping algorithms typically allow only small, highly localized differences such as single-nucleotide substitutions or small insertions or deletions. *Structural variations* are large-scale differences and need to be found with different algorithms (Alkan et al., 2011).

Re-sequencing is not necessarily done for the entire genome, but may be, for example, limited to the *exome* (Bamshad et al., 2011; Martin et al., 2013), which is the set of gene-bearing subregions of the genome. Exome sequencing is in wide use because it can help to efficiently identify the cause of genetic diseases. We discuss this in depth in Chapter 4. Another application is to study DNA methylation; this is explained in detail in Chapter 3.

In addition to DNA, living cells contain other genetic material, in particular *RNA*, which are copies of sections of DNA that usually represent genes. These *transcripts* can be converted to DNA and then sequenced (*transcriptome sequencing*). By counting the reads mapping to each gene, the expression level of each gene can be determined (Schramm et al., 2012).

SmallRNA sequencing focuses on short RNA molecules, of which microRNAs are a subset (Schulte et al., 2010; Rahmann et al., 2013). The problem arising from the fact that microRNAs ($\approx$ 22 nt) are shorter than reads is discussed in Chapter 2.

1.3 DNA

We can only give an extremely brief introduction to the basics of molecular biology here. Please see an introductory text such as that by Alberts et al. (2008) for a much fuller treatment of the subject.

DNA (deoxyribonucleic acid) is a chain molecule, where each element is one of the *nucleotides* adenine, cytosine, guanine and thymine. Nucleotides are connected by phosphodiester bonds. The part of the nucleotide that varies is a *nucleobase* or *base*. The molecule is directional. One front is the $5'$ end and the other is the $3'$ end. The nucleotides adenine–thymine and cytosine–guanine can pair with each other through relatively weak hydrogen bonds. DNA molecules in the

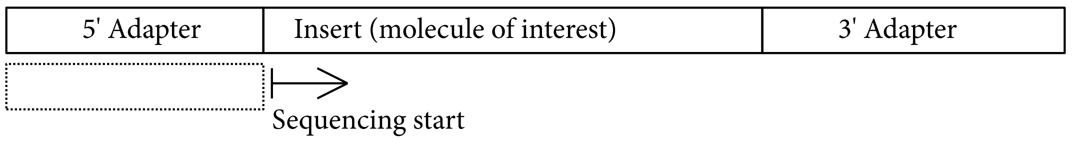


Figure 1.1 The structure of a DNA molecule that has been prepared for sequencing. The rightmost part of the 5′ adapter is the sequencing primer, which determines where the sequencing process begins. The 3′ adapter may contain a variable barcode sequence (see Section 2.1). After attaching the molecule to the surface where sequencing takes place, the molecule is partially double-stranded and also contains the dotted part, which is complementary to the 5′ Adapter.

cell are *double-stranded* most of the time, resulting in the typical helix structure: A second DNA strand, running in the opposite direction, is attached to the first such that all bases are paired. The strands are said to be *reverse complements* of each other.

1.4 Library preparation

The first step in a sequencing experiment is to obtain a tissue sample and purify it such that only DNA remains. We explain the typical subsequent biotechnological steps, which are library preparation and the actual sequencing, in as much detail as necessary to understand the properties of the data that comes out of a high-throughput sequencer.

The procedure of preparing a DNA sample for sequencing is called *library preparation*. Initially, DNA is *fragmented* that is, it is cut into short double-stranded pieces. To each fragment, *adapters* are appended (*ligated*) at both the 5′ and the 3′ end. The adapters are short, double-stranded DNA molecules with a known base sequence. Next, the fragments are copied (*amplified*) with polymerase chain reaction (PCR). Single-stranded (denatured) DNA is then, with the help of the adapters, attached to the location where sequencing takes place. The part of the molecule that we are interested in is also called an *insert*. The structure of a DNA molecule fully prepared for sequencing is shown in Figure 1.1. Adapters can serve other purposes, discussed in Section 2.1.

The base after the 5′ adapter is the first base that is sequenced. The sequencer then proceeds along the insert towards the 3′ end. How exactly the sequencing process is done depends on the specific technology used.

1.5 Sequencing technologies

We describe here the Illumina, SOLiD, 454 and Ion Torrent sequencing technologies. Illumina is currently the most popular one, and many pipelines, including the one we suggest in Chapter 4, are adjusted to work well with its data. Data from the SOLiD, 454 and Ion Torrent instruments, on the other hand, have certain interesting characteristics that require specialized algorithms (see Section 2.4 and Chapter 5).

Other technologies include Pacific Biosciences' Single-Molecule Real-Time (SMRT) sequencing (Eid et al., 2009), which is often used due to its capability of sequencing very long reads (10 000 bp), and Oxford Nanopore's technology (Clarke et al., 2009), which works entirely different from the others but is not yet commercially available.

1.5.1 Pyrosequencing with 454

Pyrosequencing, also *sequencing by synthesis*, was invented by Ronaghi et al. (1996) and later developed further into a highly parallel sequencing technology by Margulies et al. (2005). The technology was commercialized by 454 Life Sciences (now owned by Roche) and is known as *454 sequencing*. It was the first available high-throughput DNA sequencing technology.

Sequencing by synthesis starts from the single-stranded DNA template that is connected to the initial double-stranded adapter. Many copies of the same template are attached to tiny beads that are placed in miniature wells serving as reaction chambers. Each bead yields a single sequenced read. The sequencing process consists of multiple cycles. In each cycle, nucleotides of a single type are flowed over the wells and extend the sequencing primer if the next free bases on the template strand are complementary. The activity of the enzyme that catalyzes this reaction is measured optically through its intermediate release of pyrophosphate. All remaining free nucleotides are then removed and a different type of nucleotide is added.

The intensity of the light is proportional to the number of incorporated nucleotides. This means that not a single nucleotide is sequenced in each step, but a run of the same nucleotide, called a *homopolymer*. Using initial known "key" sequences, the signal intensities are normalized for each read such that an intensity of 1.0 represents the incorporation of a single base. Due to the linearity of the signal, 2.0 represents two bases and so on. This linearity can be maintained up to eight bases on a 454 system (Margulies et al., 2005), but errors already occur at lower intensities.

By measuring the intensity in each cycle, the nucleotide sequence of the entire template DNA fragment can be reconstructed. Achievable read lengths depend on the number of flow cycles and the number of homopolymers in the fragment. The most recent machine generation (GS FLX+) and associated chemistry (Titanium) allow read lengths of up to 1000 bp.

The sequencing results are natively output not as regular DNA sequences, but as so-called *flowgrams*, which associate each nucleotide homopolymer with its measured fractional intensity. Since there are always measurement errors, it is possible, for example, that a nucleotide homopolymer was measured at an "intensity of 2.4". In Chapter 5, we discuss in greater detail the problems arising from ambiguous and incorrect measurements in pyrosequencing and subsequently give a solution for those cases in which a related reference sequence is available.

1.5.2 Ion semiconductor sequencing

In *Ion semiconductor sequencing* (Merriman et al., 2012) by Ion Torrent (now owned by Life Technologies), the incorporation of nucleotides into the growing template strand is not detected optically, but through a change in pH value, which can be measured directly with a semiconductor chip. Apart from this fundamental difference, the obtained reads are of the same type as those by the 454 technology since also homopolymers are sequenced.

1.5.3 Illumina sequencing

The Illumina sequencing technology (Bentley et al., 2008) currently dominates the market. Illumina sequencing takes place on a glass slide called a *flowcell* whose surface is covered with short single-stranded oligonucleotide "stubs" that are reverse-complementary to the initial part of the adapter. The flowcell is divided into eight separated sections called *lanes*, which make it possible to run multiple experiments in parallel, without one sample contaminating the other.

To extend the growing template DNA strand, *reversible terminators* are used, which are modified nucleotides that can attach (hybridize) to the template, but, if they do, prevent the incorporation of further nucleotides (unlike in the 454 technology). The four nucleotides have different colors and can be detected optically. After detection, the remaining free nucleotides are removed and the modification is reversed such that the strand can be extended again in the next cycle. Thus, each cycle results in a single sequenced nucleotide and the cycle count determines the read length, which is fixed for all reads in a run. A typical read length is 100 bp[1] or rather 2×100 bp when paired-end sequencing is used (see Section 1.6).

1.5.4 Colorspace sequencing

While those sequencing technologies described previously determine the sequence of a molecule of DNA by adding single nucleotides to the growing template strand, the SOLiD technology uses *dinucleotides*, that is, pairs of nucleotides (`AA`, `AC`, …, `TT`) to query the sequence (Breu, 2010). Each dinucleotide queries two adjacent nucleotides of the DNA molecule. Through multiple hybridization and de-hybridization cycles, all adjacent groups of two bases of the template are queried. The dinucleotides are color-coded, and the final read is therefore given as a sequence of *colors* in which the i-th color is the result of querying bases i and $i + 1$. Hence, each nucleotide, except the first and last one, is interrogated twice (in different cycles). A nucleotide sequence of length m results in a read that contains $m - 1$ colors.

The colors assigned to the dinucleotides are not unique: Although there are 16 distinct two-character strings, only four colors are used, written as digits 0 to 3. The assignment to dinucleotides was chosen such that an unambiguous reconstruction of the nucleotide sequence from a colorspace read is possible, and to satisfy other requirements, such as that dinucleotide ab should have the same color as ba (Breu, 2010). The assignment is shown in Table 2.2 (p. 35).

Breu also describes the advantages of the encoding: It has, in particular, error-correcting properties that make it possible to distinguish some sequencing errors from real variants. For example, a single-base substitution leads to two adjacent colors being changed in a particular way. If the colors do not change as expected, then the event is likely not a substitution. Also, if only a single color change is observed, this is likely to be a sequencing error.

Some of the special properties of colorspace encoding are discussed further in Section 2.4. Reads from a SOLiD 3 plus sequencer have a fixed length of 35 bp or 50 bp.

1.6 Paired-end sequencing

When the DNA fragment of interest is sequenced from both ends, a pair of reads is obtained. Figure 1.2 gives an overview: The first read is sequenced as usual and its first base therefore is the one following the $5'$ adapter. Then the process is restarted on the complementary strand. This time, the first base that is sequenced is the complement of the last base of the insert. Instead of “first” and “second” we also call the reads *forward* and *reverse* read, respectively. Most often, both reads have the same length (for example, 2×100 bp in current Illumina protocols). The procedure is called *paired-end* sequencing and supported by most HTS technologies.

[1]http://www.illumina.com/systems/hiseq_comparison.ilmn

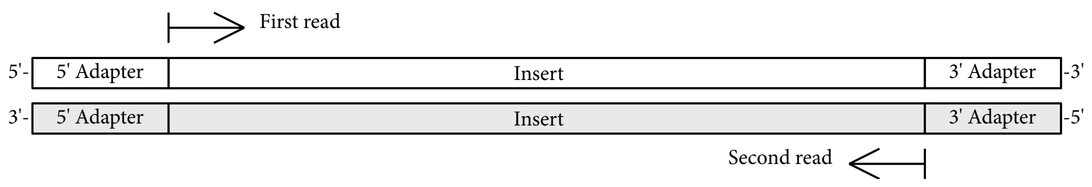


Figure 1.2 In paired-end sequencing, the molecule of interest is sequenced twice. Compared to the first read, the second read is obtained in the reverse-complementary direction. Gray shading denotes the complementary strand.

The size of the insert can be pre-determined (within bounds) during library preparation by adding a size-selection step. This can be done with gel electrophoresis, in which shorter molecules travel farther through a gel to which a voltage has been applied.

Paired-end sequencing mitigates some of the problems that are caused by the relatively short read lengths of high-throughput sequencers. For example, a read mapping algorithm can resolve ambiguous mapping locations by inspecting the mapping location of the second read. Another application is to estimate the locations of structural variations (Marschall et al., 2012).

1.7 Base quality

All sequencing technologies deliver quality values associated with each sequenced base. The values are derived from the physical measurements made during sequencing such as brightness, signal intensity, or density of the DNA molecule spots on the flowcell. Details are usually only known to the manufacturer. Quality typically decreases towards the end of the reads. Quality values can be used for pre-processing to discard low-quality reads entirely or to trim low-quality ends. Quality values are an important tool in assessing whether discovered mutations are real or sequencing errors.

A quality Q is often given as a rounded integer resulting from

$$Q = -10 \cdot \log_{10} p_e\,,$$

where p_e is the estimated probability that the base was sequenced incorrectly. For example, a probability $p_e = 0.001$ results in a quality of $Q = 30$. This convention was introduced with the phred program (Ewing and Green, 1998) and such values are therefore called *phred-scaled* qualities.

Note that $Q = 0$ is not used as an actual quality value: Since the probability that the given base is incorrect is $p_e = 10^{-\frac{Q}{10}} = 10^{0.0} = 1$, we could otherwise deduce that it must be one of the other three possible bases.

1.8 File formats

We describe some of the file formats typically used in high-throughput sequencing experiments and supported by our software where appropriate. A brief description of the *variant call format* (VCF) is postponed to Section 4.2.5.

FASTA

The FASTA format (Pearson and Lipman, 1988) is a text-based format used for storing sequences. Each sequence is introduced by a comment line that starts with the ">" character. This is the beginning of a FASTA file that contains the sequence of the $\Phi X174$ phage:

```
>phiX174
GAGTTTTATCGCTTCCATGACGCAGAAGTTAACACTTTCGGATATTTCTGATGAGTCGAAAAATTATCTT
GATAAAGCAGGAATTACTACTGCTTGTTTACGAATTAAATCGAAGTGGACTGCTGGCGGAAAATGAGAAA
...
```

FASTA files are typically used to store reference sequences; they are often gzip-compressed.

FASTQ

The FASTQ format is based on the FASTA format, but adds the ability to store quality values with each base (Cock et al., 2010). It is therefore the standard format for representing sequencing reads. Each record representing a read and its associated quality values is stored in four consecutive lines of a text file. A single record may look like this:

```
@HWI-ST552:105:C0TPMACXX:6:1101:1241:1970 1:N:0:TGACCA
TTGGGGATAGTCTGGAAAACAGAGAATGAGA
+
CCCFFFFFHHHHHJJGIIIIIJ#########
```

The first line is the read name. The format of the name is not standardized in general, but here, the standard Illumina header is used. It tells us, for example, that the read was sequenced on an instrument named "HWI-ST552" in its 105th run, that the unique flowcell identifier was "C0TPMACXX" and so on. The read itself is in the second line and the quality values are given in the last. Each quality value Q is represented as the ASCII character whose code is $Q + 33$. For example, the run of "#" characters, whose ASCII code is 35, have quality 2.

FASTQ files are often stored gzip-compressed (extension `.fastq.gz`). Paired-end reads are stored in two separate files, where the order of reads in both files must match. That is, the n-th read in the first file belongs to the n-th read in the second file.

The sequence in FASTQ files (and also in FASTA files) may contain the character 'N', which is a wildcard character that represents an unknown nucleotide.

SAM/BAM

The text-based SAM (Sequencing Alignment/Map) format (Li et al., 2009) is the standard for representing aligned reads. Each line in a SAM file is a record that stores information about a single mapped read. The most important attributes of that record are the identifier of the reference to which the read was mapped, the mapping position on that reference, information on how to reconstruct the alignment itself, and a phred-scaled mapping quality. Also, the nucleotide sequence and quality values from the original FASTQ file are stored.

A companion to the human-readable SAM format is the BAM format. It is a compressed, binary representation of the same information and is most often used in practice instead of SAM, not only because it requires less space, but also because it can be indexed for random access to any chromosome position.

1.9 Conventions

We use Σ to denote an alphabet, and $\Sigma_{\mathrm{DNA}} = \{\,\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\,\}$ is the DNA alphabet. A *string* is an element of Σ^*; for $s \in \Sigma^*$, its length is written $|s|$. We use *string* and *sequence* synonymously. We mainly use variables r, s, t for strings, and ℓ, m, n denote their lengths. A string that consists of the character $b \in \Sigma$ repeated ℓ times where ℓ is a non-negative integer is called a *run* or *homopolymer* of length ℓ and written b^ℓ. The substring $s_i, s_{i+1}, \ldots, s_{j-1}, s_j$ of s is written $s_{i\ldots j}$. When $i > j$, then it is the empty string (length zero), denoted as ε. A monospace font is used for literal strings or characters, such as $s = \mathtt{TAACG}$. Given $s = (s_1, \ldots, s_m)$ and $t = (t_1, \ldots, t_n)$, the *concatenation* of s and t, written st, is the string $s_1, \ldots, s_m, t_1, \ldots, t_n$. The power set of a set S is written $\mathcal{P}(S)$.

We use the word *query* synonymously with *read*. A reference sequence is also a *target* sequence. Although they represent chemically different entities, we do not distinguish between nucleotides and (nucleo-)bases.

The term *index* is used in two different meanings. One is the (usually numeric) value that discriminates between different elements of a collection, such as the entries of an array. The other index is a data structure that helps to locate occurrences of items of a specific type in a collection, such as an index of important terms in a book.

The term $[x]$, where x is a predicate, evaluates to one if the predicate is true and zero otherwise (Iversion bracket notation).

Units

We use nt (nucleotides) and bp (basepairs) interchangeably as length units for nucleotide sequences. Values for memory usage are given using the International Electrotechnical Commission's (IEC) binary prefixes. For example, 1 MiB $= 2^{20}$ bytes and 1 GiB $= 2^{30}$ bytes. SI-defined prefixes (k, M, G, etc.) have their usual meaning.

Algorithms

Algorithm descriptions in this work are given in imperative style on a relatively high level in order to ease readability. For example, a description of a single step may be: "Count the number of Cs or Gs in the string depending on the setting of the flag variable." This implies that in the actual implementation there is an implicit *if* statement that checks the flag, a loop that iterates over the characters of the string, and also that the result is stored in a (temporary) variable. Error conditions such as encountering characters other than A, C, G, T in DNA sequences, are ignored in the description, but are handled in the actual software that accompanies each chapter.

Some algorithms are described as so-called *generators*, a feature available in some programming languages (such as Python). A generator resembles a function, but instead of return a building up a list of results and returning the full list, a generator function *yields* each value one by one. When the generator yields, its execution is suspended and control returns to the function caller. Execution is resumed when the caller requests the next value. Generators have the advantage that memory usage is lower (compared to building the full list of results) and that they simplify programming of iterators.

As symbols for bitwise operators, we use $\&$ (*and*), $|$ (*or*), $\oplus$ (*xor*), $\neg$ (*not*), $\ll$ (*shift bits left*), and $\gg$ (*shift bits right*).

1.10 Structure

The chapters of this work loosely follow the order of steps taken when analyzing data from a high-throughput sequencing experiment: Pre-processing, read mapping, and finally some form of analysis that leads to biologically or medically interpretable results.

In Chapter 2, we discuss the removal of adapter sequences, which is an essential pre-processing step required for some types of sequencing data, in particular for smallRNA sequencing.

Chapter 3 discusses how to map a certain type of read to a reference. To study so-called DNA methylation, these reads have undergone a chemical modification with sodium bisulfite, which essentially means that some cytosines have been replaced with thymine. We propose an index data structure that allows to efficiently map those reads to a reference.

Chapter 4 focuses on the analysis steps that occur after read mapping. In this case, we consider data resulting from exome sequencing. First, we describe our implementation of a computational pipeline whose output are lists of mutations in the sequenced samples. Second, a web interface is presented that displays those mutations and allows interactive analysis by medical researchers.

Chapter 5 returns to the low-level task of aligning a read to a DNA reference string. We introduce a novel method for aligning flowgrams from the 454 or Ion Torrent instruments to a DNA sequence, which reduces the information loss otherwise incurred by the conversion of the flowgram to a regular string.

Chapters 2–5 present separate aspects of high-throughput sequencing and were written to largely stand on their own. Each of these chapters therefore concludes with a discussion and suggestions for future work considering that particular aspect alone. However, there are also connections between the four topics that become obvious only after all four of them have been covered. After summarizing the results, we therefore conclude in Chapter 6 with a discussion that includes these connections.

The four main chapters are each accompanied with software that implements the given algorithms. When inspecting the software, refer also to Appendix A.1, which explains where algorithms in the thesis can be found in each tool's source code.

2 Trimming Adapters

For certain applications of high-throughput sequencing, some of the sequenced DNA or RNA molecules are expected to be shorter than the read length. For example, microRNAs (miRNAs), which are small RNAs, are between 20 and 24 bp in length (Hafner et al., 2008), while reads have a length of at least 35 bp. The sequencing process does not stop at the end of such molecules, but continues along the molecule, reaching the adapter. Consequently, the reads that are output contain the sequence of the molecule of interest and also the adapter sequence. Therefore, an essential first task is to find the reads containing adapters and to remove the adapters where they occur. This is called the *adapter trimming problem* or *adapter contamination problem*. Only the relevant part of the read is then passed on to further analysis.

In this chapter, we first describe the variants of the adapter trimming problem known to us and the requirements that an algorithm must fulfill to solve them. We then proceed by suggesting an algorithm based on semiglobal alignment. We discuss some factors that influence sensitivity and specificity of the algorithm. A large section is devoted to the description of how the algorithm can be extended to work with colorspace reads. Finally, we describe the tool *cutadapt* and give a short overview of some of its additional features, followed by a discussion of possible future improvements of the tool and its underlying algorithms.

2.1 Introduction

Any current sequencing technology requires, as part of preparation for the actual sequencing, that an adapter or primer sequence is *ligated* to the DNA fragments of interest. The adapters serve three different purposes (Schiemer, 2011).

First, they allow the fragments to be bound to a certain physical location. For example, in Illumina sequencing, this is the surface of the flowcell. Second, the adapters contain a PCR primer sequence that directs amplification of DNA fragments. This is necessary for all but some experimental sequencing technologies (single molecule sequencing). Third, the adapter may contain a short so-called *barcode* sequence. Illumina barcodes, for example, have a length of 6 bp. They make it possible to sequence multiple samples at the same time within a single lane of the flowcell. Each sample is prepared with an adapter containing a barcode that is different from those of the other samples. The samples are pooled (*multiplexed*) and then sequenced simultaneously. After the fragments themselves have been sequenced, the barcodes are sequenced, which allows assigning each read to the correct sample (*demultiplexing*).

Therefore, usage of adapters is a technical requirement of the sequencing process. The motivation for adapter trimming is that, under some circumstances, the adapter sequence or a part of it may be seen in the sequenced reads. We call this *adapter contamination*. Before using contaminated reads, the adapters need to be found and removed.

2.1.1 Types of adapter contamination

3' adapter contamination

The most common form of adapter contamination occurs when the sequenced insert is shorter than the read. In that case, the read ends with a prefix of the 3′ adapter. We call this 3′ *adapter contamination*. A typical application in which it necessarily occurs is small RNA sequencing. Here, the sequenced molecules are known to be very short.

For example, microRNAs, which are a type of small RNA, have lengths of 20-24 nt. Since even the earliest models of commercially used high-throughput sequencers deliver reads of at least 35 nt, the adapter appears in all reads that contain a microRNA molecule. If one is interested only in microRNAs, this fact can even serve as a quality control measure: If the adapter does not appear, then the read is not from microRNA. Since not all microRNAs have been discovered, yet (this is the case even for humans), this is an advantage as the sequence of the microRNAs does not need to be known in advance.

In other applications, the library preparation includes a step in which the DNA fragments are size-selected to be longer than the reads. This ensures that all bases of each read are usable as data that can be further analyzed. Examples are whole-genome or exome sequencing. Adapters should not appear within reads in such applications, but it can still make sense to search for adapters as a quality control measure.

A 3′ adapter may also occur *within* a read. There are at least two reasons why this happens. First, although the end of the molecule may have been reached with the last base of the 3′ adapter, the sequencer will continue to sequence until it has reached its fixed read length. The called bases beyond the end of the molecule result from noise or perhaps nearby molecules. Second, the known adapter sequence may not be the true adapter sequence, but only a prefix of it, possibly because the vendor considered the full adapter sequence to be proprietary information.

5' adapter contamination

Adapters may also appear in the beginning of a read. For example, RNA immunoprecipitation followed by sequencing (*RIP-seq*) results in such reads (Philippe Loher, personal communication). Briefly, RIP-seq (Zhao et al., 2010) aims to find targets of an RNA-binding protein, that is, those RNA molecules that the protein binds to. The idea is to extract the protein using specific antibodies while it is still bound to the RNA. The proteins are then removed, leaving RNA only, which is then converted to DNA (reverse transcribed) and sequenced.

In RIP-seq, two different sets of adapters can be used. The 5′ and 3′ adapters (primers) are ligated first. Then the adapters for the actual sequencing step are added, resulting in a molecule with two nested adapter pairs (inner and outer). Since the sequencing process starts at the outer 5′ adapter, the inner 5′ adapter is part of the read. We must therefore be able to detect adapters which occur as a prefix of the read.

To find and remove the 3′ adapter in RIP-seq, the two 3′ adapter sequences can simply be concatenated and then found and removed like any other 3′ adapter. Or, since trailing bases after a 3′ adapter match are ignored, one can simply search for the inner 3′ adapter only.

Another application in which we observed 5′ adapter contamination is the bisulfite sequencing protocol as employed by Zeschnigk et al. (2009), see also Section 3.4.3. The protocol occasionally led to the 5′ adapter being ligated more than once (often twice, rarely three times) to the bisulfite-

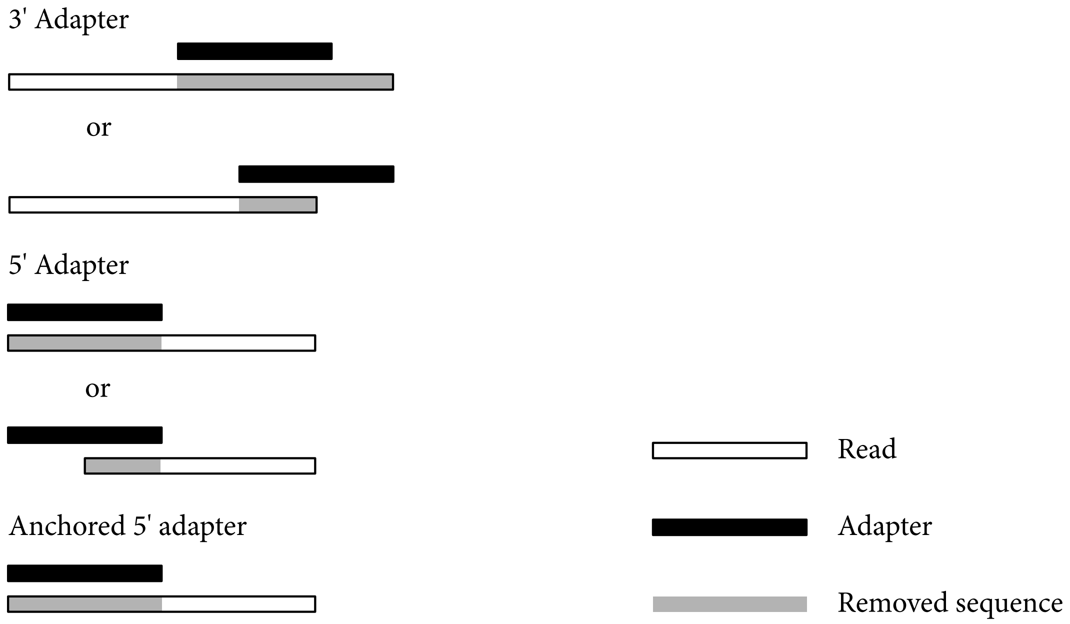


Figure 2.1 Types of adapters, how they appear within reads, and which part of the read needs to be removed if the adapter is found. A *mixed* adapter may appear in any of the shown configurations.

converted DNA molecule. In this case, sequencing will sometimes (randomly) start at the earlier adapter. Again, the adapter is then a prefix of the read.

In the same experiment, we also noted degradation of some 5′ adapters. That is, some adapters seemed to have lost their first nucleotides. In that case, the reads start with a suffix of the adapter sequence.

Mixed adapter contamination

Another observation in the bisulfite experiment was that the location of adapters may be incorrect: Sometimes the 5′ adapter was found at the 3′ end, and sometimes the 3′ adapter was found at the 5′ end. We call this *mixed adapter contamination*.

To summarize, for the purposes of adapter removal, we distinguish four different types of adapters that differ by how the adapter sequence appears in the read and which part of the read needs to be retained, see also Figure 2.1.

- A 3′ *adapter* appears either as a substring of the read, or a prefix of it is a suffix of the read. The sequence preceding the adapter needs to be retained.
- When a *regular* 5′ *adapter* occurs, a suffix of it appears as a prefix of the read. The sequence following it needs to be retained.
- An *anchored* 5′ *adapter* is a special case of the regular 5′ adapter. The entire adapter sequence must appear as a prefix of the read (no degradation allowed). As before, the sequence following it needs to be retained.

- For a *mixed adapter*, it is unknown whether it is a $3'$ adapter or a non-anchored $5'$ adapter. If it is detected to be a $5'$ adapter, the sequence following it needs to be retained and the sequence preceding it otherwise.

A universally usable algorithm must recognize all of the above types of contamination. Since the trimming behavior depends on the adapter type, we can expect the user to know beforehand which type of adapter is to be searched. If the adapter type is unknown, the “mixed” type can be used.

2.1.2 Previous work

The adapter contamination problem is not new and some partial solutions exist. There are at least two standalone tools usable for adapter trimming. *Vectorstrip* is part of the EMBOSS package (Rice et al., 2000). It was originally developed to recognize and remove vector sequence contamination from Sanger sequencing reads, which makes it cumbersome to use in high-throughput sequencing experiments. Vectorstrip does not find partial adapter matches and does not support colorspace (see below and Section 2.4). The program *fastx_clipper*, which is part of the FASTX toolkit[1] by Assaf Gordon, is another command-line tool, but it is also limited to $3'$ adapter contamination and also does not support colorspace data.

Some software libraries, such as *HTSeq*[2] by Simon Anders and Biostrings[3] offer some error-tolerant trimming routines, but HTSeq does not consider insertions and deletions, and both require the user to be able to write their own programs that use those routines.

Also, some read mapping tools, such as SOAP (version 1) by Li et al. (2008b), MAQ by Li et al. (2008a) and Novoalign[4] can trim adapters, but this is only useful if the reads are to be mapped with the respective program. Other popular mapping tools such as BWA (Li and Durbin, 2009) do not support adapter trimming. Such tools can only be used with a stand-alone adapter trimmer such as the one described here.

Furthermore, none of the mentioned tools are universally usable as they do not support all of the different types of adapter contamination that may occur.

2.1.3 Requirements

While the main goal is to find adapters of various types, high-throughput sequencing technologies and different experimental setups add further requirements. We list in this section those requirements that influence the design of the algorithm.

Error tolerance

Sequencing errors can occur anywhere in the read. That is, for each base in the read there is a nonzero probability that it is sequenced incorrectly. Sequencing errors in the insert do not matter for adapter trimming, but some form of error-tolerant pattern matching is needed to allow errors in adapter sequences. Typically, an allowed number of errors is specified by the user. Since the

[1] http://hannonlab.cshl.edu/fastx_toolkit/

[2] http://www-huber.embl.de/users/anders/HTSeq/

[3] http://bioconductor.org/packages/release/bioc/html/Biostrings.html

[4] http://novocraft.com/

adapter matches can have different lengths, this is not helpful because short matches would be allowed to have a greater number of errors relative to their length. Instead, we use an *error rate threshold*, which is the number of errors normalized by the length of the match (see Section 2.2.7).

Indels

The algorithm should be usable with data from a 454 sequencer, whose primary errors come from insertions and deletions within homopolymer runs (stretches of the same nucleotide, see also Chapter 5). One of the first applications of our suggested algorithm was adapter trimming in 454 sequencing data collected by Zeschnigk et al. (2009).

Multiple adapters

The algorithm should be able to search for more than one adapter and to determine which of the adapters (if any) occurs. It will sometimes be the case that a set of adapters is used in an experiment, and at most one of them is expected to occur in a read, for example, when the adapters contain barcodes in multiplexed samples.

A simple case is that multiple $3'$ adapters share a long enough common prefix. This can be handled by searching only for the common prefix. If it is found, the trailing variable sequence should also be removed. If the adapters vary in only a few characters, then it should also be possible to use wildcard characters (see Section 2.5.1).

If the adapters are too different, then it should be possible to specify all adapter sequences and the program must then be able to determine which of the adapters is present.

Colorspace

We would like to be able to trim reads obtained in so-called dinucleotide colorspace from an ABi SOLiD sequencer. For a study by us (Schulte et al., 2010), an ABi SOLiD sequencer was used to obtain short reads of small RNA. SOLiD sequencing is comparatively inexpensive, but yields shorter reads than other technologies. For other applications, this would be a disadvantage, but it is a good choice for small-RNA sequencing. We therefore need to support colorspace reads.

In the following, the algorithm is introduced without considering the colorspace-specific issues. In Section 2.4, we then explain implications arising from the colorspace representation and how to extend the algorithm.

2.2 Finding adapters

The adapter trimming problem is a variant of an approximate pattern matching problem with the additional constraints of being able to 1) find partial matches, 2) find matches below a given error rate threshold and 3) determine which adapter matches best if multiple adapters are searched.

The problem of allowing partial matches is similar to that of overlap detection in shotgun sequence assembly (Gusfield, 1997, Sec. 11.6.4). The alignment algorithm we introduce for adapter trimming is therefore an extension of the *semiglobal alignment* algorithm.

We first summarize the standard global and semiglobal alignment algorithms and then describe the modifications that are made to incorporate the error rate threshold.

2.2.1 Alignments

Let two strings $s, t \in \Sigma^*$ be given. Let $m := |s|$ and $n := |t|$.

Definition 1 (alignment, editing events)**.** An *alignment* between two strings $s, t \in \Sigma^*$ is a finite sequence of pairs (a^s_k, a^t_k), where $a^s_k, a^t_k \in \Sigma \cup \{ - \}$, but not both are $-$. The "$-$" is called *space*. The concatenation of all characters a^s_k and a^t_k that are not spaces must yield the strings s and t, respectively. A pair (a^s_k, a^t_k) in which a^s_k is a space is an *insertion*. If a^t_k is a space, the pair is a *deletion*. If $a^s_k = a^t_k$, it is a *match*. Otherwise, it is called a *mismatch* or *substitution*. Insertions and deletions are collectively called *indels*.

The terms insertion and deletion imply a directionality since it is necessary to define which of the two sequences is the reference sequence into which insertions and from which deletions are made. When measuring relatedness between sequences in an evolutionary sense, the designation is often arbitrary. Here, we know the true, error-free adapter sequence, and all observed editing events are expected to be sequencing errors. Therefore, we consider the adapter sequence to be the reference when discussing indels.

In the following, s is the adapter sequence and t is the read.

2.2.2 Finding optimal alignments

The *cost* of an alignment is the sum of the number of its insertions, deletions and substitutions. The *edit distance* between s and t is the minimum cost over all alignments between s and t (Levenshtein, 1966). An alignment whose cost is equal to the edit distance is an *optimal alignment*.

The edit distance can be found with the following recurrence (see Gusfield (1997) for details). Let s, t, m and n be defined as above and let $d(c_1, c_2) := [c_1 \neq c_2]$ be the unit cost function. $D(i,j)$ is the edit distance between the prefixes $s_{1..i}$ and $t_{1..j}$.

$$D(i,j) = \begin{cases} 0 & \text{for } i = 0 \text{ and } j = 0 \\ i & \text{for } i > 0 \text{ and } j = 0 \\ j & \text{for } i = 0 \text{ and } j > 0 \\ \min \left\{ \begin{array}{l} D(i-1, j-1) + d(s_i, t_j), \\ D(i-1, j) + 1, \\ D(i, j-1) + 1 \end{array} \right\} & \text{for } i > 0 \text{ and } j > 0 \end{cases} \tag{2.1}$$

By using an $(m+1) \times (n+1)$ dynamic programming table for D (also *DP matrix*), the edit distance between s and t is found by evaluating $D(m, n)$ in time $\mathcal{O}(mn)$ (Sellers, 1980). It is not necessary to keep the entire DP matrix in memory since the computations depend only on the previous row and column. Instead, storing a single row or column is sufficient. For the edit distance, memory used in addition to $\mathcal{O}(m + n)$ for the input strings is therefore $\mathcal{O}(\min \{ m, n \})$.

The simplest way to find an optimal alignment, not only the edit distance, is to keep the full DP matrix in memory and to backtrace from cell (m, n) to cell $(0, 0)$. For the backtrace, one needs to determine which of the three terms of the computation of the minimum in Equation (2.1) resulted in the value for the current cell. This can be done by either re-computing the minimum at the time of backtracing or by using a second table with "back pointers" that point to the correct cell (either *above*, *left*, or *diagonally above and left*). The memory usage for this variant of the algorithm is $\mathcal{O}(mn)$.

Hirschberg algorithm

A reduction to linear space usage is possible with the improvement by Hirschberg (1975). The idea is to first find the cell (k, l) in the middle row $k := \lfloor m/2 \rfloor$ through which the backtrace would go. As described in the original paper, this can be done by computing the edit distances between $s_{1..k}$ and prefixes of t, and edit distances between $s_{k+1..m}$ and suffixes of t with both strings reversed. Column index l is set to the length of the prefix of t at which the sum of both edit distances is minimal. By applying the algorithm recursively to the strings $s_{1..k}$ and $t_{1..l}$ and also to $s_{k+1..m}$ and $t_{l+1..n}$, the full alignment can be found (Hirschberg, 1975, Algorithm C).

Finding the cell (k, l) can also be done in the following way. In addition to table D, we maintain a second $(m+1) \times (n+1)$ table O_k (for "origin"), where k is the row we are interested in and $O_k(i, j)$ is set to the largest l' such that an optimal alignment between $s_{1..i}$ and $t_{1..j}$ passes through cell (k, l'). The desired l is found by evaluating $O_k(m, n)$. For $k < i$, the origin $O_k(i, j)$ is undefined since these rows are above k. For $k = i$, we have $O_k(i, j) = j$. For rows $i > k$, the value of $O_k(i, j)$ is propagated from either $O_k(i-1, j-1)$, $O_k(i, j-1)$, or $O_k(i-1, j)$, depending on which of the three terms in Equation (2.1) is minimal. The values in O_k have a meaning similar to the pointers used in backtracing, with the difference that they do not point to the immediately preceding cell, but to a cell in row k.

In order to get the same linear space requirement as the original Hirschberg algorithm, we note that, just as in the computation of edit distance, only a single row of the O_k table needs to be kept in memory at a time. A variant of the algorithm is to maintain an O_l table of which only a single column needs to be kept at a time.

Other cost functions

The function d may also be a non-unit cost function. For example, it may make sense to penalize insertions and deletions more strongly than substitutions in Illumina reads since substitutions are the dominant source of errors in them. Such a "weighted edit distance" would no longer be equal to the number of errors, which makes the results of the algorithm harder to interpret for the user. Also, the idea of basing the algorithm on the concept of an "error rate", which is defined in terms of the number of errors, no longer applies. In addition, it would introduce a further parameter into the algorithm that needs tuning for each dataset. We therefore restrict the algorithm to unit costs only.

Global alignment

The alignment variant described in this section is called *global alignment* since both strings are compared in full (globally). The term *alignment* is used in the literature both for the resulting object according to Definition 1 and also for the process of finding an optimal alignment. When clear from the context what is meant, we will do the same in the following.

Global alignments model a relationship between two strings by assuming that they represent two instances of the same underlying sequence which differ only in some editing events, such as those caused by mutations or sequencing errors. The hypothesis underlying global alignments is

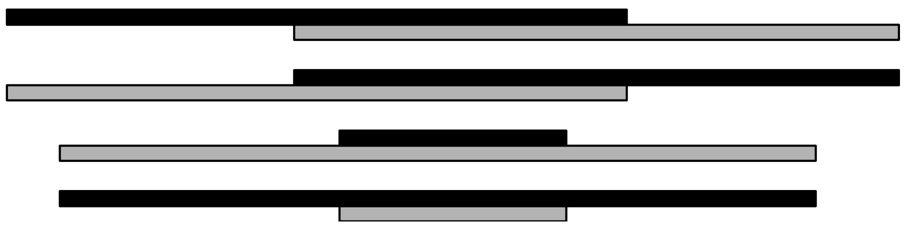


Figure 2.2 Semiglobal alignment between two sequences (marked in gray and black) allows four configurations: Either a prefix of one of the strings is a suffix of the other (top two configurations) or one string is a substring of the other (bottom two). The cases "one string is a prefix or suffix of the other" are not shown as they are special cases of the shown configurations.

that there is a start-to-end correspondence between both strings *s* and *t*; that is, the entire string *t* is an edited version of the entire string *s*.

2.2.3 Semiglobal alignment

To compare two strings that are edited fragments (substrings) of a larger string, we use *semiglobal alignment*. Semiglobal alignment allows two strings to overlap arbitrarily without penalty, while errors within the overlapping region are penalized as in regular alignments. Semiglobal alignment is described, for example, by Gusfield (1997, Chapter 11.6.4). One of the applications mentioned is the assembly of fragments from shotgun sequencing. Such fragments may overlap and when a full genome is to be assembled from them, the overlaps need to be detected.

The relationship between two strings can be: A prefix of one string may be a suffix of the other string or one string may be a substring of the other string. Therefore, there are four possible configurations for overlap between two strings, see Figure 2.2. Semiglobal alignment is also called *free-shift alignment* or *end-space free alignment*.

To achieve the desired effect of allowing arbitrary overlaps, the cost function must be modified such that it 1) can skip an arbitrary-length prefix of either *s* or *t* at no cost; and 2) can skip an arbitrary-length suffix of either *s* or *t* at no cost.

A semiglobal alignment can be considered to be a global alignment with the addition of a second type of space that incurs no cost and can occur only at the ends of the alignment. We call those *free spaces* and write them as "$\sim$". In the alignment, they are exactly those spaces that are paired with the skipped prefixes or skipped suffixes of *s* and *t*. If we allow free skipping of arbitrary-length prefixes and suffixes of both strings at the same, we arrive at the even more general local alignment (Smith and Waterman, 1981), which models that a substring of *s* is related to a substring of *t*.

Similarity scores

Usually, semiglobal alignments are not optimized in terms of cost since this can give meaningless results: By skipping both s and t entirely, an alignment with a cost of zero can always be constructed:

```
OPTIMAL~~~~~~~~~
~~~~~~~ALIGNMENT
```

Since one purpose of semiglobal alignments is to detect overlap, we would intuitively expect the `AL` prefix and suffix of the two strings to be paired (with zero errors). However, the shown alignment also has a cost of zero, and there is no guarantee for the more meaningful variant to be found. If in addition there is an error in the overlapping part, it is certain that the meaningless variant is preferred.

One solution could be to use a *similarity function* instead that assigns positive values to matches and negative values to substitutions, insertions and deletions, and that assigns a value of zero to pairs with free spaces (Gusfield, 1997). Instead of minimizing the sum of distances, an alignment would then be defined to be optimal if it maximizes the sum of similarities. Since matches improve the total similarity while free spaces do not, such an alignment tends to increase the length of its overlapping region unless there are too many differences. In the following, we will not pursue this approach, but instead show how to find useful semiglobal alignments while still using the edit distance. We believe that, in this way, results are easier to interpret since the edit distance is simply the minimal number of editing events. Thus, we suggest that the simplest way for a user to specify which alignments are acceptable and which are not is a measure that is based on the number of errors. In our case, this will be the error rate threshold.

We solve the problem of avoiding optimal, but meaningless alignments later (Section 2.2.6) and continue to describe how certain minimum-cost semiglobal alignments can be computed.

Recurrence equation

Given two strings s and t, and $m = |s|$, $n = |t|$, we use the recurrence adapted from Gusfield (1997, Chapter 11.6.4) to compute the edit distance $D'_{\text{sg}}(i,j)$ between prefixes $s_{1..i}$ and $t_{1..j}$, where a prefix of either s or t (but not a suffix) may be skipped.

$$D'_{\text{sg}}(i,j) = \begin{cases} 0 & \text{for } i = 0 \text{ or } j = 0 \\ \min \left\{ \begin{array}{l} D'_{\text{sg}}(i-1,j-1) + d(s_i,t_j), \\ D'_{\text{sg}}(i-1,j) + 1, \\ D'_{\text{sg}}(i,j-1) + 1 \end{array} \right\} & \text{for } i > 0 \text{ and } j > 0 \end{cases} \tag{2.2}$$

We see that the only difference to Equation (2.1) is to allow free spaces in the beginning by setting $D'_{\text{sg}}(0,j) = 0$ and $D'_{\text{sg}}(i,0) = 0$.

For completeness, we show how to find the full "semiglobal edit distance". In order to allow the skipping of either a suffix of s or t, the minimum of all values in row $i = m$ and column $j = n$ needs to be found:

$$D_{\text{sg}} = \min \left\{ D'_{\text{sg}}(i,j) : j = n, 0 \le i \le m \quad \text{or} \quad i = m, 0 \le j \le n \right\}$$

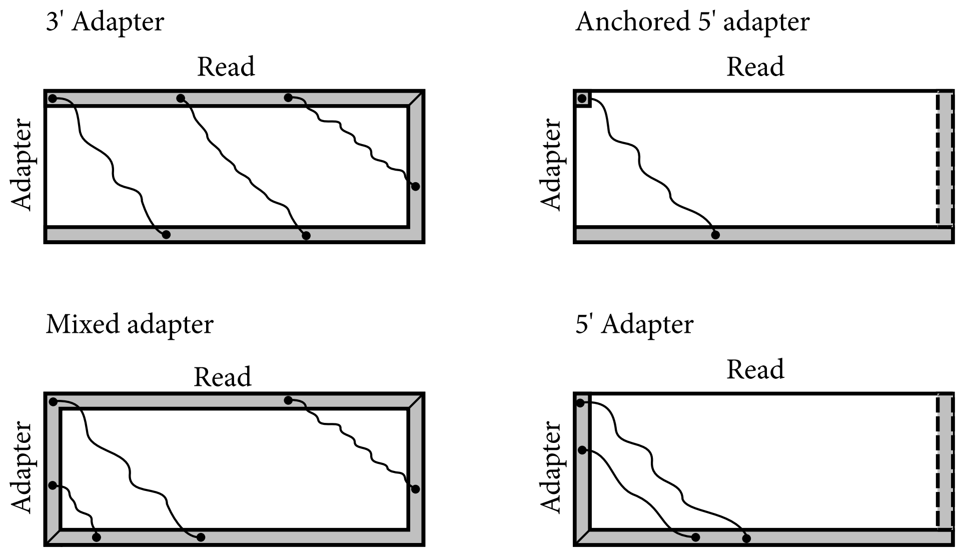


Figure 2.3 Dynamic programming matrices used for adapter alignment. The left and top gray bars indicate prefixes of the adapter and read that may be skipped. The right and bottom gray bars indicate suffixes that may be skipped. The matrix for a mixed adapter is equivalent to regular semiglobal alignment. See text for the dashed columns.

Since $D'_{\mathrm{sg}}(m, 0) = D'_{\mathrm{sg}}(0, n) = 0$ by definition, this is always zero, as discussed, but the same idea works when using a similarity function, replacing D with a similarity function S and min with max.

In the DP matrix for semiglobal alignment (equivalent to the "Mixed adapter" in Figure 2.3), allowing to skip a prefix of s or of t is equivalent to setting all values in the left column or top row, respectively, to zero. Skipping a suffix of s or of t is allowed by searching for an extremum in the bottom row or right column, respectively.

2.2.4 Adapter alignment

Since finding an adapter in a read means that the overlap between adapter and read needs to be detected, we can use semiglobal alignment for it, except that different adapter types need to be taken into account.

The adapter variants differ by which prefixes and suffixes of the read and the adapter may be skipped, see Figure 2.3. When aligning a 3′ adapter, skipping of a prefix is not allowed since the adapter is not expected to be degraded in its 5′ end. Regular 5′ adapters, on the other hand, can be degraded in the 5′ end, but they are not expected to start within the read, therefore no prefix of the read can be skipped. For anchored 5′ adapters, degradation is not allowed so that also no prefix of the adapter can be skipped.

For both 5′ adapter types, whether to allow skipping a suffix of the read is not important. If the right column in the DP matrix is reached (dashed in Figure 2.3), this implies that the read is shorter than the adapter, which should occur rarely and, if it does, the read is trimmed to a length of zero. Since read lengths are either constant (Illumina, SOLiD) or longer than typical adapters (454), it probably does not make sense to search for 5′ adapters that are longer than the read length.

Finally, for mixed adapters, all prefix and suffix skips must be allowed, resulting in a regular semiglobal alignment.

We collectively call these four restricted variants of semiglobal alignment *adapter alignment*. Formally, let $a \in \{\text{three-prime, five-prime, anchored-five-prime, mixed}\}$ be an adapter type. Let $D'_a(i,j) := D(i,j)$, except

- for $a = $ three-prime and $i = 0$, set $D'_a(i,j) := 0$,
- for $a = $ five-prime and $j = 0$, set $D'_a(i,j) := 0$,
- for $a = $ mixed, set $D'_a(i,j) := D'_{\text{sg}}(i,j)$.

Note that $D'_a(i,j)$ only accounts for the skipping of prefixes of s or t. Skipping of suffixes will be taken care of in the final algorithm.

We would like to point out why these restricted types of semiglobal alignment are needed at all. The alternative would be to always use the unrestricted DP matrix of regular semiglobal alignment as it encompasses the other types. In order to incorporate the knowledge we have about the adapter type, we would then need to check the found optimal alignment and ignore it when it does not start and end at the appropriate places. In this way, true matches can be missed. For example, if an anchored $5'$ adapter occurs once in the beginning of a read and once in the middle, but with fewer errors, the middle occurrence will be found and then discarded since it is not in the expected position. By incorporating the restrictions given by the adapter type into the alignment algorithm itself, such suboptimal occurrences are never considered.

2.2.5 Overlap

To find an adapter essentially means to find an optimal overlap between two strings. We define overlap in the following way. Let the sequence $A = \big((a^s_1, a^t_1), (a^s_2, a^t_2), \ldots, (a^s_l, a^t_l)\big)$ with $a^s_k, a^t_k \in \Sigma \cup \{-, \sim\}, k = 1, \ldots, l$ be a semiglobal alignment between s and t. The "$\sim$" represent free spaces.

Definition 2 (Overlap)**.** Let A be an alignment as above. Let $O = \Big((a^s_\alpha, a^t_\alpha), \ldots, (a^s_\beta, a^t_\beta)\Big)$ be a sub-alignment of A where α is the smallest index and β is the largest such that it contains no free spaces. O is called the *overlap*. If such indices do not exist, the overlap is an empty alignment.

Definition 3 (Overlap start and end coordinates)**.** The tuple $(\text{start}_s, \text{start}_t)$ is the *overlap start coordinate* where start_s (start_t) is the index of the leftmost character of s (t) that is not paired with a free space. The tuple $(\text{stop}_s, \text{stop}_t)$ is the *overlap end coordinate* where stop_s (stop_t) is the index of the rightmost character of s (t) that is not paired with a free space.

The strings $s_{\text{start}_s..\text{stop}_s}$ and $t_{\text{start}_t..\text{stop}_t}$ are the parts of s and t that overlap each other. The overlap start and end coordinates give the location of the cells in the dynamic programming matrix at which the overlap starts and ends, respectively. More precisely, the path that corresponds to the overlap starts in cell $(\text{start}_s - 1, \text{start}_t - 1)$ and ends in cell $(\text{stop}_s, \text{stop}_t)$.

Since free spaces are, by definition, not paired with free spaces in a semiglobal alignment, at least one of start_s and start_t is equal to one. Also, $\text{stop}_s = m$ or $\text{stop}_t = n$ or both.

Example 1. Consider the following semiglobal alignment between the strings $s =$ `ADAPTER` and $t =$ `READADPT`:

```
    1234567
~~~~ADAPTER
READAD-PT~~
123456 78
```

Here, the overlap start coordinate is $(\text{start}_s, \text{start}_t) = (1, 5)$, and the overlap end coordinate is $(\text{stop}_s, \text{stop}_t) = (5, 8)$.

2.2.6 Error rate

Since adapters can occur at variable lengths in the read, it makes sense not to count the absolute number of errors in an alignment, but to normalize this number by the length of the match. The "length" can be defined in multiple ways. We consider three variants: It can be the number of alignment columns (the length of the alignment), the number of aligned characters in the read, or the number of aligned characters in the reference (the adapter). Our aim is for each type of sequencing error to increase the error rate by the same amount. That is, when aligning reads `ADOPTER`, `ADPTER` and `ADAPETER` to the `ADAPTER`, the error rate should be $1/7$ in all cases. Hence, the denominator needs to be the number of aligned adapter characters. The other options result in undesired error rates of $1/6$ and $1/8$.

Definition 4. The *error rate* $r(A)$ or simply r is the number of errors in an adapter alignment A, divided by the number of aligned reference characters in the overlap. Letting e be the number of errors in the adapter alignment A, we can write this as

$$r(A) = \frac{e}{\text{stop}_s - \text{start}_s + 1} \,. \tag{2.3}$$

If there are no aligned adapter characters, we define the error rate to be zero.

Example 2. In this $5'$ adapter alignment of the adapter `ADAPTER`, the error rate is $r = 2/6 = 1/3$ since six adapter characters are in the overlap and there is one substitution and one insertion.

```
ADAP-TER~~~~
~RAPETERREAD
```

If the alignment is an anchored $5'$ adapter alignment instead in which the space preceding the read is not free, the error rate will be $r = 3/7$.

2.2.7 Optimization criteria

Intuitively, an alignment between an adapter and a read should be considered to be good when the overlap is large and the number of errors is low. As mentioned previously, optimizing semiglobal alignments in terms of edit distance gives meaningless results. Except for anchored $5'$ (prefix) adapters, this is also the case for adapter alignment (see Figure 2.1) since trivial solutions can be constructed that align the two strings such that there is no overlap, resulting in an edit distance of zero.

Minimizing the error rate instead is not helpful as we get the same results. One remedy could be to special-case the error rate for "no aligned characters" (that is, a denominator of zero) and set it to a value other than zero. Still, this will result in short, meaningless alignments, such as when adapter and read overlap by a single base due to chance.

Having a long overlap therefore conflicts with the aim of having a low error rate. It seems at first that we must solve a multi-objective optimization problem that involves minimizing the error rate while maximizing the overlap. There are at least two ways to deal with such a problem, and both require the introduction of a new parameter: One is to optimize a linear combination of both criteria, which we will not do here. The other is to introduce a threshold for one parameter.

Thresholding the error rate

We introduce a limit for the error rate, the *error rate threshold* ε. This parameter must be chosen by the user according to the observed sequencing error probability, see Section 2.3.1 for details on how this should be done. We use the error rate threshold as a filter in the same way as in the work by Rasmussen et al. (2006) by discarding alignments that exceed the threshold. Similar to k-differences algorithms such as that by Landau et al. (1986), we regard all alignments for which the error rate is below ε to be equally good. This makes sense since we expect a few sequencing errors to occur in each read. As long as the number of errors is not too large, an alignment with fewer errors does not necessarily better reflect reality than one with more.

Maximal number of matches

Among the remaining alignments, we can optimize for overlap length. The overlap length itself is not a good criterion: Given two alignments with the same overlap length, we should pick the one that has fewer errors. Therefore, we subtract the number of errors from the overlap length and maximize that quantity, which is equal to the number of matches in the alignment. We formalize these thoughts as follows. As before, s is the adapter and t is a read, $m = |s|$ and $n = |t|$. Let a be an adapter type.

Definition 5 (Overlap start/end diagonals)**.** A *diagonal* δ in the DP matrix is the set of all cells (i, j) for which $j - i = \delta$. The values for δ range from $-m$ to n.

The *overlap start diagonal* $\mathrm{start}_\delta := \mathrm{start}_t - \mathrm{start}_s$ of an alignment is the diagonal on which its overlap start coordinate is located. Similarly, we define the *overlap end diagonal* $\mathrm{stop}_\delta := \mathrm{stop}_t - \mathrm{stop}_s$.

Since overlaps start and end in the borders of the DP matrix, start and end diagonals can be converted back to coordinates:

$$\mathrm{start}_s = 1 + \max\{\, 0, -\mathrm{start}_\delta \,\} \tag{2.4}$$

$$\mathrm{start}_t = 1 + \max\{\, 0, \mathrm{start}_\delta \,\} \tag{2.5}$$

$$\mathrm{stop}_s = \min\{\, m, n - \mathrm{stop}_\delta \,\} \tag{2.6}$$

$$\mathrm{stop}_t = \min\{\, n, m + \mathrm{stop}_\delta \,\} \tag{2.7}$$

When these formulas result in $\mathrm{start}_t > \mathrm{stop}_t$ or $\mathrm{start}_s > \mathrm{stop}_s$, the overlap is empty.

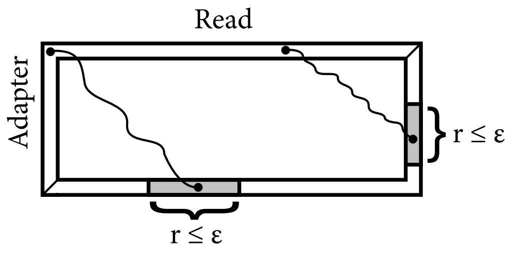


Figure 2.4 Schematic illustration of how the error rate criterion limits alignments to those that end in the shaded region. Two intervals are shown, but there can be fewer or more.

Let us now consider all adapter alignments whose overlap end diagonal is δ. For all four adapter types, δ may range between $-m$ and n, unless we choose not to allow skipping suffixes of $5'$ adapters. Then δ is limited to values between $-m$ and $n - m$.

Let A_δ be an alignment that is optimal (minimum cost) among all adapter alignments that have end diagonal δ. If there are multiple optimal alignments, choose the unique alignment that is "rightmost": That is, when backtracing from the overlap stop, this alignment is the one that prefers to go up over going diagonal and prefers diagonal over left when more than one path is possible. Thus, this alignment also has the maximal overlap start diagonal. Using the edit distance between adapter prefixes $D'_a(i, j)$ as defined in Section 2.2.4, its cost C_δ is

$$C_\delta := \begin{cases} D'_a(m, \delta + m) & \text{for } \delta \le n - m \\ D'_a(n - \delta, n) & \text{otherwise.} \end{cases}$$

In the DP matrix, this corresponds to the values of the edit distance in cells that are in the bottom row (first case) or in the right column (second case).

Let $O_\delta := \text{start}_\delta$ (for "origin") be the overlap start diagonal of A_δ. With Equations 2.3, 2.4, 2.6, we get

$$r(A_\delta) = \frac{C_\delta}{\text{stop}_s - \text{start}_s + 1} = \frac{C_\delta}{\min\{\, m, n - \delta \,\} - \max\{\, 0, -O_\delta \,\}} \,. \tag{2.8}$$

Lemma 1. $r(A_\delta)$ gives the value of the lowest possible error rate among alignments whose overlap ends in δ.

Proof. The numerator of Equation (2.8) is minimal among alignments whose overlap ends in δ by the definition of C_δ. For the denominator, consider that A_δ was chosen to be the rightmost optimal alignment among those that end in δ. Thus, O_δ is maximal among possible overlap start coordinates. Therefore, the denominator is maximal, which guarantees that $r(A_\delta)$ is minimal among all alignments that end in δ. □

Alignment candidates

Taking the error rate threshold into account, an initial set of candidate alignments $\mathcal{A}'$ is:

$$\mathcal{A}' := \{\, A_\delta : r(A_\delta) \le \varepsilon, \delta = -m, \ldots, n \,\}$$

The overlap ends of the elements of this set correspond to intervals in the bottom row and right column of the DP matrix, see Figure 2.4. Let M_δ be the number of matches in alignment A_δ. We restrict the set of alignment candidates further to those for which the number of matches is maximal and get

$$\mathcal{A} := \{ A_\delta \in \mathcal{A}' : M_\delta = \max_d M_d \} .$$

Choosing the final alignment

The set $\mathcal{A}$ can still contain more than one alignment when, for example, an adapter occurs twice in the read with the same number of errors. In the case of 3′ and mixed adapters, we choose the one with the smallest δ in order to remove as much contaminating sequence as possible.

Although multiple adapter occurrences may be rare, some people have reported using our tool for *poly(A) trimming*, which is the following: After transcription of messenger RNA, a so-called *poly(A) tail* consisting of multiple adenines is appended to the molecule before it is translated into protein, which stabilizes the RNA (polyadenylation). By searching for an adapter such as `A...A`, the poly(A) tail can be removed from reads that contain them. By choosing the leftmost occurrence of the `A...A` “adapter” in the read, the user need not concern herself with making sure that the given adapter contains as many `A`s as there are bases in the read.

For 5′ adapters, multiple alignments with the same number of matches are also possible although this is not quite as obvious as in the 3′ case. For example, when the adapter is `AGGGG` and the read is `GGGGGCCCC`, these two alignments both have four matches and an error rate of 20%:

```
AGGGG~~~~~        AGGGG~~~~
-GGGGGCCCC        GGGGGCCCC
```

We do not see a reason for preferring one over the other and therefore also choose, for simplicity, the alignment with the smallest end diagonal (the left one in the example).

Thus, for nonempty $\mathcal{A}$ and all adapter types, let $\Delta := \min \{ \delta : A_\delta \in \mathcal{A} \}$. The final alignment is then A_Δ.

2.2.8 Cutting adapters

Once the value of Δ and the corresponding value O_Δ have been found, overlap coordinates can be computed from Equations 2.4–2.7. The read needs to be trimmed to $t_{\text{stop}_t+1..n}$ for 5′ adapters, and to $t_{1..\text{start}_t-1}$ for 3′ adapters. For mixed adapters, we use the following simple heuristic to decide whether a suffix or prefix needs to be removed: We determine if the adapter occurs in the beginning of the read, that is, whether start_t is equal to one. If that is the case, the adapter is treated as a 5′ adapter and as a 3′ adapter otherwise.

Note that knowledge of the alignment A_Δ itself is not required for correct trimming since overlap start and stop coordinates can be computed from Δ and O_Δ alone.

2.2.9 Adapter alignment algorithm

Let us summarize the alignment algorithm we have so far. With standard dynamic programming, it fills matrix $C := D'_a$, which implicitly gives us C_δ for $\delta \in \{ -m, \ldots, n \}$. With the help of two auxiliary matrices (M and O, see below) values of M_δ and O_δ are found at the same time.

Error rates $r(A_\delta)$ are then computed, and all δ for which $r(A_\delta) > \varepsilon$ are not considered. Then, those δ for which M_δ is maximal are selected. Finally, if there is a tie, the smallest δ is chosen. We describe the auxiliary matrices and time/space improvements before giving the final algorithm in detail (Algorithm 1).

Auxiliary matrices

For a given δ, the values O_δ and M_δ could be found by backtracing in matrix C in time $\mathcal{O}(m+n)$. Since there can also be up to $m+n+1$ values of δ that need to be checked, this is inefficient. Instead, we introduce the two auxiliary matrices M and O, which both have dimension $(m+1) \times (n+1)$ and are updated along with C. $M(i,j)$ is defined to be the number of matches in an optimal alignment between $s_{1\ldots i}$ and $t_{1\ldots j}$. $M(i,j)$ is obtained along with $C(i,j)$ according to the edit operation determined to be optimal in cell (i,j): $M(i,j)$ is set to $M(i-1,j-1)+1$ on a match, to $M(i-1,j-1)$ on a mismatch, and to $M(i,j-1)$ or $M(i-1,j)$ on an insertion or deletion. M is initialized to zero in row $i=0$ and column $j=0$. Note that M is not the length of a longest common subsequence (lcs) since we do not optimize for number of matches. As a counterexample, consider strings ABC and CDE. The optimal alignment between them is $\begin{smallmatrix}\texttt{ABC}\\\texttt{CDE}\end{smallmatrix}$ with zero matches, but the longest common subsequence is “C” at length one.

The second matrix O is the “origin” matrix, where $O(i,j)$ is the maximal overlap start diagonal of an optimal adapter alignment between $s_{1\ldots i}$ and $t_{1\ldots j}$. A cell in this matrix is updated in the same way as M from previous values of O, except that values are only copied, never incremented. The initialization is as follows: For mixed adapter/semiglobal alignments, set $O(i,0) = -i$ and $O(0,j) = j$. For $3'$ adapters and the anchored $5'$ adapter, change the first term to $O(i,0) = 0$. For both $5'$ adapter types, also change the second term to $O(0,j) = 0$.

Single-column optimization

By using the two auxiliary matrices, we have eliminated the need for backtracing in C while finding the best alignment according to our criteria. Finding alignment A_Δ itself, which would still require backtracing, is also not necessary as only the overlap coordinates are needed for correct adapter trimming. Therefore, we can use the standard optimization of keeping only a single column of C in memory at a time. The same optimization is possible for M and O, reducing total space complexity from $\mathcal{O}(mn)$ to $\mathcal{O}(m+n)$. While the memory savings are negligible given typical values of $m \approx 25$ and $n = 100$, the optimization avoids a few memory copies and may result in better cache efficiency.

Note how this linear-space version of the algorithm is related to the variant of the algorithm by Hirschberg (1975) as described in Section 2.2.2. The main differences are that the “origin” is defined differently and that no recursion is required for adapter alignment since we are not interested in the full alignment, only in the start coordinate of the overlap.

Ukkonen's cutoff

Since we use unit costs, we can implement the optimization described by Ukkonen (1985). The idea is that, given an allowed number of errors k, one can stop computing cells in the current column of C when the cost would be larger than k. Although k varies in our algorithm since an error rate is used, we can set it to the maximum value it can attain given the known adapter length: $k := \varepsilon m$. If spaces in the left column are not free, the index of the last cell to be computed (stored in variable *last*) is initialized to k and the expected runtime is $\mathcal{O}(nk)$ as in the original algorithm.

When spaces in the left column are free, *last* needs to be initialized to m, which leads to worse runtime. In the best case, *last* decreases by one per column. Since this means that the full upper left "triangle" of the DP matrix is computed, total runtime is $\mathcal{O}(m^2 + kn)$.

When the alignment is not allowed to start in the top row of the matrix, one further optimization is to not compute columns whose indices exceed $m+k$, which is the maximum length that an alignment with at most k errors can have. Other optimizations are possible, such as not computing cells on diagonals too far away from the zero diagonal (banded alignment), see Section 2.6.1.

Remarks on anchored 5' adapters

For anchored $5'$ adapters, or in general when neither free spaces in the left and top edges of the matrix are allowed, it is not necessary to keep track of the alignment origin, as it is always the top-left cell ($O_\delta = 0$). Also, other improved alignment algorithms such as a variant of Myers bit-vector algorithm (Myers, 1999) could be used. Furthermore, the error rate computations are not necessary for this adapter type since the number of errors is always divided by the full adapter length (if the right edge of the matrix is not free or if reads are required to be long enough).

Both improvements are currently not implemented since the benefit of having a single function that can do all variants of adapter alignments and is therefore easier to maintain have so far outweighed the performance advantages.

To summarize the above thoughts, we give the full adapter alignment algorithm for locating a single adapter of any of the four discussed types within a single read.

Algorithm 1 (Locate-Single-Adapter).
Input: Adapter s (length m); read t (length n); error rate threshold ε; and adapter type.
Output: Overlap start and end coordinates of an optimal adapter alignment; number of errors and number of matches in the alignment.

1. Depending on the adapter type, set the flags *Left-Free*, *Top-Free*, *Right-Free*, *Bottom-Free* to true for those edges of the DP matrix where end spaces are free (see Figure 2.3). (The latter two flags are always true in our case.)

2. Initialize arrays C, M, and O of length $m + 1$ to zero. These arrays represent the current column of the matrices of the same name. If *Left-free* is set, the initialization of O and C is different: They are set to $C[i] = i$ and $O[i] = -i$, respectively.

3. For the cutoff optimization, set k to εm. Set *last* to m if *Left-Free* and to $k + 1$ otherwise. Since k is only an upper bound on the number of errors, whether ε is exceeded for an alignment candidate still needs to be checked below.

4. Keep track of the best alignment found so far within variables *best-cost*, *best-matches*, *best-origin* and *best-end*. *best-origin* is the overlap start diagonal, and *best-end* is the overlap end diagonal. The initialization is as follows: *best-cost* $= C[m]$; *best-matches* $= 0$; *best-origin* $= O[m]$; *best-end* $= -m$.

5. Compute the remaining columns of the matrix and update the *best-*. . . variables in each step by iterating over $j = 1$ to n and doing the following; if *Top-Free* is *not* set, then only iterate to column $\min\{n, m+k\}$:

   a) Set variables *prev-C*, *prev-M* and *prev-O* to $C[0], M[0], O[0]$. When computing cell (i, j), they hold the values $C(i-1, j-1)$, $M(i-1, j-1)$ and $O(i-1, j-1)$, which would otherwise be overwritten since only a single column of the matrix is kept in memory.

   b) If *Top-Free*, set $(C[0], M[0], O[0])$ to $(0, 0, j)$. Set them to $(j, 0, 0)$ otherwise.

   c) Compute cells in column j by iterating from $i = 1$ to *last* and doing the following.

      i. Determine the minimum of *prev-C* $+d(s_i, t_j)$ (match or mismatch), $C[i-1]+1$ (insertion), and $C[j]+1$ (deletion).

      ii. Copy $C[i]$, $M[i]$, $O[i]$ into helper variables.

      iii. Set $C[i]$ to the minimum found above. Depending on where it was found, set $O[i]$ to *prev-O* for a match or mismatch, and to $O[i-1]$ for an insertion (no change for a deletion). If there is a tie, prefer the largest diagonal to make sure that the overlap start of the longest alignment is found.

         In the same way, update $M[i]$, except that the value gets incremented by one if the edit operation is a match.

      iv. Set *prev-C*, *prev-M*, *prev-O* to values of the helper variables.

   d) While $C[\mathit{last}] > k$, decrease *last* by one. If *Free-Bottom* is set and $\mathit{last} = m$, then an alignment candidate ending at (m, j) has been found (see Ukkonen, 1985). Compute the error rate $r = \frac{C[m]}{m - \max\{-O[m], 0\}}$ and update the *best-*. . . variables with appropriate values if a better match is found. A match is better if $r \leq \varepsilon$ and either $M[m] >$ *best-matches* or $M[m] =$ *best-matches* and $C[m] \leq$ *best-cost*.

   e) If *last* is less than m, increment it.

6. When *Free-Right* is set, search for a better match in the last column using the same criteria as in step 5d), but compute the error rate as $r = \frac{C[i]}{i - \max\{-O[i], 0\}}$.

7. If *best-end* is still set to m, check whether the *best-*. . . values represent an alignment whose error rate is below the threshold. Return "no alignment found" if that is not the case. Otherwise, convert *best-origin* and *best-end* to a pair of coordinates and return them along with *best-cost*, and *best-matches*.

2.2.10 Removing one of multiple adapters

For completeness, we give the full algorithm that searches and removes the best matching adapter from a single read.

Algorithm 2 (REMOVE-BEST-ADAPTER).
Input: A read; a list of adapters, their types and associated error rates threshold.
Output: A trimmed read.

1. For each adapter:
    a) Run Locate-Single-Adapter and store the returned values.
2. Determine the adapter that received the largest number of matches ($M_\triangle$). This is assumed to be the best matching adapter.
3. Remove the adapter and any trailing or preceding sequence (according to adapter type) from the read.
4. Return the modified read.

2.3 Reducing false positives and false negatives

While Algorithm 1 is exact in the sense that it finds all occurrences of the adapter sequence below a given error rate threshold, it is still possible to get incorrect results.

Assume the adapter alignment algorithm is used as a binary classifier, that is, it tells us whether a read contains an adapter or not. If a read contains an adapter and the classifier agrees then that is a *true positive* (TP). If the classifier does not, it is a *false negative* (FN). Similarly, *true negatives* (TN) and *false positives* (FP) occur when the read does not contain an adapter.

In adapter alignment, false negatives occur when an adapter occurrence is not found because the error rate is too strict. False positives occur when part of the read randomly matches the adapter sequence. We will discuss both types of errors below.

The *sensitivity* is the ratio $TP/(TP+FN)$, and the specificity is the ratio $TN/(TN+FP)$; higher ratios are better. We first discuss how to achieve good sensitivity.

2.3.1 Choosing the maximum error rate

To improve sensitivity, the number of false negatives must be reduced. This requires that the error rate threshold is set sufficiently high such that not to too many adapters are missed. However, setting it overly large increases false positives. We suggest to compute a good threshold from a given *target sensitivity t*. This is inspired by the "missing prob" parameter in BWA (Li and Durbin, 2009) (command-line parameter `-n`).

The *sequencing error rate* of a set of reads is the estimated overall probability for a sequenced base to be incorrect. It is typically below 5%. Our own measurements on a SOLiD dataset from 2008, using the quality values associated with each sequenced base or color, resulted in 3% error rate and in 0.75% on an Illumina HiSeq 2000 dataset from 2011. Assuming that we know that rate or an estimate of it, we can set the target sensitivity t to a value such as $t = 0.95$ and compute an appropriate error rate threshold ε_t that allows us to reach that sensitivity. We first consider only full adapter occurrences of length m and deal with partial matches in the next section.

Let the random variable X be the number of observed sequencing errors. The probability of seeing at most x sequencing errors in m bases given a sequencing error rate of p is given by the cumulative distribution function of the binomial distribution:

$$P(X \le x) = \sum_{\ell=0}^{x} \binom{m}{\ell} (1-p)^{m-\ell} p^{\ell}$$

Table 2.1 Suggested error rates for different adapter lengths m and sequencing error rates p. For a given p, each table shows, for multiple adapter lengths m, the number of errors k that should be allowed in order to reach a target sensitivity of $t = 0.95$. The columns $\varepsilon_t^{\min}$ and $\varepsilon_t^{\max}$ show the range of error rate thresholds that achieve this. Given values for $\varepsilon_t^{\min}$ are rounded away from zero; values for $\varepsilon_t^{\max}$ are rounded towards zero.

$p = 0.01$

m	k	$\varepsilon_t^{\min}$	$\varepsilon_t^{\max}$
10	1	0.10	0.19
20	1	0.05	0.09
30	1	0.04	0.06
40	2	0.05	0.07

$p = 0.02$

m	k	$\varepsilon_t^{\min}$	$\varepsilon_t^{\max}$
10	1	0.10	0.19
20	2	0.10	0.14
30	2	0.07	0.09
40	2	0.05	0.07

$p = 0.05$

m	k	$\varepsilon_t^{\min}$	$\varepsilon_t^{\max}$
10	2	0.20	0.29
20	3	0.15	0.19
30	4	0.14	0.16
40	4	0.10	0.12

This is equal to the expected sensitivity. We can therefore determine the number of errors k that must be allowed within matches of length m in order to achieve the desired sensitivity t:

$$k = \min\{\, x : P(X \leq x) \geq t, x \geq 0 \,\}\,.$$

This absolute number of errors must be converted to an error rate threshold ε_t that must fulfill $\lfloor \varepsilon_t m \rfloor = k$ in order to allow k errors in matches of length m. We get this range for ε_t:

$$\frac{k}{m} \leq \varepsilon_t < \frac{k+1}{m}$$

Hence, the threshold should be at least $\varepsilon_t^{\min} := \frac{k}{m}$, but note that floating-point precision is an issue here: On a typical machine, $\lfloor 98 \cdot \frac{8}{98} \rfloor$ results in 7, not 8. One can solve this by adding a small constant value to the numerator. The threshold should be at most about $\varepsilon_t^{\max} := \frac{k+0.99}{m}$. Some suggested error rate thresholds that achieve a target sensitivity of $t = 0.95$ in full-length matches are shown in Table 2.1.

Varying sequencing error rates

In many sequencing technologies, the sequencing error rate is position-specific and tends to increase towards the end of the read. Depending on the experimental setup and in particular on the average insert size, we may expect $3'$ adapters to occur in those low-quality parts of the read, too. The converse also applies: When trimming $5'$ adapters, they tend to be within the high-quality regions of the read. The choice of error rate threshold according to Table 2.1 should therefore not depend on the global sequencing error rate, but, if possible, on an estimate of the sequencing error rate within adapter-bearing regions of reads.

2.3.2 Partial-match bias

When we start to take partial matches into account, it becomes relevant that the number of errors in a match can only be an integer. Let ℓ be the match length and assume that $\varepsilon > 0$. The product $\ell\varepsilon$ is the number of errors that would be allowed. Since the actual number of errors in a match is never fractional, effectively $\lfloor \ell\varepsilon \rfloor$ errors are allowed. For $\varepsilon = 0.11$, both functions are shown in

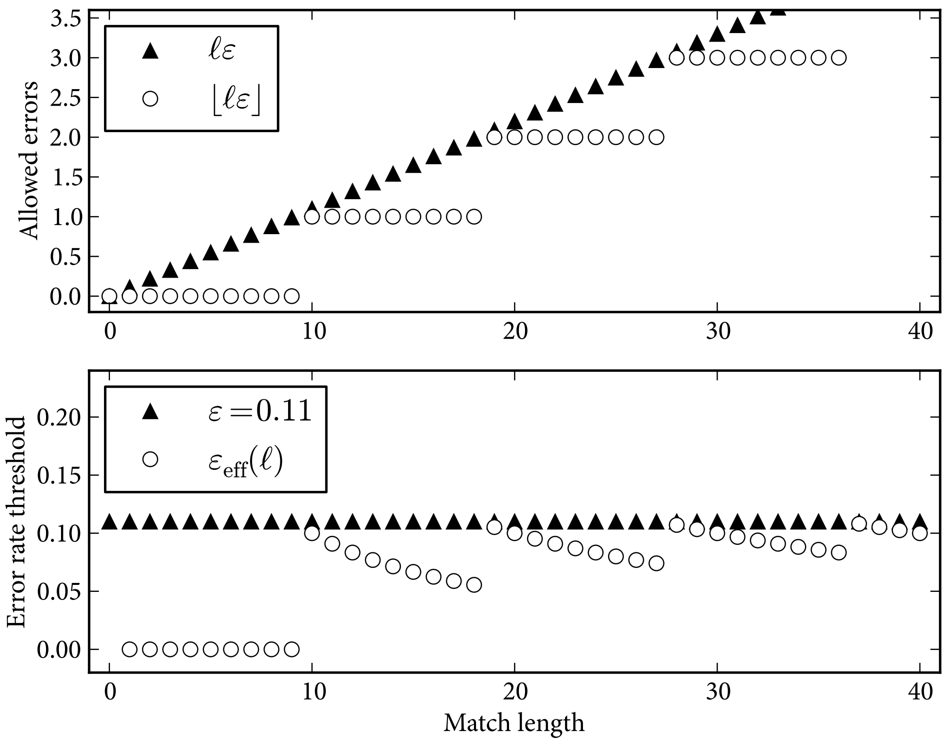


Figure 2.5 For error rate $\varepsilon = 0.11$, the top plot shows the fractional number of errors theoretically allowed vs. the actual value. The bottom plot compares actual and effective error rate.

Figure 2.5.

Let us define the *effective error rate* as $\varepsilon_{\text{eff}}(\ell) := \frac{\lfloor \varepsilon\ell \rfloor}{\ell}$. It varies over different match lengths. The sensitivity is dependent on the effective error rate. As a consequence, fewer matches are found for lower effective error rates.

As we can see in the example for $\varepsilon = 0.11$ shown in Figure 2.5, ε_{eff} is quite close to ε at length 10, but decreases further and further up to length 18. As long as the number of absolute errors remains constant, we suspect that this phenomenon is likely not relevant in practice as the difference is small when going from ℓ to $\ell + 1$.

More of a problem could be those lengths where $\varepsilon_{\text{eff}}(\ell)$ is at a local minimum and then “jumps” to a local maximum at $\varepsilon_{\text{eff}}(\ell + 1)$. That is, where the function allows k errors at length ℓ, but $k + 1$ errors at length $\ell + 1$, such as $\ell = 9$ and $\ell = 18$ in Figure 2.5. As a result, the sensitivity suddenly increases at these transitions. For many applications, this second issue is also not relevant in practice. When trimming reads that come from an mRNA sequencing experiment, the effect is not visible. The only real problem are false positives themselves, not that the sensitivity varies with match length.

However, when there are many partial matches and when one is interested in comparing the number of matches at certain lengths, the bias can influence the results. We describe a specific case that we observed when trimming smallRNA data, as also described by Rahmann et al. (2013).

Bias in a small-RNA sequencing study

In the study by Rahmann et al., microRNA (miRNA) molecules were sequenced on a SOLiD sequencer at read length 35 bp. The most abundant miRNAs in this study have lengths between 20 and 23 nt. Therefore, the match lengths in reads with miRNAs is most often expected to be between 15 and 12 nt, corresponding to 14–11 colors in SOLiD reads.

The adapter has a length of 18 and since raw SOLiD data has a high sequencing error rate, the error rate threshold was initially set to $3/18 \approx 17\%$. Unfortunately, this choice exposes the bias at match lengths 11 vs. 12 since $\lfloor 11 \cdot 0.17 \rfloor = 1$ and $\lfloor 12 \cdot 0.17 \rfloor = 2$. These lengths correspond to miRNAs of 22 and 23 nt. Both are abundant in the sample, but 23 nt-miRNAs were underrepresented due to reduced sensitivity of adapter trimming. The solution in this case was to increase the error rate threshold to 20 %, which results in two errors being allowed for match lengths from 10 to 14, thus covering the range of interesting miRNAs with a constant number of allowed errors. As a result, the number of reads containing 23 nt miRNAs increases by 18 %, while the number of reads trimmed to shorter lengths remains almost unchanged. We conclude that the bias can be reduced by choosing the error rate threshold such that the steps of the floor function are not located within interesting length regions.

While this problem seems to be an inherent problem of the error rate, it is actually a problem of allowing approximate partial matches. Having a constant number of errors over all match lengths is certainly counter-intuitive: While allowing four errors in an adapter of length 40 is appropriate, allowing four errors in a partial match of length 10 is not. Therefore, there must be different numbers of absolute errors and the error rate is one way to interpolate between them.

2.3.3 False positives through random matches

Apart from false negatives, which occur when we miss real adapter occurrences, there can also be false positive matches that happen when an adapter is found although the corresponding sequence is not an adapter. The fewer false positives are found, the greater is the specificity.

Adapter sequences are artificial sequences that we must assume are designed to be different from naturally occurring DNA (at least for those organisms whose genome is known). Also, the chance that an adapter, whose typical length is between 20 and 40 bp, occurs at its full length by chance in a piece of unknown DNA is negligible. We can therefore exclude this type of systematic error error from consideration. On the other hand, when we allow partial matches, we need to be aware that short, random matches do occur. We will in the following give a rough estimate of the number of bases lost due to random matches.

Let us consider a $3'$ adapter first. Assume that no errors are allowed and that the characters of the alphabet Σ are distributed uniformly within reads and adapters. Let $\sigma = |\Sigma|$. The probability of a length-k prefix of the adapter to randomly match a length-k suffix of the read is

$$P(s_{1\dots k} = t_{m-k+1\dots m}) = \sigma^{-k} .$$

The expected number M of matching – and therefore removed – characters, ignoring matches longer than m, is

$$M(m) = \sum_{k=1}^{m} kP(s_{1\dots k} = t_{m-k+1\dots m}) = \sum_{k=1}^{m} k\sigma^{-k} .$$

Before proceeding, we need the following lemma.

Lemma 2. Let $|r| < 1$. Then

$$\sum_{k=1}^{m} kr^k = \frac{r - r^{m+1}(1 + m - rm)}{(1 - r)^2}.$$

Proof. We show this by using

$$r\frac{d}{dr}\sum_{k=1}^{n} r^k = \sum_{k=1}^{n} kr^k$$

and applying the formula for the sum of the first m terms of a geometric series

$$\sum_{k=1}^{m} r^k = \frac{r(1 - r^m)}{1 - r}.$$

□

We let $r = \sigma^{-1}$ in the lemma and get

$$M(m) = \frac{\sigma^{-1} - \sigma^{-m-1}\left(1 + m - \frac{m}{\sigma}\right)}{(1 - \sigma^{-1})^2}$$

To get an upper bound, let m tend to infinity:

$$\lim_{m\to\infty} M(m) = \frac{\sigma^{-1}}{(1 - \sigma^{-1})^2} = \frac{\sigma}{(\sigma - 1)^2}$$

For the DNA alphabet with $\sigma = 4$, the value is $\frac{4}{9}$. That is, one loses approximately 0.44 bases per read on average. For regular 5′ adapters, we get the same result. For a mixed adapter, the number of randomly matching bases is doubled since the adapter can match a prefix or a suffix of the read. If there are multiple adapters, the value $\frac{4}{9}$ must be multiplied by the number of adapters, assuming independence between adapter sequences.

We can see that the error rate threshold ε does not need to be taken into account for realistic values of ε: Even for ε as high as 0.2, at least four bases need to match exactly, which is sufficiently rare to have any discernible effect.

Minimum overlap

While the value of 0.44 bases lost per read is already quite low, we can reduce it even further by introducing a *minimum overlap* parameter o. Any match in which the overlap is shorter than o is ignored. For example, setting o to three reduces the number of bases lost per read to $\frac{4}{9} - \frac{1}{4} - \frac{2}{16} = \frac{10}{144} \approx 0.07$. Note that this will slightly increase the number of false negatives.

Some analysis pipelines may include a step in which so-called *RNA editing* is analyzed, which is a physical process by which RNA molecules are modified after being transcribed from DNA. This needs to be considered when tuning the minimum overlap parameter. When o is too small, false positive matches may result that are detected as spurious “RNA shortening”. However, increasing *o* to avoid this may lead to spurious “RNA extension” being detected for inserts that are slightly below the read length. We suggest that the safest approach is to choose a large value for *o* and to analyze RNA editing only for those reads that are trimmed at all. Obviously, it is also beneficial when the read length can be chosen to be clearly larger than the longest RNA molecule that is to be analyzed, such as when sequencing small RNA.

2.4 Trimming colorspace reads

We show in this section how to trim adapters from colorspace reads (see Section 1.5.4). Since the colors in a colorspace read result from querying two adjacent nucleotides, some effects occurring at the first and last bases of the insert, where a color covers both the insert and an adapter sequence, need to be taken into account.

In the absence of sequencing errors, colorspace reads can be converted to the standard nucleotide representation and then treated in the same way as reads from other sequencing technologies. Sequencing errors, however, propagate along the sequence during the conversion and result in mostly unusable reads. Better results can be achieved by modifying the algorithms such that they can work with colorspace data natively. We show how to extend the adapter trimming algorithm such that it works well with colorspace reads.

As discussed in Section 1.5.4, colorspace sequencing allows to distinguish, to a degree, sequencing errors from true editing events since only certain color change patterns can be observed when an editing event occurs. For adapter trimming, we expect the only source of differences between read and adapter to be sequencing errors. Thus, every color that has changed in comparison to the reference can be counted as a sequencing error, and there is no need to check whether the pattern of color changes corresponds to an allowed one. On the other hand, the sequencing error rate of the SOLiD system is often given in terms of errors after correction. Only then is it comparable to the error rates of other technologies. For adapter trimming, the higher error rate before correction needs to be taken into account when deciding which error rate threshold to use.

Converting strings to and from colorspace

The string of colors one observes when sequencing a DNA fragment with a SOLiD sequencer is called the *colorspace representation* of the read. We call the conversion into colorspace *encoding* and the conversion from colorspace *decoding*. In contrast, a string of nucleotides is sometimes said to be in *nucleotide space*.

To encode a read, Table 2.2 can be used as a look-up table for all dinucleotides of the read. For example, `GGCAG` is encoded into the sequence of colors $0, 3, 1, 2$, which we write as 0312 for short.

This simple mapping is not injective as there are, for a given colorspace sequence, four nucleotide sequences that map to it. For example, `AATGA` also encodes into 0312. The ambiguity is obvious by considering the pigeonhole principle: A nucleotide string of length m is encoded into a colorspace string of length $m - 1$. While there are 4^m nucleotide strings of length m, there are only 4^{m-1} colorspace strings of length $m - 1$.

To resolve the ambiguity and make decoding possible, one of the original nucleotides needs to be known. In SOLiD sequencing, this is the first nucleotide. It is clear that none of the bases of

Table 2.2 Assignment of colors to nucleotide pairs in colorspace encoding. Colors are written as digits 0 to 3. Since the assignment is symmetric, the labels "First" and "Second" may be swapped.

		Second base			
	$\oplus$	A	C	G	T
First base	A	0	1	2	3
	C	1	0	3	2
	G	2	3	0	1
	T	3	2	1	0

the insert are known (otherwise, one would not need to sequence it). Therefore, the sequencing process begins one base earlier. That is, the first queried dinucleotide and therefore the first sequenced color covers the last base of the 5′ sequencing adapter (or primer) (called the *primer base*) and the first base of the insert. When the standard primer is used, the primer base is a T.

Given a dinucleotide ab, we write its encoded color as $a \oplus b$. This notation was chosen because encoding a dinucleotide is equivalent to the bitwise "exclusive or" operation (XOR, $\oplus$) on the two nucleotide characters, assuming that the nucleotides A, C, G, T are encoded as 0, 1, 2, 3, respectively. Breu (2010) introduced the $\oplus$ symbol in his notation, but does not explicitly describe the connection to XOR.

Let x be a DNA fragment $x = (x_1, \dots, x_k)$, $x_i \in \Sigma_{\text{DNA}}$. Let $\Sigma_{\text{color}} := \{0, 1, 2, 3\}$ be the colorspace alphabet. The colorspace representation x^{cs} of x is

$$x^{cs} = (x_1, c_1, c_2, \dots, c_{k-1}) \quad \text{where } c_i \in \Sigma_{\text{color}} \text{ and } c_i = x_i \oplus x_{i+1}.$$

A given instance of a colorspace read such as $(\mathtt{T}, 0, 3, 1, 2)$ is usually written in the abbreviated form "T0312". The T is the primer base. This is also the format in which the read is stored in FASTQ files. We also introduce x^{*cs}, which is equal to x^{cs} without the primer base:

$$x^{*cs} = (c_1, c_2, \dots, c_{k-1}) \quad \text{where } c_i \in \Sigma_{\text{color}}, c_i = x_i \oplus x_{i+1}$$

Decoding

The conversion from colorspace to nucleotide space needs to start at the known nucleotide, in our case in the beginning at x_1. To find x_2, we can search for the row that contains c_1 in the column labeled x_1 in Table 2.2 and then proceed to x_3 and so on. Alternatively, we note that the exclusive or is its own inverse. That is, $(a \oplus b) \oplus b = a$. For the base at index $i + 1$, we get

$$x_{i+1} = (x_i \oplus x_{i+1}) \oplus x_i = c_i \oplus x_i\ .$$

The nucleotide sequence can therefore be computed iteratively from left to right, using the previously decoded nucleotide in the current iteration and starting with the known nucleotide x_1.

2.4.1 Aligning colorspace reads

Since each color in a colorspace read is decoded with the help of the preceding decoded nucleotide, a sequencing error in the form of an incorrect color propagates from the point at which it occurs to the end of the read. That is, all the bases following the incorrect one are also decoded incorrectly (unless a compensating second sequencing error occurs). When attempting to align or map these converted reads to a reference, a run of mismatches is produced that prevents a correct analysis. Therefore, colorspace-aware read mappers such as BFAST (Homer et al., 2009) use the unconverted colorspace read and instead convert the reference to colorspace. Sequencing errors thus appear as mismatches of single colors only. After mapping, the most likely nucleotide sequence of the read is then inferred, guided by the reference (Li et al., 2008a; Homer et al., 2009).

For adapter trimming, we follow the same approach. Thus, to align a colorspace read to an adapter, we first convert the adapter to colorspace and then use the adapter alignment algorithm (Algorithm 1) to find it within the read, except that the cost function is defined on colors instead of nucleotides. It is not necessary to infer a most likely nucleotide sequence since the correct adapter sequence is already known.

While the alignment algorithm remains almost unchanged, care needs to be taken when removing a found adapter, resulting from the way in which string concatenation works in colorspace.

2.4.2 Concatenating strings in colorspace

A read that contains an adapter can be modeled as the concatenation of the string that represents the insert and the string that represents the adapter. Removing an adapter can then be seen as undoing this concatenation. This is the same in both nucleotide- and colorspace, but concatenation works differently in colorspace.

Let $x = (x_1, \dots, x_k)$, $y = (y_1, \dots, y_\ell)$, with $x_i, y_j \in \Sigma_{\text{DNA}}$ and $k, l > 1$. When these two nucleotide-space strings are concatenated and subsequently encoded into colorspace, the resulting string is

$$(xy)^{cs} = (x_1, x_2 \oplus x_3, \dots, x_{k-1} \oplus x_k, x_k \oplus y_1, y_1 \oplus y_2, \dots, y_{\ell-1} \oplus y_\ell) .$$

It contains one color $x_k \oplus y_1$ that belongs to both x and y since it encodes the transition from one string to the other. We call it a *bridge color*.

Example 3. The colorspace representations of the two strings x = `AAAA` and y = `TGTG` are x^{cs} = `A000` and y^{cs} = `T111`, respectively. The concatenated string xy = `AAAATGTG` gets encoded to $(xy)^{cs}$ = `A0003111`. The “3” is the bridge color.

Another example for a bridge color is the first color of each SOLiD read. It encodes the transition from the primer base to the first base of the insert.

Example 4. Sequencing `GGCAG`, preceded by the standard primer base `T`, results in the read `T10312`, where the first 1 is the bridge color.

Bridge colors can cause spurious mismatches when they are not accounted for. Consider a colorspace read of the primer `T` concatenated with the insert. When the read is mapped to a location on the reference in which the sequence of the insert is not preceded by a `T`, there will be a mismatch at that position. See Figure 2.6 for an example. One solution is to discard the first color before mapping the read, as done by BWA (Li and Durbin, 2009). The problem is then that one of the dinucleotides that interrogates the first base of the insert is missing. A substitution of

Read			T		A		C		C		C
Read^{cs}			T	**3**		1		0		0	
Ref.^{cs}	G	0		**2**		1		0		0	
Ref.	G		G		A		C		C		C

Figure 2.6 An example for a spurious mismatch between the colorspace read of the DNA fragment `ACCC` and the reference sequence `GGACCC`. Since the `ACCC` is preceded by a `T` in the read and by a `G` in the reference, the colors differ (bold) and result in the mismatch.

the first base would therefore result in only one color change instead of two. As a consequence, BWA ignores the first base when computing the most likely nucleotide sequence. BWA also does not call the last nucleotide of a read since it is supported by only one color. For BWA, this results in reads that are two bases shorter than the nominal read length given by the manufacturer.

Especially when sequencing short molecules, which is a primary application for adapter removal, discarding even a single base represents a proportionally large loss of data (5% for a microRNA molecule of 20 nt). For adapter removal, we show how to retain as much information as possible. The procedure is different for 5′ and 3′ adapters. We describe the simpler case of removing a 3′ adapter first.

2.4.3 Removing 3′ adapters

Let $t = bus$ be a sequenced fragment (in nucleotide space), where $u = u_1, \ldots, u_k$ is the insert, $b \in \Sigma_{\text{DNA}}$ is the primer base, and s is the adapter. We assume the adapter is a suffix of the read, but the following thoughts hold also if a prefix of it appears or when it is followed by other nucleotides. The sequencer gives us the colorspace read

$$t^{cs} = (b, b \oplus u_1, u_1 \oplus u_2, \ldots, u_{k-1} \oplus u_k, u_k \oplus s_1, s_1 \oplus s_2, \ldots, s_{m-1} \oplus s_m) \,.$$

After adapter removal, the read should look as if the adapter had not been sequenced, that is, it should look as if $t_{\text{trimmed}} = bu$ had been sequenced. The result of adapter trimming must therefore be the colorspace sequence

$$t^{cs}_{\text{trimmed}} = (b, b \oplus u_1, u_1 \oplus u_2, \ldots, u_{k-1} \oplus u_k) \,.$$

In other words, the suffix $(u_k \oplus s_1, s_1 \oplus s_2, \ldots s_{m-1} \oplus s_m)$ of t^{cs} needs to be searched and removed. This is not possible, however, as the bridge color $u_k \oplus s_1$ is unknown since it depends on the unknown base u_k. The simple solution in this case is to search instead for $s^{*cs} = (s_1 \oplus s_2, \ldots s_{m-1} \oplus s_m)$ and then cut the read one base earlier than what the obtained overlap start coordinate indicates. See Figure 2.7 for an example.

Note that the string s^{*cs}, which we search for, has only $m - 1$ colors whereas the adapter has m nucleotides. This is caused by leaving out the bridge color from the search pattern, which could be considered to represent a “free mismatch”. If the color is inconsistent with the transition from u_k to s_1, no error would be detected. Without knowing u_k, this cannot be avoided, but if adapter detection was done as part of the read mapping process, it would be possible, see Section 2.7.2.

The full algorithm is summarized in the following.

Adapter sequence:	330201030313112312
Original read:	$\underline{\text{T3}}$000232100101222222$\underline{\text{3330201030313112}}$
Trimmed read:	000232100101222222

Figure 2.7 Trimming a 3′ colorspace adapter. Here, also the removal of the primer base and the first color is shown, which is necessary if the read is processed further with BWA. The removed characters are underlined.

Algorithm 3 (Cut-3′-Adapter-Colorspace).
Input: Colorspace read t^{cs}; adapter s; error rate threshold ε.
Output: A prefix of t^{cs}.

1. Compute s^{*cs} (the colorspace representation of adapter s without the initial nucleotide s_1).
2. Compute t^{*cs} by removing the primer base from t^{cs}.
3. Search for s^{*cs} within t^{*cs} by calling Locate-Single-Adapter (Algorithm 1) with the adapter type set to "3′ adapter" and the error rate threshold set to ε.
4. If no match was found, return t^{cs} unchanged.
5. Otherwise, set t^{*cs}_{trimmed} to $t^{*cs}_{1\ldots\text{start}_t-2}$. Prepend the primer base and return the result.

Note that the algorithm works correctly for reads in which the insert is empty: The single bridge color between primer and adapter is correctly removed. If that bridge color is also missing due to a deletion, then $\text{start}_t - 2$ gets negative, and, by definition, an empty read is correctly returned.

2.4.4 Removing 5′ adapters

When trimming 5′ adapters in both the anchored or non-anchored case, there are two problems: First, we need to take into account that the first color is a bridge color from the primer into the adapter, not into the insert. Second, to remove the adapter from the read, the two bridge colors between primer/adapter and adapter/insert cannot simply be discarded as when trimming a 3′ adapter, but need to be re-encoded.

Let the sequenced fragment in nucleotide space be $t = bsu = (b, s_1, \ldots, s_m, u_1, \ldots, u_k)$, where b is the primer base, s is the 5′ adapter, and u is a prefix of the insert. In the non-anchored case, s is a suffix of the adapter, but the following considerations still apply. The colorspace read is:

$$t^{cs} = (b, b \oplus s_1, s_1 \oplus s_2, \ldots, s_{m-1} \oplus s_m, s_m \oplus u_1, u_1 \oplus u_2, \ldots, u_{k-1} \oplus u_k) \tag{2.9}$$

This includes the two bridge colors $b \oplus s_1$ and $s_m \oplus u_1$. As before, the trimmed read should be $t_{\text{trimmed}} = (b, u_1, \ldots, u_k)$ in nucleotide space. In colorspace, this is

$$t^{cs}_{\text{trimmed}} = (b, b \oplus u_1, u_1 \oplus u_2, \ldots, u_{k-1} \oplus u_k)\ . \tag{2.10}$$

Let us now consider the search pattern. As in the case of the 3′ adapter, searching for s^{*cs} works, but we can do better since the primer b is known for each read. For anchored 5′ adapters, also s_1 is known and therefore the first bridge color $b \oplus s_1$ can be computed and included in the pattern.

Assuming that we have removed the primer base from both the read and the adapter before alignment, the final search pattern is therefore $(bs)^{*cs} = (b \oplus s_1, s_1 \oplus s_2, \ldots, s_m)$ for anchored $5'$ adapters.

If the adapter is not anchored, the first color in the read t^{cs} is not necessarily $b \oplus s_1$, but $b \oplus s_i$ for some $i \geq 1$. The correct search pattern is $(bs_{i \ldots m})^{*cs}$ for an unknown i. Since i is unknown, we actually get patterns for $i = 1, \ldots, m$ where the pattern for $i+1$ is in general not a suffix of the one for i. Fortunately, we do not need to deal with all patterns separately, but can use semiglobal alignment with the search pattern $(bs)^{*cs}$ as before, but need to alter the cost function in column $j = 1$. At a cell $(i, 1)$ in this column, one would usually compare the first color of the read to the i-th color of the pattern, which is $s_{i-1} \oplus s_i$ for $i > 1$ and $b \oplus s_1$ for $i = 1$. The only adjustment we need to make is to instead compare the first color of the read to $b \oplus s_i$ for cells in column 1, rows $i > 1$. No changes are needed for the other columns.

For how to actually remove the adapter, let us look at the untrimmed read again, copying Equation (2.9):

$$t^{cs} = (b, \boxed{b \oplus s_1}, s_1 \oplus s_2, \ldots, s_{m-1} \oplus s_m, \boxed{s_m \oplus u_1}, u_1 \oplus u_2, \ldots, u_{k-1} \oplus u_k)$$

The boxes mark bridge colors. We get the position of the second box from the adapter alignment step. In the trimmed read (Equation (2.10)), the primer base is as before and also the colors to the right of the second box:

$$t^{cs}_{\text{trimmed}} = (b, \boxed{b \oplus u_1}, u_1 \oplus u_2, \ldots, u_{k-1} \oplus u_k)$$

The box marks a new bridge color $b \oplus u_1$ that does not appear in the original read, but it can be computed since we now know that the second bridge color is $s_m \oplus u_1$. Decoding that gives us u_1 and then encoding yields the new bridge color:

$$b \oplus u_1 = b \oplus (s_m \oplus u_1) \oplus s_m$$

See Figure 2.8 for an example.

Quality values

So far, quality values (see Section 1.7) have not been considered as they must simply be trimmed in the same way as the read that they belong to: For each base or color that is kept, the corresponding quality value must be retained. In the case of $5'$ adapter trimming in colorspace, however, the trimmed read contains a new, computed color that is not part of the original read. The corresponding quality should describe the probability of the dinucleotide that covers the primer base and u_1 being sequenced incorrectly, but that measurement was never made. We suggest to use the quality value associated with the second bridge color $(s_m \oplus u_1)$. While that belongs to a different dinucleotide, both are the result of interrogating one known and the same unknown base.

Algorithm 4 (CUT-5′-ADAPTER-COLORSPACE).
Input: Colorspace read t^{cs}; $5'$ adapter s; adapter type (anchored $5'$ or non-anchored $5'$); error rate threshold ε.
Output: An adapter-trimmed version of t^{cs}.

Read	T		G		T		T		C		A		A		A		A
Read^{cs}	T	$\boxed{1}$		1		0		2		$\boxed{1}$		0		0		0	
Adapter			G		T		T		C								
Adapter^{*cs}				1		0		2									
Trimmed read^{cs}	T	———	———	———	———	3	———	———	———	———	———	0		0		0	

Figure 2.8 Trimming the 5′ adapter GTTC from the colorspace read T11021000. The boxes mark the two bridge colors in the original read. The new bridge color is the single 3 in the last row.

1. Let $b = t_1$ be the primer base of t^{cs}.
2. Set t^{*cs} to t^{cs} without the primer base.
3. Set s' to bs, that is, the adapter prepended by the primer base of the read.
4. Compute $(bs)^{*cs}$.
5. If the adapter is anchored, run Locate-Single-Adapter (Algorithm 1). If the adapter is non-anchored, then run the version of Locate-Single-Adapter instead that uses the modified cost function as described above. The parameters are the adapter type, $(bs)^{*cs}$, t^{*cs} and ε.
6. If no result was returned, return t^{cs} unchanged. Otherwise, set i to $\text{stop}_t + 1$. This is the index of the second bridge color in t^{*cs}.
7. Decode the second bridge color by setting d to $s_m \oplus t_i^{*cs}$.
8. Set t^{cs}_{trimmed} to $(b, b \oplus d, t_i^{*cs}, t_{i+1}^{*cs}, \ldots, t_{n-1}^{*cs})$ and return it.

2.5 Implementation in cutadapt

The above algorithms were implemented in a tool named *cutadapt* (Martin, 2011). The program runs on Ubuntu Linux, Windows and Mac OS X. Cutadapt is written in Python, except for the adapter alignment algorithm, whose prototype was developed in Python but then re-implemented in C as a Python extension module, and again rewritten in Cython[5], which is a Python-like language that is also compiled to C. The tool has been published under the MIT Open Source license at http://code.google.com/p/cutadapt/. It has been downloaded thousands of times and is being successfully used by researchers as part of their sequencing pipelines (van Bakel et al., 2011; Jünemann et al., 2012; Vesely et al., 2012).

2.5.1 Features

Cutadapt was written to be usable in practice. In addition to the adapter trimming algorithms described in this chapter, it contains many features that have not been described so far. They, for example, help the user deal with different input and output formats and further filter and

[5]http://www.cython.org/

modify the reads. Many of the features were added following user feedback, and a few have been contributed in the form of source code patches by the users themselves (marked below).

For increased maintainability and to allow us to deliver production-quality code, all of the features and trimming algorithms are extensively unit-tested. We use the Python unit testing framework *nosetest*[6].

Input and output formats Input to the program can be given in FASTA or FASTQ format or – needed for SOLiD data – as a pair of `.csfasta` and `.qual` files. The output format is either FASTA or FASTQ, depending on whether input data contains quality values or not. Also, the program works with colorspace FASTQ files from the Sequence Read Archive (SRA, Leinonen et al., 2011; Kodama et al., 2012) that contain a fake additional quality in the beginning.

Compressed input and output Any input or output file can be gzip- or bzip2-compressed. Transparent compression or decompression is automatically enabled when a given filename ends in `.gz` or `.bz2`.

Quality values As discussed in Section 1.7, sequencing data contain quality values for each sequenced base. The program can read ASCII-encoded quality values from FASTQ files according to different vendor standards and from SOLiD-specific `.qual` files. The reads are trimmed appropriately and written with correctly trimmed quality values to the output file.

Low-quality bases Using the same algorithm as BWA (Li and Durbin, 2009), low-quality ends of reads can be trimmed before locating adapters.

Discarding reads In some cases, reads that contain adapters should not be analyzed at all. To support this, there is a mode in which a read with an adapter is discarded entirely, instead of being trimmed appropriately. Use of this option requires a more careful choice of the minimum overlap parameter. It should be set to a value larger than its default of three in order to avoid discarding too many reads with random adapter matches.

Allowed lengths Reads that are not within a specified length range after trimming can be discarded.

Double encoding When trimming colorspace reads, cutadapt can produce output compatible with MAQ (Li et al., 2008a) and BWA (Li and Durbin, 2009). These tools require FASTQ files in which the colors are not encoded by the digits 0–3, but by the letters ACGT (so-called double encoding), and in which the primer base and the first color are removed.

Repeated search Due to technical problems, it is possible that, at library preparation time, an adapter is added to each DNA fragment multiple times. If it is a $5'$ adapter, only the leftmost occurrence would be found and removed. If it is a $3'$ adapter, but occurs only in a degraded form, only the rightmost occurrence would be removed. We allow dealing with such reads by searching and removing adapters repeatedly until either no adapter can be found anymore or until a maximum number of iterations is reached. The number of iterations is set to one by default.

[6]http://readthedocs.org/docs/nose/

Wildcard characters Both the adapter and the read may contain 'N' characters, which are wildcard characters that match any character at no cost. This is useful when sets of adapters are used that differ in only a few characters, for example when sequencing barcoded fragments. (This feature was contributed externally.)

For completeness, we point out two differences between the implementation available online and the work described in this chapter: First, non-anchored $5'$ adapters are also allowed to start *within* reads, that is, in contrast to Figure 2.1, the DP matrix for them is the same as for mixed adapters (semiglobal alignment). This behavior is currently retained only for backwards compatibility and will be changed in a future version of the tool. The second difference is that the part of Algorithm 4 where a modified cost function is used to trim non-anchored $5'$ adapters is not implemented.

2.5.2 Performance evaluation

The focus while developing cutadapt was foremost on correctness, usability, maintainability and feature-richness. Choosing Python as the programming language facilitates reaching those goals, but since Python is interpreted, there is some loss of performance compared to an implementation in a lower-level language. To partially mitigate that loss, the alignment function is kept in a compiled *Python extension module* that contains machine-level instructions. Calling the function incurs the same overhead as calling a Python function, but the execution of the function itself is as fast as it would be for any other function implemented in a compiled language such as C.

In theory, adapter trimming with cutadapt is dominated by the time needed to compute alignments, which is $\mathcal{O}(NM)$, where N is the total number of the characters in all reads and M is the total number of characters in all adapters. For those adapter types where the "Ukkonen cutoff" (stopping the calculation of a column when the number of encountered errors is too large) is applicable, a better estimate is $\mathcal{O}(N\varepsilon M)$ expected runtime. As we will see below, however, time spent aligning adapters to reads takes up only a fraction of total processing time. As first anecdotal evidence for that, consider the addition of the Ukkonen cutoff parameter to the algorithm. Compared to a version of the alignment algorithm that computes the full alignment matrix, the runtime of the alignment algorithm itself was sped up by more than a factor of three, but the total runtime decreased by only 30%.

Software

Our benchmarks were done on a single core of an Intel Core 2 Quad processor (Q9400) running at 2.66 GHz with Python 2.7 under a 64 bit Ubuntu Linux 11.04. The extension module was compiled with Cython 0.17.2 to C code, which in turn was compiled to a shared library (".so", *shared object*) with GCC 4.5.2. The cutadapt version is 1.3.

Improving runtime

As a runtime heuristic, the implementation does initially not run the alignment procedure, but searches for an exact match of the adapter sequence within the read first. If a match is found, the alignment procedure can then be skipped. In many real-world datasets, the error rate is low and adapters often occur in full. Also, since a fast, built-in function is used (the `find()` method of string objects) for the exact search, this results in noticeable speedups for typical datasets.

Datasets

We evaluate cutadapt's performance on both simulated and real datasets. For the simulated datasets, we use the randomly generated adapter sequence `GCCTAACTTCTTAGACTGCCTTAAGGACGT` (length 30).

Simulated We generate a FASTQ file with 10 million 100 bp reads sampled randomly from human chromosome 1. The quality value for each base is set to "I", which corresponds to quality 40 in the standard encoding. The quality value is not used, but required for the FASTQ file format.

SimulatedAdapt0.0 Into 50% of the reads of the above file, we insert an adapter sequence. The start position of an adapter is chosen randomly from $\{ 1, \ldots, 100 \}$ (using a uniform distribution). After the adapter is inserted, the read is truncated to length 100.

SimulatedAdapt0.1, SimulatedAdapt0.2 These two datasets are generated in the same way from *Simulated*, except that each base in the adapter was mutated before insertion with a probability of 0.1 or 0.2, respectively.

Exome We obtained dataset SRR636532 from the Sequence Read Archive. This is a paired-end exome sequencing dataset published as part of a study by Harbour et al. (2013), which we further describe in Section 4.5.3. We use here only the forward read of the first 10 million read pairs. Each read has a length of 101 bp. This is an example of a dataset where we expect little adapter contamination. Systematically trying out all Illumina adapters known to us, we determined that the adapter that should be removed is the Illumina TruSeq adapter `AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC`.

SmallRNA This dataset stems from a smallRNA sequencing experiment ("sequencing of small RNA from the H1 cell line"). It was picked randomly from those smallRNA sequencing datasets available in SRA that also provide the adapter sequence in their description. Single-end reads of length 43 bp were obtained on an Illumina Genome Analyzer II.[7] Reads are of comparatively low quality. For example, 10% contain at least one `N` character. As above, only the first 10 million reads are used.

Colorspace This is a smallRNA sequencing dataset of a neuroblastoma patient and chosen because it is an example for adapter-containing colorspace reads. The dataset has previously been used in two of our studies (Rahmann et al., 2013; Schulte et al., 2010). It is available under accession SRR029974 from the SRA. The length of the single-end reads is 35 bp. Again, only the first 10 million reads are used. The adapter sequence is `CGCCTTGGCCGTACAGCAG`, which is `330201030313112312` in colorspace.

Table 2.3 Benchmark results for 5′ adapter trimming on simulated and real datasets. All datasets contain 10 million reads. Runtimes were measured as the minimum of six runs.

		Total	Trimmed			Time
Dataset	ε	Bases (Mbp)	Reads (%)	Bases (Mbp)	Bases (%)	Read (µs)
Simulated	0.1	1 000	1.82	0.6	0.06	32
SimulatedAdapt0.0	0.1	1 000	49.91	252.7	25.27	33
SimulatedAdapt0.1	0.1	1 000	32.07	162.8	16.28	37
SimulatedAdapt0.1	0.2	1 000	49.16	246.5	24.65	41
SimulatedAdapt0.2	0.1	1 000	9.00	33.0	3.30	33
SimulatedAdapt0.2	0.2	1 000	33.23	158.4	15.84	39
Exome	0.1	1 010	4.97	6.5	0.65	35
SmallRNA	0.1	430	75.73	135.1	31.42	38
SmallRNA	0.2	430	82.09	145.6	33.86	42
Colorspace	0.1	350	45.38	66.5	19.01	34

Results

Results of running the cutadapt program on all datasets are shown in Table 2.3. The minimum overlap parameter was set to 3 and we see that almost no adapters are trimmed in the simulated dataset without adapters. The predicted value of 0.07 bases lost per read (Section 2.3.3) matches the observed value of 0.06% of bases per read. Comparing the percentage of trimmed reads in dataset SimulatedAdapt0.1 for $\varepsilon = 0.1$ and $\varepsilon = 0.2$ confirms that the error rate threshold needs to be higher than the actual sequencing error rate in order for the program to find most adapters. Also, the percentage of trimmed bases is as expected (25%) only for SimulatedAdapt0.0 with $\varepsilon = 0.1$ and SimulatedAdapt0.1 with $\varepsilon = 0.2$. The modest increase in the percentage of trimmed bases in the SmallRNA dataset going from $\varepsilon = 0.1$ to $\varepsilon = 0.2$ suggests that a further increase of ε will likely not improve results any further. The difference in the percentage of trimmed reads between the SmallRNA ($\varepsilon = 0.1$) and Colorspace datasets suggests that enrichment of small RNA molecules was less effective in the latter. Runtimes for $\varepsilon = 0.1$ are relatively constant at values around 35 µs and increase to slightly above 40 µs for $\varepsilon = 0.2$.

Not shown in this table are the runtimes of the adapter alignment function itself, which is about 7 µs and therefore represents only a fifth of the total runtime. That is, about 20% is spent within compiled C code on computing alignments, and the remaining CPU time is spent within Python on parsing the input file, calculating statistics, and writing output files. For comparison, note that mapping a read of length 100 with BWA takes 400 µs on average on the same machine.

2.6 Future work

We consider two types of improvement to the algorithm and to the program that warrant future research. The first is performance improvement, the second is the addition of more features, in

[7] See http://www.ncbi.nlm.nih.gov/sra/SRX007166 for more details.

particular support for paired-end reads. There are also other feature requests, but most of them entail simple implementation or engineering work, such as support for multi-threading or other file formats. We will not discuss those here; see the issue tracker at http://code.google.com/p/cutadapt/issues/list for a current list.

2.6.1 Performance improvements

Since the current performance bottleneck is the interpretation of the code by the Python interpreter, an obvious way to speed up execution is to re-write the essential parts of the tool in a non-interpreted language. At that point, the alignment algorithm itself may become the bottleneck and alternatives may be sought for. It is likely that the best improvements can be achieved by using separate algorithms for some adapter types. For example, an anchored 5′ adapter will benefit from banded alignment, in which cells in the matrix that are too far from the diagonal are skipped (proposed, among others, by Fickett, 1984).

Another option is to use a bitparallel alignment algorithm (Myers, 1999). Since typical adapters are shorter than available register widths of 64 bits, there is not even a need to split a too long pattern into multiple registers. This limits the implementation complexity and reduces runtime overhead. Another bitparallel alternative is a Shift-And algorithm that allows approximate matches (Wu and Manber, 1992). For adapter alignment, these algorithms cannot be used unchanged. We must take partial matches and the error rate threshold into account. Partial matches, at least for 3′ adapters, are straightforward to handle, but the error rate poses a problem since the overlap start coordinates need to be tracked and the overhead for that could dwarf the benefits gained from the faster algorithm.

In general, any approximate pattern matching algorithm that slides a window from left to right over the text and searches within that window from left to right may potentially be used for partial 3′ adapter matching (Navarro and Raffinot, 2002, some of the approaches in Chapter 6 apply). For 5′ adapters, the same algorithms could search on the reverse read.

When it is of importance that many adapters can be searched for efficiently, more complicated data structures may be appropriate, such as an error-tolerant version of the Aho-Corasick algorithm (Aho and Corasick, 1975).

2.6.2 Paired-end reads

The most often requested feature for cutadapt is better support for trimming of paired-end reads (see Section 1.6). We will explain in this section how paired-end reads are currently supported in the tool and what could be done to get better results.

As with single reads, adapter contamination is possible in paired-end reads if the insert is shorter than the read length. In the forward read, the 3′ adapter appears towards the end of the read, whereas in the reverse read, the reverse complement of the 5′ adapter appears, also towards the end of the read (see Figure 1.2).

Since paired-end reads are stored in two separate FASTQ files, the cutadapt tool can currently deal with paired-end reads by trimming both files separately. First, the forward reads are trimmed with the 3′ adapter and with the adapter type set to "3′ adapter". Second, the reverse reads are trimmed with the reverse complement of the 5′ adapter and again the adapter type is set to "3′ adapter".

While this approach works, it ignores some of the information that is available: For inserts shorter than the read length, not only do adapter sequences appear in both reads, but also the insert is the same in both reads. Let ℓ_1 and ℓ_2 be the respective read lengths and L be the insert length. Assume for the moment that $L \geq \ell_1$, $L \geq \ell_2$, that is, no adapters occur. If $\ell_1 + \ell_2 > L$, then there is at least one base that is sequenced twice. More precisely, the suffixes of length $\ell_1 + \ell_2 - L$ of both reads come from the same part of the insert. This also means that the forward read and the reverse complement of the second read overlap. The tool FLASH by Magoč and Salzberg (2011) uses this fact: It detects overlapping regions and merges the reads into single, longer reads, which in turn improves the output from assembly algorithms.

For adapter trimming, paired-end reads with adapters ($L < \ell_1$ or $L < \ell_2$) could reduce false positives matches. The idea is that there are now more conditions that can be checked to assert that an adapter is found. Previously, the criterion (for a $3'$ adapter) was that the adapter must appear as a suffix of the read. For paired-end reads, the criteria are: 1) The first ($3'$) adapter must be a suffix of the forward read; 2) the second (reverse-complemented $5'$) adapter must be a suffix of the reverse-complemented reverse read; 3) the part preceding the adapter in the forward read must be the same as the part following the adapter in the reverse-complemented reverse read. All conditions can be checked error-tolerantly at the same time by using a single DP matrix as shown in Figure 2.9. The idea is that we compute a semiglobal alignment of two sequences: The first is the forward read (“First read” in the figure) concatenated to the reverse-complemented $5'$ adapter (“Reverse adapter”), and the second is the reverse-complemented reverse read (“Second read”) followed by the $3'$ adapter. We need to allow free spaces in the left column and in the bottom row, and, as previously, we search for an alignment that is as long as possible but has few errors. There are four regions in the matrix. The bottom-right one aligns the forward adapter to the first read, and the top-left one aligns the reverse adapter to the second read. This covers conditions 1) and 2). The bottom-left region aligns a prefix of read 1 to a suffix of read 2. Together with the fact that we allow only a single path, this covers condition 3). The top-right region can be ignored and does not need to be computed.

In principle, this method gives us all the information needed to properly trim both reads while being more specific than trimming both reads separately.

2.7 Discussion

Adapter trimming must accurately model the underlying sequencing process in order for subsequent steps in a high-throughput sequencing pipeline to give correct results. The algorithms presented in this chapter do this, and their implementation within cutadapt is faster than read mapping by an order of magnitude. In the following, we discuss alternative approaches of solving the adapter trimming problem in practice.

2.7.1 The error rate threshold

We argued that the error rate threshold is a natural way for users to specify how error-prone they believe their data to be. This is supported by the fact that almost none of the hundreds of messages regarding cutadapt that the author received express a misunderstanding regarding the error rate. On the other hand, the error rate threshold causes the bias described in Section 2.3.2.

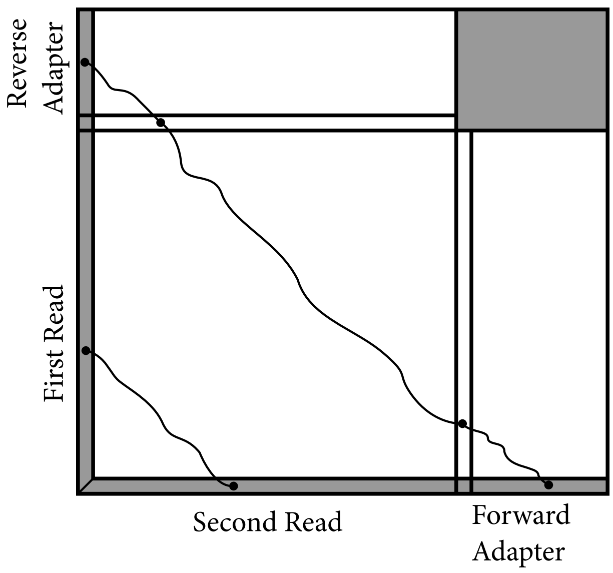


Figure 2.9 Matrix for aligning paired-end reads that may contain adapters. The “Reverse adapter” denotes the reverse-complemented 5′ adapter. The “Second read” denotes the reverse-complemented reverse read. The longer of the two paths represents an alignment that correctly identifies where the reverse adapter is in the second read (top-left region), where the the two reads overlap (bottom-left region) and where the forward adapter is in the first read (bottom-right region). The shorter path represents an alignment in which no adapter was found, but where the reads overlap. The shaded top-right region is not computed.

For those use cases in which that bias may be important, a different model based on scores might work better.

Computation of the error rate also requires maintaining the origin matrix O, which may not need to be done when scores are used. The matrix is also needed to know where to trim the read, but that could be solved for 3′ adapters by searching for the reverse of the 3′ adapter in the reverse of the read; then only the overlap end coordinate, which is available without the O matrix, is needed.

2.7.2 Combining trimming and mapping

Adapter trimming and read mapping are currently modeled as two separate problems. It may be beneficial to treat them as one. Consider the case of spurious matches of a 3′ or non-anchored 5′ adapter. Currently, when the last few bases of the read match the adapter, there is no way to decide whether that is a real match or due to chance. The minimum overlap parameter also does not help to make that decision, it only reduces the overall number of false positives.

A much better criterion would be available if the location that the read maps to on a reference genome were known: If the last few bases match the adapter, but not the reference, then a true adapter is likely present. The minimum overlap could therefore be lowered, increasing sensitivity.

An existing mapping algorithm would need to be modified to allow simultaneous alignment to the reference and to the adapter, for example by appending the adapters as additional reference sequences and then allowing a single “jump”, that is, a long deletion, at little or no penalty to the beginning of a 3′ adapter. This splits the alignment into one part that is on the regular reference and one part that aligns a suffix of the read to a prefix of the adapter. Read mappers that find local alignments, such as BWA-SW by Li and Durbin (2010), and those used for split-read mapping such as *splitseek* by Ameur et al. (2010) could be used for that purpose, except that they need to be modified to ensure that the alignment to the adapter starts with its first base. A similar method would work for non-anchored 5′ adapters.

A problem is that we cannot expect every read mapper to include adapter trimming routines as flexible as those in a tool such as cutadapt. One only needs to consider that the command-line interface to cutadapt currently has 33 distinct options, and that it is a valuable analysis tool by itself. A worthy goal therefore is to define a standard way of exchanging information between the processing pipelines. In the future, a trimming tool would then be able to mark the found adapters, for example by writing the characters in lower-case in the FASTQ file or by using *soft clipping* in a SAM/BAM file (Li et al., 2009), and would not actually remove them. Instead, the read mapper would then decide whether to cut.

3 Mapping Bisulfite Sequencing Reads with a *q*-Gram Index

In addition to the nucleobases adenine, cytosine, guanine and thymine, DNA in some organisms – and vertebrates in particular – may contain methylated cytosine, which influences gene expression and is associated, for example, with gene silencing and genomic imprinting. To study methylation patterns at nucleotide resolution, the method of *bisulfite sequencing* has been developed twenty years ago. Treatment with sodium bisulfite changes cytosines depending on their methylation state. Sequencing the modified DNA and then comparing it to the unmodified reference yields the desired information. High-throughput sequencing technologies have allowed to investigate methylation of ever larger amounts of DNA, up to the level of whole-genome studies.

A crucial algorithmic step in any bisulfite sequencing pipeline is that of mapping the bisulfite-modified reads to the reference genome. In this chapter, we will propose one such method, developed for a study comparing DNA methylation between human blood and sperm samples in a subset of DNA regions called *CpG islands*.

The algorithms have been implemented within the Verjinxer (versatile Java-based indexer) software framework, which is available from http://code.google.com/p/verjinxer/. Verjinxer was originally written by S. Rahmann, and has been heavily extended by us while we added bisulfite mapping capabilities. The software is dual-licensed under the GNU General Public License and the Artistic License.

3.1 Introduction

We discuss methylation in the biological context, describe the bisulfite sequencing method and introduce the *q*-gram index for use in read mapping.

3.1.1 Biological background

A nucleotide is said to be methylated if a methyl group (CH_3) is attached to it. Methylation has different functions. In bacteria, for example, the mechanism that detects and corrects copying errors during DNA replication relies on methylated adenines to distinguish the original from the copied strand.

The prevalent form of methylation in vertebrates is cytosine methylation. In the following, we will consider only this form of methylation. Since the methyl group – if present – is attached to the fifth carbon atom, a methylated cytosine is called *5-methylcytosine*.

Methylation in vertebrates occurs almost exclusively at cytosines that are followed by a guanine. The sequence "CG" is also called a *CpG dinucleotide*, where the "p" represents the phosphodiester bond between C and G, clarifying that base-pairing (involving hydrogen bonds) is *not* meant. Since CG is reverse-complementary to itself, there is, for each methylcytosine, also a cytosine on the opposite strand. Enzymes such as DNMT1 in mammals have a *maintenance methyltransferase activity*, which means that they methylate one strand if they detect methylation on the other. Most of the time, therefore, both cytosines are methylated.

Methylated cytosines influence gene expression, mainly by preventing transcription factors from binding to the DNA. Methylation also achieves what is called *genomic imprinting*. This means that some genes are expressed differently depending on whether they are on the maternally or paternally inherited chromosome.

Methylation patterns can be passed from one generation to the next and therefore represent another level of heritable information aside from the information contained in the base sequence alone. Other heritable information exists that is not stored within the DNA sequence itself, an example being histone modification (histones are proteins around which the DNA winds in order to take up less space in the cell). The study of heritable information not stored in the DNA is called *epigenetics*.

CpG depletion

CpG dinucleotides occur at a lower frequency than would be expected from single-base frequencies alone. The human genome (excluding the mitochondrial chromosome), for example, contains 20% G and 20% C (the *GC content* is 40%). The expected frequency of CpG dinucleotides, using a first-order model, is therefore $0.2 \cdot 0.2 = 4\%$, but the observed frequency is less than 1%, as can be computed from the openly available GRCh37 reference sequence (Church et al., 2011). See also the book by Scherer (2008, page 11).

This bias can be explained by considering the repair mechanisms for imperfect DNA replication. As summarized by Alberts et al. (2008, Chap. 5), nucleobases within DNA undergo spontaneous changes called *deamination*. For unmethylated bases, deamination gives rise to bases that do not naturally occur within DNA (hypoxanthine, xanthine and uracil) and the damage can be detected and repaired efficiently by enzymes recognizing these bases. Since methylcytosine deaminates into the natural base thymine, detection is harder and damage repair is less efficient. Over an evolutionary time scale, this leads to a depletion of methylated Cs in the genome. Since methylation is used to inactivate most genes within germ-line cells, most CpG dinucleotides are methylated. And, since only cytosines in CpG dinucleotides are methylated, a depletion of CpG dinucleotides results.

CpG islands

Although CpGs are rare in the genome on average, there are regions where they are more frequent, called *CpG islands (CGIs)* (Bird et al., 1985). Methylation correlates with CpG islands: CpGs within CpG islands are usually not methylated (Cooper et al., 1983), while those outside usually are (Larsen et al., 1992). There are two reasons for the existence of CpG islands. First, they are associated with genes that need to remain active in germ-line cells and are therefore unmethylated. Second, CGIs are within functionally important locations that do not tolerate damaged CpGs and selection therefore favors those individuals that contain undamaged CpGs (Alberts et al., 2008, Chap. 7). CpG islands show a high correlation with promoter regions of genes and are therefore used as markers that help to discover new genes.

Attempts have been made to characterize CpG islands by their sequence composition alone. Gardiner-Garden and Frommer (1987) define a CpG island to be a region of DNA that is at least

200 bp in length, has a GC content of at least 50%, and has an observed/expected ratio of CpG dinucleotides of 0.6 or more.

An updated CGI definition was given by Takai and Jones (2002). After human chromosomes 21 and 22 had been sequenced and annotated, the authors could computationally apply the previous definition to the two chromosomes and compare the locations of the found islands to the positions of known genes. The authors show that many of the islands are, in fact, not associated with genes, but with so-called *Alu* repetitive elements (*Alu* repeats). For CpG islands to serve as gene markers, this is not desirable. Takai and Jones then find that adjusting the criteria for CpG islands reduces the fraction of islands classified as *Alu* repeats (and those classified by them as "unknown") substantially. Their updated definition states that a CpG island is a region of DNA at least 500 bp in length that has a GC content of at least 55% and an observed/expected CpG ratio of at least 0.65. Consequently, the number of CpG islands on chromosomes 21 and 22 decreases from 14 062 to 1 101, which is closer to the actual number of genes on those chromosomes. While the new definition strongly improves specificity, this comes at a loss of sensitivity. In particular, the number of exon-associated CpG islands also drops from 757 to 120 on chromosomes 21 and 22, according to the authors. For the purpose of read mapping, an overly broad definition leads to a larger reference and is therefore not a problem, but losing sensitivity could impair mapping accuracy. Thus, we rely on the older definition by Gardiner-Garden and Frommer in the following. For more details, see also the summary by Bock et al. (2007).

3.1.2 Detecting methylation

Since current sequencing technologies cannot distinguish methylated from unmethylated cytosines, multiple approaches for determining methylation of a sample of DNA have been developed. There is, for example, MeDIP (methylated DNA immunoprecipitation) (Weber et al., 2005), which uses antibodies to extract those fragments from the DNA that contain methylcytosine. Using microarrays (*MeDIP-chip*) or high-throughput sequencing (*MeDIP-seq*), the fragments that contain at least one methylcytosine can be identified. Since it is not possible to determine which of the cytosines in a single fragment are methylated, methylation can only be determined at the level of fragments, i. e., at a low resolution.

The other major technique is *bisulfite sequencing* (Frommer et al., 1992), which can determine methylation patterns at the level of single nucleotides. It has only recently become economically feasible to use this method on a larger scale due to the arrival of high-throughput sequencing technologies.

3.1.3 High-throughput bisulfite sequencing

Bisulfite sequencing relies on a chemical reaction between the DNA and sodium hydrogen bisulfite (simply *bisulfite* in the following), which acts differently on methylated and unmethylated cytosines (Frommer et al., 1992). Bisulfite sequencing has been called the "gold standard" of methylation analysis since it works at nucleotide resolution. Certain phenomena can only be investigated with a method that works at the nucleotide level (Lister and Ecker, 2009).

The first studies combining bisulfite treatment with high-throughput sequencing investigated methylation in *Arabidopsis thaliana* (Lister et al., 2008; Cokus et al., 2008). Meissner et al. (2008) determined methylation patterns in mice using an Illumina Genome Analyzer. Methylation in

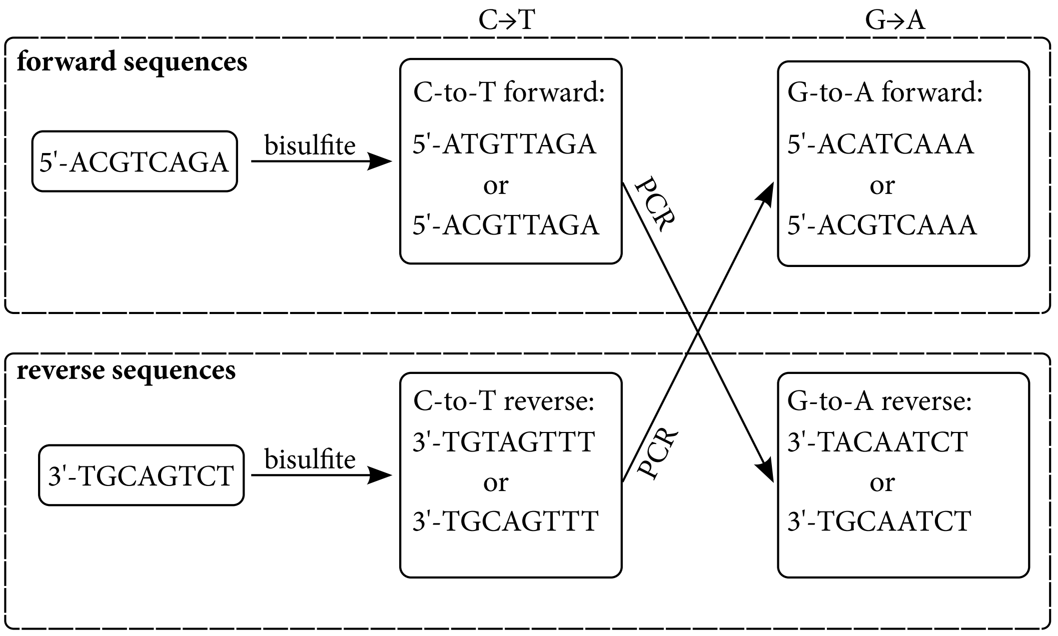


Figure 3.1 The six different types of sequences that arise during bisulfite sequencing of the double-stranded DNA fragment ACGTCAGA. The 5′ denotes the beginning of the forward sequence and 3' the beginning of the reverse sequence. All sequences in the top dashed box have a reverse complementary partner in the bottom dashed box.

human CpG islands was analyzed by Zeschnigk et al. (2009). The underlying bisulfite-aware read mapping algorithm is described in detail in this chapter. The biological results of that work are summarized in Section 3.4.3. The first study to perform whole-genome methylation analysis was performed later by Lister et al. (2009).

Bisulfite sequencing protocol

To determine the methylation pattern of a double-stranded DNA fragment, bisulfite sequencing uses the following protocol, summarized in Figure 3.1. We assume that methylation occurs only at cytosines in CpGs context. Because of methyltransferase activity, we also assume that a CpG is methylated if and only if its partner on the opposite strand is methylated.

Initially, double-stranded DNA fragments potentially containing both methylated and non-methylated cytosines are denatured into single-stranded molecules. Let us call the strand that is in the same orientation as the reference sequence the forward strand and the other the reverse strand (also called Watson and Crick strands).

When sodium bisulfite is added, unmethylated cytosines within both strands are deaminated to uracil, but 5-methylcytosines are left unaffected.

Next, bisulfite-treated DNA is amplified by polymerase chain reaction (PCR), during which the copied fragments will incorporate thymine instead of uracil in their base sequence. The intermediate uracil-containing fragments represent only a small fraction of all fragments and we ignore them henceforth. Therefore, unmethylated Cs in the original fragment will appear to have been replaced by Ts. We say that the bisulfite-converted fragments are of the *C-to-T forward* or

C-to-T reverse type, depending on their orientation relative to the reference sequence (middle column in Figure 3.1). Note that bisulfite conversion introduces an asymmetry: The converted C-to-T forward and C-to-T reverse fragments are, in general, not reverse complementary.

PCR amplification not only creates copies of a template fragment, but also copies of the reverse complement. Here, the reverse complements of C-to-T forward and of C-to-T reverse are therefore created. Since the complementary bases of `C` and `T` are `G` and `A`, respectively, those fragments will appear to have undergone G-to-A substitutions (right column in Figure 3.1). More exactly, those guanines that are paired with an unmethylated cytosine in the original fragment will appear to have been replaced by adenines. We therefore say that the resulting fragments are of the *G-to-A forward* or *G-to-A reverse* type. These fragments are also, in general, not reverse complementary.

Finally, the amplified fragments are sequenced. The distinction between fragment types is not maintained and can only be recovered computationally.

Incomplete bisulfite conversion

In reality, bisulfite conversion is not perfect and there are parts of the DNA that are not converted. Bisulfite conversion works only on single-stranded DNA and a source of error is therefore incomplete denaturation of the double-stranded DNA. Since very short double-stranded regions of a few nucleotides are unstable, the resulting errors are localized over longer stretches of the sequence.

3.1.4 Choosing the references

In standard, non-bisulfite sequencing, a read originates from either the forward or reverse strand and it is unknown which one it is. The standard solution is to map the read and also its reverse complement to the forward reference, which is equivalent to mapping it to the forward and reverse reference, but uses less memory.

In bisulfite sequencing, the read originates from one of four strands. By using the same idea of mapping the read and its reverse complement, only two references are needed. Those two references must not be reverse complements of each other. There are two sensible choices: Either designate both C-to-T strands or both forward strands as reference. The first option allows one to ignore the G-to-A strand. With the second option, the reverse sequence can be ignored.

In our software, we need to be able to deal with incomplete bisulfite conversion and therefore need the unconverted sequence as a third reference. It is therefore simplest to use all three forward references (top dashed box in Figure 3.1).

3.1.5 The *q*-gram index

A string of length q is a *q-gram* or *q-mer*. Both terms often imply that the q-gram or q-mer is a substring of another string. In a string of length n, there are $\max\{0, n-q+1\}$ positions at which a q-gram starts. We assume $q \ll n$ in the following and use $n-q+1$ only.

The *q-gram index* of a text $r \in \Sigma^*$ of length $n = |r|$ is an abstract data type that supports a single operation: Given a q-gram u, return all occurrences of u within r in constant time per

Table 3.1 The 3-gram index of the reference r = `CCACACCC`, using the alphabet $\{$ `A`, `C` $\}$. The shown q-code is only valid for an alphabet of size 2.

q-code	q-gram	list of positions
0	AAA	
1	AAC	
2	ACA	3
3	ACC	5
4	CAA	
5	CAC	2, 4
6	CCA	1
7	CCC	6

occurrence. More precisely, the q-gram index of r is the function

$$Q_r : \Sigma^q \to \mathcal{P}(\{ 1, \ldots, n - q + 1 \})$$

where, with $u \in \Sigma^q$

$$Q_r(u) = \{ i : r_{i \ldots i+q-1} = u \} \ .$$

See Figure 3.1 for a simple example of a q-gram index.

We also introduce the *q-code* of a q-gram, which is the integer encoding of the q-gram to base $\sigma := |\Sigma|$, assuming characters are ordered `A`, `C`, `G`, `T`.

Implementation with two arrays

A q-gram index can be implemented with the help of two arrays. For simplicity, let the first element in them have index zero. The first array P stores all q-gram positions ordered by the q-code of the respective q-gram. We call a subarray of P that contains all positions belonging to the same q-gram a *bucket*; buckets may be empty. The second array B has length $\sigma^q + 1$ and is indexed by q-code. The element $B[c]$ is the start index of the bucket in P that contains the positions for the q-gram with code c. We set $B[|\Sigma^q|]$ to the length of P minus one. All q-gram positions for q-code c are found at $P\big[B[c], \ldots, B[c+1] - 1\big]$.

Enumerating all k occurrences of a requested q-gram within r takes time $\mathcal{O}(k)$, as required. Memory usage is $\mathcal{O}\big((n + \sigma^q)\lceil \log n \rceil\big)$ bits since there are $n - q + 1$ positions in the buckets and σ^q q-grams. For an efficient implementation, the $\lceil \log n \rceil$ term is rounded up to the nearest multiple of the machine word size.

For example, memory usage for indexing the human genome with $n = 3 \cdot 10^9$ and $q = 12$ and a word size of 4 bytes is 12 GiB for the positions plus $4 \cdot 4^{12} = 64$ MiB for the bucket start indices. Values of up to $q = 16$ may be realistic on current machines, requiring 12 GiB for P as before, but 16 GiB for the B array.

Similarity to hashing

The q-gram index can be seen as an associative array that maps a q-gram to a list of positions. Array B can be considered a hash table, where the hash function maps each q-gram to its q-code. The values in the hash table are the indices into the array P. Another interpretation is that the index is a hash table that stores (q-gram, position) pairs and resolves collisions of equal q-grams through direct chaining.

Many authors therefore call read mapping methods that use q-gram indices or similar data structures *hashing-based methods* since mapping a q-gram to its q-code is one specific hashing function. It follows that the q-gram indexing scheme can be generalized by using different hashing functions.

Tools that use a *q*-gram index

The q-gram index or a variant of it is typically used to heuristically speed up read mapping or for finding local alignments. The approach is to *seed and extend*: First, find short matches (*seeds*) between a query and reference sequence and then extend those matches to full alignments if possible. As the seed finding phase must be fast, a q-gram index is often used here. The index can be created over the query or reference sequence. After constructing the index, matches between query and reference are found by iterating over the sequence that is not indexed and looking up its q-grams in the index.

BLAST (Altschul et al., 1990) finds local alignments. It saves memory by indexing the (short) query sequence. When it is used for protein sequences, not only all q-grams of the query are looked up in the index, but also those that are similar to within a pre-determined threshold in order to find *maximal segment pairs*, which are then extended to local alignments. Later versions (Altschul et al., 1997) also support alignments with gaps.

Choosing the q parameter involves a tradeoff between sensitivity and speed: A large q leads to fewer random matches that need to be verified and therefore makes the algorithm faster. It also reduces sensitivity since the chance of finding the seed at all is reduced (depending on the error rate). For the converse reasons, a small q increases processing time but improves sensitivity.

The authors of PatternHunter (Ma et al., 2002) show that through the use of non-contiguous, so-called *gapped q-grams*, one can improve speed and sensitivity at the same time. Gapped q-grams are defined by a *shape* of length w with q ones, such as 1101101. The program creates an index of each w-gram of the input sequence, but those characters that are at a position where the shape is zero are removed beforehand. For example, the original w-gram `ATGAATC` would be indexed as `ATAAC`.

MAQ (Li et al., 2008a) is a read mapping tool that creates six indices of the first 28 characters of the input reads. The chosen shapes guarantee that up to two mismatches in that region are tolerated. BFAST (Homer et al., 2009) indexes the reference with multiple gapped q-gram indices and allows configurable shapes. Found seeds are extended by gapped alignment.

Rasmussen et al. (2006) show that in a local alignment of a given minimum length and within a given error rate threshold, a certain number of q-gram matches must necessarily be found in parallelogram-shaped regions of the alignment matrix. Since this is not a heuristic, full sensitivity can be achieved. Their SWIFT program finds those regions in the alignment matrix with the help of a q-gram index. The RazerS read-mapping tool (Weese et al., 2009) is an advanced implementation of the same concept, also reaching full sensitivity.

GASSST (Rizk and Lavenier, 2010) is another full sensitivity read mapper. It focuses on quickly filtering false positive q-gram hits. Multiple filters are applied in order of fastest to most complex. For efficiency, the index contains not only the positions of the q-grams, but also the actual base sequence of up to 16 bp to the left and to the right of each q-gram occurrence. GASSST needs $16n$ bytes of memory (more than 48 GiB for the human genome).

3.1.6 Bisulfite read mapping tools

A bisulfite read mapping tool can solve the read mapping problem for bisulfite-treated reads. We give here an overview of some of the existing programs and methods.

Lister et al. (2008) studied methylation patterns in the *Arabidopsis thaliana* genome. It is one of the first studies that used bisulfite sequencing combined with high-throughput sequencing. The authors approach the bisulfite mapping problem by mapping reads to one reference in which cytosines are replaced by thymines and another reference in which guanines are replaced by adenines. Unmapped reads are afterwards mapped to the original reference. C-to-T and G-to-A modifications therefore do not result in mismatches, but there are mismatches for methylated cytosines. If the degree of methylation is high and the error threshold too low, reads are unmapped, leading to a loss of sensitivity (Xi and Li, 2009). Lister et al. have not made their software publicly available.

Cokus et al. (2008) also studied methylation in *Arabidopsis*. Their tool *CokusAlignment* does not work with the standard representation of a read as a sequence of bases with associated quality values, but with data one step earlier in the Illumina base-calling pipeline, where each base of the read is represented by a mixture of four color intensities. Thus, a read can be seen as a sequence of probability vectors (position weight matrix, PWM). CokusAlignment views also the reference as a PWM by assigning, at each position, very low, but non-zero probabilities to each non-reference base. The authors then define a function that incorporates a background model and gives the likelihood of a read originating from a given reference location. Bisulfite modification can be accounted for through a straightforward modification of that function. Their mapping tool uses a prefix trie of the reference to search for the best read location, computing partial scores at the nodes and pruning branches that cannot yield good results. CokusAlignment is available for download. It uses more than 32 GiB of memory for human genomes and is slow, according to Xi and Li (2009)

MAQ (Li et al., 2008a) also has a bisulfite mapping mode, which is not mentioned in the original publication, but which has been successfully used by Ziller et al. (2011). MAQ is not suited for long 454 reads as it does not map with indels and allows read lengths of at most 63 bp.

The following tools appeared after the article this chapter is based on (Zeschnigk et al., 2009) had been published.

One of the first tools aimed at being generally usable for bisulfite mapping is BSMAP (Xi and Li, 2009). It uses a hashing-based method implemented on top of SOAP (Li et al., 2008b). It was published after our tool, but the authors independently arrive at a similar idea to ours, which is to use an index of a simulated bisulfite-treated reference genome. One of the major differences to our software is that BSMAP indexes the C-to-T forward and C-to-T reverse sequences, and therefore does not need to consider G-to-A changes. After looking up q-grams of reads in its index, BSMAP uses bitparallel operations to allow C-to-T matches in the remainder of the read without counting them as mismatches. BSMAP is six times faster than CokusAlignment at the same sensitivity. The BSMAP algorithm is also used in RRBSMAP (Xi et al., 2012). The first version of BSMAP has been superseded in speed by tools that were developed later (Chatterjee et al., 2012).

One of those faster tools is RMAPBS, part of an extension to the RMAP read mapper (Smith et al., 2008, 2009). RMAP indexes the reads and then scans the reference sequence. Bisulfite mapping in RMAPBS is achieved by treating Ts in the reads as wildcard characters that match both a C and a T in the reference, and allowing a match between a C in the read and a C in the reference only when in CpG context.

Chen et al. (2010) later published BS Seeker, which does not implement an aligner itself, but wraps Bowtie (Langmead et al., 2009). BS Seeker converts either all `Cs` to `Ts` or `Gs` to `As`, in both reads and the reference, then maps the converted reads to the converted references. Nonunique alignments are discarded, and the original sequences are used to compute an accurate number of mismatches. Alignments with too many errors are discarded. The paper compares BS Seeker's runtimes to those of BSMAP, RMAP and MAQ. BSMAP is shown to be orders of magnitude slower than the other three even on small references. MAQ is dismissed due to low accuracy. On a human-sized reference, RMAP is also shown to be slower than BS Seeker.

BRAT (Harris et al., 2010) converts the reference to two bitstrings. In the first, a bit is set to one if and only if the reference has a C or T at that position. In the second, the bit is set to one for positions with C or G. All q-grams of both bitstrings are then indexed in separate hash tables. The bisulfite reads are converted in a similar way and first looked up in the C/T reference hash table, automatically allowing C-to-T and G-to-A matches. A lookup in the second hash table filters out the G-to-C and A-to-G matches. Under some conditions, BRAT is faster than mrsFAST (Hach et al., 2010) and BSMAP. BRAT uses $12n$ bytes of memory ($24n$ for paired-end reads), where n is the length of the reference, plus some memory for the reads. Mapping only to human chromosome 1 requires 4 GiB. BRAT does not map with indels. A more memory-efficient successor named BRAT-BW that uses the Burrows-Wheeler transform (Burrows and Wheeler, 1994) instead of hashing has recently become available (Harris et al., 2012).

3.2 The bisulfite *q*-gram index

In this section, we introduce another variation of the q-gram indexing scheme that enables us to map reads from bisulfite sequencing experiments to a reference genome, which we call the *bisulfite read mapping problem*, and to subsequently analyze methylation patterns. We use the standard seed-and-extend approach to map bisulfite-treated reads. The index allows to find seeds that contain substitutions from bisulfite conversion without counting them as errors. The mapping procedure itself is described in Section 3.3.

3.2.1 Simulated bisulfite treatment

The bisulfite q-gram index is a q-gram index over a simulated bisulfite-treated version of the reference sequence. Since methylation of cytosines is unknown during index creation, all possible arising q-grams are simulated. The index answers the following question: Given a q-gram of a read from a bisulfite sequencing experiment, from which positions in the reference can it have originated? The index must cope with reads of both the C-to-T and the G-to-A type, but also those reads of fragments that were partially or fully unconverted.

We allow methylation of cytosines only in CpG context. This is a simplification, but close to reality in vertebrate genomes, and it reduces memory usage. Therefore, when we compare a read that comes from the C-to-T forward strand to the reference, the characters should be equal, except that every C in the reference not followed by a G must be replaced with a T and that every C in the reference that is followed by a G can be either C or T, see also Figure 3.1. This mnemonic, also for the G-to-A forward strand, may be helpful:

$$\texttt{C} \to \texttt{T} \text{ but } \texttt{CG} \to \texttt{CG}|\texttt{TG} \quad \text{and}$$

$$\texttt{G} \to \texttt{A} \text{ but } \texttt{CG} \to \texttt{CG}|\texttt{CA}$$

Definition 6 (bisulfite compatibility). Let $r \in \Sigma^*_{\text{DNA}}$ be the reference sequence and $u \in \Sigma^q_{\text{DNA}}$ a q-gram. A position i in r is *C-to-T bisulfite compatible* with u if the following holds for all $j = 1, \ldots, q$:

$$\begin{array}{lllll} u_j = \texttt{T} & \text{if} & r_{i+j-1} = \texttt{C} & \text{and} & r_{i+j} \neq \texttt{G}, \\ u_j \in \{\,\texttt{C}, \texttt{T}\,\} & \text{if} & r_{i+j-1} = \texttt{C} & \text{and} & r_{i+j} = \texttt{G}, \\ u_j = r_{i+j-1} & \text{otherwise.} & & & \end{array}$$

The position is *G-to-A bisulfite compatible* with u if the following holds for all $j = 1, \ldots, q$:

$$\begin{array}{lllll} u_j = \texttt{A} & \text{if} & r_{i+j-1} = \texttt{G} & \text{and} & r_{i+j-2} \neq \texttt{C}, \\ u_j \in \{\,\texttt{G}, \texttt{A}\,\} & \text{if} & r_{i+j-1} = \texttt{G} & \text{and} & r_{i+j-2} = \texttt{C}, \\ u_j = r_{i+j-1} & \text{otherwise.} & & & \end{array}$$

We do not define bisulfite compatibility between two q-grams as compatibility requires knowledge of the context on the reference. In the C-to-T case, we need to know the character following the q-gram on the reference. In the G-to-A case, we need to know the character preceding it.

Example 5. Given a position on the reference where the substring `CACGAC` occurs, the q-grams `TATGAT` and `TACGAT` are C-to-T compatible with it. If the substring is followed by `G`, then also `TATGAC` and `TACGAC` are compatible.

Definition 7. The *bisulfite q-gram index* of r is the function

$$B_r : \Sigma^q_{\text{DNA}} \to \mathcal{P}(\{\,1, \ldots, |r| - q + 1\,\})$$

$$B_r(u) = \{\, i : i \text{ is C-to-T or G-to-A bisulfite compatible with } u \quad \text{or} \quad r_{i\ldots i+q-1} = u \,\}\,.$$

This mathematical definition is slightly more general than the actual implementation, in which the positions are stored as a list in *ascending order* (see below).

In order to be able to process reads that were not bisulfite converted, it is necessary to include in the index the positions where the unmodified q-gram occurs. As a consequence $B_r(u) \subseteq Q_r(u)$ for all $u \in \Sigma^q_{\text{DNA}}$. In total, the index covers all types of reads shown in the upper box in Figure 3.1.

No alphabet reduction

The bisulfite q-gram index does not employ a reduced alphabet as other tools do, such as BS seeker (Chen et al., 2010). The idea of a reduced alphabet is that all Cs are (unconditionally) replaced by Ts (or Gs by As) in both the read and the reference, neutralizing all differences caused by bisulfite-treatment so that any standard read mapping tool can be used afterwards. In contrast, the advantage of our approach is that less information is lost that can potentially be used to assign a read unambiguously to a position. A q-gram that contains at least one C can potentially be mapped to fewer locations. For example, consider the read `CGA` and the reference `CGATGA`. Reducing the alphabet results in read `TGA`, reference `TGATGA`. The read can therefore be mapped to two locations (start positions 1 and 4). By taking into account the `C` in the read, it is clear that only the first alternative is correct.

3.2.2 Index structure and creation

The bisulfite q-gram index is stored in the same arrays B and P described in Section 3.1.5. In summary, array B is an array of bucket start indices, indexed by the q-code of the q-gram. Array P contains start positions within r such that $P\big[B[c], \dots, B[i+1]-1\big]$ for q-code c of q-gram u gives, in ascending order, the content of the set $B_r(u)$.

In the regular q-gram index, array P is a permutation of the integers $1, \dots, |r| - q + 1$. An important difference is that, in the bisulfite version, positions can occur more than once in P.

Index creation

The index is created in a pre-processing step and stored on disk. We split the description of its creation up into two algorithms. One simulates bisulfite treatment for each q-gram of the reference, and the second uses those results to actually build the index. The second algorithm could also be used to build a regular q-gram index. For the sake of a straightforward description, we give simplified versions of the algorithms that do not consider some practical issues. These are then discussed further below.

In the following, a *lone C* is a C that is not in CpG context.

Algorithm 5 (Simulate-Bisulfite-C-to-T).
Input: $r \in \Sigma_{\text{DNA}}^*$ and q.
Output: Lists of bisulfite-simulated q-grams (C-to-T and unmodified q-grams only).

1. u is the current q-gram on the reference. Since we start before the first character of the reference, initialize it to `A...A` (length q).

2. *active* is a list of bisulfite-simulated q-grams. It is initialized to a list that contains u.

3. $lone \leftarrow -1$ tells us the position of the last lone C. The invariant is that r_{lone} is the most recently seen lone C. If $lone = -1$, no lone C has been seen so far.

4. Iterate from $i = 1$ to $i = |r| - q + 1$. The current character is therefore r_i:

    a) Drop the first character from u and append r_i.

    b) Remove all q-grams from *active* that start with `C`. Remove the first character from those that remain.

    c) If r_i is a lone C, that is, if $r_i =$ `C` and $r_{i+1} \neq$ `G`, set ch to `T` and set $lone$ to i. Set ch to r_i otherwise.

d) If *ch* $\neq$ `C`, append *ch* to all strings in *active*.

e) Otherwise, update *active* such that it contains two copies of each of its previous elements, where one has a `T` and one a `C` appended to it.

f) If $i < q$, continue with the next iteration of the loop.

g) If $lone \geq i - q$, that is, if there is a lone C in the current q-gram, yield $active \cup \{ u \}$. Otherwise, yield *active* only since we know that the unmodified q-gram is already one of its elements.

The full algorithm SIMULATE-BISULFITE, which is not shown here, also takes G-to-A substitutions into account.

For simplicity, the algorithm above is described in terms of q-grams, that is, strings. In the actual implementation, integer values (q-codes) are used that represent the q-grams and string manipulations (dropping the first character, appending a character, etc.) are implemented as bit-level operations. Assuming that a q-gram fits into a machine word, Algorithm 5 needs time $\mathcal{O}(1)$ per simulated q-gram. In the worst case, r consists entirely of consecutive CGs, and therefore each q-gram of r (except for the last) gives rise to $2^{q/2}$ simulated q-grams (assuming even q). Worst-case runtime is therefore $\mathcal{O}(|r|2^{q/2+1})$ when we also simulate G-to-A conversion. The average runtime is much better, as we show in Section 3.4.1.

The bisulfite index is created in two passes over the reference.

Algorithm 6 (CREATE-BISULFITE-Q-GRAM-INDEX).
Input: Reference $r \in \Sigma^*_{\text{DNA}}$ and q.
Output: A q-gram index consisting of the two arrays B (bucket starts) and P (positions).

1. Let *sizes* be an array of length 4^q, all values initialized to zero. It stores the size of the buckets of P.

2. Iterate over SIMULATE-BISULFITE(r, q); let *qgrams* be the list of q-grams returned in each iteration: For each q-gram in *qgrams*, let *qcode* be its q-code and increment $sizes[qcode]$ by one.

3. Initialize B with the help of *sizes*: $B[c] \leftarrow \sum_{d<c} sizes[d]$ (this can be done in $\mathcal{O}(|B|)$).

4. Set $B[4^q]$ to the sum over all sizes and initialize P to that length.

5. Copy B into *starts*. Iterate, as above, over SIMULATE-BISULFITE(r, q), but this time, keep track of the current position i on the reference and for each q-code, write it into P with $P\big[starts[qcode]\big] \leftarrow i$ and increment $starts[qcode]$.

6. Return B and P.

The asymptotic runtime of Algorithm 6 is the same as that of Algorithm 5 plus $\mathcal{O}(4^q)$ for creation of B. The worst-case size of the bisulfite q-gram index is the same, that is, $\mathcal{O}(|r|2^{q/2+1} + 4^q)$. See the results in Section 3.4.1 for actual values, which are much lower.

Table 3.2 Low-complexity filtering by keeping only those q-gram buckets containing q-grams with at least $d = 3$ or $d = 4$ distinct characters.

		q-grams kept		Ratio	
q	4^q	$d \geq 3$	$d \geq 4$	$d \geq 3$	$d \geq 4$
4	256	168	24	65.62%	9.38%
5	1 024	840	240	82.03%	23.44%
6	4 096	3 720	1 560	90.82%	38.09%
7	16 384	15 624	8 400	95.36%	51.27%
8	65 536	64 008	40 824	97.67%	62.29%
9	262 144	259 080	186 480	98.83%	71.14%
10	1 048 576	1 042 440	818 520	99.41%	78.06%
11	4 194 304	4 182 024	3 498 000	99.71%	83.40%
12	16 777 216	16 752 648	14 676 024	99.85%	87.48%
13	67 108 864	67 059 720	60 780 720	99.93%	90.57%
14	268 435 456	268 337 160	249 401 880	99.96%	92.91%

Using less main memory

The second phase of the algorithm as given creates the full P array in memory. Main memory usage can be reduced by constructing P incrementally in multiple passes over r. Each pass fills in subarray $B[c_k, \ldots, c_{k+1} - 1]$ and works only on those simulated q-grams whose q-code is at least c_k and at most $c_{k+1} - 1$. Each c_k is chosen such that the $sizes[k + 1] - sizes[k]$ positions in that bucket range fit into memory and such that all q-codes have been processed in the end.

Wildcard characters

Non-nucleotide characters in the reference occur, in particular, at the centromere and telomere regions, where long stretches of consecutive ‘N’ characters indicate an unknown sequence. We therefore ignore the start position of all q-grams in those regions. Those positions are not added to the index. If only a few characters are non-nucleotide characters, another solution (currently not implemented) is to simulate the possible q-grams similar to how the bisulfite q-grams are simulated. For example, the IUPAC character K is “G or T”. The q-gram AAKAA would be interpreted as being compatible with both AAGAA and AATAA.

In some references, lowercase characters indicate repeat-masked regions. We have therefore implemented the option to ignore all q-grams that contain a lowercase character.

Low complexity *q*-gram filtering

In the Verjinxer software, it is also possible to filter repetitive sequences at the q-gram level. This filter was implemented by the original author for regular q-gram indexing, but can also be used for bisulfite q-gram indexing. The idea is to discard all q-grams that use fewer than the four available alphabet characters. The filter discards all buckets belonging to those q-grams that contain fewer than a specified number d of distinct characters. Only values less than four make sense for this parameter. See Table 3.2.

3.2.3 Multiple reference sequences

Our assumption so far has been that there exists a single reference sequence. A realistic genome that we would like to use as reference consists of multiple parts, which we call *chromosomes* for simplicity, even if they do not correspond to a biological chromosome. We can deal with those in the following ways.

The first option is to append a sentinel character that is otherwise unused in the alphabet to each chromosome. Before index construction, the chromosomes plus sentinels are concatenated and treated as a single reference. The position of each sentinel is recorded in a separate array. Index creation needs to ignore q-grams with sentinel characters and read mapping needs to be modified to stop processing when a sentinel character is encountered. The mapping algorithm remains unchanged otherwise so that the coordinates of mapped reads are obtained relative to the concatenated reference. Each coordinate is finally converted to a pair of the correct chromosome name and the coordinate relative to the chromosome start, using either binary search or linear search if there are few chromosomes.

Alternatively, the chromosomes can be concatenated directly and non-nucleotides be replaced with random nucleotide characters (Li and Durbin, 2009). Reads overlapping a chromosome border are filtered out in a postprocessing step. Again, binary or linear search is used to map reference coordinates to chromosome positions.

When the term *reference* occurs in the following, the concatenated reference is meant.

3.3 Mapping a bisulfite read

In the mapping algorithm, we basically follow the seed-and-extend paradigm employed in tools such as BLAST (Altschul et al., 1990), MAQ (Li et al., 2008a) and BFAST (Homer et al., 2009). In general, this paradigm describes the idea of initially locating short and usually exact matches (the *seeds*), which are then extended error-tolerantly.

In our version of this approach, a seed is a maximal exact match between the reference and the read, where "exact" means that differences due to bisulfite treatment are allowed. We also call this a *maximal bisulfite match*. How seeds are found with the help of the bisulfite q-gram index is described in Section 3.3.1.

When a short match is extended to a longer one (error-tolerantly or not), and bisulfite rules apply, then it is necessary to know whether the read is of the C-to-T or G-to-A type in order for the costs to be computed correctly. This is described in Section 3.3.2.

Finally, we explain in Section 3.3.6 how seeds are extended into an alignment that covers the entire read.

3.3.1 Finding seeds

Let $t \in \Sigma_{\text{DNA}}^n$ be a single read. If we look up one of its q-grams $u = t_{i \dots i+q-1}$ in the bisulfite q-gram index, we get a list of positions on the reference r. We call the pair (i, j), where j is one of the positions, a *(bisulfite) hit*. The *diagonal* of the hit is $j - i$. An exact bisulfite match between read and reference that is longer than q results in consecutive hits that lie on the same diagonal.

Briefly, the seed finding algorithm determines the list of hits for each position $1, \dots, n-q+1$ of the read, extends each to the right as far as possible and reports them, taking care to exclude hits that are part of an already reported match.

When extending a hit to the right, we need to be aware that it may have arisen from the simulated C-to-T or G-to-A strand or from the unmodified sequence, and that the type must be consistent in an exact match. As there are multiple ways to ensure this, we describe the subroutine for hit extension in the next section and assume here simply that it exists.

Algorithm 7 (Find-Seeds)**.**
Input: The read t; reference r. The bisulfite q-gram index.
Output: Maximal bisulfite matches between read and reference as triples (i, j, ℓ), where ℓ is the match length.

1. If read length $n < q$, return an empty result list.
2. Initialize an empty list *active*. Each entry in the list is a pair $(start, length)$.

   Each entry describes the remainder of a match that has been found in a previous iteration. In other words: The *start* positions describe the hits that we expect to find in the next iteration and which should be ignored because the matches they belong to have already been reported.
3. Iterate over the q-grams of the read with start positions $i = 1, \ldots, n - q + 1$:
   a) For each entry in *active*, increase its *start* by one and decrease its *length* by one. Remove those entries whose length is less than q.
   b) Retrieve a list of hits by looking up the current q-gram code in the q-gram index. Let *hits* be that list. Due to index construction, the hits are in ascending order by reference position.
   c) Iterate over *active* and *hits* simultaneously in the same way as in a "merge" step in mergesort.

      There are three cases:
      i. If the position is found in both *active* and *hits*, then a previously found match continues. Keep the entry in the *active* list.
      ii. If a position is in *active*, but not in *hits*, then the entry in the q-gram index for this position is missing. This can be due to a skipped q-gram in the reference because of a wildcard character, an empty bucket in the index due to a too high q-gram frequency, or discrepancies in how hits are extended (see below) and how the index is constructed. In any case, keep the entry in the *active* list.
      iii. If the position is in *hits*, but not in *active*, the beginning of a new exact match has been found. To turn it into a maximal exact match, compute its length with one of the algorithms of Section 3.3.2. Add the position in the reference and the length of the match to the *active* list, and yield the match.

Some implementation details

In the implementation, the list *active* is stored as two arrays *activepos* and *activelen* that have as many entries as there are positions in the largest bucket of the q-gram index. An entry is removed from both lists by setting the entry in *activelen* to zero and a check is added in order to skip such an entry when iterating over active matches.

When querying the index to retrieve the list of *hits*, it is sufficient to return a pointer to the start of the appropriate bucket and the length of that bucket. This avoids some copying.

We also introduce a minimum match length $\ell_{\min} \geq q$. Maximal bisulfite matches are only reported if they are not shorter. This reduces the number of false positives that need to be processed in the next stage. Iteration over the read is modified to iterate from $i = 1$ to $n - \ell_{\min} + 1$.

3.3.2 Extending hits

Given a hit (i, j), the question arose in the previous section how to extend it in order to find a maximal match. Without bisulfite conversion, we could simply compare the subsequent characters on reference and read until a mismatch occurs. The problem is that bisulfite reads can be of the C-to-T or G-to-A type and that we must first determine which of the two we have. At this point, we will also take into account that bisulfite reads can be fully or partially unconverted. We therefore describe two different modes of extending a hit.

Strict hit extension requires that the bisulfite substitution rules are strictly followed. That is, in a C-to-T match, every C must be replaced by T, unless the C is in CpG context. In a G-to-A match, every G must be replaced with an A, unless the G is in CpG context.

Relaxed hit extension lifts the restriction of not allowing lone Cs or lone Gs in the read. Thus, in a C-to-T match, every C can be replaced with a T. In a G-to-A match, every G can be replaced with an A. This does not only allow incomplete bisulfite conversion, but also cytosine methylation outside of CpG context.

3.3.3 Strict hit extension

Extending a hit while strictly observing bisulfite replacement rules is straightforward to implement with a deterministic finite automaton. We start out by describing regular languages for the different types of matches. We use the alphabet $\Sigma_{\text{DNA}} \times \Sigma_{\text{DNA}}$, where the first character is from the reference and the second from the read. In the following, we write the characters on top of each other as in an alignment. This is the language of all valid C-to-T bisulfite matches:

$$L_{\text{C-to-T}} = \left(\begin{array}{c|c|c|c|c|c} \mathsf{A} & \mathsf{T} & \mathsf{G} & \mathsf{CG} & \mathsf{CG} & \mathsf{C} \\ \mathsf{A} & \mathsf{T} & \mathsf{G} & \mathsf{CG} & \underline{\mathsf{TG}} & \mathsf{T} \end{array}\right)^*$$

And this describes the valid G-to-A bisulfite matches:

$$L_{\text{G-to-A}} = \left(\begin{array}{c|c|c|c|c|c} \mathsf{A} & \mathsf{T} & \mathsf{C} & \mathsf{CG} & \mathsf{CG} & \mathsf{G} \\ \mathsf{A} & \mathsf{T} & \mathsf{C} & \mathsf{CG} & \underline{\mathsf{CA}} & \mathsf{A} \end{array}\right)^*$$

The two underlined subexpressions are redundant and are added for clarity. We can immediately write down the language of matches for which we cannot give the type since that is equal to the intersection of both languages:

$$L_{\text{unknown}} = L_{\text{C-to-T}} \cap L_{\text{G-to-A}} = \left(\begin{array}{c|c|c} \mathsf{A} & \mathsf{T} & \mathsf{CG} \\ \mathsf{A} & \mathsf{T} & \mathsf{CG} \end{array}\right)^*$$

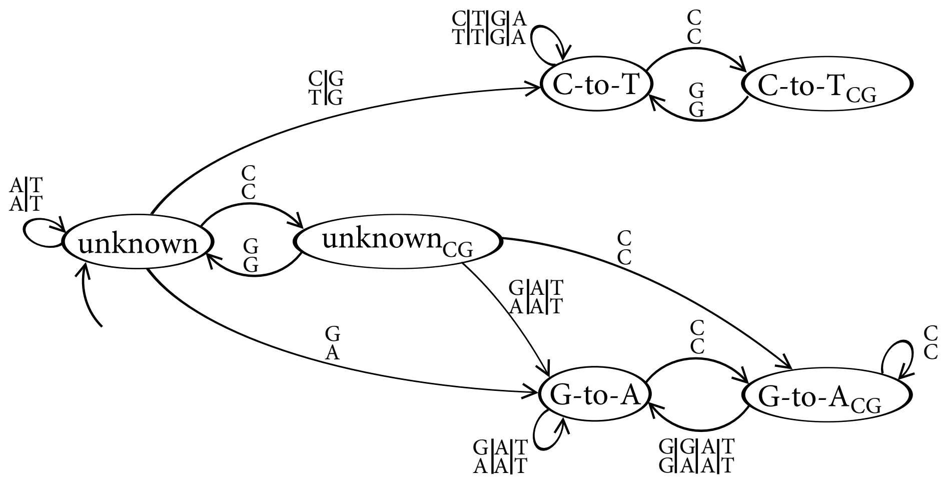


Figure 3.2 The bisulfite matching automaton for strict hit extension. All transitions that are not shown lead to an implied *Mismatch* state. The leftmost state is the start state. All shown states are accepting states.

Finally, an arbitrary bisulfite match can be described by the language $L_{\text{unknown}}\,(L_{\text{C-to-T}}|L_{\text{G-to-A}})$. The concatenation of L_{unknown} is also redundant, but helpful to understand the automaton.

We can now construct the deterministic finite automaton (DFA) that recognizes the language of all matches. We call it the *bisulfite matching automaton* and it is shown in Figure 3.2. The automaton has one mismatch state (not shown) and six regular states. Two states each represent unknown, C-to-T and G-to-A match types. The automaton starts out in an "Unknown" state, and remains in one of the two unknown states until a character pair occurs that is only allowed in one of the C-to-T or G-to-A states, to which it then transitions.

The resulting algorithm is straightforward.

Algorithm 8 (Extend-Bisulfite-Hit-Strict).
Input: Reference r of length m; read t of length n; and a hit (i, j).
Output: Length and type of the strict match.

```
1. Initialize the bisulfite matching automaton state to Unknown and set length to zero.
2. Repeat while $i \leq m$ and $j \leq n$:
    a) Transition in the automaton according to $(r_i, t_j)$.
    b) If the automaton is in Mismatch state, break out of the loop.
    c) Increment length, i and j.
3. Return length and the type of the match, depending on the last non-Mismatch state the
   automaton was in.
```

Some optimization of this basic algorithm is possible. We can, for example, add a cache that maps pairs of q-grams to a pre-computed automaton state. Only those q-grams that do not lead to *Mismatch* are stored to avoid quadratic memory usage. Thus, we can avoid iterating over the first q characters of the hit, for which we know that no mismatch can occur.

Another option is to store the type of the hit in the bisulfite q-gram index itself. This does require two bits of extra storage per position, however, since we need to distinguish between *unknown*, C-to-T and G-to-A.

We note that the strict mode is not as useful in practice, in contrast to the relaxed mode, which we describe next.

3.3.4 Relaxed hit extension

The rule for a relaxed match is: Every character must match, except that in the C-to-T case, a C in the reference additionally matches a T in the read, and in the G-to-A case, a G in the reference additionally matches an A in the read. Note that this includes exact matches and we therefore cover not only the case of methylation outside of CpG context, but also full or partial incomplete bisulfite conversion. The idea of the following algorithm is that the first character pair that is not equal determines the match type.

Algorithm 9 (Extend-Bisulfite-Hit-Relaxed).
Input: Reference r; read t; and a hit (i, j).
Output: Length and type of the relaxed match.

1. Set *length* to zero. Set the *type* to unknown.
2. While the end of neither r nor t has been reached and $r_i = t_j$, increase i, j and *length*.
3. If neither the end of r nor t has been reached, compare r_i and t_j:
   a) If $r_i = \mathtt{C}$ and $t_j = \mathtt{T}$, loop over the remaining characters while $r_i = t_j$ or $r_i = \mathtt{C}$ and $t_j = \mathtt{T}$, increasing *length* for each character. Set *type* to C-to-T.
   b) If $r_i = \mathtt{G}$ and $t_j = \mathtt{A}$, do the same, but allow $r_i = \mathtt{G}$, $r_j = \mathtt{A}$ instead. Set *type* to G-to-A.
4. Return *length* and *type* as result. The type is *unknown* if no mismatch was found before the end of one of the strings is reached or when the first mismatch is neither C-to-T nor G-to-A.

3.3.5 Bitwise-parallel relaxed hit extension

If a 2-bit representation is used for both reference and read, Algorithm 9 would need to extract groups of 2-bit characters from each machine word in order to loop over the strings. This can be sped up by comparing $w/2$ characters at a time, where w is the machine word (register) width. We assume that a register is wide enough to hold an entire read.

The method described here is similar in spirit to the one described by Xi and Li (2009), who use bit operations to count mismatches between reference and bisulfite-modified query. In contrast to their method, we do not need to compute a separate "bitwise mask" of the reference, which they seem to compute in a non-bitparallel way. Additionally, we can distinguish between C-to-T and G-to-A reads. Our strategy is to compute the match length with low-level instructions twice: Once allowing C-to-T and once allowing G-to-A replacements. The longer match then determines the returned match type.

We use the encoding $\mathtt{A} = 00_2$, $\mathtt{C} = 01_2$, $\mathtt{G} = 10_2$, $\mathtt{T} = 11_2$, which allows us to compute the complementary character with a binary *not* ($\neg$). Let $x = x_1x_0$ be an encoded character on the

Table 3.3 Character comparison with XOR (left) and with the $\text{bcomp}_{\text{C-to-T}}$ (bisulfite comparison) function. The 00_2 values that signify character matches are underlined. Note the differences in the `C` row. x: Character on reference. y: Character on read.

$x \oplus y$			y			
			A	C	G	T
			00	01	10	11
x	A	00	$\underline{00}$	01	10	11
	C	01	01	$\underline{00}$	11	$\boxed{10}$
	G	10	10	11	$\underline{00}$	01
	T	11	11	10	01	$\underline{00}$

$\text{bcomp}_{\text{C-to-T}}(x, y)$			y			
			A	C	G	T
			00	01	10	11
x	A	00	$\underline{00}$	01	10	11
	C	01	01	$\underline{00}$	01	$\underline{00}$
	G	10	10	11	$\underline{00}$	01
	T	11	11	10	01	$\underline{00}$

reference and $y = y_1 y_0$ the corresponding character in the read (most significant bit written left). To check whether they are equal, the bitwise XOR ($\oplus$) can be used, since $x \oplus y = 00_2$ if and only if $x = y$, that is, when $x_1 = y_1$ and $x_0 = y_0$, see Table 3.3.

For C-to-T reads, the result should additionally be 00_2 when $x = 01_2$ (`C`) and $y = 11_2$ (`T`). We can use this function (bisulfite comparison):

$$\text{bcomp}_{\text{C-to-T}}(x, y) = (x \oplus y) \mathbin{\&} \big(01_2 \mid (x \mathbin{\&} 10_2) \mid \neg(x \ll 1)\big)$$

Table 3.3 proves its correctness. The idea is that the right-hand side acts as a bit mask that is 01_2 if $x = 01_2$ (`C`) and 11_2 otherwise. Why is this mask not dependent on y? Inspecting the `C` rows in Table 3.3, we see that indeed clearing the top bit changes two cells, but that only the C-to-T substitution ends up as 00_2 and therefore being considered equal.

The corresponding function for G-to-A type matches can be constructed by noting that `C/G` and `T/A` are complementary. Thus,

$$\text{bcomp}_{\text{G-to-A}}(x, y) = \text{bcomp}_{\text{C-to-T}}(\neg x, \neg y) \ .$$

The comparison functions also work when x and y are full machine words containing a 2-bit encoded DNA string. Instead of the bit masks 01_2 and 10_2, we need to use the masks $0101\ldots01_2$ and $1010\ldots10_2$.

A set bit in the result of one of the bcomp functions gives us the position of a mismatch. Our ability to determine the type *and* length of a match therefore depends on the ability to find the position of the first set bit. Fortunately, many modern processors have a *bit scan* or *count zeros* instruction, which find the position of the first or last nonzero bit or, similarly, count the number of leading or trailing zeros in a word. For example, on Intel 80386, one such instruction is BSR (*bit scan reverse*) (Intel, 2012). In the following, let $\text{clz}(x)$ be a function that returns the number of leading zeros in x and that returns w if x is zero.

Match length

Let x and y be two words of length w that encode $w/2$ characters of the reference and of the read, respectively. The length of a C-to-T bisulfite match is:

$$\ell_{\text{C-to-T}}(x,y) = \left\lfloor \frac{\operatorname{clz}\left(\operatorname{bcomp}_{\text{C-to-T}}(x,y)\right)}{2} \right\rfloor$$

The length of a G-to-A bisulfite match is:

$$\ell_{\text{G-to-A}}(x,y) = \left\lfloor \frac{\operatorname{clz}\left(\operatorname{bcomp}_{\text{C-to-T}}(\neg x,\neg y)\right)}{2} \right\rfloor$$

We summarize the results in the following algorithm.

Algorithm 10 (Extend-Bisulfite-Hit-Relaxed-Bitparallel).
Input: Reference r and read t in 2-bit encoding; and a hit (i,j).
Output: Length and type of the relaxed match.

1. Set x to $r_{i\ldots i+w-1}$, and y to $t_{j\ldots j+w-1}$. If read or reference are too short, pad x on the right with ones and y on the right with zeros. (One could instead fall back to Algorithm 9).

2. Compute $ct \leftarrow \ell_{\text{C-to-T}}(x,y)$ and $ga \leftarrow \ell_{\text{G-to-A}}(x,y)$.

3. If $ct > ga$: Return length ct and type C-to-T.

4. If $ct < ga$: Return length ga and type G-to-A.

5. Otherwise (ct and ga are equal): Return length and type *unknown*.

The algorithm can be extended to work with reads that are potentially longer than $w/2$. On a current machine with $w = 64$, it can compare 32 characters in a single step, but larger registers are often available. The Advanced Vector Extensions (AVX), for example, offer 256 bit registers.

3.3.6 Extending seeds

Those maximal bisulfite matches found that are not shorter than the minimum seed length $\ell_{\min}$ are used as seeds when aligning the full read to the reference.

Assume the seed was found at (i,j) with length ℓ. The seed is first extended to the right until the end of the read by aligning $r_{i+\ell\ldots|r|}$ to $t_{j+\ell\ldots|t|}$ and allowing to skip a suffix of the reference at no cost. This is the same variant of semiglobal alignment as used for aligning an “anchored 5’ adapter” to a read, described in Section 2.2.4 (page 20), except that a suffix of the read is aligned to a suffix of the reference, see also Figure 2.3 in that section (upper right matrix).

A different cost function also needs to be used. We distinguish again a strict and a relaxed version of alignment. For relaxed alignment, the cost function is the same as unit costs, except that the cost of having a C in the reference and a T in the read is zero. For strict alignment, the cost function is slightly more complicated. It needs to count also non-bisulfite-conversions as errors:

$$d_{\text{C-to-T}}(r_i, t_j) = \begin{cases} 0 & \text{if } r_i \neq \texttt{C} \text{ and } r_i = t_j \\ 0 & \text{if } r_i = \texttt{C} \text{ and } t_j = \texttt{T} \\ 0 & \text{if } r_i = t_j = \texttt{C} \text{ and } (r_{i+1} = \texttt{G} \text{ or } t_{j+1} = \texttt{G}) \\ 1 & \text{otherwise} \end{cases} \tag{3.1}$$

The *or* condition of the third case warrants further explanation. When we encounter a C on the reference and a C in the read, why is it sufficient that *one* of them is followed by a G? Consider the following three alignments (the reference is in the top row), which all involve a CG substring:

```
TC-GT    TCAT    TCGT    TCTGT
TCAGT    TCGT    TCAT    TC-GT
```

They describe events that can be either sequencing errors or actual mutations. In all cases, it is possible that the actual genome sequence truly contains a CpG, which can be methylated. If it is methylated, incomplete bisulfite conversion has not occurred and should not be penalized. All of these alignments will therefore receive a cost of one instead of two. However, since the underlying cause of the event is unknown, such sites will be counted as neither methylated nor non-methylated in the following.

Let the alignment algorithm that uses cost function $d_{\text{C-to-T}}$ be called ALIGN-C-TO-T. We can further define a function $d_{\text{G-to-A}}$ (not shown) that works on G-to-A reads. Let the resulting algorithm be called ALIGN-G-TO-A. We now have an easy way of extending the seed of a C-to-T type match to the left: Take the reverse complement of the reference prefix $r_{1\ldots i-1}$ and align it to the reverse complement of the read prefix $t_{1\ldots j-1}$ with ALIGN-G-TO-A.

When extending a G-to-A type seed, the roles of ALIGN-C-TO-T and ALIGN-G-TO-A are reversed. At this point, we are done if we trust that the read type determined by the hit extension is correct and if it is not "unknown". Otherwise, we suggest the following algorithm for better results.

Algorithm 11 (EXTEND-SEED).
Input: Seed coordinate (i, j); length ℓ; reference r; read t.
Output: Alignment of substrings of r and t and start coordinates on r and t.

1. For both possible read types C-to-T and G-to-A, do the following.
   a) Count the number of unconverted Cs or Gs (depending on current read type) in the seed region and store that in *unconv*.
   b) Use the proper combination of algorithms ALIGN-C-TO-T and ALIGN-G-TO-A as described above to extend the seed to the entire read, assuming the current read type.
   c) Set $total_{\text{type}}$ to the sum of *unconv* and the numbers of errors returned by the two alignment functions.
2. Assume that the read type which collected the smaller number of total errors is the correct one. Construct the full alignment from the three segments and return it, along with the start coordinates of the aligned strings on r and t.

Before proceeding, we discard those alignments whose error rate (number of errors divided by aligned reference length) is above a given error rate threshold ε as previously discussed in Section 2.2.7.

3.3.7 Determining methylation patterns and rates

By inspecting an alignment, the methylation state of individual CpGs can be easily determined. We classify a CG in the reference aligned to a CG in the read as methylated. It is classified as unmethylated if it is aligned to a TG in a C-to-T read or to a CA in G-to-A read. As discussed in the previous section, if a CG in the reference is aligned to neither CG nor TG/CA in the read, then that CG is ignored.

In our study (Zeschnigk et al., 2009), we report both individual methylation patterns of single reads and also aggregated methylation rates. The methylation rate of a single read is determined as $\frac{\#\text{methylated}}{\#\text{methylated}+\#\text{unmethylated}}$ if methylation status of at least one CG could be determined. The read is removed from further consideration otherwise.

Reads with a degree of methylation above 75% and below 25% are referred to as fully methylated and unmethylated, respectively. Others are referred to as partially methylated.

3.3.8 Finding unconverted parts of reads

In our study (Zeschnigk et al., 2009), a problem in the experimental setup had the effect that reads were partially not bisulfite-converted. Since the problem is due to an enzymatic reaction that starts at one end of the DNA fragment, we know that either a prefix or a suffix is unconverted, but we do not know which one and also not how long it is. To process such data, we therefore always use the "relaxed" algorithms for hit and seed extension. We then try to recognize the unconverted part of the read in the way described below and ignore it when computing the methylation pattern and rate.

The first step is to identify those cytosines in the alignment whose conversion status we know for certain. For C-to-T reads, a C aligned to a T is *converted*. A C that is not in CpG context and that is aligned to a C is *unconverted*. Similar rules hold for G-to-A reads.

We assign a score of $+1$ to converted and -1 to unconverted bases and obtain the sequence $Z = (z_1, \dots, z_{|z|})$, where $z_i \in \{-1, +1\}$. Assume that a prefix of the read is converted. Then ideally, there is an index k' such that $z_i = +1$ for all $i < k'$ and $z_i = -1$ for all $i \geq k'$. To take errors into account, we define the best k to be the one that maximizes the sum over scores in the converted prefix of the read:

$$k_{\text{prefix}} = \operatorname*{argmax}_{k} \sum_{i<k} z_i$$

When a suffix of the read is converted, the best k is

$$k_{\text{suffix}} = \operatorname*{argmax}_{k} \sum_{i<k} (-z_i) \ .$$

In both cases, when there is a tie, choose the option that leads to a shorter converted part of the read. To determine whether a prefix or suffix is bisulfite-converted, compute both k_{prefix} and

k_{suffix} and pick the one for which the number of errors is lowest, where errors are unconverted bases within the converted region and converted bases within the unconverted region.

Furthermore, the number of errors is added to the number of errors already determined for the alignment. If the error rate becomes greater than ε, the alignment is discarded. Otherwise, the unconverted part is removed.

3.3.9 Counting bisulfite strings

In this section, we answer the question of how many possible strings of length n exist that are found as a substring in a bisulfite-modified read following strict bisulfite modification rules. We consider only C-to-T read types. This allows us to answer, for example, how many buckets are unused in a bisulfite q-gram index that includes only C-to-T simulated q-grams.

Strict bisulfite matching means that no lone Cs occur: Any C must be followed by G, unless it is the last C of the string. Equivalently, the substrings `CA`, `CC` and `CT` must not occur in the string.

Definition 8. Given a string $s \in \Sigma^n_{\text{DNA}}$, we say that it is a *bisulfite string* if

$$s_i = \mathtt{C} \quad \Rightarrow \quad s_{i+1} = \mathtt{G} \quad \text{for all} \quad i = 1, \ldots, n-1 \, .$$

Let $\mathcal{B}_n$ be the set of all bisulfite strings of length n, and $b_n := |\mathcal{B}_n|$. We first estimate b_n before giving the exact solution. Since $\mathcal{B}_n$ is a subset of Σ^n_{DNA}, b_n is less than 4^n for large enough n. If no `C` occurs in the string, no restrictions apply and therefore all strings over a three-character alphabet without `C` are a subset of $\mathcal{B}_n$, giving us $b_n \geq 3^n$. Since the alphabet is closer to three than to four (thirteen out of sixteen dinucleotides are allowed), we estimate that $b_n \in \Theta\big((3+\varepsilon)^n\big)$, where $0 < \varepsilon < 1$ and ε is a little closer to 0 than to 1.

Lemma 3. The number of bisulfite strings is

$$b_n = \frac{1+\alpha}{\alpha-\beta}\alpha^n - \frac{1+\beta}{\alpha-\beta}\beta^n \tag{3.2}$$

where

$$\alpha = \frac{3+\sqrt{13}}{2} \quad \text{and} \quad \beta = \frac{3-\sqrt{13}}{2} \, .$$

Proof. The problem of counting bisulfite strings is similar to Exercise 7.42 in the book by Graham et al. (1994) and is solved here along the same lines of thought. See also Section 6.6 in the book. We start by giving a recurrence for b_n, then derive a generating function and find a closed form for its coefficients.

Let $\mathcal{A}_n$ be the set of bisulfite strings that do not end in `C`, and $\mathcal{C}_n$ the set of bisulfite strings that do end in `C`. Let $a_n := |\mathcal{A}_n|$ and $c_n := |\mathcal{C}_n|$ be their respective sizes. Then

$$b_n = a_n + c_n \, . \tag{3.3}$$

The recurrence for a_n is

$$\begin{aligned} a_0 &= 1 \\ a_n &= 3a_{n-1} + c_{n-1} \;. \end{aligned}$$

That is, to get a bisulfite string not ending in `C`, we can either extend a bisulfite string not ending in `C` with `A`, `G` or `T`; or we extend one ending in `C` with `G`. Also, $a_0 = 1$ since the empty string does not end in `C`. We follow Graham et al.'s convention of letting $x_i = 0$ for $i < 0$. Then, using Iverson bracket notation, this can be rewritten as

$$a_n = 3a_{n-1} + c_{n-1} + [n = 0] \;.$$

The recurrence for c_n is

$$\begin{aligned} c_0 &= 0 \\ c_n &= a_{n-1} \;. \end{aligned}$$

That is, a bisulfite string ending with `C` can be obtained by simply appending a `C` to one that does not end in `C`. Also, there are no strings of length `0` that end with `C`.

From the above two recurrences, we see that the generating functions $A(z) = \sum_n a_n z^n$ and $C(z) = \sum_n c_n z_n$ are

$$A(z) = 3zA(z) + zC(z) + 1 \quad \text{and} \tag{3.4}$$
$$C(z) = zA(z) \;. \tag{3.5}$$

Inserting Equation (3.5) into Equation (3.4) and solving for $A(z)$ gives us

$$A(z) = \frac{1}{1 - 3z - z^2} \;.$$

The generating function $B(z) = \sum_n b_n z^n$, taking Equation (3.3) into account, therefore is

$$B(z) = A(z) + C(z) = \frac{1 + z}{1 - 3z - z^2} \;.$$

To find its coefficients, we can use the same method as in Graham et al. (1994, Chapter 6.6), where a closed form for the Fibonacci series is derived, whose generating function $\frac{z}{1-z-z^2}$ is similar to the above. First, assume that the partial fraction expansion is

$$\frac{1+z}{1-3z-z^2} = \frac{P}{1-\alpha z} + \frac{Q}{1-\beta z} = \frac{P - P\beta z + Q - Q\alpha z}{(1-\alpha z)(1-\beta z)} \;.$$

We thus need to solve

$$(1-\alpha z)(1-\beta z) = 1 - 3z - z^2 \quad \text{and} \tag{3.6}$$
$$P + Q - (P\beta + Q\alpha)z = 1 + z \;. \tag{3.7}$$

Choosing $\alpha = \frac{3+\sqrt{13}}{2}$ and $\beta = \frac{3-\sqrt{13}}{2}$ satisfies Equation (3.6):

$$\begin{aligned}(1-\alpha z)(1-\beta z) &= \left(1 - \frac{3+\sqrt{13}}{2}z\right)\left(1 - \frac{3-\sqrt{13}}{2}z\right)\\ &= 1 - \frac{3-\sqrt{13}}{2}z - \frac{3+\sqrt{13}}{2}z + \left(\frac{3+\sqrt{13}}{2}\right)\left(\frac{3-\sqrt{13}}{2}\right)z^2\\ &= 1 - \frac{6}{2}z + \frac{1}{4}(9-13)z^2\\ &= 1 - 3z - z^2\end{aligned}$$

Setting $z = 0$ in Equation (3.7) gives us $Q = 1 - P$, and then setting $z = 1$ in the same equation results in

$$P + (1-P) - \big(P\beta + (1-P)\alpha\big) = 2\,.$$

And therefore

$$P = \frac{1+\alpha}{\alpha+\beta} \quad \text{and} \quad Q = \frac{1+\beta}{\alpha-\beta}\,.$$

According to Graham et al., Equation (6.118), the coefficients of the partial fraction expansion

$$B(z) = \frac{P}{1-\alpha z} + \frac{Q}{1-\beta z}$$

are

$$b_n = P\alpha^n + Q\beta^n,$$

which gives us the lemma

$$b_n = \frac{1+\alpha}{\alpha-\beta}\alpha^n - \frac{1+\beta}{\alpha-\beta}\beta^n\,.$$

□

The formula can be simplified as described by Graham et al. We observe that the right-hand term in Equation (3.2) is small and gets smaller with increasing n. It can be omitted if we instead round the first term to the closest integer. If we additionally insert constants α and β, we get

$$b_n = \left\lfloor \frac{5\sqrt{13}+13}{26}\left(\frac{3+\sqrt{13}}{2}\right)^n + \frac{1}{2}\right\rfloor\,.$$

Since $\frac{3+\sqrt{13}}{2} \approx 3.30277$, this finding is in line with our initial estimate. Some values for b_n are shown in Table 3.4.

Table 3.4 Some values for the number b_n of bisulfite strings of length n, compared with the number 4^n of DNA strings of length n.

n	b_n	4^n	$\frac{b_n}{4^n}$ (%)
0	1	1	100.00
1	4	4	100.00
2	13	16	81.25
3	43	64	67.19
4	142	256	55.47
5	469	1 024	45.80
6	1 549	4 096	37.82
7	5 116	16 384	31.23
8	16 897	65 536	25.78
9	55 807	262 144	21.29
10	184 318	1 048 576	17.58
11	608 761	4 194 304	14.51
12	2 010 601	16 777 216	11.98
13	6 640 564	67 108 864	9.90
14	21 932 293	268 435 456	8.17
15	72 437 443	1 073 741 824	6.75
20	28 468 099 417	1 099 511 627 776	2.59
25	11 188 035 508 324	1 125 899 906 842 624	0.99
30	4 396 926 422 870 754	1 152 921 504 606 846 976	0.38

3.3.10 Bucket compression

Individual buckets of the q-gram index as created by Algorithm 6 contain q-gram start positions in ascending order. We explore in this section whether it makes sense to store only the differences (Δ values) between the positions in order to save memory.

Büttcher et al. (2010, Chap. 6.3) have already summarized applicable methods for compressing occurrence lists in the context of *inverted text indices*, which are used in (web) search engines. The authors describe how to compress *posting lists*, which contain document indices. These are, for our purposes, equivalent to positions within q-gram buckets.

Given a text of length n, the index needs to store roughly n positions of q-grams. Since the differences between positions in each bucket can be on the order of n, we need to use a variable-width encoding to save any memory.

Büttcher et al. describe multiple such codes, but argue that some of them, such as Elias' γ and δ code (Elias, 1975) cannot be decompressed quickly enough. One of the advantages of the q-gram index is the speed with which single buckets can be accessed, so we follow this argument and concentrate on byte-aligned codes although these may not result in the best compression.

The *Base 128 variable integer code* or *varint* is a byte-aligned code that can represent arbitrary integers. It is, for example, implemented in the Protocol Buffers serialization format by Google[1]. It represents each integer as a series of bytes in which the lower seven bits of each byte contain the actual value of the integer. The most significant bit of each byte indicates whether another byte follows. This representation needs 8 bits for values up to $2^7 - 1$, 16 bits for values up to $2^{14} - 1$, and so on. The same idea, but with 16 bit words with a payload of 15 bits each, is the *Base* 2^{15} *variable integer code*.

Let us estimate the average Δ value. If n positions are distributed over 4^q buckets, there are on average $\frac{n}{4^q}$ positions per bucket. The average distance between two positions is therefore roughly $\frac{n}{n/4^q} = 4^q$, ignoring the first position in each bucket. We come to the same conclusion if we argue that, in a random text (of infinite length), the average distance between two occurrences of a given q-gram is 4^q. Thus, we would need $2q$ bits per difference on average.

With realistic q-gram lengths $q \geq 12$ when indexing the human genome, each difference requires 24 bits on average. The base 128 varint encoding therefore needs four bytes on average, which is not better than storing absolute positions (four bytes each).

Since we know that the average value requires around 24 bits, it makes sense to consider a custom byte-aligned value-length integer code that needs fewer bits than base 128 and base 2^{15} varint codes for typical data. We therefore define two versions of a base 2^{23} varint code. The first version (named 23+7) always uses at least three bytes, of which 23 bit are usable. If more bits need to be encoded, further bytes are used. That is, the code uses $24 + 8k$, $k \geq 0$ bits per value. The second version (named 23+15) also uses three bytes for small values, but then adds groups of two bytes. Thus, the code uses $24 + 16k$, $k \geq 0$ bits per value.

So far, our thoughts hold only for the regular q-gram index. When we consider the bisulfite q-gram index, we need to consider that it contains more positions than the regular index and that a few buckets are much larger than others due to bisulfite simulation (as seen in Table 3.4). Both properties decrease the average difference between positions.

[1] https://developers.google.com/protocol-buffers/

3.4 Results

We evaluate size and compression of the bisulfite q-gram index and then show mapping results for a bisulfite-sequencing dataset. We use the following two references.

CpG islands

A reference sequence (named "CGIs") of 27 639 known CpG islands was created by retrieving CpG island locations from the UCSC database[2] and extracting the corresponding substrings from the human genome (NCBI Build 36.1), including a context of 200 bp before and after each island. The resulting FASTA file contains 32 156 976 bp in total, of which 9.1% are marked as repeats (lowercase letters), which are ignored at index creation time.

Chromosome 1

The second reference (named "Chr1") is human chromosome 1 of the GRCh37 sequence (Church et al., 2011). It has a length of 249 250 621 bp. Of these, 23 970 000 bp (10.6%) are 'N' characters, which are ignored at index creation time. Lowercase letters are not ignored.

3.4.1 Index size

In the worst case, runtime of the creation and memory consumption of the bisulfite q-gram index is exponential in q for both the B and P arrays. As described in Section 3.2.2, the worst case is reached only for sequences consisting entirely of CpG dinucleotides. For realistic data, this is not the case. As discussed in the Section 3.1.1 (page 50), only 1% of observed dinucleotides are CpGs and we therefore expect a more benign behavior for the size of P.

Let us define the *bisulfite factor* of a q-gram index to be the number of positions stored divided by the number of q-grams on the reference that were not skipped. For a regular q-gram index, this is equal to 1. For a bisulfite q-gram index, each non-skipped q-gram gives rise to an unconverted q-gram, simulated C-to-T q-grams, and simulated G-to-A q-grams. If no CpG dinucleotides occur, there is one of each and the bisulfite factor will be slightly below 3 since there is some overlap if no `C`, no `G` or none of both occurs. If CpGs occur, the bisulfite factor increases accordingly. The factor tells us how much more work is needed to construct a bisulfite q-gram index instead of a regular index.

In Table 3.5, we have summarized sizes of the bisulfite q-gram index for both datasets, given varying values of q. For the index over the full chromosome, the bisulfite factor starts out at 2.83 for $q = 9$. As q increases, more CpGs per q-gram occur and the factor rises slowly up to 3.16 at $q = 14$, confirming our expectation that the exponential increase in q of runtime and memory usage is almost negligible in practice for typical sizes of q.

Since CpG density is much higher in CpG islands by definition, the bisulfite factor is larger in the CGIs dataset. Its increase is also more pronounced, growing from 4.57 at $q = 9$ to 6.62 for $q = 14$.

Table 3.5 Size of bisulfite and regular q-gram indices for the CGIs and Chr1 references. The "size" is the number of positions stored in the index, either in total or per bucket. For the CGIs reference, the regular index gets noticeably smaller with increasing q since it consists of multiple reference sequences (see main text), in which q-grams overlapping a border are ignored.

		Total size		Average bucket size		
Dataset	q	Regular	Bisulfite	Regular	Bisulfite	Bis. factor
CGIs	9	31 935 549	145 957 159	121.82	556.78	4.57
CGIs	10	31 907 909	157 250 825	30.43	149.97	4.93
CGIs	11	31 880 269	169 171 565	7.60	40.33	5.31
CGIs	12	31 852 629	181 869 010	1.90	10.84	5.71
CGIs	13	31 824 989	195 629 713	0.47	2.92	6.15
CGIs	14	31 797 349	210 426 758	0.12	0.78	6.62
Chr1	9	225 280 317	637 414 158	859.38	2431.54	2.83
Chr1	10	225 280 279	655 561 520	214.84	625.19	2.91
Chr1	11	225 280 241	671 771 220	53.71	160.16	2.98
Chr1	12	225 280 203	686 524 952	13.43	40.92	3.05
Chr1	13	225 280 165	700 154 548	3.36	10.43	3.11
Chr1	14	225 280 127	712 809 260	0.84	2.66	3.16

3.4.2 Analyzing bucket compression

The q-gram indices that are listed in Table 3.5 are also analyzed for compressibility according to Section 3.3.10. To denote the encoding, we write "$x + y$" (such as 23+7). This means that the first group of bits has a payload of x bits and all further groups, if they exist, have a payload of y bits. Each group, as discussed previously, needs one additional bit of storage to indicate whether another group follows. For each non-empty bucket, its first value is left unchanged and all remaining ones are replaced with their difference to the previous value. All values are then encoded.

A typical implementation of a q-gram index using 32-bit words for positions needs 4 bytes per position. As we see in Table 3.6, this can be reduced considerably by the described encoding schemes, excluding the $23 + 15$ variant, which needs more than four bytes per position for large q.

For all datasets, either the $15 + 7$ or the $23 + 7$ encoding uses fewest bytes, where $23 + 7$ tends to be the better one for q greater than twelve. The predefined $7 + 7$ and $15 + 15$ encoding schemes always perform worse. As we expect, fewer bytes are used per position in a bisulfite index compared to the regular index for otherwise the same parameters. This is due to the higher density of positions, leading to smaller differences. For the converse reason, more bytes are used when q gets larger.

Interestingly, even in those datasets with very small average bucket sizes, compression is advantageous. This can be explained by the nonuniform distribution of their sizes, as illustrated by the large differences between median and average.

Although the largest analyzed reference here is chromosome 1, we expect the observed trend to

[2]http://hgdownload.cse.ucsc.edu/goldenPath/hg18/database/cpgIslandExt.txt.gz

Table 3.6 Compressibility of the q-gram index by variable-length encoding of differences between positions. Values in the *Encoding* column are the average bytes per position. They should be compared to the standard scheme that typically uses 4 bytes per position. The best values in each row are set in bold.

			Bucket size		Encoding				
Dataset	q	Type	Avg.	Median	7+7	15+7	15+15	23+7	23+15
CGIs	9	Bis.	556.8	154	2.318	**2.264**	2.527	3.000	3.000
	9	Reg.	121.8	62	2.767	**2.681**	3.355	3.000	3.001
	10	Bis.	150.0	31	2.541	**2.455**	2.897	3.002	3.003
	10	Reg.	30.4	14	2.980	**2.886**	3.676	3.013	3.026
	11	Bis.	40.3	6	2.748	**2.653**	3.240	3.014	3.027
	11	Reg.	7.6	3	3.242	**3.107**	3.816	**3.092**	3.183
	12	Bis.	10.8	1	2.939	**2.840**	3.502	3.045	3.091
	12	Reg.	1.9	1	3.493	3.354	3.870	**3.254**	3.509
	13	Bis.	2.9	0	3.119	**3.012**	3.673	3.100	3.200
	13	Reg.	0.5	0	3.664	3.554	3.891	**3.433**	3.866
	14	Bis.	0.8	0	3.293	**3.176**	3.777	**3.178**	3.357
	14	Reg.	0.1	0	3.754	3.674	3.900	**3.564**	4.128
Chr1	9	Bis.	2431.5	780	2.403	**2.330**	2.656	3.001	3.002
	9	Reg.	859.4	626	2.822	**2.723**	3.431	3.003	3.006
	10	Bis.	625.2	175	2.618	**2.518**	3.010	3.006	3.012
	10	Reg.	214.8	131	2.996	**2.902**	3.722	3.020	3.039
	11	Bis.	160.2	38	2.846	**2.722**	3.307	3.032	3.065
	11	Reg.	53.7	25	3.291	3.121	3.836	**3.097**	3.193
	12	Bis.	40.9	8	3.063	**2.939**	3.529	3.113	3.227
	12	Reg.	13.4	5	3.577	3.430	3.882	**3.333**	3.666
	13	Reg.	3.4	1	3.733	3.648	3.904	**3.586**	4.173
	14	Reg.	0.8	0	3.803	**3.753**	3.918	**3.730**	4.460

Table 3.7 Statistics of read lengths. Processed reads are those that have undergone adapter removal.

	Median length	Average length	Total no. of nucleotides	No. of sequences
Blood reads	120	135.4	22081676	163034
Processed blood reads	119	134.1	21870876	163034
Sperm reads	117	133.4	17291091	129620
Processed sperm reads	115	132.0	17115073	129620

also apply to indices over the full human genome reference. Since the average distance between occurrences of the same q-gram remains the same, the same number of bits will be required to encode position differences.

Overall, bucket compression works best on bisulfite q-gram indices. It offers significant memory savings of around 40% for small q and still around 10% for large q when the appropriate custom encoding is used.

3.4.3 Bisulfite sequencing of human CpG islands

We applied our method to a bisulfite sequencing dataset of CpG-rich DNA fragments (Zeschnigk et al., 2009). Two DNA samples (blood of a female and sperm) were obtained and digested with an optimized mix of restriction enzymes in order to enrich fragments associated with CpG islands. Adapters were ligated and single-stranded DNA was bisulfite treated. The adapters contain no unmethylated cytosines and are therefore unaffected by bisulfite conversion. After PCR, prepared fragments were sequenced on a 454 Roche Genome sequencer FLX. Processing of sequencing data is fully automated through the use of a Makefile.

Reads

As a first processing step, cutadapt (Chapter 2) is used to remove adapter contamination from the reads. Since some reads contain multiple adapters or adapter fragments, possibly due to problems during adapter ligation, the program was set to repeat the read trimming process for each read up to three times (“`--times 3`”). See Table 3.7 for read lengths before and after pre-processing.

Bisulfite read mapping

For mapping, a bisulfite q-gram index of the CGIs reference with $q = 10$ is created. Reads and their reverse complements were then mapped in the “relaxed” mode, with an error rate threshold of 5% and minimum seed length $\ell_{\min} = 25$. Unconverted parts of reads are excluded from further consideration (see Section 3.3.8), and also those parts of the read that do not cover a CpG island. All remaining reads that map uniquely to one position on the reference are then classified into fully methylated, partially methylated or unmethylated (Section 3.3.7).

Results

12 358 blood reads and 10 216 sperm reads could be mapped uniquely to a CpG island. In the X-chromosomal CGIs of the blood sample, we find a generally higher rate of methylation than

in autosomes. This is to be expected since inactivation of one of the X-chromosomes in females is achieved through methylation. An unexpected result, however, is that we see many partially methylated CGIs on this chromosome. Partial methylation is observed in only 3.8% of autosomal CGIs, but in 25% of those on the X chromosome. Statistical analysis of the methylation rates shows the difference to be significant. See the article by Zeschnigk et al. for more results.

3.5 Future work

The bisulfite read mapping software presented comprises two components, the bisulfite index (and its construction algorithm) and a mapping algorithm that uses that index. We discuss some improvements that can be made to these two components.

A problem that many q-gram indexing methods have is the large memory usage. The bisulfite index, with triple the size of a regular index, suffers even more from that problem. Our work on index compression mitigates the problem only partially. Some of the following suggestions therefore involve a reduction of memory usage. The first two have already been implemented in the Verjinxer software, but were not yet evaluated.

Read indexing

A surprisingly simple to implement change in the algorithm is to index the reads instead of the reference as is done, for example, in the MAQ software (Li et al., 2008a). The index itself is created as a regular (non-bisulfite) index. To find hits, bisulfite simulation is done for each q-gram of the reference (from left to right) and the resulting q-grams are then then looked up in the index. Further processing (seed extension etc.) is the same as before.

This method reduces memory usage if the total length of all reads is less than the length of the reference. It therefore does not apply to current datasets that often contain tens of millions of reads. The Verjinxer software supports mapping by indexing the reads.

Strided indexing

To reduce memory consumption, one could include not all positions i in the index, but only those that are multiples of a constant d, called *stride*, as is done in the BLAT software (Kent, 2002). The case $d = 1$ is the same as before and values up to $d = q$ may make sense, which means that all non-overlapping but adjacent q-grams are included. Thinning out positions in this way reduces sensitivity, but also improves runtime as there are fewer positions in each bucket. Algorithm 7 (Find-Seeds) needs to be adjusted, too, since the shown version assumes in step 3a that adjacent q-grams that belong to the same maximal bisulfite match have coordinates that differ by exactly one. Also, the algorithm needs to extend each seed to the left, not only to the right. For a stride $d > 1$, the memory usage could be reduced further by storing not position i, but i/d, thus reducing the number of bits needed for each position if an appropriate encoding scheme is chosen. For example, for $d = 8$, three bits can be saved per position.

Some preliminary support for BLAT-like strided indexing has been implemented in the Verjinxer software.

True hashing

The bucket array B grows exponentially in q. As soon as the number of buckets is about the same as the number of positions that need to be stored, it makes sense to limit B to a fixed length. As done in regular hash tables, positions for q-code c are then stored at $B\left[c \mod |B|\right]$. The usual considerations for hash tables apply. For example, $|B|$ should be a prime number in order to distribute positions more evenly across buckets. Index construction remains almost unchanged, except for the modulo term, but obtaining a set of hits for a given q-gram needs to be modified: Each retrieved position needs to be checked for whether it is compatible with the requested q-gram. The overhead for this is likely small as relaxed/strict hit extension already needs to compare the corresponding substrings on reference and read.

Avoiding C-to-T/G-to-A type classification

Since the bisulfite index mixes C-to-T- and G-to-A-type simulated q-grams, the appropriate type needs to be determined after a position has been looked up. This classification could be avoided in at least two ways. The first idea is store the type in the index along with the position. Since the types are C-to-T, G-to-A and "unconverted", this requires two additional bits, which could be stored in the same machine word as the position. This leaves only 30 bits for the position for typical word sizes and should therefore be combined with one of the schemes for index compression proposed in Section 3.3.10.

The second idea is to split up the index into three indices: two indices over the C-to-T and G-to-A-converted references, respectively, and one regular q-gram index of the unconverted reference. On-disk memory usage would increase slightly since each position is stored at least three times unlike in the combined index, where some positions are stored only once. A read would then be mapped against all three indices, where the regular q-gram index would be used last and serves as a kind of "fallback". When reads are partially unconverted, the regular index needs to be used at the same time as one of the others, which likely makes mapping more complicated.

Improved mapping algorithms

The relatively basic mapping algorithm described in this chapter may be considered to be a proof-of-concept only. Other algorithms can be used on top of the bisulfite q-gram index. Particularly interesting may be to find a bisulfite version of Lemma 2 in the paper describing the SWIFT software (Rasmussen et al., 2006), which is also the basis for RazerS (Weese et al., 2009). Both tools allow read mapping at full sensitivity. Other techniques, such as using gapped q-grams or multiple indices with differently shaped q-grams (see Section 3.1.5 on p. 55), may also be used and are orthogonal to the bisulfite index itself.

3.6 Discussion

We could show that mapping reads with the bisulfite q-gram index works well on real-world data, enabling advances in medical research. However, one needs to be aware that the basic algorithms in the presented tool were finished already in 2009 and that it therefore belongs to the first generation of bisulfite read mappers. As a testament to this, note that the Verjinxer software does not support the SAM/BAM format because the format had not been invented at the time. First-generation tools – including ours – are comparatively slow and use large amounts of memory. This does not diminish our contribution regarding the mapping algorithm: The concept of simulating bisulfite treatment for each q-gram has been imitated and varied successfully in other mapping tools that appeared subsequently.

Many recent read mapping tools, including those with bisulfite capabilities, tend to employ data structures based on the FM index (Ferragina and Manzini, 2000), which in turn uses the Burrows-Wheeler transform (Burrows and Wheeler, 1994). Asymptotic runtimes are close to the suffix tree (depending on implementation and on the machine model), but memory usage is much smaller.

The advantage of the q-gram index is its extreme simplicity coupled with the ability to instantly obtain a requested list of q-gram positions. So far, the disadvantage of its large memory usage has negated that advantage and compressed indices dominate. However, compression comes with a runtime overhead, and it is possible that, with rising main memory sizes but genome sizes remaining constant, the scale may tip in favor of q-gram indices again.

4 Analyzing Exome Sequencing Data

The *exome* is the set of all exons of a genome. Since genetic diseases are usually caused by protein-changing mutations, it is most efficient to limit the search for mutations to the exome, which makes up only about 1%–2% of the genome in humans. Efficient techniques have recently been developed to enrich exonic DNA in regular DNA samples, making targeted sequencing possible. As human whole-genome studies are, as of now, still too expensive for routine studies, exome sequencing has become the method of choice for identifying genes that underlie genetic diseases.

While many bioinformatics tools exist that help in the analysis of exome sequencing data, these are usually command-line tools, many of which need to be run in a particular order for a full analysis, resulting in a cumbersome workflow with repetitive tasks. We have therefore developed a pipeline called *Exomate* that automates most of the tasks and provides an interactive web interface to the medical researcher who can easily get desired analysis results and interactively re-run parts of the analysis with adjusted parameters.

We used the pipeline in studies that identify mutations in uveal melanoma (Martin et al., 2013), in patients with Nager syndrome (Czeschik et al., 2013) and Oto-facial syndrome (Voigt et al., 2013). It is also being used to study mutations related to Goldenhar syndrome, retinoblastoma and other diseases.

Exomate is available under the MIT license at https://bitbucket.org/marcelm/exomate/.

4.1 Introduction

After we motivate our research, we give an overview of what exome sequencing entails and describe work by other groups whose existing tools we have integrated into our software.

4.1.1 Structure and motivation

Exomate is comprised of three components. The first part is the computational backend. It is probably that component which is most similar to other pipelines that were created at the same time in other research groups, but not publicly available when we started. The backend was created to automate all computational tasks involved in exome sequencing and to make them reproducible.

The second component of Exomate is the database. Using a relational database engine is a crucial difference to many other pipelines and provides greater flexibility. Other algorithms often work with specially formatted files and manipulate them in various ways in order to annotate, aggregate, extract, sort and retrieve the desired information. However, relational database engines were designed for exactly those tasks. By storing most of our data in the database, we can leverage existing optimizations that went into making the database engine fast. Also, queries that otherwise require running one or more specialized tools can be formulated by writing one or a few lines of code in the standardized Structured Query Language (SQL).

The third part of Exomate is a web frontend that was created out of the desire to remove a bottleneck in multidisciplinary collaborations between computer scientists or bioinformaticians on the one side, and medical researchers without computer science background on the other side. A “traditional” model of cooperation is as follows: The computer scientist is asked to perform some data analysis, does so and hands the results to the medical researcher. Then there is feedback from the recipient and the process is iterated when problems need to be fixed, parameters need to be changed, or when new data arrives. The problem, of course, is that the assistance of bioinformaticians/computer scientists is required for virtually everything. For small groups, this does not scale: If the responsible persons leave, the knowledge is gone, too.

Our solution to this is to introduce a piece of easily usable software in the form of a web frontend that is placed in the middle and gives medical researchers more direct access to the data, without requiring them to become programmers. Data analysis is then done through the interaction between the medical researcher and the software, thus reducing reliance on a human “middle man”. This does not obviate the need for communication: Discussions about what to implement and how to do so still take place, but the analyses themselves can be done independently.

The separation into three components is not only on the conceptual level, but on a technical one: Each component can, if desired, be run on a different machine.

4.1.2 Exome sequencing

Exome sequencing was pioneered by Ng et al. (2009), who enriched DNA fragments that contain coding sequences by using two custom microarrays before sequencing those fragments on an Illumina Genome Analyzer II. The authors later applied their method to discover the cause of Miller syndrome (Ng et al., 2010).

It is estimated that 85% of disease-causing mutations are within the exome (Antonarakis et al., 1995). Combined with the fact that the price for sequencing an exome is about one tenth of sequencing an entire genome (Baker, 2011), exome sequencing has found widespread adoption as the method of choice for discovering the cause of genetic diseases.

Exome sequencing has been used to find the causative gene for Miller syndrome (Ng et al., 2010), to find genes involved in Schinzel-Giedion syndrome (Hoischen et al., 2010), to identify mutations related to Coffin-Siris syndrome (Santen et al., 2012), and in many other studies, see also the reviews by Bamshad et al. (2011) and Gilissen et al. (2011, 2012).

Exome capturing

Commercial so-called *exome capture kits* that enrich exonic DNA in DNA samples have become available by vendors such as Agilent, Illumina and Roche. The exome data with which our pipeline was tested and to which some of the examples in this chapter refer was obtained mainly with *Roche NimbleGen SeqCap EZ Human Exome Library v2.0* (EZ2) and *v3.0* (EZ3)[1]. Earlier capture kits were based on microarrays, but have been superseded by kits that work in solution with free-floating biotinylated oligos, called *baits*. Exonic DNA fragments hybridize to the baits, which in turn bind to magnetic beads. These are then extracted and washed, leaving the enriched molecules (Bamshad et al., 2011, Box 1).

[1]http://www.nimblegen.com/products/seqcap/ez/v2/ (EZ2) and …/v3/ (EZ3)

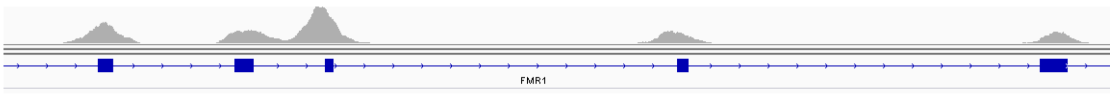


Figure 4.1 This image shows the coverage of some exons (thick bars) of the *FMR1* gene in an exome sequencing dataset of a sample of blood from a healthy patient. The shown region is chromosome X, positions 147 016 188 to 147 025 995. The SeqCap EZ 3.0 capture kit was used. We can see that usable coverage extends into introns (thin line with dashes). Screenshot take from the Integrative Genomics Viewer (IGV).

The regions of DNA captured by the kits differ from the coding regions. The *target regions* are those chosen by the manufacturer to be included. They also vary from kit to kit depending on how many probes are used. Since the capture probes have a certain length, the regions are extended for very short exons, yielding the *capture regions*. For example, SeqCap EZ 2.0 targets regions that total 36.5 Mbp, but captures 44.1 Mbp. SeqCap EZ 3.0 captures 64 Mbp. Both SeqCap EZ 2.0 and 3.0 also capture some regions annotated as microRNA, which are disregarded in the current version of our pipeline.

Since captured DNA fragments extend into regions outside the capture targets, it is possible to find mutations not only in coding regions themselves, but also in short sections of intronic and UTR regions that are close to coding regions, see also Figure 4.1. This makes it possible to reliably analyze intronic splice-site affecting variants.

Note that the targets of exome sequencing are the coding regions only although the term "exome" implies otherwise. Non-coding parts of exons (5′ UTR and 3′ UTR regions) are usually not captured. Also, exome capture is more accurately described as *enrichment*, that is, it is not perfect and in reality some non-exome DNA fragments are also enriched and sequenced, but their coverage is low.

A comparison of three commercially available exome capture kits was done by Asan et al. (2011), who conclude that all kits have similar specificity, allow to find very similar sets of variants, but that the solution-based methods are easier to handle in the lab than the array-based methods.

Variants

Exomate focuses on the analysis of *variants*, which we define to be sequence differences between the sample of interest and a reference sequence. Currently, this is the human reference sequence (build 37) by the Genome Reference Consortium (GRC) (Church et al., 2011). The differences can be substitutions, insertions and deletions.

If a particular variant recurs in a population, the site of the variant is said to be *polymorphic*. A *single nucleotide polymorphism* (SNP) is a polymorphic site with recurring substitution variants of a single base. Differences between individuals of a population are largely due to SNPs. The dbSNP (Sherry et al., 2001) is a public database of verified polymorphisms. It mostly contains SNPs, but also polymorphic insertions and deletions.

A *mutation* is a change in DNA sequence at a point in time in a single cell. If the mutation occurs in a germ-line cell, it spreads to every cell of the child that develops from that cell. It is called *de novo* mutation. A *somatic* mutation arises in a non-germ cell, such as in a tumor.

The main task of the Exomate software is to determine which of the variants that were found are mutations that are relevant for a disease. De novo mutations can be found by sequencing both a patient and their parents (a *trio*), and excluding variants found in the patient that also occur in the parents. Somatic mutations are found by sequencing both affected and unaffected tissue of a patient and discarding those variants that are found in both samples.

Note that terminology is not strictly followed in the literature. For example, the process of finding variants is often termed *SNP calling* although *variant calling* is more accurate. In this text, we prefer the term "variant" as it includes SNPs and mutations.

Types of variants

Variants in coding regions can be distinguished by their effect on the resulting protein. Insertions and deletions of a number of bases that is a multiple of three add or remove entire amino acids. Otherwise, a frameshift occurs and the remainder of the amino acid chain is changed. Substitution mutations can be classified into the following types. Those that change the resulting amino acid are called *missense* and those that do not are called *synonymous*. If a regular codon becomes a stop codon, this is a *nonsense* mutation. If a a stop codon changes into a regular codon, this is a *read-through* mutation. In exome sequencing, we are typically interested in *non-synonymous* (indel, missense, nonsense, read-through) variants.

4.1.3 Related work

The filtering strategy of our suggested pipeline builds upon the work by Ng et al. (2010). The authors search for variants that either are non-synonymous, affect a splice site, or are indels. They exclude variants from further consideration that are also found in dbSNP, in eight HapMap control exomes (International HapMap 3 Consortium et al., 2010), or classified as *not damaging* by Polyphen (Adzhubei et al., 2010).

Our pipeline uses many tools of the Genome Analysis Toolkit (GATK, DePristo et al., 2011) and follows the recommended workflow given in their document "best practice exome variant detection v1".[2] Alternative tools for variant calling are, among others, ATLAS2 (Challis et al., 2012) and SAMTools' mpileup command (Li et al., 2009).

The idea to create a web frontend for an exome sequencing pipeline at all and the idea to model "abstract variants" was taken from an existing exome sequencing pipeline developed by Tim Strom's human genetics group at the Helmholtz center in Munich (personal communication).

4.1.4 Structure of the software

The first component of Exomate is an automated pipeline that performs all computations necessary to get from raw FASTQ files as obtained from the sequencer to high-quality variant calls. It is described in Section 4.2. The second component is a database in which variants, annotations and metadata are stored, described in Section 4.3. The third component is a web frontend that constructs database queries from user requests, submits those requests to the database and displays the results in a web browser, described in Section 4.4.

[2]The document used to be available at http://www.broadinstitute.org/gsa/wiki/index.php/Whole_exome_v1, but is no longer online. The third, similar version of those guidelines can currently be found at http://gatkforums.broadinstitute.org/discussion/15/.

All three components can use custom computational resources by running on different machines. Most resources are needed by the computational pipeline at those times when new datasets arrive. It should ideally run on a system with as many CPUs as possible. A cluster may be preferable. For day-to-day work, the database server is the bottleneck since it runs almost all interactive computations. It should therefore run on a system with high single-core performance and enough main memory to cache most of the database contents. The web frontend needs few resources as it only constructs and submits queries and shows the result to the user.

Since the Exomate software is centered around variant calls, we will describe its constituent parts in the order in which the information regarding variants flows through them: Computation, database, and web frontend. Note that not all information flows in that direction since the pipeline also accesses the database to read metadata from it.

4.2 Computational pipeline

The task of the computational pipeline is to process sequencing data in order to obtain high-quality variant calls. It is fully automated through the use of *Snakemake* (Köster and Rahmann, 2012).

Workflow description

Snakemake offers a workflow description language that extends the Python language in order to allow concisely specifying *rules* that describe how to transform one type of file to another. In the style of GNU make, whether a rule applies solely depends on the file name patterns given as input and output. File names can contain wildcards resulting, for example, in a rule that specifies how to transform "something.bam" into "something.sorted.bam" through the use of the program "samtools sort". The tool automatically creates a dependency graph (a DAG, directed acyclic graph) from all rules, given a set of target files, linearizes the jobs that need to be executed through topological sort, and distributes them to any number of CPU cores or to a batch processing system. We chose Snakemake since its usage of Python in the workflow description (named *Snakefile*) allows general-purpose programming in the same language as that used for the rest of the pipeline. This allows, for example, re-use of modules that have been written for database access. See also the paper by Köster and Rahmann (2012) for more advantages over other workflow systems. For an excerpt of the Snakefile used in Exomate, see Figure 4.2.

Input data

Two manual steps are necessary. First, the raw sequencing data needs to be copied into the working directory in FASTQ format. Second, the metadata with information about what is stored in the files needs to be added to the database (explained in Section 4.3.1). After that, all the following steps are run automatically.

4.2.1 Variant calling pipeline

The steps of the variant calling pipeline, modeled on the GATK's "best practice" document and extended, are briefly summarized in Figure 4.3 and explained below.

```
rule validate:
  input:
    bam="{file}.bam"
  output:
    validated="{file}.bam.validated"
  log:
    "{file}.bam.validatelog"
  shell:
    'picard-tools ValidateSamFile I={input.bam} > {log} 2>&1 &&'
    ' touch {output.validated}'
```

```
rule samtools_index:
  '''Index a BAM file'''
  input:
    bam="{name}.bam"
  output:
    bai="{name}.bam.bai"
  shell:
    "samtools index {input.bam}"
```

```
rule bwa_sampe:
  '''Create BAM file from a pair of sai files output by bwa aln'''
  input:
    BWAREF,
    "mapped/{ds}.1.sai", "mapped/{ds}.2.sai",
    "reads/{ds}.1.fastq.gz", "reads/{ds}.2.fastq.gz"
  output:
    bam="mapped/{ds}.unsorted.bam"
  log:
    "{ds}/bwa-sampe.log"
  run:
    sample_name = session.query(Sample).join(Library).join(Unit).\
      filter(Unit.prefix == wildcards.ds).one().accession
    rgline = "@RG\tID:{id}\tSM:{sample}\tPL:Illumina".format(
      id=wildcards.ds, sample=sample_name)
    shell(
      'bwa sampe -r "{rgline}" {input} 2> {log} | sqt-samfixn |'
      ' samtools view -bS - > {output.bam}')
```

Figure 4.2 Three exemplary rules from the Exomate Snakefile, simplified for presentation. Shell commands in a "`shell:`" section are executed after replacing "{placeholders}" with their actual values. The "`run:`" section contains regular Python code. The `session.query` line queries the database for the correct sample accession number of the given FASTQ file.

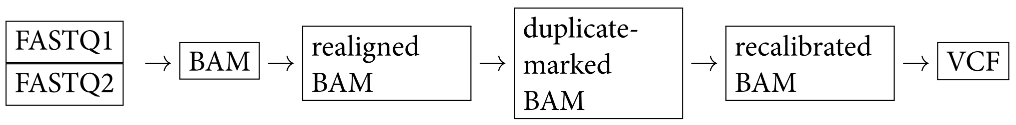


Figure 4.3 Overview of data flow from raw input sequences (two FASTQ files for paired-end sequences) to files with variant calls (VCF). This is essentially the GATK's "best practice" workflow. The diagram is simplified and leaves out some intermediate steps. Also not shown is the possibility to have multiple BAM files leading to a single VCF output file.

Read Mapping

Paired-end or single reads are mapped with BWA (Li and Durbin, 2009). The SAM/BAM files (Li et al., 2009) for the resulting mapped reads can store meta information about the dataset (see Section 1.8). Such metadata are read, for each dataset, from the appropriate database tables and added to the BAM file by supplied command-line parameters for BWA. BWA works in two stages. The first (`bwa aln`) can run multi-threaded, while the second (`bwa sampe`) cannot. For each stage, a rule in the Snakefile exists that sets the appropriate number of threads. This enables Snakemake to properly schedule execution of those rules in parallel if more than one dataset is being mapped at a time.

Most BWA parameters have been left at their default values since the program was designed with Illumina data in mind. However, we do set a quality threshold for the removal of low-quality read ends. This was verified on a low-quality dataset containing 48.5 M reads. Without quality trimming, 91.6% of all reads were mapped. With the quality-trimming threshold set to 10 ("-q 10"), this grew to 95.7%.

Indel realignment

Read mapping tools map a single read at a time and cannot take other reads that map to the same location into account. This has been shown to lead to the detection of spurious variants in the vicinity of indels (Homer and Nelson, 2010). For an example, see also Figure 4.4. The authors propose a software (SRMA) that does local re-alignment around indels, which reduces the number of spurious variants. The GATK authors have independently added a similar feature, called multiple sequence realignment (McKenna et al., 2010) to the GATK, available as the IndelRealigner tool. The latter is used in the Exomate pipeline.

Re-alignment is typically necessary when the ends of reads optimally align to an inserted or deleted sequence that is also repetitive. As shown in Figure 4.4, the read mapper will not output an alignment that (correctly) includes an insertion or deletion, but instead tends to produce an alignment that may contain mismatches. Considered in isolation, this is optimal because no or a few mismatches result in a better score than an insertion or deletion in typical scoring schemes. When there are multiple such reads, the mismatches will result in spurious variant calls. When the alignments of the other reads are considered, indel re-alignment can determine that the overall optimal configuration of reads is one in which all (or half for a diploid organism) contain the same indel, reducing the number of mismatches and therefore the number of spurious variant calls (Homer and Nelson, 2010) and at the same time increasing support for the indel.

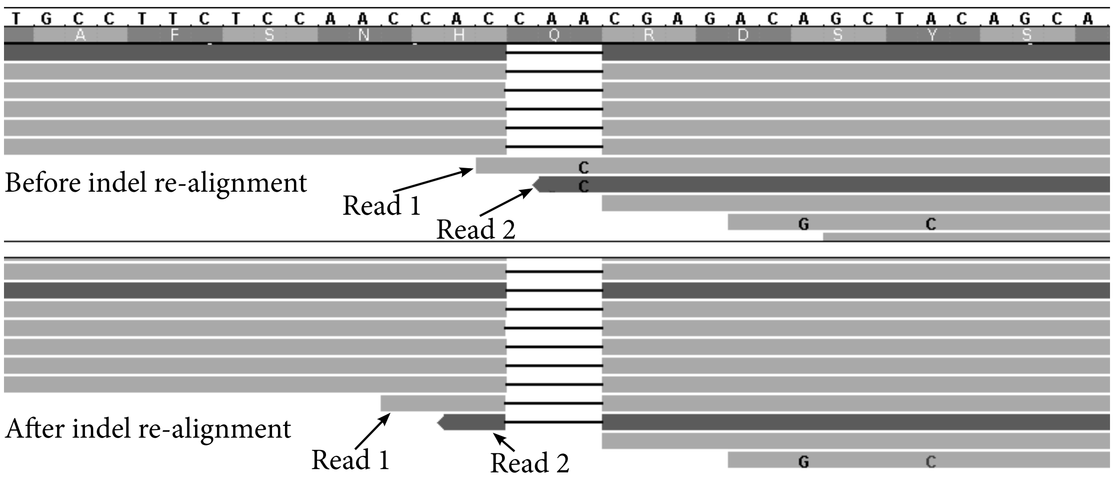

T G C C T T C T C C A A C C A C C A A C G A G A C A G C T A C A G C A
A F S N H Q R D S Y S
Before indel re-alignment
Read 1
Read 2
C
C
G
C
After indel re-alignment
Read 1
Read 2
G
C

Figure 4.4 An example showing the effect of indel realignment. Horizontal bars are reads. Characters within reads mark bases that differ from the reference (given at the top). Read 1 starts with the bases CCAC and is optimally aligned to the reference CCAA with one substitution. The same holds for Read 2, which begins with AC. The re-aligner can inspect the other reads mapping to the same location and determine that in fact a deletion of the bases CAA – which would usually incur a lower score than a single mismatch – is more consistent with the other observations. In this way, a spurious SNP substituting A with C can be avoided. Edited screenshot of the Integrative Genomics Viewer (IGV, Thorvaldsdóttir et al., 2012).

Removal of PCR duplicates

Because DNA fragments are amplified through PCR before sequencing (see Section 1.4), it is possible that there are multiple reads that originate from the same fragment. If there is an error in an early cycle of the PCR, many reads may result that show the same error. A variant caller thus incorrectly finds a high-quality SNP since it is supported by many reads.

A larger problem is that efficiency of PCR amplification depends on the sequence content, which works better with higher GC content, and is therefore biased (*PCR bias*, Polz and Cavanaugh, 1998). This means that different positions on the reference have differing coverages depending on sequence content. Since coverage, among other things, is used to compute variant quality, this makes quality values from different reference positions harder to compare. PCR bias can also skew the expected 1:1 ratio of alleles in diploid genomes.

To identify PCR duplicates, we use the tool MarkDuplicates from the Picard tool collection[3]. For single-end reads, it considers all reads to be from the same fragment whose first bases are mapped to the same coordinate and which have the same (forward or reverse complement) orientation relative to the reference.[4] For paired-end reads, two read pairs are considered to be from the same fragment if the smallest and largest coordinates match (the first base of the first read and the last base of the second read, whose reverse complement was mapped). Also, orientations of corresponding reads must match. MarkDuplicates keeps only that read or read pair whose overall base quality is largest. The tool can either discard duplicate reads or set a flag in order for

[3]http://picard.sourceforge.net/
[4]This is only documented in the Picard source code itself in `MarkDuplicates.java`.

them to be ignored in further processing. Marking reads like this has the advantage that it is still possible to get the duplicate reads if desired, for example by displaying them interactively in the Integrative Genomics Viewer (IGV, Thorvaldsdóttir et al., 2012).

Reads or read pairs can be considered duplicate also when they map to the same location by chance only but are in fact independent. For single-reads, this limits the coverage to twice the read length. The closer one gets to that, the more reads are discarded incorrectly. For exome sequencing, where target coverage is around 20 or 30 and read lengths are 100, the effect is small. For paired-end reads, the effect is even smaller since not only the start position but also the end position of the fragment needs to match and because insert sizes are highly variable.

For an exome sequencing dataset, typically between 10% and 20% of reads are marked as duplicates. If there are problems during sample/library preparation, this can grow to 90%.

Recalibrate base qualities

Next, the GATK BaseRecalibrator (DePristo et al., 2011, Online Methods) is used to recalibrate base qualities in the re-aligned, duplicate-marked BAM files. Using the known alignments of reads to the reference, the tool estimates the true sequencing error rate for each quality value and then adjusts the base qualities accordingly. Mismatches at positions that are also found in dbSNP are ignored. This step improves the accuracy of variant call qualities computed in the following step since these rely on base qualities.

Call variants

Variant calling is the process of finding differences between the sample and the given reference, done in our case with the Genome Analysis Toolkit (GATK) (McKenna et al., 2010). Since there are always sequencing errors, which could be mistaken for variants, calls are assigned a quality value by the GATK, which gives an estimate of the reliability of a variant call. Variants are discovered with the GATK's UnifiedGenotyper, which finds both indel and substitution variants in the sample. Variant lists are emitted in standard variant call format (VCF) files.

We set most parameters to their standard values, except that the minimum quality for a variant to be emitted at all is set very low. In the GATK documentation, a threshold of 50 is recommended while we set it to around 10. This increases the number of calls: Around 30% of calls in our database have a quality of less than 50. While this naturally means that many false positives are included, it gives us the freedom to dynamically set the quality threshold to any desired value when querying the database, choosing the tradeoff between sensitivity and specificity. Also, it makes it possible to use different thresholds within different parts of a single query, see Section 4.4.3.

Dealing with repeatedly sequenced samples

Some samples are sequenced multiple times. Reasons for this include a low yield (too few reads) in the first sequencing run or the desire to use an improved exome capture kit. The question arises how this data should be treated. If the reason for repeated sequencing was bad quality, then the bad data should simply be discarded. If the data was good, then it makes sense to use the data of both runs. The GATK UnifiedGenotyper supports this natively: If it receives multiple

Table 4.1 The effect of merging samples from two different sequencing runs for one of the samples that was sequenced twice. Merging was done by providing two input BAM files to GATK UnifiedGenotyper. Only exonic variants (within 1000 bp of an exon) are considered. Q is the variant quality. *On target* refers to the number of bases within uniquely mapped reads that are not marked as duplicates and that mapped to the captured region as annotated by the manufacturer.

	Bases	On target	No. of variants		
	(Gbp)	(Gbp)	$Q \geq 20$	$Q \geq 50$	$Q \geq 200$
Run 1	14	0.4	92 500	63 197	22 841
Run 2	11	3.5	296 320	216 800	120 898
Union of 1 and 2			309 734	222 558	121 982
BAM merged	25	3.9	332 776	245 814	133 420

BAM input files, these are merged on the fly. For GATK to correctly detect that the files should be merged, the metadata in the BAM header needs to have the same sample name in both files.

Merging could also be done by calling variants in the input files separately, followed by taking the union of the resulting sets of variants. This alternative method leads to worse results, as we show in Table 4.1. The difference is that merging on the BAM file level increases the total coverage and therefore leads to variant calls with higher average quality.

Table 4.1 shows data for a sample of a healthy mother of a patient diagnosed with Coffin-Siris syndrome. The sample was captured initially with the NimbleGen 2.1M microarray-based exome capture kit, yielding unsatisfactory coverage in the first sequencing run. For the second run, the exome was re-captured with SeqCap EZ v3, resulting in sufficient coverage.

Discard non-exonic variants

As shown in Section 4.1.2, exome capture also yields non-exonic reads that cover introns or the UTR. In our pipeline, we are not interested in the variants that arise from those regions, except for intronic variants very close to an exon border as these may influence splicing. Before importing the variants into the database, those located farther away than 1000 bp from an exon are removed, which reduces the total number of calls with a quality of at least 100 by 50% on average.

The exon annotation is taken from the Ensembl (Flicek et al., 2013) gene annotation track[5], currently at version 70 (see also Section 4.3.3), and VCF files are filtered with the `bedtools window` program by Quinlan and Hall (2010).

Import variants into database

Finally, all found variants are imported into the database. Before importing, indels are normalized as described in Section 4.2.5.

[5]available at ftp://ftp.ensembl.org/pub/release-70/gtf/homo_sapiens/

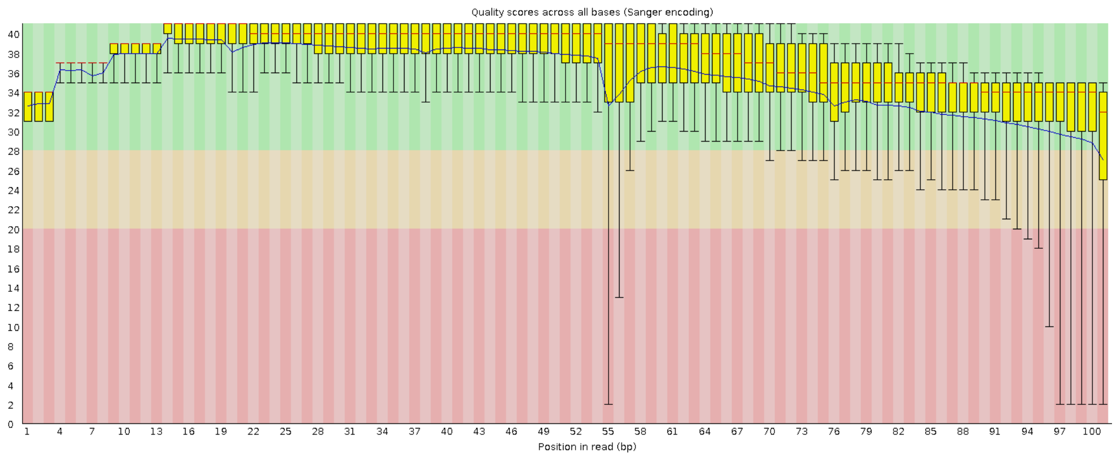


Figure 4.5 A plot of base quality distributions created by the FastQC program. Each column is a boxplot of quality values at that position in all reads. One can see the Illumina-typical decrease of quality values towards the end of the read. This example also shows an untypical decrease in quality at cycles 55–57. The full FastQC report in HTML format is available from within the web frontend through a hyperlink.

4.2.2 BAM file validation

Due to our experience with poor error reporting in some existing tools, we added extensive validation to the pipeline. The read mapper BWA ignores improperly paired FASTQ files, that is, if the n-th reads in both files are not actually from the same fragment. We detect this by checking whether corresponding read names are equal. The command `samtools index` ignores defective (truncated) BAM files. We added a validation step using Picard's ValidateSamFile command in order to detect such problems as early as possible. Also, BAM files end with a typical signature (a compressed block of length zero). Since not all downstream tools check for the existence of that signature themselves, we do so and stop further processing if we encounter an incompletely written BAM file.

4.2.3 Quality control

While the previous sections have described the main processing steps needed to go from raw reads to called variants, we have implemented other steps that mainly consist in gathering statistics for quality control purposes.

Quality control of raw sequencing data

We first count the number of reads in all input FASTQ files. Each file is also analyzed by the FastQC[6] program. We mainly use it to compute the distribution of quality values (see Figure 4.5).

[6]http://www.bioinformatics.babraham.ac.uk/projects/fastqc/

Verification of sequencing and library preparation

For each of the final BAM files, we determine the mapping rate (the fraction of reads that could be mapped), which is at least 90% for almost all of our datasets (96% on average). We also determine the fraction of mapped reads that were marked as duplicate. The duplicate rate in our datasets is highly variable: It ranges from 2% to 90%, with a median of 15%. Most of the datasets with high duplicate rates ($\geq 50\%$) were obtained within two early runs of the sequencer for which library preparation was problematic. More recent datasets generally remain below 20% duplicates.

Verification of exome capture

For all exome capture kits that we use, annotation files are available. Each mapped BAM file is run through the `bedtools coverage` tool with the appropriate annotation track, resulting in information on the coverage in the captured regions. Through further processing, we extract numbers for the "breadth" of coverage, which answers: What is the percentage of bases (relative to all captured bases) that is covered by at least 1, 5, 10 or 20 reads? Good datasets result in 80% to 90% of captured bases covered at least 20-fold.

On-target bases

As a global indicator of the efficiency of library preparation, sequencing and exome capture, the number of "on-target" bases is computed for each dataset. We define this to be the number of bases in reads that map to a target region of the capture kit (flanking regions are ignored) and that have not been marked as duplicates. The average over our datasets is 2.9 Gbp.

Chromosome-level mapping rates

The number of reads mapping to each chromosome is obtained with `samtools idxstats` and stored in a separate table in the database. We describe in Section 4.4.5 how that data is used to estimate patient gender and also to detect chromosome loss or duplication.

4.2.4 Empirical runtime

We use the exome dataset generated by DePristo et al. (2011), which is publicly available, to measure the runtime of the variant calling pipeline. The sequenced exome belongs to the NA12878 individual of the 1000 Genomes Project (The 1000 Genomes Project Consortium et al., 2010). It contains 13.5 Gbp and is therefore representative of our 169 exomes, whose median size is 13.7 Gbp (average: 13.1 Gbp; standard deviation: 4.3 Gbp). The pipeline was run on an Intel Core i7-3770 CPU (at 3.40 GHz) and with 16 GiB RAM. The processor has four hyperthreading cores allowing for eight threads. The pipeline was allowed to use six threads.

The runtimes are shown in Table 4.2. We see that at least two typical datasets can be processed per day (about 12 hours per dataset) on that machine. Throughput is limited by the non-parallel tasks, but can be improved if multiple datasets are being processed simultaneously.

4.2.5 Indel normalization

The main tasks that we are concerned with when answering user queries are set operations on sets of variants. For example: Given all variants found in a tumor sample, subtract those variants

Table 4.2 Processing times for DePristo et al.'s NA12878 exome sequencing dataset. Given times are wall-clock time. Where possible, tools were set to use six threads. Tools that cannot run multi-threaded are shown under the "1 thread" heading. Not included are runtimes for FastQC, computation of statistics, and importing of variants into the database, which are less than 1 hour in total.

Task	Time (hours)	
	1 thread	6 threads
Read mapping		3.40
Sorting	0.73	
Indel Realignment 1/2		0.75
Indel Realignment 2/2	0.91	
Duplicate marking	0.59	
Recalibration 1/2		0.69
Recalibration 2/2	1.16	
Variant Calling		3.26
Total	3.89	8.10

also found in healthy tissue. For that, the variants need to be comparable, which is sometimes not the case for variants coming from different sources. The problem is that there can be multiple equivalent alignments that represent the same insertion or deletion event.

Let us look at how a variant is represented in a VCF (variant call format) file[7]. The same representation is used within our database. A variant is described by four values: The chromosome ("CHROM" as per the VCF specification); the position in 1-based coordinates relative to the beginning of the chromosome (POS); the original sequence on the reference (REF); and the alternative sequence that replaces it (ALT). Each variant is stored in a single line and additional, optional fields on the same line allow to describe quality values, allele counts, coverage, inferred genotype (homozygous or heterozygous) etc. that are associated with the sample-specific call.

Example 6. A single-nucleotide variant on chromosome 7, position 5000, where the reference has a `C`, while a `T` was observed instead, is described by $(7, 5000, \texttt{C}, \texttt{T})$. Below, we write this as 7:5000 `C`$\rightarrow$`T`.

For insertions and deletions, the position must refer to one base *before* the actual event, which implies that the first characters in REF and ALT are equal. The event is an insertion if ALT is longer than REF and a deletion if ALT is shorter than REF.

Example 7. The deletion of a single `T` on chromosome 2, position 4444, assuming that the previous base on the reference is an `A`, is described by 2:4443 `AT` $\rightarrow$ `A`.

Unfortunately, a single event can be described by multiple tuples. The chosen representation depends on the way in which the alignment was found. For example, the insertion of the sequence `AC` at position 100 on chromosome X can be described by X:100 `C`$\rightarrow$`CAC`. A different tool may find that the insertion is X:99 `G`$\rightarrow$`GCA`. This is the alignment that is described by the first variant:

[7]Specification:
http://www.1000genomes.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-41

```
GC--T
GCACT
```

And this is the alignment described by the second variant:

```
G--CT
GCACT
```

Both are optimal alignments between GCT and GCACT under linear and affine gap scores. There is no reason to prefer one over the other, and which one is chosen depends only on the way in which ties are resolved in the alignment algorithm.

Some mappers (including BWA) have standardized on computing indels that are shifted to the left as far as possible. This can be achieved at the time of backtracing by choosing a mismatch over an insertion or deletion if there is a tie between the scores.

In order to normalize indels that were not computed with BWA (such as some of those found in dbSNP), we can use the following simple algorithm (instead of re-computing the alignment).

We first observe that an insertion described by $r_1 \to a_1, \ldots, a_n$ with $r_1 = a_1$ and $n > 1$ can be moved one position to the left without changing the alignment score or cost if $r_1 = a_n$. A deletion described by $r_1, \ldots, r_m \to a_1$ with $r_1 = a_1$ and $m > 1$ can be moved one position to the left if $r_m = a_1$. Together, an insertion or deletion described by $r_1, \ldots, r_m \to a_1, \ldots, a_n$ with $r_1 = a_1$ and either $n > 1, m = 1$ or $n = 1, m > 1$ can be moved one position to the left without changing the alignment score if $r_m = a_n$.

The resulting algorithm needs, in addition to the variant itself, also the nucleotide sequence of the chromosome that it refers to.

Algorithm 12 (Normalize-Indel).
Input: POS, REF, ALT and the base sequence of CHROM. The lengths of REF and ALT must be different, and both must start with the same character.
Output: Normalized POS, REF, ALT.

```
1. While the last character of REF and the last character of ALT are equal:
    a) Delete the last character of both REF and ALT.
    b) Prepend to both REF and ALT the character found in the reference before POS.
    c) Decrease POS by one.
2. Return modified POS, REF and ALT.
```

Each shifting of the indel by one position can be done in constant time if appropriate data structures are used (linked lists) for REF and ALT.

Example 8. With REF=C and ALT=CAC and assuming that the preceding characters are GA, the algorithm finds a leftmost representation of the insertion in three steps. The characters that are compared are underlined:

	Step 1	Step 2	Step 3
REF → ALT:	C → CAC	A → ACA	G → GAC
Alignment:	$\texttt{GA\underline{C}--T}$	$\texttt{G\underline{A}--CT}$	$\texttt{\underline{G}--ACT}$
	$\texttt{GACA\underline{C}T}$	$\texttt{GAC\underline{A}CT}$	$\texttt{GA\underline{C}ACT}$

4.3 Database

The database is the second main part of our exome sequencing analysis tool. It stores both patient metadata and the variant data. Using an SQL database for metadata is the natural choice as the different entities such as patients and samples and their relationships are easily represented as tables, which are the basic components of SQL databases. We also store the variants themselves in the database. This is unusual since other workflows use VCF files directly. This is exemplified by the existence of dedicated tools for the purpose of indexing, intersecting, merging, and filtering VCF files. However, all those operations are typical tasks of a database system and database engines are already highly optimized to perform those tasks. By putting the variant data into the database, it is possible to leverage those optimizations. In addition, all processing remains in a single domain, which simplifies program structures.

The most important advantage, however, is that it enables us to provide dynamic, interactive querying of variant data for the user. For example, when viewing a particular set of variants, the user can interactively try out different filtering parameters, such as whether to discard variants found in dbSNP or in other samples. On each change, a new query is issued to the database server and the result is available within a few seconds. This would be much harder to achieve with a workflow purely based on VCF files.

In the current implementation, we rely on the PostgreSQL database[8]. Since a database abstraction layer is used, it is possible to easily switch to a different engine.

In the following sections, we give an overview of the entities that are stored in the database. We strive for the database tables to be in third normal form (see Codd, 1971).

4.3.1 Patients, samples and other metadata

In addition to storing the variants, we keep track of where they come from. Inspired partially by the SAM/BAM specification (Li et al., 2009), we model the following basic entities: patient, sample, library and sequencing unit. Further metadata tables exist for sequencing runs, capture kits and diseases. A simplified entity-relationship diagram (without attributes) is shown in Figure 4.6.

We now describe the tables, including their attributes, in more detail. All tables except “Call” and “Annotation” have a primary integer key named “id”, even when this is not explicitly mentioned below. The id is distinct from the “accession” attribute, which is also present in some tables. The accession is a user-visible, permanent identifier for an object, while the id is used only internally.

[8]http://www.postgresql.org/

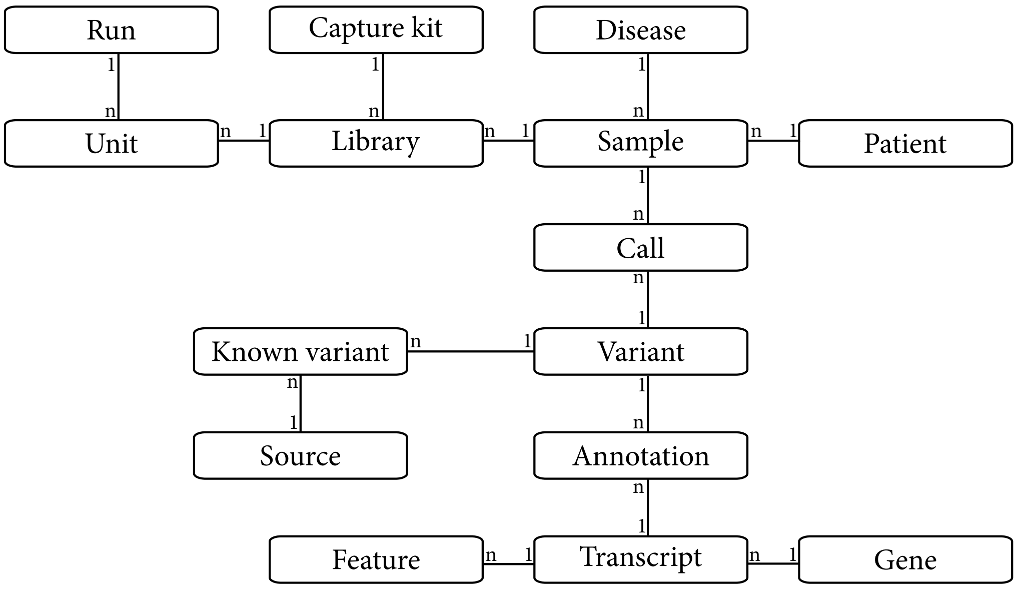


Figure 4.6 This simplified entity-relationship diagram shows the main tables in the database. One-to-many relationships are shown with a "1" and "n" at the ends of their connection. Note that the "Annotation" and "Call" table create many-to-many ($m : n$) relationships between Variant/Transcript and Sample/Variant, respectively.

Patient

The attributes of each patient are: accession, gender, id of mother, id of father. The accession consists of a unique number that identifies the patient, prefixed with "P". To maintain privacy, no names, birthdays etc. are kept in the database, except for the gender, which can be used as a plausibility check (see Section 4.4.5). The association between names and accession numbers is supposed to only be available to the medical researchers, who maintain a separate spreadsheet on their own secured systems with that information. The mother and father id attributes are self-referencing foreign keys that model family relationships between patients. If the intention is to model a sibling relationship between two patients, then records for their parents must be created even if those parents' entries would otherwise be unnecessary.

Disease

A disease simply has an id and a name. To model healthy samples, this table always contains the entry "healthy".

Sample

A sample is a piece of tissue (blood, tumor) taken from a single patient. Its attributes are: accession, patient id, disease id, tissue. The patient id references the patient that is the source of the sample and the disease id references the disease that is associated with this sample. The tissue is either "blood" or "tumor". Note that we consider samples instead of patients to have diseases:

This makes it possible to classify a blood sample as "healthy" and a different sample from the same patient as being associated with a specific tumor.

The sample accession numbers play a central role within Exomate as they are the identifiers that are used most often by the medical researchers. They are often also used to identify patients. This works because a sample is associated with exactly one patient. Multiple samples from a single patient are possible.

Capture kit

An entry in the capture kit table has an id, a short name, such as "SeqCap EZ v3", and also the path to an annotation track in the file system that specifies the capture targets of the capture kit. These files are usually in GFF (General Feature Format) and available from the manufacturer's web site.

Library

The attributes of a library are: sample id, capture kit id. A library is a sample that has been prepared for sequencing. Having separate tables for library and sample allows us to describe the case that one sample has been captured by multiple kits, where, for example, the second one is a more recent kit that captures a larger part of the exome. If we were not concerned with exome sequencing only, further attributes could indicate perhaps whether whole genome sequencing or bisulfite sequencing has been performed.

Run

A run groups all sequenced datasets that were obtained during a single run of the sequencing instrument. Its attributes are: date, machine name (assumed to be globally unique) and a flowcell identifier string. The flowcell identifier is assigned by the manufacturer and also unique.

Unit

A "sequencing unit" or simply "unit" is a sequenced library. Modeling this is necessary since a library can be sequenced more than once with different parameters. There is a bijection between units and the FASTQ files with raw sequencing data that are output by the sequencer. Its attributes are: FASTQ file name, library id, lane, barcode, run id. The lane is the lane number of the Illumina sequencer and the barcode is the barcode sequence used within that lane when multiplexing was used.

Statistics

Unit-specific statistics are stored in a statistics table with data such as read counts, fraction of mapped reads, fraction of PCR duplicates, etc.

Importing metadata

New patients, samples, etc. can be added and edited through an administrative interface on the web page. For bulk imports of an entire run, a spreadsheet table template exists. It needs to be filled out manually (one sample per row), but can then be imported automatically by a command-line import script.

4.3.2 Known and discovered variants

There are two types of variants: Those that were discovered (called) in our samples and those that are known from external sources. Called variants, or "calls", come with an estimate of the genotype (homozygous or heterozygous) and with quality values both for the variant call itself and the genotype. External or "known" variants come from dbSNP, but also from data released by the 1000 Genomes project, from the HapMap Project (International HapMap 3 Consortium et al., 2010), and from the Exome Variant Server (EVS)[9]. These variants are known in the sense that we assume that most of them are not associated with a disease and that they are therefore not interesting candidates for disease-causing mutations.

Since variants often occur in more than one sample and additionally in datasets of known variants, we normalize the tables by splitting variants up into abstract variants, calls, and known calls, as described below. Annotations are stored in a separate table.

Abstract variants

We use an idea by Tim Strom (Helmholtz Centre Munich, Institute of Human Genetics, personal communication) and store variants in their own table without associating them with a specific sample. Using the same convention as in the VCF input files, an entry in the variants table therefore consists of only the chromosome name, position on the chromosome, the reference sequence and the altered sequence. Each variant also has a unique integer id that serves as the primary key.

One of the main low-level tasks of the pipeline is to perform set operations on lists of variants. This is realized with SQL JOIN operations. By using the numeric variant identifiers, this can be done efficiently since only integers need to be compared in the database engine instead of full (chromosome, position, reference, alternative) tuples.

Calls

An entry in the calls table represents a single called variant. It references the id of an abstract variant and a sample id, and also contains further information, such as the assigned quality value of the call, quality value of the genotype, strand bias, total read depth, number of reads supporting the reference allele and number of reads supporting the alternative allele.

Sources

This table simply lists all sources for known variants, such as data from the 1000 Genomes project, different dbSNP versions, etc.

Known variants

The third table stores those variants that are known from external sources. Each entry references a variant and a source by id. If an entry refers to a dbSNP entry, it also contains the dbSNP

[9]http://evs.gs.washington.edu/EVS/

Table 4.3 A snapshot of the Exomate database in February 2013. For dbSNP variants, only "exonic" variants are counted, that is, those that are within 1000 bp of an exon as annotated by Ensembl. The dbSNP 137 re-discovery rate is the number of calls that also occur in dbSNP divided by the total number of calls (see Section 4.4.5).

Patients	128
Samples	154
Units	172
Calls	41 390 180
Unique called variants	2 612 406
dbSNP 137 exonic variants	9 582 261
dbSNP 137 re-discovery rate	94.4%
Unique variants overall	10 763 318

rsid, the allele frequency and flags that indicate whether this variant is clinical: Such variants are possibly associated with a disease and if they are found in a sample, they should therefore not be discarded.

Global statistics of the number of different types of variants within the database are shown in Table 4.3.

4.3.3 Annotations

A variant annotation tells us what effect the variant has on a gene transcript within which the variant is located. Since multiple transcripts may overlap the variant's location, there can be multiple annotations for a single variant. An annotation is therefore not stored along with the variant itself, but in a separate *annotations* table. A single annotation associates a variant id and a transcript id with the following information (substitution variants only): Reference codon (string of length three), alternate codon, reference amino acid, alternate amino acid. To indicate a stop codon, a special symbol is used as "amino acid". Further, the variant's type is classified; it can be an insertion or deletion, or a synonymous, nonsense, missense or read-through substitution. Additionally, the variant is classified by its position relative to the transcript into the regions upstream, UTR, coding, intron and downstream. Also, the distance to the next splice site (defined as an exon-intron border), is recorded. If the distance is negative, the variant is located on an intron, and within an exon otherwise. A value of zero does not occur.

To simplify some queries, the annotations table is not in third normal form. For example, reference amino acid and alternate amino acid are each a function of the respective codon and would usually have to be stored in a separate table. Also, whether a variant is insertion, deletion or substitution is a function of the lengths of the variant's reference and alternate sequences. We consider each annotation and variant to be immutable, however, and can therefore not suffer from update anomalies that can occur in non-normalized tables.

On average, we have 3.1 annotations per variant and 1.1% of variants have annotations with differing amino acid changes.

Ensembl

Variants are annotated with the help of the public Ensembl gene annotation track, available for download as a GTF file. A custom script first imports the data into the database into three tables for genes, transcripts and features. A feature is an interval on the reference genome (start and stop coordinate) that has a type, which can be *exon*, *coding sequence*, *start codon* or *stop codon*. Each feature references a transcript, and a transcript is defined by all features that reference it. Each transcript, on the other hand, references a gene. Each gene usually has multiple transcripts that reference it due to alternative splicing.

Using the information available in the features and transcripts tables, variants are annotated by a script[10] that considers feature start and stop positions and variant positions as "events" and iterates over merged events in ascending order. If a position belonging to a variant is seen, the variant is annotated with the currently active features.

4.3.4 SIFT

SIFT is a tool by Ng and Henikoff (2003, 2001) that assigns scores to substitution variants which help to estimate whether a mutation is detrimental or not. A database of pre-computed scores can be downloaded[11] and imported into our own database. (SIFT predictions are currently not shown in the web frontend.)

4.4 Web frontend

The most visible part of Exomate is the interactive web frontend. Its main task is to create database queries derived from user input and to display the results. The emphasis is on creating the queries: These encode the central logic necessary to reduce the lists of variants to those that are likely interesting in the context of the study being done.

In a simplified view, we could consider everything done up to this point as pre-processing: mapping, variant calling, annotation, gathering statistics. The actual work of selecting the appropriate data is left to the frontend. Other workflows rely on software such as vcftools[12] that annotate, intersect, subtract etc. VCF files. Within Exomate, these tasks are encoded as SQL database queries and computed by the database server. As should be clear now, this allows a highly interactive analysis where queries with different parameters can be run quickly by a non-bioinformatician.

4.4.1 Implementation

The web frontend is written in Python with the Flask[13] web-framework. Flask deals with such low-level tasks as interpreting request URLs, routing them to the correct function, and parsing parameters. To interface with the database, we rely on the object-relational mapping (ORM) library SQLalchemy[14]. ORM makes it possible to view each row of a database table as an ob-

[10] written by Christopher Schröder and Manuel Allhoff, former student assistants in our group
[11] ftp://ftp.jcvi.org/pub/data/sift/Human_db_37_ensembl_63/
[12] http://vcftools.sourceforge.net/
[13] http://flask.pocoo.org/
[14] http://www.sqlalchemy.org/

ject, where the table columns are represented as attributes of the object. More importantly, SQLalchemy allows us to construct queries incrementally and dynamically. Instead of going into detail here, we refer to the SQLalchemy documentation and give a short example that should help to understand the basic idea.

Example 9. We show the code for querying either *all* samples or only those that have a specific disease.

Assuming that `db` is an open database session and that `disease` references a *Disease* object, the code would look similar to this in Python with SQLalchemy:

```
1 q = db.query(Sample)
2 if disease is not None:
3     q = q.filter(Sample.disease == disease)
4 samples = q.all()
```

Line 1 constructs a query for *all* samples; line 3 adds a filter, but only if a disease was specified. The actual query is then sent to the database in the last line. In raw SQL, this corresponds to choosing between either

```
SELECT * FROM samples;
```

or

```
SELECT * FROM samples WHERE disease_id = ...;
```

Note that the correct disease id is filled out by SQLalchemy automatically. In the same way, more complex queries can be constructed. See the Exomate source code for some of them.

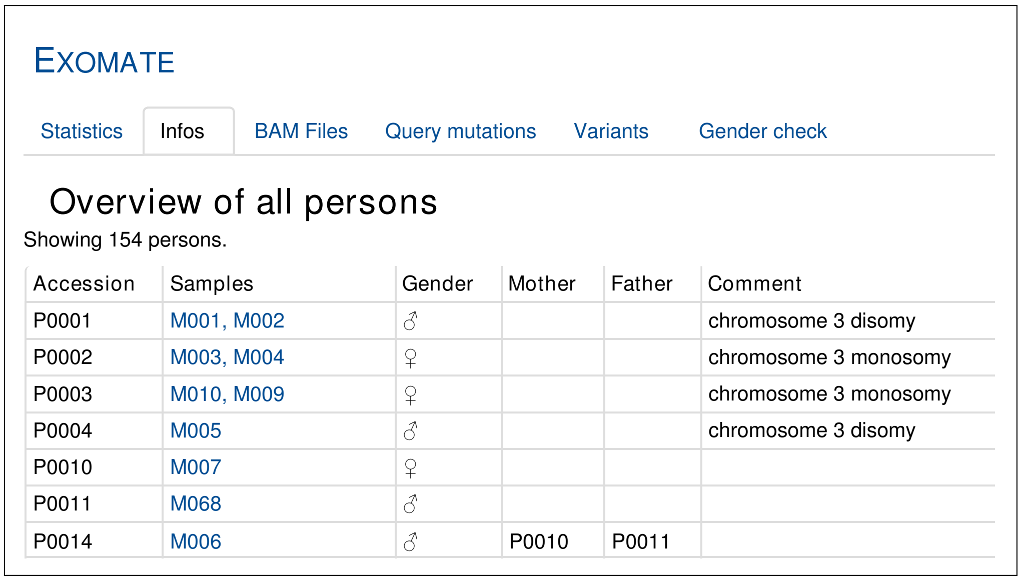
4.4.2 Simple queries

Some of the pages generated by the web frontend simply list the content of a particular table. An example of an overview of all persons is shown in Figure 4.7. Similar pages exist for samples, units and diseases. Libraries are never shown to the user. Instead, the capture kit information is shown along with the corresponding units. There is also a page that shows all the quality control statistics that are described in Section 4.2.3.

Another page allows the user to search the entire database for any variant given by its coordinates (chromosome and position), or for all variants in a region. Both calls and known variants are displayed. See Figure 4.10 on page 111 for an example, explained in Section 4.5.

4.4.3 Criteria for filtering mutations

Our central task can be described as follows: Find all variants in a certain sample, but only those that are probably interesting. The list of variants is simply the list of calls found in the corresponding VCF file. More difficult is to answer what "interesting" means. We describe here the filtering criteria that can be relevant, many of which are optional and can be switched on and off dynamically. Some of these have already been used by Ng et al. (2009).



Overview of all persons

Showing 154 persons.

Accession	Samples	Gender	Mother	Father	Comment
P0001	M001, M002	♂			chromosome 3 disomy
P0002	M003, M004	♀			chromosome 3 monosomy
P0003	M010, M009	♀			chromosome 3 monosomy
P0004	M005	♂			chromosome 3 disomy
P0010	M007	♀			
P0011	M068	♂			
P0014	M006	♂	P0010	P0011	

Figure 4.7 Extract of patient/person overview page. P0001 through P0004 are uveal melanoma patients. Chromosome 3 status within the tumor was added as a patient-specific comment (see Section 4.5.3). A trio of a patient and his parents is also shown. Sample names have been anonymized.

Variant region

Typically, we are interested only in variants that are located within a coding region. For a complete picture, one may also include intronic variants or those located in the 5′ or 3′ untranslated region (UTR). Theoretically, variants located in promoter regions are also of interest, but these regions are not captured.

Splice sites

Splice-site mutations can have a large functional impact as changed splicing leads to radically different protein products. In order for an intron to be removed by the splicing machinery, consensus sequences in three regions must be recognized (Alberts et al., 2008, Chap. 6, Fig. 6-28). One of the regions (the *branch site*) is typically located too far within the intron to be consistently seen by exome sequencing and therefore ignored here. The two other regions are the exon/intron boundaries. The consensus sequences extend a few bases into the exon and up to twelve bases into the intron at the 3′ splice site (Zhang, 1998). Splicing is quite tolerant of changes within those sequences, excluding the four bases directly adjacent to the two exon/intron boundaries. In our software, we define a splice-site mutation to be any that has a distance of at most n bases to the closest exon/intron border, where n is per default set to twelve and can be changed dynamically for each query. Splice-site mutations, including those on introns, are never discarded.

Variant types

Synonymous variants do not change the amino acid sequence of the resulting protein and are therefore discarded from the result list by default. An exception are synonymous variants that occur close to a splice site. These are not discarded since the nucleotide sequence is relevant for splicing, not the amino acid sequence.

Known in dbSNP

Variants that occur in our samples of interest but that are also in dbSNP are, by default, discarded. The assumption is that variants in dbSNP either come from healthy individuals or, if not, that the associated disease is at least sufficiently different from the disease that is being investigated. This assumption is not always valid. Some entries in dbSNP are marked as "clinically significant", and these are therefore not discarded. When searching for a recessive trait, the responsible variant can also occur in dbSNP without being specially marked since the affected individual is heterozygous for that variant and therefore does not show the phenotype. Currently, the alternative of not discarding dbSNP variants is infeasible as too many remain.

Found in other samples

A second source of variants that can be ignored are the variants found in healthy samples. Studies usually include not only samples of an affected patient or affected tissue, but also controls. For solid tumors, the control is typically blood from the same patient, and for patients with rare Mendelian diseases, the parents are often sequenced (resulting in *trios*). Discarding these variants should, in the best case, restrict the set to de-novo/somatic mutations only. In our own experiments, this does not work as many variants of the controls are simply not found, and usually hundreds of variants remain unless variants found in other, unrelated samples and dbSNP are discarded. According to Ng et al. (2009, Suppl. Fig. 2), the need to rely on dbSNP decreases as more and more controls are available.

Using own controls has the further advantage that some variants arising from sequencing errors are ignored automatically. Such errors are often sequence-specific (Nakamura et al., 2011; Allhoff et al., 2013), and will therefore affect both the sample of interest and the control.

Quality thresholds

There are two independently adjustable thresholds on variant call qualities. The first limits the shown calls to those above the threshold (*affected call quality threshold*). This must be set to some reasonable, large enough value in order not to show too many artifacts. The second threshold is applied to the calls from control exomes used for filtering (*unaffected call quality threshold*). Since we want to maximize the effectiveness of the filter, this threshold is set to a low value and therefore includes – intentionally – many sequencing artifacts.

4.4.4 Mutation query implementation

The following description of a variant query only approximates what is really done by the program. The actual code uses SQLalchemy to construct a single SQL query incrementally. That is, when we describe below that the "set of variants is reduced", this actually means that the query is extended by appropriate criteria. The idea behind this is to let the SQL database engine optimize the final query in the best way. For example, instead of retrieving two sets of variants separately from the database and subtracting one from the other on the client side, it is much more efficient to use a proper SQL JOIN clause and let the server return the end result, especially if millions of variants are involved, as is the case here.

Algorithm 13. The following algorithm summarizes the procedure to find “interesting” mutations, defined by user-supplied parameters.
Input: A database connection and query parameters.
Output: A list of (Call, Annotation) tuples.

1. Get a list of all unaffected samples. These contain the variants that are to be discarded. The list is provided by the user, who can explicitly specify samples by accession or indirectly through a list of diseases. This is usually left at its default of discarding variants found in “healthy” samples.

2. Obtain all calls from unaffected samples, excluding those below the unaffected quality threshold.

3. Get a list of all affected samples, that is, those that interest us.

4. Query the database for all calls from affected samples whose quality is better than the affected call quality threshold, and subtract (using a LEFT OUTER JOIN) those that represent the same variant (identical variant id) as one of the unaffected calls obtained above.

5. Get the list of “Known sources” specified by the user for filtering. If that list is nonempty, further subtract from the set of calls obtained so far those calls that represent the same variant as one of the known variants from the given sources. Optionally, calls that are marked as “clinical” are not subtracted.

6. Associate (join) each call to its corresponding annotation, applying further filter criteria: Variants must be on the coding region and be not synonymous unless they are close to a splice site. If specified, intron and UTR variants are also retained.

7. Return the resulting list of (Call, Annotation) tuples.

4.4.5 Quality control queries

In addition to directly displaying statistics, further quality control is performed by analyzing the calls that belong to a single sample. The advantage here is again that the queries can be changed dynamically, for example by setting different variant quality thresholds.

dbSNP re-discovery rate

The dbSNP re-discovery rate, for example described by Challis et al. (2012), is the fraction of those variants discovered in an exome that were also found in dbSNP. The rate gives a rough estimate of the variant calling pipeline’s specificity. The value is computed on the fly when the sample detail page is retrieved. On average, the dbSNP re-discovery rate of our data is 94.4% (see Table 4.3) when only variants with a minimum quality of 200 are considered.

Table 4.4 Captured bases on chromosomes X and Y for three capture kits. Capture targets are taken from the manufacturer's annotation.

	Captured bases		
Capture kit	Chromosome Y	Chromosome X	Ratio Y/X
NimbleGen 2.1M Array	62 443	1 482 139	4.2%
SeqCap EZ v2	102 743	1 661 562	6.2%
SeqCap EZ v3	232 073	2 404 880	9.7%

Checking patient gender

As a first measure against incorrect patient metadata, we use the number of reads mapping to sex chromosomes to estimate each sample's gender. For each sample, we compute the ratio y/x, where y is the number of reads mapping to the Y chromosome and x is the number of reads mapping to the X chromosome.

In females (two X chromosomes), one might expect that this ratio is zero since no reads should map to the Y chromosome. However, since there are regions on X and Y that contain the same genes and have similar sequence (pseudoautosomal regions), some reads will map to the Y reference even in females. Therefore, the ratio should be low, but not zero.

In males (both X and Y chromosome), we expect the y/x ratio to be (roughly) the same as the ratio between the number of bases captured on Y vs. those captured on X. This ratio depends on the capture kits used, but is at least 4.2%; see Table 4.4 for details.

Our largest observed y/x ratio within a sample known to be female is 1.4%, while the lowest y/x ratio within a known male sample is 4.7%. See Figure 4.8. We therefore use a simple linear classifier with a threshold of 3%: Ratios below are classified as female, above as male. Within Exomate, this result is presented in the form of a table that lists all samples, ordered by their y/x ratio. If annotated and predicted gender do not match, the sample is highlighted. This check has so far resulted in the correction of three annotations. The check is crude, but works accurately. The only incorrect classifications are due to two male tumor samples classified as female, but it is likely that here either partial or full Y chromosome loss has occurred.

Chromosome number anomalies

Other chromosomes can also be lost (partially or fully) in tumors, such as chromosome 3 in uveal melanoma (see Section 4.5.3) or chromosome 13 in retinoblastoma. Such large-scale deletions are easier to detect with other methods, but are also detectable in exome sequencing data. We have implemented two methods.

First, for a single dataset, we compute the ratio $\frac{n_c/n}{b_c/b}$, where n_c is the number of reads that map to a chromosome c, n is the total number of mapped reads in the dataset, b_c is the number of bases captured in chromosome c, and b is the total number of captured bases. Deviations from a value of 1 point to a chromosome number anomaly.

The second idea is based on the observation that missing chromosomes or large deletions lead to regions without heterozygous calls. This is called *loss of heterozygosity* and has been used successfully to detect deletions (Bignell et al., 2004). We use this technique here only as a quality check in order to verify that our known information about chromosomal loss match the data.

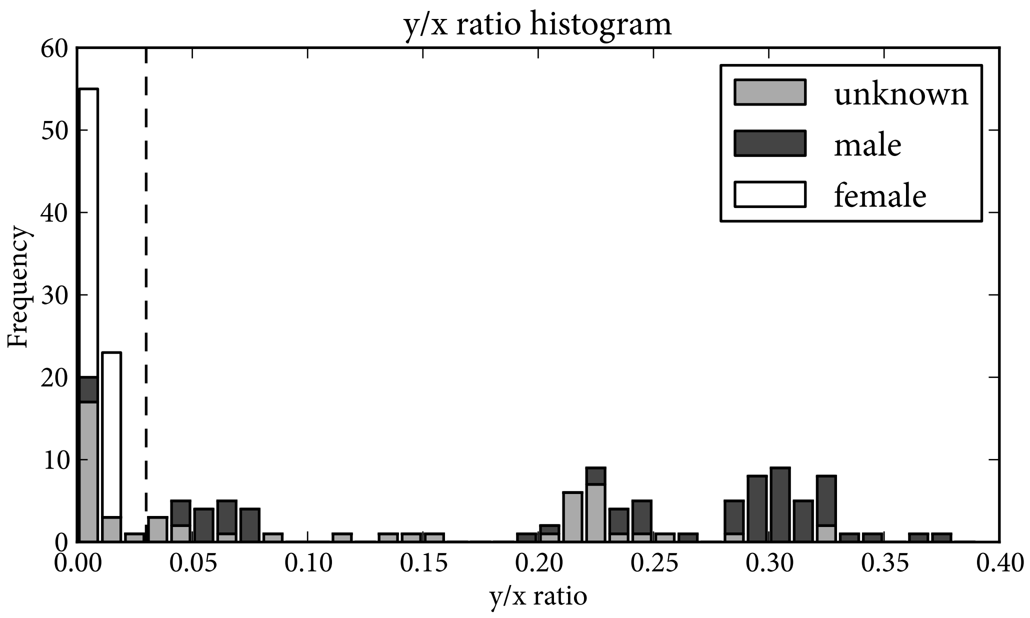


Figure 4.8 Histogram of ratio of reads mapped to the Y- vs. the X chromosome. The dashed line marks the threshold used for gender classification. The male samples classified as female are tumor samples (see text).

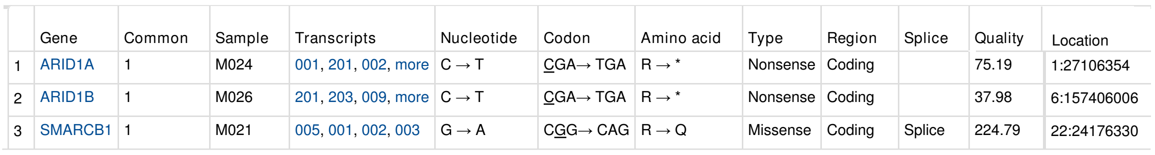
For that, each sample overview page in Exomate shows a table that lists all chromosomes and the ratio n_{het}/n, where n_{het} is the number of heterozygous calls on a chromosome, and n is the total number of calls on that chromosome. A chromosome where that ratio is low compared to the others is a good candidate for having been lost.

Checking relatedness

A further check (currently not implemented) could test whether two samples really are related in the way the metadata indicates. For example, for parent/child relationships, most variants in the child should be explained by matching variants in one of the parents. In tumor/normal sample pairs from the same patient, the variants in the tumor should be mostly a subset of variants in the normal cells. If non-matching samples are detected, it may be possible to find the sample most closely related to a given one.

4.5 Results

Exomate has been and is being used in multiple projects. We present three of the projects, two of which have resulted in manuscripts accepted for publication. All show that Exomate is a valuable tool that helps in identifying disease-causing genes and discovering mutations in exome sequencing data.



	Gene	Common	Sample	Transcripts	Nucleotide	Codon	Amino acid	Type	Region	Splice	Quality	Location
1	ARID1A	1	M024	001, 201, 002, more	C → T	CGA→ TGA	R → *	Nonsense	Coding		75.19	1:27106354
2	ARID1B	1	M026	201, 203, 009, more	C → T	CGA→ TGA	R → *	Nonsense	Coding		37.98	6:157406006
3	SMARCB1	1	M021	005, 001, 002, 003	G → A	CGG→ CAG	R → Q	Missense	Coding	Splice	224.79	22:24176330

Figure 4.9 The result table returned by Exomate when querying for mutations on genes encoding SWI/SNF subunits within Coffin-Siris patients. Some columns were removed for better readability.

4.5.1 Coffin-Siris syndrome

Coffin-Siris syndrome is a genetic disease associated with mental retardation and body anomalies (Santen et al., 2012). Seven affected patients were sequenced by colleagues at the University Hospital Essen. This was one of the first exome datasets available to us, and it was used to calibrate the pipeline and refine the methods. We assumed that Coffin-Siris was a monogenic disease (caused by mutations in a single gene). Candidate genes were therefore those that were simultaneously mutated in a large subset of the seven patients. Unfortunately, no single gene could be identified. Later, a study by Santen et al. (2012) was published, in which the cause for Coffin-Siris syndrome was pinpointed to be mutations in the gene *ARID1B*. At the same time, Tsurusaki et al. (2012) come to the conclusion that mutations in an entire set of genes (including *ARID1B*) that encode subunits of the "SWI/SNF" protein complex are common to patients with Coffin-Siris syndrome, in contrast to our assumption of a single gene. We verified with Exomate that one of our patients had a mutation in *ARID1B* and that two other patients had mutations on *ARID1A* and *SMARCB1*, respectively (see Figure 4.9), which also belong to the SWI/SNF complex.

4.5.2 Nager syndrome

Nager syndrome belongs to a type of diseases called acrofacial dysostoses. We sequenced exomes of two patients with Nager syndrome and identified mutations in gene *SF3B4*, using Exomate. With Sanger sequencing of ten further patients, we found pathogenic mutations of *SF3B4* in seven out of twelve patients, confirming that *SF3B4* is a cause for Nager syndrome (Czeschik et al., 2013).

Before we finished our study, Bernier et al. (2012) independently published their own results, also identifying mutations in *SF3B4* as a cause of Nager syndrome. Our results agree in that both studies do not find pathogenic mutations in *SF3B4* in one third of Nager patients. It is therefore likely that Nager syndrome is a condition with genetic heterogeneity, in which a mutation in *SF3B4* is only one of several possible causes, but the other causal genes are currently unknown.

4.5.3 Uveal melanoma

In our largest study (Martin et al., 2013), we investigate uveal melanoma, which is a malignant eye tumor. Tumor and blood samples of 22 patients were exome-sequenced. Uveal melanoma can be divided into two major classes (Tschentscher et al., 2003). In one class, tumor cells have a normal chromosome 3 count (disomy 3), and in the other, cells are characterized by loss of an entire chromosome 3 (monosomy 3). Metastasis disease and poor survival are strongly associated with monosomy 3 (Prescher et al., 1996). Exome sequencing identified recurrent mutations in multiple genes, among them *GNAQ*, *GNA11* and *BAP1*. These genes are known to be frequently mutated in uveal melanoma.

The study shows the potential of exome sequencing for identifying new candidate genes. One of the candidates was *SF3B1*, which had previously been described in connection with other tumors, and another was *EIF1AX*, which had previously not been described as cancer gene. Mutations in these genes were all verified by Sanger sequencing. To evaluate the mutation frequency and pattern of these genes in uveal melanoma, the relevant exons of *SF3B1* and *EIF1AX* were Sanger-sequenced in additional 66 tumors. The main result of the study is that mutations of these two genes are mutually exclusive and largely restricted to the class of tumors with disomy 3.

After submission of our initial manuscript, a study by Harbour et al. (2013) was published, in which the authors also observe recurrent mutations of *SF3B1* in uveal melanoma. Since their exome sequencing data are publicly available from the Sequence Read Archive (SRA) under accession SRA062369, we downloaded the data and ran our pipeline on it.

All three mutations in *SF3B1* that were found by Harbour et al. change position 198 267 484 on chromosome 2 from `G` to `A`. With Exomate, we can confirm the existence of those mutations in their data. The result of the crucial query is shown in Figure 4.10. In the figure, we see the three mutations described by Harbour et al. and an additional one affecting sample SRS378747, which is supposed to be the healthy control sample from one of the patients according to SRA. It is not clear where this discrepancy comes from. The figure also shows two of the *SF3B1* mutations that we observe in our own exome data.

4.6 Future work

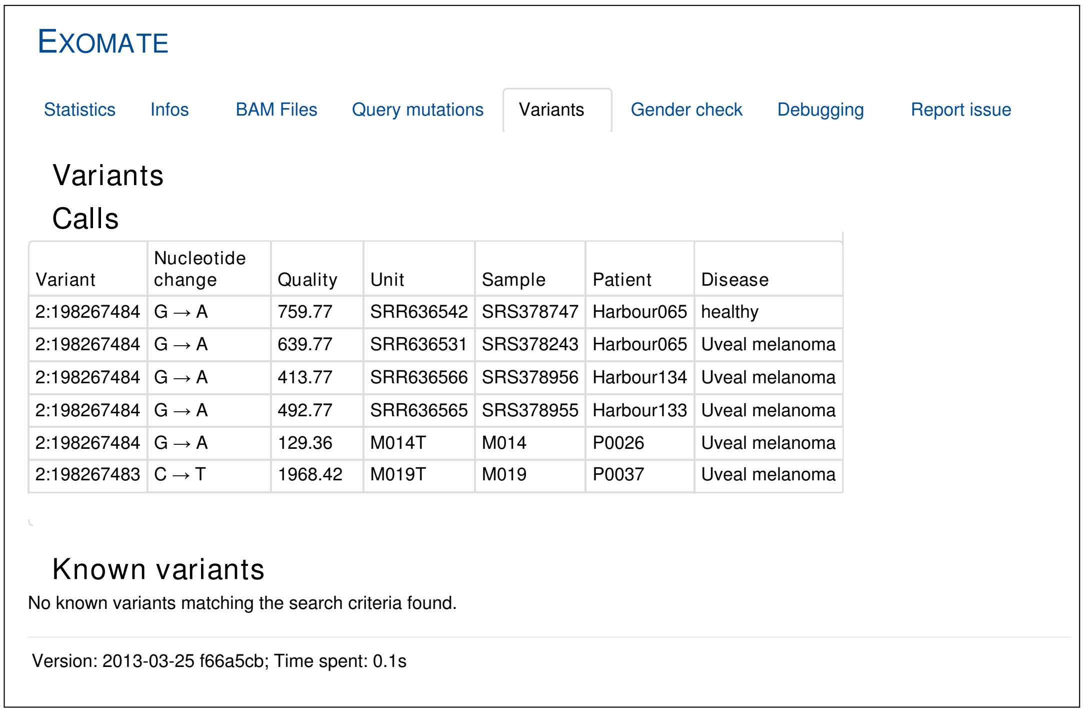
The software described in this chapter is in daily use at the University Hospital of Essen within the Human Genetics research group. Its development is still ongoing at a rapid pace. We describe here some possibilities for future work. First, we list planned extensions for which it is already quite clear how to proceed in adding them. In Section 4.6.1, we describe those issues for which more research is needed.

Tumor/normal variant calling

When samples of tumor and normal tissue from the same patient are available, both datasets can be analyzed at the same time. The idea is that a variant that would usually be missed in one of the samples since evidence for it is too weak can still be found if information from both samples and the knowledge that the samples are closely related is taken into account. SomaticSniper (Larson et al., 2012), for example, is able to find SNPs in tumor/normal pairs. With such a tool, it should be possible to eliminate some of the false positive candidates.

Do not discard rare variants

For those variants that are discarded, we would like to ensure that they are most likely not relevant for the disease being studied. We initially discarded all variants that also occur in dbSNP. A first improvement was to keep those variants marked as “clinical” (Section 4.4.3). A problem is that absence of that marker does not imply that the variant is not clinical. Only 60 325 of



Variants

Calls

Variant	Nucleotide change	Quality	Unit	Sample	Patient	Disease
2:198267484	G → A	759.77	SRR636542	SRS378747	Harbour065	healthy
2:198267484	G → A	639.77	SRR636531	SRS378243	Harbour065	Uveal melanoma
2:198267484	G → A	413.77	SRR636566	SRS378956	Harbour134	Uveal melanoma
2:198267484	G → A	492.77	SRR636565	SRS378955	Harbour133	Uveal melanoma
2:198267484	G → A	129.36	M014T	M014	P0026	Uveal melanoma
2:198267483	C → T	1968.42	M019T	M019	P0037	Uveal melanoma

Known variants

No known variants matching the search criteria found.

Version: 2013-03-25 f66a5cb; Time spent: 0.1s

Figure 4.10 Reproducing the results by Harbour et al. The page shows results for a search for variants within 10 bp of pos. 198 267 484 on chromosome 2. On the bottom, we see that no known variants (in dbSNP) were found. See text for explanation of the results in the top table.

8 944 641 (0.7%) exonic variants in dbSNP 135 have that marker. As dbSNP continues to grow, it makes therefore sense to only discard those variants that occur in a sufficiently large subset of the population. About 6.7 of 9.6 million exonic variants in dbSNP 137 are annotated with an allele frequency. Setting a minimum allele frequency of 1% leaves 2.1 million variants that can be filtered. A similar threshold could be used for discarding variants that occur in our own individuals, where one could, for example, only discard those that occur in at least two unrelated healthy persons.

Integrate more external data sources

At the moment, we still see too many spurious variants in Exomate's output. One way to improve specificity is to add other external sources of known variants, which would be added to our own variants from healthy samples and dbSNP variants. One such source is the data released by the 1000 Genomes Project (The 1000 Genomes Project Consortium et al., 2010), but that data has already been integrated into dbSNP. A promising candidate for inclusion is data from the NHLBI Exome Sequencing Project (ESP), available for download via the Exome Variant Server (EVS)[15]. It includes variants from over 6500 samples obtained by exome sequencing. Variants have mostly not been verified with Sanger sequencing, which makes the data less reliable than dbSNP, but this is not a problem since it is only meant to be used for filtering. Since there are so many variants in EVS, the previous point about not discarding rare variants becomes even more important and should be addressed first.

Include structural variations

As many other tools, we currently ignore structural variations, which are large-scale insertions, deletions, inversions etc. (see the review by Alkan et al., 2011). Long indels can be found with a tool such as CLEVER (Marschall et al., 2012) or Pindel (Ye et al., 2009), but it is unclear whether this works well for exome sequencing data with its uneven distribution of reads over the genome.

Pseudoautosomal regions

Due to the pseudoautosomal regions (PAR) on the X and Y chromosomes, some reads are seen mapping to the Y chromosome in females. The errors introduced are small, but if the gender of the patient is known, one could map samples from females to a reference that does not contain a Y chromosome.

Ignore long transcripts

When searching for genes that are commonly mutated in a set of samples, long transcripts often appear as candidates, simply because they have a larger chance of being mutated in all samples. Accordingly, we often see *TTN* and *MUC16*, which are the largest proteins (Scherer, 2008, p. 26). Possible solutions are to create a blacklist of genes that are never to be displayed or to mark very long transcripts in the user interface.

[15]http://evs.gs.washington.edu/

4.6.1 Open problems

In this section, we mention those issues whose existence we have recognized, but for which a solution is probably not as straightforward as in the previous section and which therefore require further research.

Non-existence of variants

In its current design, the database can only indicate existence of a variant, but not non-existence. The problem is as follows: If no entry exists in the calls table for a given variant, there are two possibilities. The first is that the variant caller encountered a sufficient number of reads covering that position that support the hypothesis that there is no difference to the reference. The second option is that not enough reads cover the position to support any hypothesis. When, for example, a variant is found in a child but not its parents, it is a candidate for being a *de novo* mutation only if the coverage in both parents at that position is sufficient. To reflect this, one would need to model three types of variants: Those that exist, those that do not exist, and those for which one does not know.

One way to solve the problem could be to automatically post-process result lists of candidate variants by inspecting the appropriate BAM files. The disadvantage of this approach is that it forces us to leave the database domain. Storing full existence/nonexistence/unknown information in the database for each sample and each coordinate is not an option as this requires too much space and slows down all queries. A second option is therefore to summarize the data by storing, for each sample, a set of intervals that describe regions with high (or low) coverage. We have conducted initial experiments which suggest that the number of intervals needed is below 100 000 per sample, making this approach feasible.

Automating testing

Little of Exomate's functionality that is exposed via the web interface is tested automatically for correctness. The largest problem is that we do not know whether the result set returned by a particular query is correct and that it therefore does not make sense to check whether it remains the same after changing the algorithm. This does not mean that the software is unreliable: The results given in the previous section show that results are accurate. Also, many manual checks have been done by the users. False positives and negatives were reported by them and investigated by us. Often, these reports indicate not a problem in the pipeline itself or in the algorithms, but in a misunderstanding of how a particular parameter works. Incorrect metadata were also the cause of false results. Rare problems in the software were subsequently fixed and a decreasing number of error reports over time increase our confidence in the results, but of course, this is not a substitute for automatic testing.

An initial solution to part of the problem is to automate some queries that search for variants verified by Sanger sequencing, making sure that these are shown. Some checks could also verify that false positives do not re-appear after a modification to the software and when new data is imported. The best variants for verification may be those from published studies whose sequencing data is available, such as the one by Harbour et al. As we have described in the last section, even that may be unreliable as we have encountered possibly incorrect metadata, at least in that study.

Correctness of metadata

In our own software we also need to be concerned with problems in metadata. Errors are hard to detect and the checks described in Section 4.4.5 can detect only a few classes of errors. Some may be caught by using check digits in sample numbers, and by using barcodes. In fact, connecting Exomate to a full laboratory information management system (LIMS) that is able to properly track samples could help, but is currently out of scope. We cannot give a solution here, but only emphasize that procedures need to be established that ensure metadata is double-checked before being added to the database.

4.7 Discussion

Using a relational database

The decision to build on a database instead of using a file-based workflow was influenced by the realization that most of the variant data one works with during exome sequencing can be viewed as tables and that the specialized tools used to manipulate them arguably re-implement the same algorithms as those already available in a database – except for the fact that the database engine likely has received more performance tuning and testing than the specialized tool. This assumption has so far turned out to be true: It is usually quite easy to implement a new, custom analysis on top of the database with a few lines of SQL. Adding to this a web-accessible user interface takes only a few more lines in Python, and is, for simple queries, therefore done very quickly.

Using SQL also has its problems. It is a very concise language that essentially allows us to express the properties of the desired result objects in very few lines of code. In fact, we are not only allowed to but often have to express the entire idea in a few lines. Typical software engineering techniques such as refactoring and using subroutines that help to make the code easier to understand are not easily possible or lead to major performance loss. In that regard, usage of the object-relational mapping library alleviates this problem to a large degree, as it makes it possible to break up a query into re-usable parts which lend themselves to refactoring. However, as the underlying language is SQL, it still takes some time to design complex queries.

Development model

One of our aims was to remove the "human bottleneck" in exome analysis by adding the web interface, which is always ready to answer a research question. We noted that it requires some effort in order for that bottleneck to not re-appear, by following a simple rule: If a request for an analysis comes in, one must add the possibility for such a query to the software, and not give the answer oneself. Inspired by agile software development models, we also try not to offer too many query parameters prematurely, but generalize only when there is evidence that the flexibility is required.

Proceeding in the described way improves reproducibility of experiments since the source code of the query algorithms serve as documentation that describes how a result was obtained.

Another observation that has likely been made before by many other software developers is the following. It is crucial for the developer to understand how the user interacts with the software in order to recognize those tasks that could be automated. Many people without formal computer science training or programming experience will happily work around missing functions in the software that they use by effectively implementing their own algorithms "by hand". An example could be the copying and pasting of a gene name from a result page into the search field of a third-party website, whereas adding an appropriate hyperlink to the result page would be a matter of minutes for the software engineer.

Exomate in its current form is a practical tool that has already shown its usability in practice, but we also consider it to be easy to build upon in order to meet new challenges in exome sequencing.

5 Aligning Flowgrams to DNA Sequences

We return in this chapter to the low-level task of aligning a read to a reference, but focus on reads represented as flowgrams, that is, those from the 454 or Ion Torrent sequencer. Incorrectly measured homopolymers are the major source of errors in such data.

Recent work has focused on improving the accuracy of the initial conversion of flowgrams to DNA sequences (base calling) in order to facilitate read mapping and downstream analysis of sequence variants. However, base calling always incurs a loss of information by discarding fractional intensity information. We introduce here a method to directly align flowgrams to DNA sequences. In this way, base calling can be avoided entirely. We call our algorithm *flowgram-string alignment*. It is based on dynamic programming, but covers more cases than standard local or global sequence alignment. We also propose a scoring scheme that takes into account sequence variations (from substitutions, insertions, deletions) and sequencing errors (flow intensities contradicting the homopolymer length) separately. This allows to resolve fractional intensities, ambiguous homopolymer lengths and editing events at alignment time by choosing the most likely read sequence given both the nucleotide intensities and the reference sequence. We also demonstrate the advantages of flowgram-string alignment compared to base-called alignments. A proof-of-concept implementation called *FlowG* is available under the MIT license from http://www.rahmannlab.de/software .

5.1 Introduction

We describe in this section the type of measurement errors arising in 454 sequencing when not the full information available in the flowgram is used, and discuss previous work attempting to incorporate more of that information into the alignment algorithm. We then motivate why it seems hard to incorporate editing events into the algorithm.

5.1.1 Information loss from base calling

A flowgram is a sequence of nucleotides, each of which is paired with its fractional measured intensity (see Section 1.5.1). By rounding intensities in a flowgram to the nearest integer, a regular DNA sequence can be inferred, a step known as *base calling*. For example, a thymine at intensity 2.4 would be called as `TT`. Subsequently, standard read-mapping and sequence alignment algorithms can be used to compare the obtained sequence reads with reference sequences.

Recent work has focused on improving base calling from straightforward rounding to the nearest integer towards more elaborate statistical methods based on HMMs (Golan and Medvedev, 2013). Nevertheless, base calling always incurs a loss of information by replacing the fractional intensities with integer lengths. For example, the distinction between a `C` observed at an intensity of 5.4 vs. one at an intensity of 4.6 is lost. Both are called as `CCCCC`, but in the first case, alignment to six `C`s is much more plausible than in the latter case.

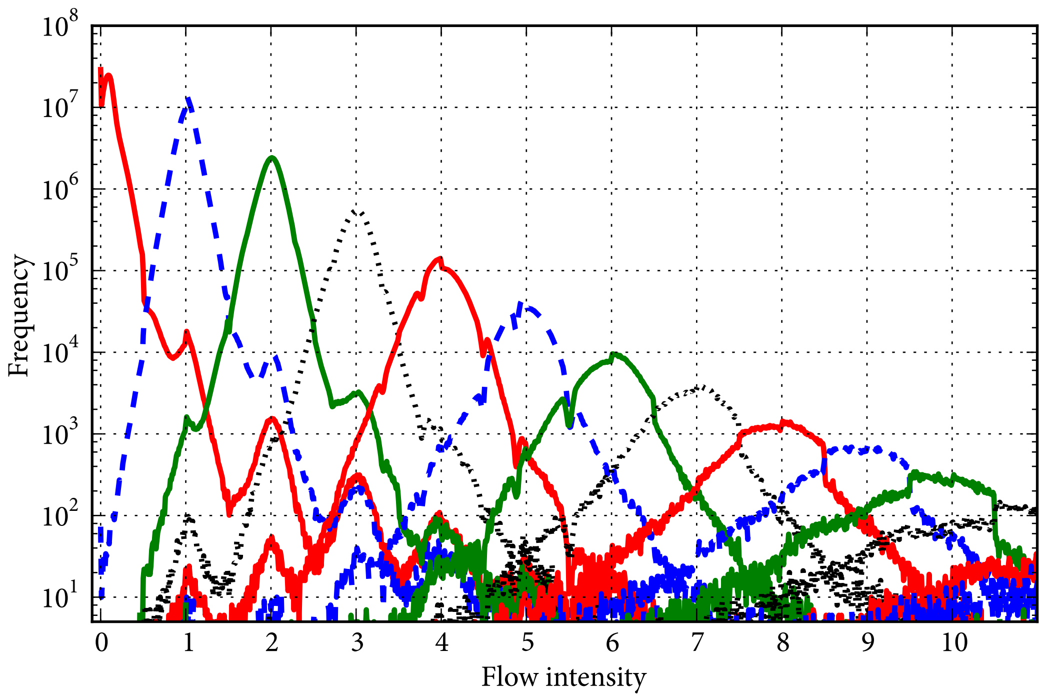


Figure 5.1 Distribution of flowgram intensities given homopolymer length ℓ. The data was obtained by mapping base-called reads to an *Arabidopsis* reference and correlating genomic homopolymer lengths to flowgram intensities, assuming insertions of the same character into runs are overcalls and deletions within a run are undercalls. Some artifacts likely remain from this heuristic procedure, including smaller peaks at integer intensities that are not equal to the run length, but the graph looks remarkably similar to Figure 3 in the article by Balzer et al. (2010).

For a homopolymer of length ℓ, rounding works correctly if the measured intensity is in the interval $[\ell - 0.5, \ell + 0.5)$. If it is outside that range, a *homopolymer error* occurs. If the intensity is $\ell + 0.5$ or larger, this is called an *overcall* and if it is smaller than $\ell - 0.5$, it is an *undercall*.

Homopolymer errors occur because the distributions of flowgram intensities observed for different run lengths overlap each other, as seen in Figure 5.1. The overlap gets more pronounced the longer the homopolymer is (Balzer et al., 2010). Within an alignment to a reference sequence, homopolymer errors result in spurious insertions (for overcalls) or deletions (for undercalls). These are the dominant form of errors in 454 sequencing data (Margulies et al., 2005).

5.1.2 Previous work avoiding base calling

We put forward the hypothesis that it makes more sense to invent alignment algorithms that directly work on flowgrams, instead of on a base-called sequence. A few publications on flowgram-based alignment already exist, but none clearly separates the two processes of sequence editing and flowgram under- and overcalling.

Vacic et al. (2008) model the distribution of flowgram intensities and derive a probabilistic model to compute the log-odds score that a given flowgram originates from a given genomic sequence. Using an enhanced suffix array of the run-length compressed reference genome, positions with a high score are then found efficiently. Their software FLAT is intended for mapping sequenced small RNA molecules to a reference and not for aligning diverged DNA sequences, so they do not take into account editing events. We use a similar way of deriving log-odds scores for differences between aligned reference and flow intensity.

Quince et al. (2009) use an algorithm adapted from global alignment (Needleman and Wunsch, 1970) to align two flowgrams, first converting the reference sequence into flowspace. The authors' idea is to introduce gaps only in steps of four in order to take into account the cyclic nature of the flow order. The remaining description in the paper is brief, but one can deduce that a single flow is aligned to a homopolymer. It is unclear how editing is handled. The cost function used is $-\log P(f \mid \ell)$, where $P(f \mid \ell)$ is the probability of observing flow intensity f given a homopolymer of length ℓ.

Lysholm et al. (2011) propose a different method of aligning flowgrams, which is an extension of the Smith-Waterman local alignment algorithm (Smith and Waterman, 1981) and can handle substitutions and indels with affine gap costs. FAAST's alignment is computed between the reference string and the base-called flowgram. Its modified scoring system reduces gap costs at points of uncertain homopolymer lengths.

Our contributions

In contrast to previous work, we do neither convert the reference into flowspace nor the flowgram to a string. Instead, we present the first algorithm that directly aligns a flowgram to a reference sequence, being aware of two processes in between: sequence editing between the reference and the (unknown) sequenced sample, and sequencing errors resulting in imprecise flow intensities.

After stating basic definitions (Section 5.1.3), we introduce a dynamic programming algorithm for optimal flowgram-string alignment (Section 5.2). The key component is a detailed scoring scheme that models both sequence editing events and flow intensity measurement errors; it is described in detail in Section 5.3, where we also explain how the scoring parameters can be set to reasonable values. In Section 5.4, we demonstrate how flowgram-string alignment improves upon aligning a base-called sequence. A discussion and outlook on future work concludes the chapter.

5.1.3 Basic definitions and ideas

Let Σ_{DNA} be the DNA alphabet as in preceding chapters. Remember that b^ℓ is the character $b \in \Sigma_{\text{DNA}}$ repeated ℓ times, where ℓ is a non-negative integer. Such a string is called a *homopolymer* of length ℓ.

The output of a 454 or Ion Torrent sequencer for a single read is a sequence of pairs of a nucleotide character and an intensity, called a flowgram (Margulies et al., 2005), which we now define formally.

Definition 9 (Flow)**.** A *flow* is a pair (b, f), where $b \in \Sigma_{\text{DNA}}$ is the *flow character* (or *flow nucleotide*) and $f \in \mathbb{R}_0^+$ is the *flow intensity*.

In analogy to exponentiation, we also write a single flow as $b_i^{f_i}$, that is, as the flow character followed by the flow intensity as a superscript. For example, instead of $(\mathtt{A}, 3.4)$, we write $\mathtt{A}^{3.4}$.

Definition 10 (Flowgram)**.** A *flowgram* is a finite sequence $F = (F_1, F_2, \ldots, F_m)$ of flows $F_i = (b_i, f_i)$. The *flowgram length* is m. For $k = 1, \ldots, m-1$, we require that $b_i \neq b_{i+1}$.

The four nucleotides are typically added in repeating cycles. We assume in the following that the order is $(\mathtt{T}, \mathtt{A}, \mathtt{C}, \mathtt{G}, \ldots)$, the typical order used in 454 instruments (that for Ion Torrent is different). We may also say that a read is in *flowspace* to indicate that it is a flowgram.

Given flowgram $F = (F_1, \ldots, F_m)$, the sequence $F_{j\ldots k} := (F_j, \ldots, F_k)$ is a *subflowgram*. For $j = 1$, it is a *flowgram prefix*, and for $k = m$, it is a *flowgram suffix*.

Example 10. A possible measured flowgram for the sequence TTCGG is $\mathtt{T}^{2.3}\mathtt{A}^{0.1}\mathtt{C}^{0.9}\mathtt{G}^{1.9}$.

The flowgrams output by both 454 and Ion Torrent sequencers are stored in Standard Flowgram Files (SFF). The cycle order is stored once globally, and each flow intensity is stored as a 16-bit unsigned integer value and scaled such that a value of 100 represents an intensity of 1. It is thus a fixed-point number representation with two decimal places after the decimal separator and allows intensity values from 0.00 up to $(2^{16} - 1)/100 = 655.35$.

The SAM/BAM file format has also been extended to allow storing flowgram information. For this, an array data type was added to the specification and the tags “FO” (flow order) and “FZ” (flow intensities) were standardized. Both SFF and SAM/BAM formats additionally store a basecalled regular DNA sequence for each read.

Definition 11 (Canonical flowgram)**.** Given a string s, the *canonical flowgram* for s is the flowgram that arises when we substitute all runs of character b of length n with the flow b^n and insert appropriate flows of intensity zero in between in order to get the correct order of nucleotides according to cycle order.

Example 11. The canonical flowgram for TCTT (using cycle order TACG) is $\mathtt{T}^1\mathtt{A}^0\mathtt{C}^1\mathtt{G}^0\mathtt{T}^2$.

Definition 12 (Canonical DNA sequence)**.** Given a flowgram F, let $\bar{F}$ be the flowgram with each intensity rounded to the nearest integer. The *canonical DNA sequence* for F is the DNA sequence which has the canonical flowgram $\bar{F}$, if it exists, and is undefined otherwise.

Note that a canonical DNA sequence for $F = \mathtt{T}^{1.1}\mathtt{A}^{0.1}\mathtt{C}^{0.4}\mathtt{G}^{0.2}\mathtt{T}^{2.3}$ does not exist, as rounding leads to $\bar{F} = \mathtt{T}^1\mathtt{A}^0\mathtt{C}^0\mathtt{G}^0\mathtt{T}^2$, but TTT has a canonical flowgram starting with $\mathtt{T}^3$. (A base caller that is cleverer than rounding (Golan and Medvedev, 2013) calls TCTT in this example instead of a non-existing canonical DNA sequence.)

5.1.4 How editing changes the flowgram

To align a flowgram to a reference sequence, we could convert the reference to its canonical flowgram and perform alignment between the resulting two flowgrams (Quince et al., 2009; Vacic et al., 2008). We show in this section that this approach complicates the modeling of editing events.

Editing events occuring in base space change the associated canonical flowgram in various ways that depend on the sequence context, that is, on the bases adjacent to the change. Changes that are compatible with the flow order only change an intensity, but those that are incompatible may insert or delete a full cycle of four flows.

Example 12 (Events compatible with flow order)**.** Insertion of a third T into TT changes the canonical flowgram from $\mathtt{T}^2$ to $\mathtt{T}^3$. A substitution that changes TAG to TCG changes the canonical flowgram from $\mathtt{T}^1\mathtt{A}^1\mathtt{C}^0\mathtt{G}^1$ to $\mathtt{T}^1\mathtt{A}^0\mathtt{C}^1\mathtt{G}^1$.

Example 13 (Events incompatible with flow order)**.** A substitution that changes TAC into TGC changes the canonical flowgram from $\mathtt{T}^1\mathtt{A}^1\mathtt{C}^1$ to $\mathtt{T}^1\mathtt{A}^0\mathtt{C}^0\mathtt{G}^1\mathtt{T}^0\mathtt{A}^0\mathtt{C}^1$, which has an additional cycle. The insertion of A into TT results in the flowgram $\mathtt{T}^1\mathtt{A}^1\mathtt{C}^0\mathtt{G}^0\mathtt{T}^1$ with an additional cycle. Deleting the G from TGT shortens the flowgram from $\mathtt{T}^1\mathtt{A}^0\mathtt{C}^0\mathtt{G}^1\mathtt{T}^1$ to $\mathtt{T}^2$.

Algorithmically dealing with editing events in flowspace

A method that changes a canonical flowgram according to a given substitution event may work as follows. Assume standard cycle order and, without loss of generality, that the substitution occurs within a run of the character T of length $n \geq 1$. First, split $\mathtt{T}^n$ into $\mathtt{T}^k$ and $\mathtt{T}^{n-k-1}$, where $k \in \{ 0, n-1 \}$ is the number of characters preceding the substituted base. Insert flows $\mathtt{A}^{x_\mathtt{A}}$, $\mathtt{C}^{x_\mathtt{C}}$, $\mathtt{G}^{x_\mathtt{G}}$ between them, where one of the x_b is set to one and the others are zero, depending on which character is substituted. Thus, $\mathtt{T}^n$ is replaced with

$$\mathtt{T}^k\mathtt{A}^{x_\mathtt{A}}\mathtt{C}^{x_\mathtt{C}}\mathtt{G}^{x_\mathtt{G}}\mathtt{T}^{n-k-1} .$$

When $k = 0$ or $k = n-1$, this transformation can result in three consecutive flows with intensity zero, which is not a valid canonical flowgram. Therefore, in a second step, check whether this occurs (potentially twice) and fix the flowgram by removing the three zero flows and by merging the flanking flows.

Example 14. We consider the substitution that changes ACCTTT into ACCCTT. The canonical flowgram is $\mathtt{A}^1\mathtt{C}^2\mathtt{G}^0\mathtt{T}^3$, which is initially transformed into $\mathtt{A}^1\mathtt{C}^2\underline{\mathtt{G}^0\mathtt{T}^0\mathtt{A}^0}\mathtt{C}^1\mathtt{G}^0\mathtt{T}^2$. Since the underlined subflowgram is not allowed, it is removed and $\mathtt{C}^2$ and $\mathtt{C}^1$ are merged, resulting in $\mathtt{A}^1\mathtt{C}^3\mathtt{G}^0\mathtt{T}^2$.

Insertions are handled in a similar way: by inserting a full cycle at the appropriate position, possibly splitting up a flow if the insertion is within a run, and then repairing the resulting flowgram if necessary. For deletions, no cycle needs to be added, only the intensity of the affected flow needs to be decreased before repairing.

The method could be sped up by tabulating the induced changes from all possible editing events, taking adjacent bases into account. When we start to take into account that multiple editing events can occur simultaneously, it becomes clear that the approach of converting the reference to flowspace is not constructive.

Thus, our approach is to align a flowgram directly to a reference sequence, converting neither the flowgram to a string nor the reference to a flowgram.

Direct alignment between flowgram and reference sequence

Our main idea is to conceptually model a two-stage process (sequence editing, errors during sequencing) within one model and scoring function. It is best visualized with Figure 5.2.

Fractional intensities and ambiguous run lengths are resolved at alignment time by choosing the most likely read sequence given flowgram and reference sequence. In contrast to the work by Vacic et al. (2008), we also model differences due to editing events. Every observed flow b^f must

Reference r Substring t	editing $\longrightarrow$	Read/sample s Homopolymer b^ℓ	sequencing $\longrightarrow$	Flowgram F Flow b^f
(known)		(unknown)		(observed)

Figure 5.2 Differences between an observed flowgram F and a reference sequence r arise from two different processes that cannot be distinguished by the observer: Sequence editing, responsible for differences between the sequenced sample and the reference in databases, and sequencing/measurement errors (intensity overcalls and undercalls) incurred during the sequencing process.

be explained by a substring t of the reference. The substring and the flow need not necessarily agree: If there is a non-b character in t, then there is a substitution or an insertion, and if f deviates from $|t|$, then there is an insertion or deletion event or a homopolymer error. Thus, a flow b^f can be explained as a sequenced homopolymer b^ℓ (where ℓ is integer), which in turn is an edited version of t (compare Figure 5.2).

5.2 A flowgram-string alignment algorithm

We define flowgram-string alignments and give the recurrence for finding an optimal alignment. The scoring functions are described later in Section 5.3.

5.2.1 Alignments

Definition 13 (Flowgram-string alignment)**.** A *flowgram-string alignment* $\mathcal{F}$ between a flowgram F of length m and a string s of length n is a finite sequence of pairs $\mathcal{F} = (F'_i, t_i)$, where each F'_i is a flow or the space character $(-)$ such that the concatenation of all non-space F'_i is the flowgram F, and where the t_i are (possible empty) substrings of s such that their concatenation is equal to s.

Example 15. Given flowgram $F = \mathtt{T}^{1.0}\mathtt{A}^{2.1}\mathtt{C}^{1.1}\mathtt{G}^{2.2}\mathtt{T}^{3.1}\mathtt{A}^{0.1}\mathtt{C}^{2.7}\mathtt{G}^{0.2}$ and string $s = \mathtt{TAAGGATGTCC}$, a possible alignment is (see also Figure 5.3):

$\mathtt{T}^{1.0}$	$\mathtt{A}^{2.1}$	$\mathtt{C}^{1.1}$	$\mathtt{G}^{2.2}$	$-$	$\mathtt{T}^{3.1}$	$\mathtt{A}^{0.1}$	$\mathtt{C}^{2.7}$	$\mathtt{G}^{0.2}$
T	AA	ε	GG	A	TGT	ε	CC	ε

We see that flowgram-string alignment can describe all editing events: $\mathtt{T}^{3.1}$ aligned to TGT involves a mismatch (G instead of T); $\mathtt{C}^{1.1}$ aligned to ε means that there is an insertion; and the space aligned to an A is a deletion. We will also see in Section 5.3 that, with the proper scoring function, flowgram-string alignment can distinguish between homopolymer errors and insertions. The scoring function will inform us whether the rightmost flow $\mathtt{G}^{0.2}$ aligned to an empty string ε of the reference needs to be interpreted as a homopolymer error of 0.2 or as an insertion. Our alignment algorithm picks the option with the better score. Note also the asymmetry between deletion and insertion: Only characters are deleted, and only flows are inserted.

A flowgram-string alignment describes how (1) editing events and (2) sequencing errors due to over- or undercalling add up to result in an observed flowgram. In contrast to previous flowgram

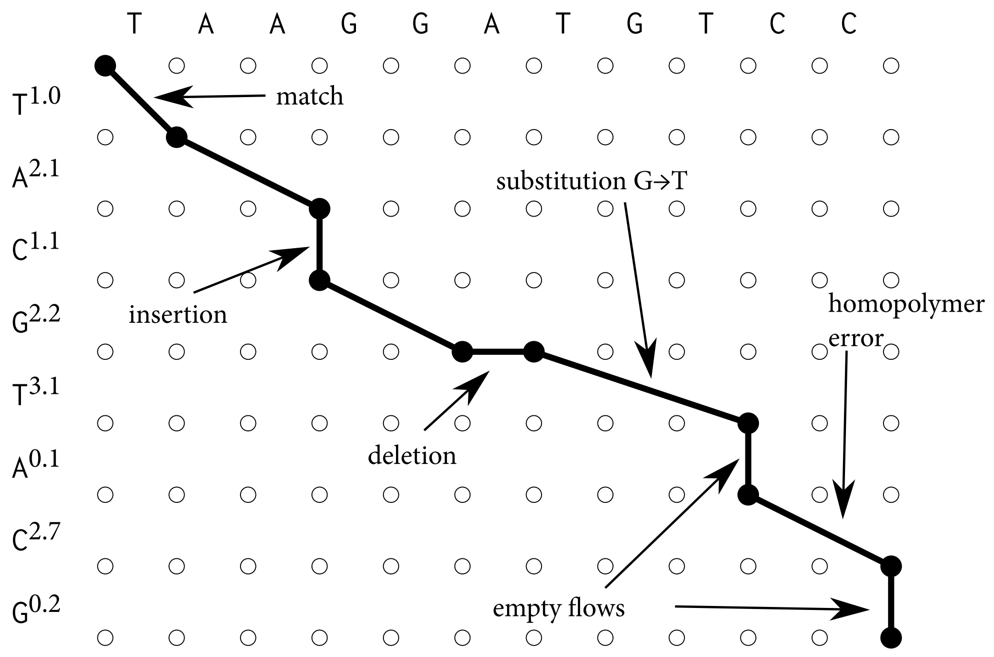


Figure 5.3 A visualization of the alignment of Example 15. The black path represents the alignment. A flowgram alignment is a path from the top left node to the bottom right node along edges allowed by Equation (5.1). One difference to global alignment is that the path may skip an arbitrary number of columns, but it cannot skip rows.

alignment ideas, there is no need to convert the reference to a flowgram or to convert the flowgram into a string. Instead, a flowgram-string alignment describes a direct relationship between flowgram and reference.

5.2.2 The flowgram-string alignment graph

A flowgram-string alignment can be interpreted as a path through a graph of $(m+1) \times (n+1)$ vertices $(i, j) \in \{0, \ldots, m\} \times \{0, \ldots, n\}$ that has different types of edges with different scores (see Figures 5.3 and 5.4). The score of an alignment $\mathcal{F}$ is the sum of the scores of the individual edges used by the alignment, and so finding the optimal alignment is equivalent to finding a highest-scoring path. The following two edge types exist:

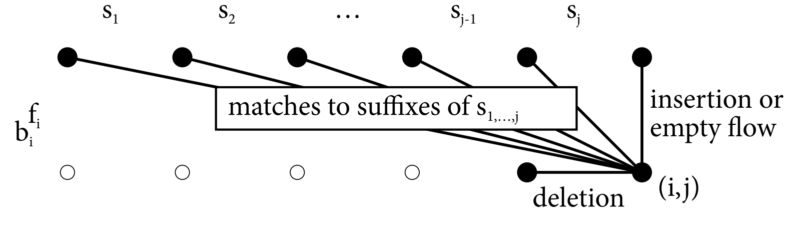


Figure 5.4 Visualization of different types of edges of the alignment graph and of the recurrence for cell (i, j) in the dynamic programming matrix

- Horizontal edges that connect (i,j) to $(i,j-1)$. These represent deletions, that is, the flowgram indicates that a nucleotide from the reference is missing in the sample. The score $del < 0$ is assigned to these edges.

- Vertical and diagonal edges that connect (i,j) to $(i-1,j-k)$ for all $k \in \{0,\dots,j\}$. For $k = 0$, the edge is vertical and interpreted as either an empty flow aligned to an empty substring, or as an insertion, where the flowgram indicates a homopolymer not present in the reference. These edges use a complex scoring function $v(b,f,t)$ for aligning flow b^f to substring $t = s_{k+1\dots j}$. This scoring function is central to our method and discussed in detail in Section 5.3.

5.2.3 Recurrence

Let (b,f) be a flow and $t \in \Sigma^*_{\text{DNA}}$ a string. We assume that scoring parameter *del* and scoring function $v(b,f,t)$ are available (see Section 5.3).

Let $S(i,j)$ be the optimal score between the length-i prefix of flowgram F of total length m and the length-j prefix of string s of total length $n = |s|$. The recurrence for $S(i,j)$ follows from the structure of the alignment graph, in which the optimal flowgram-string alignment is a highest-scoring path, analogously to standard global alignment. Other variants (local, semiglobal, etc.) are possible; for ease of exposition, we focus on the global case. We have

$$
\begin{aligned}
S(0,j) &= j \cdot del, \\
S(i,0) &= \sum_{k=1}^{i} v(b_k,f_k,\varepsilon), \\
S(i,j) &= \max \left\{ \begin{array}{l} S(i,j-1) + del, \\ \max\limits_{k=0,\dots,j} \big(S(i-1,k) + v(b_i,f_i,s_{k+1\dots j})\big) \end{array} \right\} . \qquad (5.1)
\end{aligned}
$$

The two cases for $S(i,j)$ correspond to the two types of edges. The inner maximization corresponds to the vertical and diagonal edges, in which the score of aligning the current flowgram to all suffixes of $s_{1\dots j}$ (including the empty suffix for case $k = j$) is found. It is the main difference to regular global alignment. With dynamic programming, $S(m,n)$ can be computed in time $\mathcal{O}(mn^2)$, assuming v can be evaluated in constant time.

5.3 Scoring

The score $v(b,f,t)$ for pairing flow (b,f) with string t must take into account two different processes – editing and measurement – that cannot be distinguished by an observer (Figure 5.2). First, editing events occur that change a substring t of the reference into a homopolymer b^ℓ, but ℓ is unknown. Second, an intensity f is measured for b^ℓ. We score the first process by $s_{\text{edit}}(b,\ell,t)$, which is the score of an optimal alignment between t and b^ℓ. The score $\sigma(f,\ell)$ is assigned to measuring intensity f for a homopolymer run of length ℓ; we assume that it does not depend on the nucleotide b.

Since ℓ is unknown, to obtain $v(b,f,t)$ we maximize over all possible lengths in order to pick the most plausible explanation:

$$v(b,f,t) := \max_{\ell=0,1,2,\dots} \big(s_{\text{edit}}(b,\ell,t) + \sigma(f,\ell)\big) \tag{5.2}$$

As we will see in the two following subsections, this potentially infinite maximization is in fact finite, since a value of $\ell \gg \max\{\,|t|, f\}$ will yield a strongly negative score in both terms and cannot achieve the maximum. In practice, positive scores are only obtained if $f \approx |t|$ for a choice of ℓ close to both f and $|t|$.

To reconstruct the most plausible process to flow b^f via homopolymer b^ℓ from sequence t, we also store the value of ℓ maximizing $v(b,f,t)$ in Equation (5.2),

$$L(b,f,t) := \operatorname*{argmax}_{\ell=0,1,2,\dots} \big(s_{\text{edit}}(b,\ell,t) + \sigma(f,\ell)\big)\ . \tag{5.3}$$

It is this (unknown but inferred) value of $\ell = L(b,f,t)$ that links the two processes of sequence editing and sequencing.

As we will show, the score $v(b,f,t)$, and hence $L(b,f,t)$, depends on b and t only through the number $e = e(t,b)$ of characters in t that are equal to b and the number $\bar{e} = \bar{e}(t,b)$ of characters different from b (see Section 5.3.1). Therefore we can write $v(b,f,t) = v'(f,e,\bar{e})$ and $L(b,f,t) = L'(f,e,\bar{e})$. Tables of both L' and v' are pre-computed for realistic flow intensities $f \in \{\,0.00, 0.01, \dots, 9.99\,\}$ and values of e and $\bar{e}$, both in $\{\,0,\dots,9\,\}$, that is, 100 000 values overall. As the recurrence (5.1) considers different substrings t that differ in length by 1, the b characters in t can be counted in amortized constant time for each t, so each value of $v(b,f,t)$ is available in constant time. The non-tabulated rare cases can be computed on demand without measurably affecting the running time.

Reconstruction

As discussed, the most likely underlying DNA sequence when pairing flow (b,f) with t is the run b^ℓ with $\ell = L(b,f,t)$. Concatenating these runs for all flows in an optimal flowgram alignment thus results in the most likely sequence of the DNA fragment that was sequenced. We call this a *reconstructed* read or sequence.

5.3.1 Scoring of editing events

In this section, we derive the edit score $s_{\text{edit}}(b,\ell,t)$ to align two sequences: b^ℓ and t. This is, in fact, a classical sequence alignment problem, with the special property that one sequence b^ℓ is a homopolymer. Instead of using a standard global alignment algorithm every time when s_{edit} is called, we can give a closed formula because of the special structure.

We assume that scores for insertion (*ins*), deletion (*del*), mismatch (*mis*) and match (*mat*) are available and fulfill $ins, del < mis < 0 < mat$.

Let e be the number of characters in t that are equal to b, and let $\bar{e} = |t| - e$ be the number of characters in t that are not equal to b. If $|t| = \ell$, the score is composed of only match (e times) and mismatch ($\bar{e}$ times) scores. If t is longer than ℓ, then $|t| - \ell$ characters must be deleted from t to obtain length ℓ, and it is advantageous to delete only non-b characters, as long as there are any. If t is shorter than ℓ, we have e matches and $\bar{e}$ mismatches, and $\ell - |t|$ characters must be inserted into t. Thus the score for aligning t to b^ℓ can be expressed as

$$s_{\text{edit}}(b,\ell,t) = \begin{cases} e \cdot mat + \bar{e} \cdot mis & \text{if } \ell = |t|, \\ e \cdot mat + \bar{e} \cdot mis + (\ell - |t|) \cdot ins & \text{if } \ell > |t|, \\ \min\{\, e, \ell \,\} \cdot mat + \max\{\, \ell - e, 0 \,\} \cdot mis + (|t| - \ell) \cdot del & \text{if } \ell < |t| \,. \end{cases} \tag{5.4}$$

The parameter values for *mat*, *mis*, *ins*, *del* must be compatible with the scores for scoring flow intensities f against substring lengths ℓ, which we discuss in Section 5.3.3. We come back to choosing appropriate values in Section 5.3.4.

5.3.2 Alternative formulation

Equation (5.2) contains an implicit iteration over all homopolymer lengths ℓ. We give here an alternative formulation as an explicit algorithm. The idea is to start with an empty string (length $\ell = 0$) and iteratively add appropriate characters of t to it. This is equivalent to what is done in Equation (5.2) and serves simply as an alternative explanation. In our Python implementation, this algorithm is also slightly faster (by a constant factor) due to fewer function calls.

Algorithm 14 (Compute-V)**.**
Input: Flow character b; flow intensity f; string t.
Output: $v(b, f, t)$.

1. Compute the value $e(t, b)$ or look it up in a table.

2. Initialize *length* to zero and *editscore* to $|t| \cdot del$. That is, start out by assuming that all characters of t are deleted.

3. While $length < e$, increase *length* by one and add $del - mat$ to the *editscore*. That is, we incrementally undo deletions of characters equal to b and gain matches. Keep track of the best total score *editscore* $+\sigma(f, \ell)$.

4. While $length < |t|$, increase *length* by one and add $del - mis$ to the *editscore*. That is, we also undo deletions of non-b characters, gaining mismatches. Again, keep track of the maximal total score as above.

5. Keep increasing *length* by one, but add *ins* to *editscore* in each iteration. Stop iterating when $\sigma(f, \ell)$ starts to decrease. This assumes that σ decreases monotonically to the left and to the right of a global maximum. Again, keep track of the best total score.

6. Return the best total score.

5.3.3 Scoring of flow intensities against substring lengths

Here we describe how to set the scores $\sigma(f, \ell)$ for scoring the event that a flow of intensity f is aligned to a DNA sequence of length ℓ. Our approach is similar to that of Vacic et al. (2008), but we go further by analyzing the resulting empirical scores parametrically.

Intuitively, the score should be positive if $f \approx \ell$ and drop into the negative range when $|f - \ell|$ gets large. A consistent set of score values is obtained by using log-odds scores (Dayhoff et al., 1978; Müller et al., 2001), having their roots in the theory of score matrices for amino acids, such as the famous PAM matrices (Dayhoff et al., 1978). There the score Σ_{ij} between amino acids i and j is computed as the log-odds $\Sigma_{ij} = \log_{10}\left(P_{ij}/(\pi_i \cdot \pi_j)\right)$, where P_{ij} is the probability of observing i and j paired in an alignment and π_i, π_j are the background frequencies of amino acids i, j, respectively. Moreover, the joint probabilities P_{ij} depend on the divergence time t of the aligned sequences, and so different score matrices $\Sigma_{ij}^{(t)}$ are used for differently diverged sequences.

Here we follow a similar idea for deriving scores for evaluating differences between f and ℓ. We estimate frequencies from (assumedly correctly) aligned flowgrams to DNA sequences. For ease of exposition, we do not discuss different divergence times, and we assume that the flowgrams have been obtained from the DNA reference by sequencing, or at least from a very closely related reference sequence.

Given a large number of such aligned flowgram-DNA alignments, we construct a count matrix $C = (C_{f,\ell})$ for all reasonable genomic lengths $\ell \in \{0, 1, 2, \dots\}$ and flow intensities $f \in \{0.00, 0.01, 0.02, \dots, 1.00, \dots\}$, such that $C_{f,\ell}$ counts the number of times we observe a flow of intensity f aligned to a genomic sequence of length ℓ. The result is shown in Figure 5.1. We obtain a joint probability matrix $P = (P_{f,\ell})$ by dividing C through the sum of its entries. Background frequencies $\pi = (\pi_\ell)$ for genomic lengths are obtained as marginal probabilities $\pi_\ell = \sum_f P_{f,\ell}$, and similarly background frequencies $\tau = (\tau_f)$ for flow intensities. The score component for aligning a flow of intensity f to a homopolymer of length ℓ is defined in units of nats as

$$\sigma(f, \ell) := \ln \frac{P_{f,\ell}}{\tau_f \cdot \pi_\ell} \,.$$

To obtain such scores, we used three SFF files containing *Arabiopsis* reads (from an unspecified strain)[1]. To measure only the effects of homopolymer errors, only reads aligning close-to-perfectly to the *A. thaliana* reference sequence were considered further, and the empirical joint distribution of flow intensities f and homopolymer lengths ℓ was tabulated where $f \in [\ell - 1, \ell + 1]$.

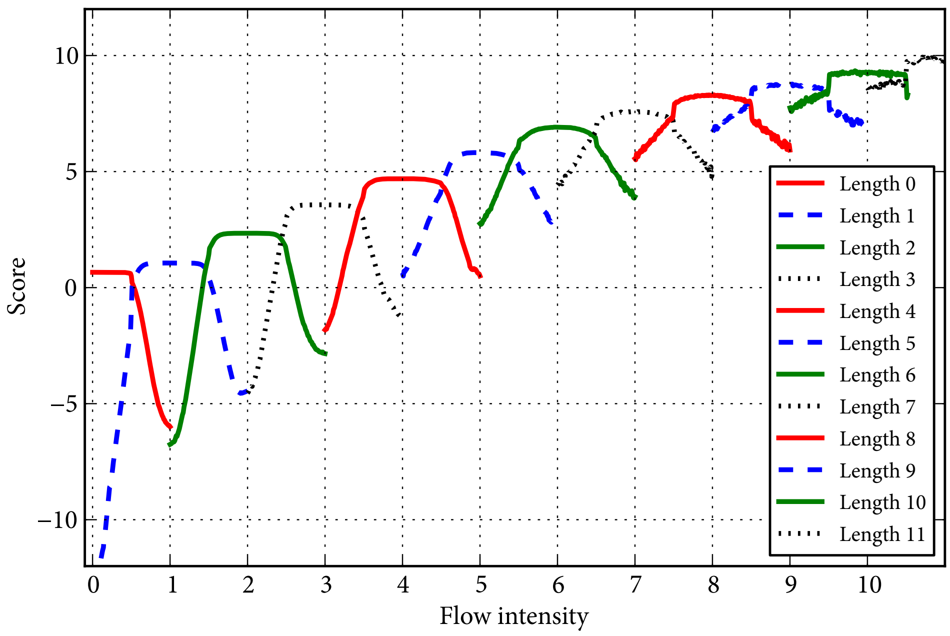
The resulting scores are shown in Figure 5.5, one curve for each ℓ with sufficient data (top) and one curve for some chosen intensities (bottom). Unsurprisingly, the maximum score occurs at flow $f = \ell$ for each ℓ. More remarkably, the score stays almost constant at the same level in the interval $f \in [\ell - 0.5, \ell + 0.5]$. At $\ell \pm 0.5$, there appears to be a sudden drop in the scoring function, beyond which we can observe an affine-linear course in the intervals $[\ell - 1.0, \ell - 0.5]$ and $[\ell + 0.5, \ell + 1.0]$. Therefore, for each ℓ, the score function can be described by five parameters, namely the values of $S_{\ell,f}$ for $f \in \{\ell - 1.0,\ \ell - 0.75,\ \ell,\ \ell + 0.75,\ \ell + 1.0\}$. Scores at other values of f are obtained by linear resp. constant interpolation (or extrapolation outside the 1.0-neighborhood). The parameters for lengths $\ell \leq 7$ are shown in Figure 5.6. As empirical data becomes sparser for larger ℓ, it is advisable to extrapolate the parameters instead of relying on data.

In summary, we implement $\sigma(f, \ell)$ for each ℓ as a piecewise affine function consisting of three components, given by the empirically determined parameters shown in Figure 5.5 (top).

5.3.4 Parameters for editing events

It remains to appropriately set the match, mismatch, insertion and deletion score parameters *mat*, *mis*, *ins* and *del*, respectively. These depend on the assumed degree of divergence of the

[1]We thank Bernd Weisshaar from the Chair of Genome Research, Bielefeld University, for providing the data.



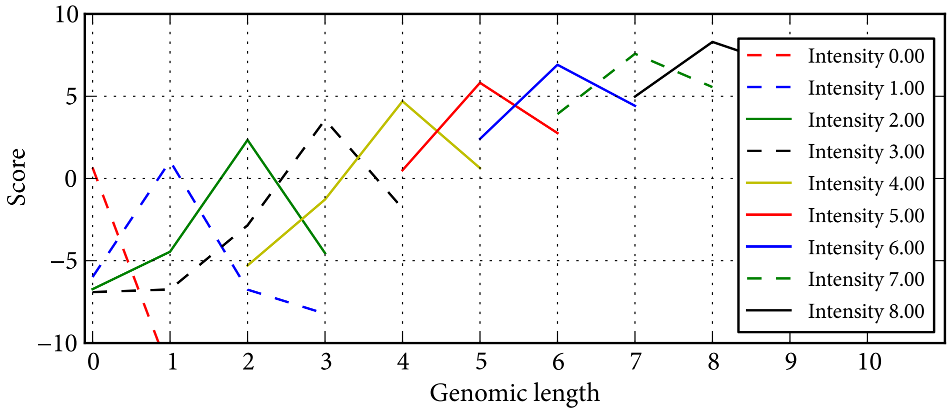


Figure 5.5 Empirically determined log-odds scores $\sigma(f, \ell)$, computed from the *Arabidopsis* reference frequencies shown in Figure 5.1. *Top:* The score is shown for different genomic sequence lengths ℓ (see legend) as a function of flow intensity f. *Bottom:* The score is shown for different intensities f as a function of genomic sequence length.

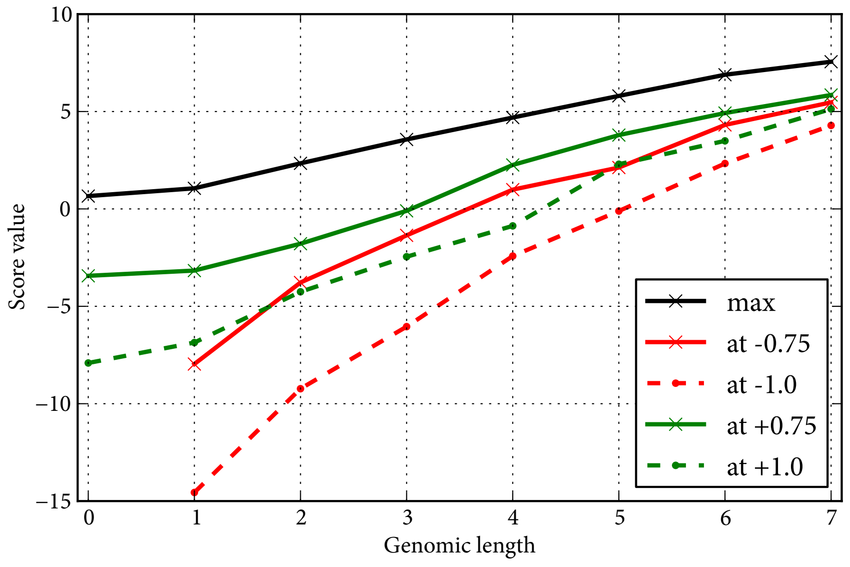


Figure 5.6 As each function in Figure 5.5 can be described by a piecewise affine function with three components, one of them constant, we estimated five parameters from five characteristic score values: for each length ℓ, the score values at $f \in \{\,\ell - 1.0, \ell - 0.75, \ell, \ell + 0.75, \ell + 1.0\,\}$. The plot shows these five score values as a function of ℓ.

sequenced sample and the reference and can be obtained by (approximate) log-odds.

Assuming 3% divergence (i.e., 97% matches, as opposed to about 30% in alignments of random sequences), and rare insertions/deletion with a rate of 1/3000, it is reasonable to use

- $\mathit{mat} \approx \ln \frac{0.97}{0.3} = 1.173 \approx 1.2$,
- $\mathit{mis} \approx \ln \frac{0.03}{0.7} = -3.1498 \approx -3.1$,
- $\mathit{ins} = \mathit{del} \approx \ln \frac{1/3000}{C} \approx -8.0$ with some $C \approx 1$.

These are the scores that we use for evaluation; other assumptions will result in different scores. It is important to use the same logarithm (and scaling, if any) as for $\sigma(f, \ell)$ in order to keep both score components compatible.

5.4 Evaluation

Before we evaluate flowgram-string alignment against base-called alignment, let us illustrate typical miscalls made by base calling. Obviously, the most common case is that a homopolymer length is simply off by 1 because of rounding in the wrong direction. If a length of at least one remains, this can always be corrected by post-processing the alignments. However, there are more complex errors, such as spurious indels in the middle or between two homopolymers. Some real examples of problematic alignments which can resolved by our algorithm are illustrated in Table 5.1.

Table 5.1 Example errors made by alignment after base calling in contrast to flowgram-string alignment: (A) The undercalled $\mathtt{C}^{0.4}$ would be reported as a single-base deletion, but flowgram alignment reconstructs a read that includes the missing C. (B) The $\mathtt{G} \rightarrow \mathtt{A}$ substitution is mistakenly reported as a single-base deletion because of the low flow value. For flowgram-string alignment with the surrounding context, the case that AAGAA generated $\mathtt{A}^{4.4}$ is plausible. (C) Similarly, but slightly more complex, the $\mathtt{CA} \rightarrow \mathtt{AC}$ flip is mistakenly reported as a 2 bp deletion after base calling. For our method, however, two mismatches plus small intensity errors are more plausible than two deletions.

	Flowgram alignment				Reconstructed read	Alignment of base-called read
(A)	$\mathtt{A}^{2.0}$	$\mathtt{C}^{0.4}$	$\mathtt{G}^{1.0}$		AACG	AA-G
	AA	C	G			AACG
(B)	$\mathtt{T}^{0.9}$	$\mathtt{A}^{4.4}$	$\mathtt{C}^{1.1}$		TAAAAAC	TAA-AAC
	T	AAGAA	C			TAAGAAC
(C)	$\mathtt{T}^{0.9}$	$\mathtt{A}^{3.4}$	$\mathtt{C}^{3.4}$	$\mathtt{G}^{2.0}$	TAAAACCCCGG	TAAA--CCCGG
	T	AAAC	ACCC	GG		TAAACACCCGG

We now demonstrate that flowgram-string alignment reduces the number of differences between observed sequence and reference that are due to sequencing errors, but leaves actual mutation events untouched. We simulate DNA fragments of *E. coli K12* (NC_000913); call this the *original* data. We introduce mutations by adding 3% substitutions and 0.05% indels (*mutated* data). Then the 454 sequencing process is simulated with *flowsim* (Balzer et al., 2010). Reads in the resulting SFF file are base-called by rounding flow intensities (*basecalled*) and aligned to the original sequence. Alternatively, we use our flowgram alignment algorithm to align each flowgram to the *original* sequence. During the process, the most likely base-space *mutated* sequence is reconstructed using function $L(b, f, t)$ from Equation (5.3) (*reconstructed*).

All differences between *mutated* and *basecalled* are necessarily due to sequencing errors and wrong base calls. Similarly, all differences between *mutated* and *reconstructed* are due to sequencing errors and errors by our alignment method. Figure 5.7 compares histograms of the number of differences, measured by unit-cost edit distance. The differences are considerably reduced for flowgram-string alignment in comparison to base-calling: The distribution is shifted towards the left side. Thus, flowgram alignment is able to distinguish editing events and true mutations.

5.5 Future work

There are several ways to extend this work. For example, finding a more robust way to estimate the divergence rate between reference and sample than guessing it before computing alignments would be of interest. On the practical side, several optimizations of the basic alignment algorithm are desirable, improving the running time from $O(mn^2)$ to $O(mn)$ by restricting the considered predecessors in each node (i, j) of the alignment graph (compare Figure 5.4). It is clear that for a flow b^f, the best choices for t and ℓ have $|t| \approx \ell \approx f$.

Extending the algorithm to be able to use affine gap costs would be of high practical relevance. This is not entirely trivial, as gaps could extend over several flows, which in the current model

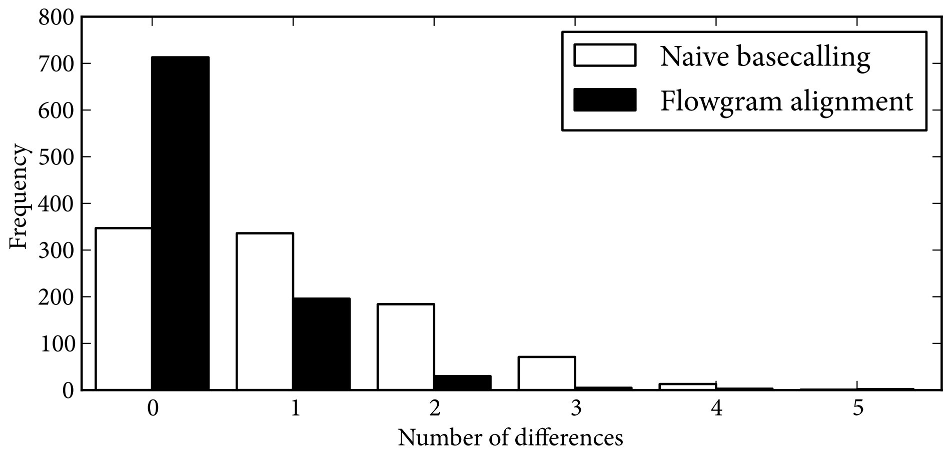


Figure 5.7 Histograms of the number of differences due to sequencing and base-calling errors for naive base calling and our method. Note how the latter distribution is shifted towards zero.

can only be considered separately.

Our approach used a scoring function derived from alignments of 454 data, but we expect that it works well also on Ion Torrent datasets with a re-estimated score function. In fact, preliminary work indicates that intensity distributions for each length overlap much stronger in Ion Torrent data, making it even more important to avoid basecalling.

It is currently not clear how to ensure that the flowgram alignment does not involve three or more consecutive zero-length runs. For example, the algorithm in its current form aligns the flowgram $\mathtt{T}^1\mathtt{A}^0\mathtt{C}^0\mathtt{G}^0\mathtt{T}^1\mathtt{T}^1$ to the string TT by pairing each $\mathtt{T}^1$ flow with a T character. In the alignment graph, the restriction is that the path must not contain three consecutive vertical edges that correspond to empty flows ($\ell = 0$).

5.6 Discussion

We presented a dynamic programming alignment algorithm that optimally aligns flowgrams output by 454 or Ion Torrent sequencers to DNA reference sequences directly, without explicit base calling. Our approach can also be interpreted as calling bases *conditional* on the reference we align to, that is, doing both steps at the same time instead of sequentially. Our algorithm is based on a two-stage process model (Figure 5.2) that explains both sequence editing and homopolymer sequencing errors. In particular, in the process, we can reconstruct the most plausible homopolymer length ℓ for each flow b^f and thus separate flow intensity over- and under-calling from sequence editing. Our method is the first one that cleanly separates the two processes.

A major challenge is to design a scoring scheme for flowgram-DNA alignment that is of low complexity (that is, has few parameters) and statistically well-founded. We started from a classical log-odds framework (Dayhoff et al., 1978) that was also used by Vacic et al. (2008). Going a step further, we noted that for each length ℓ, the score function has a simple three-component piecewise affine form that can be described by only five parameters. This yields the first low-complexity scoring scheme for directly aligning 454 flowgrams to DNA sequences.

6 Conclusion

In this final chapter, we start with a summary of the main results from the preceding four chapters (Section 6.1). While those chapters are related in that they all demonstrate how to tackle challenges that arise in the processing of high-throughput sequencing data, the four topics were presented mostly independently. Therefore, it is now appropriate to point out the connections between them (Section 6.2). We finally discuss some thoughts on the development of algorithms and tools for high-throughput sequencing in Section 6.3.

6.1 Summary

Adapter trimming

DNA sequencing reads of short molecules can contain adapter sequences. We give an algorithm for removing different types of $5'$ and $3'$ adapters. It is based on semiglobal alignment and finds overlapping regions between read and adapter. Using auxiliary matrices in the dynamic programming algorithm, only alignments below a given error rate threshold are found, where the error rate is defined as the number of errors divided by the number of aligned adapter characters. We show how to use only linear space while still keeping track of all information necessary to correctly locate adapter sequences. Performance is improved by using a version of banded alignment through Ukkonen's cutoff idea. We also show how the algorithm can be extended for removing adapters from colorspace reads. Due to the color encoding using overlapping dinucleotides, this is straightforward for $3'$ and slightly more complicated for $5'$ adapters. The *cutadapt* tool is introduced, which is easy to use, contains many additional features for making pre-processing adapter-contaminated HTS reads simple, and is in use by many other researchers.

Bisulfite q-gram indexing

For mapping of bisulfite-treated reads, the bisulfite q-gram index is proposed, which is a data structure that indexes all q-grams in a reference and also those that arise due to simulated bisulfite treatment (it includes all C-to-T- and G-to-A-compatible positions). Simulation theoretically leads to an exponential increase in index size, but on real-world data, the size of the index is only triple that of the regular q-gram index. We show how to create the index in time proportional to its memory usage.

We describe how to map reads with the index with the seed-and-extend paradigm, first finding short matches (hits) with the help of the index, and extending them to longer ones (still without errors) in either a "relaxed" or a "strict" mode. Strict hit extension can be done by using a deterministic finite automaton (DFA) that works on pairs of characters. Relaxed hit extension, on the other hand, allows incomplete bisulfite conversion within reads and is therefore more relevant in practice. We give an efficient bitparallel algorithm for relaxed hit extension and type determination (C-to-T or G-to-A) that works on multiple DNA characters at the same time. We describe how to extend seeds to an alignment that covers the full read and how to detect those parts of each read that are not bisulfite-converted.

A lemma is derived stating that the number of bisulfite strings of a given length n is approximately $1.19 \cdot 3.3^n$. The result can be used to compute the fraction of the buckets in the bisulfite index that do not contribute to mapping of bisulfite reads.

We investigate how storing the differences between positions in each bucket instead of full four-byte words helps to compress the index, and find that a considerable compression of 25% can be achieved with our custom byte-aligned encoding scheme. This result also applies to non-bisulfite q-gram indexing. We finally apply the full read mapping algorithm to a dataset of 454 bisulfite sequencing data using the *Verjinxer* tool and observe that the fraction of partially methylated CpG islands located on the human X chromosome is larger than previously thought (Zeschnigk et al., 2009).

Exome sequencing

We describe the *Exomate* tool, which is used for the analysis of exome sequencing data and in particular assists in finding potentially disease-causing mutations in syndrome and tumor patients. The tool is split up into three parts. The first is an automated pipeline that implements all tasks needed to find variants (mutations), such as read mapping, quality recalibration, indel realignment and variant calling. The second part is a relational database, into which variants and metadata about patients, samples, the sequencing instrument, etc. are imported. The third part of Exomate is a web interface. In conjunction with the database, it was designed for highly interactive analysis of the variant data by medical researchers, without in-depth knowledge of command-line tools. The challenge lies in filtering the enormous amounts of variants by discarding those that are assumed to be not disease-causing. This can be done, for example, by quality filtering, discarding variants known in public databases (dbSNP), and discarding variants seen in an unaffected sample. All thresholds and filtering options can be set on the fly, resulting in dynamically generated SQL queries that finish within a few seconds. Extensive quality control was implemented through various queries that check for known chromosome loss, patient gender and dbSNP re-discovery rate.

Exomate is still in active development, but has already been used successfully in multiple studies (Martin et al., 2013; Czeschik et al., 2013; Voigt et al., 2013), and results of two other studies could be confirmed (Harbour et al., 2013; Santen et al., 2012).

Flowgram alignment

In sequencing technologies that obtain reads as flowgrams, measurement errors of nucleotide intensities are the dominant cause of sequencing errors (over- and undercalls). Instead of rounding fractional intensities to obtain a regular sequence, which can then be aligned, we integrate base calling into the alignment algorithm, avoiding information loss from rounding. This has been suggested before, but our method (flowgram-string alignment) is the first to use a well-founded statistical model. The score function has two components that model the processes taking place: 1) editing of the sample compared to the reference and 2) measurement errors during sequencing. A recurrence equation is given that leads to a dynamic programming algorithm on the alignment matrix between flowgram and DNA reference string.

For the first score component, we give a closed formula for alignment scores between a DNA string and a homopolymer, which is then used to show that the score can be evaluated in constant time by tabulating the values in a pre-processing step. For the second score component, we use correctly aligned reads to estimate empirical frequencies for intensity measurements vs. sequence length, from which we derive log-odds scores and then approximate them by a function with five parameters.

To evaluate the method, we compare flowgram-string alignment to regular alignment by simulating the two-stage sequencing process and find that our method considerably reduces the number of spurious editing events introduced by measurement errors.

6.2 Connections between topics

Having summarized the previous chapters, we can now point out the connections between them that have not been discussed until now. Some of them also show directions for future work.

The initial adapter trimming algorithms and the bisulfite read mapping work have been presented separately, but were in fact developed as part of the same methylation study (Zeschnigk et al., 2009). Proper trimming of adapters was a pre-condition for the success of read mapping in that experiment. Fortunately, there is no need to make the read trimming algorithms aware of bisulfite treatment: Adapters can be added after bisulfite treatment and will therefore remain unchanged. Alternatively, adapters with only methylated cytosines can be used; these will also not be changed. The third option, if the adapters have been modified, is to trim the modified sequence, which also does not require any changes in the trimming step.

Adapter trimming can also be a valid pre-processing step in exome sequencing. Feedback from users of the cutadapt tool shows that this is done in practice, and we tested whether it makes sense to do so on our data. Our target insert sizes are around 300 bp. Since actual lengths follow a normal distribution, a large variance could lead to some inserts being shorter than our 100 bp read length. Testing a few datasets for adapter contamination, we found that less than 0.4% of bases were trimmed. Since the improvements are marginal in relation to the needed processing time, and also because other factors such as quality trimming have a much larger effect, we chose not to include the step in the exome sequencing pipeline. A compromise in the future may be to trim only the first 1 million reads or so of a dataset and, as a further quality control measure, to mark those datasets in the web interface where the rate of trimmed reads exceeds a critical threshold.

Considering that one of the aims of our adapter trimming algorithm was to make it usable with 454 sequencing data, it appears that using flowgram alignment in a (not yet discussed) semiglobal variant would be an obvious improvement over the current method, which is based on edit distance. However, we see this as low priority future work for two reasons. First, the strength of our flowgram alignment algorithm is in distinguishing sequencing errors from true editing events, but there is expected to be no editing in adapter sequences. The second reason is that our assumption for adapter trimming has been that a score-based alignment method should not be used since the results are less comprehensible by the user.

In contrast, flowgram alignment is a good candidate for being used in methylation studies. Due to the effective reduction of the alphabet size from bisulfite treatment, homopolymer lengths of the `T` and `A` nucleotides increase and, compared to regular reads, result in more high-intensity flows, which cannot be measured well. Within our group, research is in progress in the area of amplicon sequencing, where short pre-determined segments of DNA are sequenced in multiple individuals. Preliminary results indicate that flowgram alignment improves correct detection of methylation.

Tentatively, we also suggest to draw a connection between our exome sequencing pipeline and methylation studies. Such studies are now performed at a much larger scale, on the level of whole genomes. While we have not investigated this further, it seems plausible that some of the ideas proposed for the exome sequencing pipeline and web interface can be transferred to that area. For example, the structure for meta data is likely very similar as also patients, samples, tissues, etc. need to be modeled. Computational pipelines for methylation studies are today also based on BAM files and would likely integrate well into the workflow due to the similarity to processing of exome data.

6.3 Discussion

The development of algorithms and tools for the analysis of high-throughput DNA sequencing data remains challenging. It needs to adjust quickly to new ways for conducting experiments and to innovations on the technological side. Different technologies result in different data and the algorithms building upon them need to take that into account. The work presented in this thesis is a good illustration of that point, with its algorithms specially designed for colorspace and flowgram data. Even if the technology yields data that is very close to our model of DNA as a simple string over a four-letter alphabet, as Illumina reads are, an understanding of low-level details of the full process is required. As an example, consider the case of PCR duplicates in the reads or sequence-specific errors that may lead to the detection of spurious mutations.

Developing algorithms for high-throughput sequencing is therefore a process that needs to be closely coupled to the underlying technology. For the foreseeable future, as the technology continues to improve, this will arguably remain that way. Developing HTS algorithms seems to be a typical situation in which methods of agile programming are required – in this case, an iterative cycle of programming, development of theoretical models, and user feedback. The cutadapt program may serve as an example of this: Over the course of its development and while it was already in use, the alignment algorithm was re-implemented multiple times until it converged to the version presented here. This is entirely acceptable since it is unrealistic to expect that all requirements are known in advance.

We do not only cover novel algorithms in this thesis, but have also implemented them within tools. Our opinion of how important this is can be found in the title of this work, which includes the word "tools". The area of research we are concerned with here is aimed at gaining biological or medical insights through computational methods. That is only possible if the algorithms can actually be used in order to test them and to apply them. Usability can come at different levels. The cutadapt software is usable by other computer scientists or those familiar with the command line, which is still the level at which much day-to-day bioinformatics work is conducted. The Exomate software, on the other hand, is an attempt to improve accessibility even further and therefore removing even more hurdles for research.

Overall, we believe to have reached our aim of improving algorithms on the bioinformatics side in order to help fellow researchers gain biological knowledge.

A Appendix

A.1 Software

The following software was created as part of this thesis. All tools are available under an Open Source license.

Cutadapt (Chapter 2)

The cutadapt software for adapter trimming is available at http://code.google.com/p/cutadapt/, while current source code is hosted at https://github.com/marcelm/cutadapt/. The work presented in Chapter 2 refers to the state of the software at commit `036d487b52`, which is a pre-release of cutadapt version 1.3.

Relevant source files are the following. File `cutadapt/calign.pyx` contains the Cython implementation of adapter alignment (Algorithm 1, page 27), which locates adapters through a variant of semiglobal alignment. File `cutadapt/adapters.py` contains definition of adapter types, trimming of $3'$ colorspace adapters (Algorithm 3, page 38) and $5'$ colorspace adapters (Algorithm 4, page 38). The main script is in `cutadapt/scripts/cutadapt.py` and includes read filtering and conversion routines listed in Section 2.5.1 on page 40.

Verjinxer (Chapter 3)

The Verjinxer software (versatile Java-based indexer) can be found at http://code.google.com/p/verjinxer/. The version of the software used for the paper by Zeschnigk et al. (2009) is available by checking out the tag named "methylationpaper" from version control. File `src/verjinxer/sequenceanalysis/BisulfiteQGramCoder.java` implements simulation of bisulfite treatment as in Algorithm 5, page 59. Creation of the q-gram index (Algorithm 6, page 60) is implemented in `src/verjinxer/QGramIndexer.java`. Within file `src/verjinxer/QgramMatcher.java`, finding maximal bisulfite matches (Algorithm 7, page 63) is implemented.

The B and P arrays containing the q-gram index are written by Verjinxer into files named `name.qbck` and `name.qpos`, respectively.

Exomate (Chapter 4)

The Exomate software is available at https://bitbucket.org/marcelm/exomate. The state of the software as described in Chapter 4 is that of April 2013. The repository includes also contributions from colleagues and student assistants.

The variant calling pipeline (Section 4.2) is implemented in `pipeline/Snakefile`.

Algorithm 12 (page 96) for indel normalization is in `exomate/variants.py` within the function `normalized_indel`. The database layout (Section 4.3) is defined in `exomate/models.py`.

Function `affected_mutations` in `exomate/views.py` implements the SQLalchemy query for interesting mutations (Algorithm 13, page 106). The same file also contains all quality-control queries (Section 4.4.5, page 106).

FlowG (Chapter 5)

A proof-of-concept implementation of flowgram-string alignment written in Python is available from http://www.rahmannlab.de/software. Scoring-related functions, in particular Equations (5.2), (5.3) and (5.4) are implemented in `flowg/scoring.py`. The dynamic programming implementation of the recurrence given in Equation (5.1) is found in `flowg/align.py`. Algorithm 14 is implemented in the same file. Emphasis is currently on readability and no attempt has been made to improve performance, except that diagonal edges in recurrence 5.1 are computed only up to a fixed number of cells to the left.

A.2 Contributions to co-authored articles

This thesis contains results that were obtained in cooperation with other researchers. My contributions to multi-author publications are described in the following. As advisor, Sven Rahmann assisted in all stages of research.

I implemented the exome-analysis pipeline described in Chapter 4 and performed all exome-sequencing-related bioinformatics analyses for the following two articles:

> Czeschik, Voigt, Alanay, Albrecht, Avci, FitzPatrick, Goudie, Hehr, Hoogeboom, Kayserili, Simsek-Kiper, Klein-Hitpass, Kuechler, López-González, Martin, Rahmann, Schweiger, Splitt, Wollnik, Lüdecke, Zeschnigk, and Wieczorek (2013). *Clinical and mutation data in 12 patients with the clinical diagnosis of Nager syndrome.*

> Martin, Maßhöfer, Temming, Rahmann, Metz, Bornfeld, van de Nes, Klein-Hitpass, Hinnebusch, Horsthemke, Lohmann, and Zeschnigk (2013). *Exome capture identifies recurrent somatic mutations in EIF1AX and SF3B1 anti-correlated in uveal melanoma with disomy 3.*

I performed adapter removal (Chapter 2), bisulfite read mapping (Chapter 3) and subsequent bioinformatics analyses:

> Zeschnigk, Martin, Betzl, Kalbe, Sirsch, Buiting, Gross, Fritzilas, Frey, Rahmann, and Horsthemke (2009). *Massive parallel bisulfite sequencing of CG-rich DNA fragments reveals that methylation of many X-chromosomal CpG islands in female blood DNA is incomplete.*

The bioinformatics analyses in the following articles were shared equally between Sven Rahmann, Tobias Marschall and me. I extended the adapter removal software (Chapter 2) to work with colorspace:

> Schulte, Marschall, Martin, Rosenstiel, Mestdagh, Schlierf, Thor, Vandesompele, Eggert, Schreiber, Rahmann, and Schramm (2010). *Deep sequencing reveals differential expression of microRNAs in favorable versus unfavorable neuroblastoma.*

> Rahmann, Martin, Schulte, Köster, Marschall, and Schramm (2013). *Identifying transcriptional miRNA biomarkers by integrating high-throughput sequencing and real-time PCR data.*

The following paper contains an abridged version of Chapter 5. Major contributions by my co-author are found in Sections 5.3.3 (Scoring of flow intensities) and 5.3.4 (Parameters for editing events). Also, restructuring and rewording of the text was done by my co-author for the article and incorporated back into the chapter.

> Martin and Rahmann (2013). *Aligning Flowgrams to DNA Sequences.*

Bibliography

I. A. Adzhubei, S. Schmidt, L. Peshkin, V. E. Ramensky, A. Gerasimova, P. Bork, A. S. Kondrashov, and S. R. Sunyaev. A method and server for predicting damaging missense mutations. *Nature Methods*, 7(4):248–249, Apr. 2010. doi:10.1038/nmeth0410-248. (Page 86.)

A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975. doi:10.1145/360825.360855. (Page 45.)

B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 5th edition, 2008. (Pages 2, 50 and 104.)

C. Alkan, B. P. Coe, and E. E. Eichler. Genome structural variation discovery and genotyping. *Nature Reviews Genetics*, 12(5):363–376, May 2011. doi:10.1038/nrg2958. (Pages 2 and 112.)

M. Allhoff, A. Schonhuth, M. Martin, I. Costa, S. Rahmann, and T. Marschall. Discovering motifs that induce sequencing errors. *BMC Bioinformatics*, 14(Suppl 5):S1, 2013. doi:10.1186/1471-2105-14-S5-S1. (Page 105.)

S. F. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, Oct. 1990. (Pages 55 and 62.)

S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997. (Page 55.)

A. Ameur, A. Wetterbom, L. Feuk, and U. Gyllensten. Global and unbiased detection of splice junctions from RNA-seq data. *Genome Biology*, 11(3):R34, 2010. doi:10.1186/gb-2010-11-3-r34. (Page 48.)

S. E. Antonarakis, M. Krawczak, and D. N. Cooper. The nature and mechanisms of human gene mutation. In C. Scriver, A. Beaudet, W. Sly, and D. Valle, editors, *The Metabolic and Molecular Bases of Inherited Disease*, chapter 13, pages 259–291. McGraw-Hill, New York, 7th edition, 1995. (Page 84.)

Asan, Y. Xu, H. Jiang, C. Tyler-Smith, Y. Xue, T. Jiang, J. Wang, M. Wu, X. Liu, G. Tian, J. Wang, J. Wang, H. Yang, and X. Zhang. Comprehensive comparison of three commercial human whole-exome capture platforms. *Genome Biology*, 12(9):R95, 2011. doi:10.1186/gb-2011-12-9-r95. (Page 85.)

M. Baker. Sorting out sequencing data. *Nature Methods*, 8(10):799–803, 2011. doi:10.1038/nmeth.1702. (Page 84.)

S. Balzer, K. Malde, A. Lanzén, A. Sharma, and I. Jonassen. Characteristics of 454 pyrosequencing data–enabling realistic simulation with flowsim. *Bioinformatics*, 26(18):i420–i425, Sept. 2010. doi:10.1093/bioinformatics/btq365. (Pages 118 and 130.)

M. J. Bamshad, S. B. Ng, A. W. Bigham, H. K. Tabor, M. J. Emond, D. A. Nickerson, and J. Shendure. Exome sequencing as a tool for Mendelian disease gene discovery. *Nature Reviews Genetics*, 12(11):745–755, Nov. 2011. doi:10.1038/nrg3031. (Pages 2 and 84.)

D. R. Bentley, S. Balasubramanian, H. P. Swerdlow, G. P. Smith, J. Milton, C. G. Brown, K. P. Hall, D. J. Evers, C. L. Barnes, H. R. Bignell, J. M. Boutell, J. Bryant, R. J. Carter, R. Keira Cheetham, A. J. Cox, D. J. Ellis, M. R. Flatbush, N. A. Gormley, S. J. Humphray, L. J. Irving, M. S. Karbelashvili, S. M. Kirk, H. Li, X. Liu, K. S. Maisinger, L. J. Murray, B. Obradovic, T. Ost, M. L. Parkinson, M. R. Pratt, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, Nov. 2008. doi:10.1038/nature07517. (Page 4.)

F. P. Bernier, O. Caluseriu, S. Ng, J. Schwartzentruber, K. J. Buckingham, A. M. Innes, E. W. Jabs, J. W. Innis, J. L. Schuette, J. L. Gorski, P. H. Byers, G. Andelfinger, V. Siu, J. Lauzon, B. A. Fernandez, M. McMillin, R. H. Scott, H. Racher, FORGE Canada Consortium, J. Majewski, D. A. Nickerson, J. Shendure, M. J. Bamshad, and J. S. Parboosingh. Haploinsufficiency of SF3B4, a component of the pre-mRNA spliceosomal complex, causes Nager syndrome. *American Journal of Human Genetics*, 90(5):925–933, May 2012. doi:10.1016/j.ajhg.2012.04.004. (Page 109.)

G. R. Bignell, J. Huang, J. Greshock, S. Watt, A. Butler, S. West, M. Grigorova, K. W. Jones, W. Wei, M. R. Stratton, P. A. Futreal, B. Weber, M. H. Shapero, and R. Wooster. High-resolution analysis of DNA copy number using oligonucleotide microarrays. *Genome Research*, 14(2):287–295, Feb. 2004. doi:10.1101/gr.2012304. (Page 107.)

A. Bird, M. Taggart, M. Frommer, O. J. Miller, and D. Macleod. A fraction of the mouse genome that is derived from islands of nonmethylated, CpG-rich DNA. *Cell*, 40(1):91–99, Jan. 1985. (Page 50.)

C. Bock, J. Walter, M. Paulsen, and T. Lengauer. CpG island mapping by epigenome prediction. *PLoS Computational Biology*, 3(6):e110, June 2007. doi:10.1371/journal.pcbi.0030110. (Page 51.)

H. Breu. A theoretical understanding of 2 base color codes and its application to annotation, error detection, and error correction. Technical report, Applied Biosystems by Life Technologies Corporation, 2010. White Paper. (Pages 5 and 35.)

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Systems Research Center, digital, Palo Alto, California, 1994. (Pages 57 and 82.)

S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval – Implementing and Evaluating Search Engines*. MIT Press, 2010. (Page 75.)

D. Challis, J. Yu, U. S. Evani, A. R. Jackson, S. Paithankar, C. Coarfa, A. Milosavljevic, R. A. Gibbs, and F. Yu. An integrative variant analysis suite for whole exome next-generation sequencing data. *BMC Bioinformatics*, 13:8, 2012. doi:10.1186/1471-2105-13-8. (Pages 86 and 106.)

A. Chatterjee, P. A. Stockwell, E. J. Rodger, and I. M. Morison. Comparison of alignment software for genome-wide bisulphite sequence data. *Nucleic Acids Research*, Feb. 2012. doi:10.1093/nar/gks150. (Page 57.)

P.-Y. Chen, S. J. Cokus, and M. Pellegrini. BS Seeker: precise mapping for bisulfite sequencing. *BMC Bioinformatics*, 11:203, 2010. doi:10.1186/1471-2105-11-203. (Pages 57 and 58.)

D. M. Church, V. A. Schneider, T. Graves, K. Auger, F. Cunningham, N. Bouk, H.-C. Chen, R. Agarwala, W. M. McLaren, G. R. S. Ritchie, D. Albracht, M. Kremitzki, S. Rock, H. Kotkiewicz, C. Kremitzki, A. Wollam, L. Trani, L. Fulton, R. Fulton, L. Matthews, S. Whitehead, W. Chow, J. Torrance, M. Dunn, G. Harden, G. Threadgold, J. Wood, J. Collins, P. Heath, G. Griffiths, et al. Modernizing reference genome assemblies. *PLoS Biology*, 9(7):e1001091, July 2011. doi:10.1371/journal.pbio.1001091. (Pages 50, 76 and 85.)

J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature Nanotechnology*, 4(4):265–270, Apr. 2009. doi:10.1038/nnano.2009.12. (Page 3.)

P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, Apr. 2010. doi:10.1093/nar/gkp1137. (Page 7.)

E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971. (Page 97.)

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, Mar. 2008. doi:10.1038/nature06745. (Pages 51 and 56.)

D. N. Cooper, M. H. Taggart, and A. P. Bird. Unmethylated domains in vertebrate DNA. *Nucleic Acids Research*, 11(3):647–658, Feb. 1983. (Page 50.)

J. C. Czeschik, C. Voigt, Y. Alanay, B. Albrecht, S. Avci, D. FitzPatrick, D. R. Goudie, U. Hehr, A. J. Hoogeboom, H. Kayserili, P. O. Simsek-Kiper, L. Klein-Hitpass, A. Kuechler, V. López-González, M. Martin, S. Rahmann, B. Schweiger, M. Splitt, B. Wollnik, H.-J. Lüdecke, M. Zeschnigk, and D. Wieczorek. Clinical and mutation data in 12 patients with the clinical diagnosis of Nager syndrome. *Human Genetics*, 2013. doi:10.1007/s00439-013-1295-2. (Pages 83, 109, 134 and 139.)

M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. *A model of evolutionary change in proteins*, volume 5, pages 345–352. National Biomedical Research Foundation, 1978. (Pages 126, 127 and 131.)

M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. D. Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature Genetics*, 43(5):491–498, May 2011. doi:10.1038/ng.806. (Pages 86, 91, 94 and 95.)

D. Earl, K. Bradnam, J. St John, A. Darling, D. Lin, J. Fass, H. O. K. Yu, V. Buffalo, D. R. Zerbino, M. Diekhans, N. Nguyen, P. N. Ariyaratne, W.-K. Sung, Z. Ning, M. Haimel, J. T. Simpson, N. A. Fonseca, I. Birol, T. R. Docking, I. Y. Ho, D. S. Rokhsar, R. Chikhi, D. Lavenier, G. Chapuis, D. Naquin, N. Maillet, M. C. Schatz, D. R. Kelley, A. M. Phillippy, S. Koren, et al. Assemblathon 1: a competitive assessment of de novo short read assembly methods. *Genome Research*, 21 (12):2224–2241, Dec. 2011. doi:10.1101/gr.126599.111. (Page 2.)

J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, A. Bibillo, K. Bjornson, B. Chaudhuri, F. Christians, R. Cicero, S. Clark, R. Dalal, A. Dewinter, J. Dixon, M. Foquet, A. Gaertner, P. Hardenbol, C. Heiner, K. Hester, D. Holden, G. Kearns, X. Kong, R. Kuse, Y. Lacroix, S. Lin, et al. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, Jan 2009. doi:10.1126/science.1162986. (Page 3.)

P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar. 1975. doi:10.1109/TIT.1975.1055349. (Page 75.)

B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Research*, 8(3):186–194, Mar 1998. (Page 6.)

P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, Washington, DC, USA, 2000. IEEE Computer Society. (Page 82.)

J. W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1 Pt 1):175–179, Jan. 1984. (Page 45.)

P. Flicek, I. Ahmed, M. R. Amode, D. Barrell, K. Beal, S. Brent, D. Carvalho-Silva, P. Clapham, G. Coates, S. Fairley, S. Fitzgerald, L. Gil, C. García-Girón, L. Gordon, T. Hourlier, S. Hunt, T. Juettemann, A. K. Kähäri, S. Keenan, M. Komorowska, E. Kulesha, I. Longden, T. Maurel, W. M. McLaren, M. Muffato, R. Nag, B. Overduin, M. Pignatelli, B. Pritchard, E. Pritchard, et al. Ensembl 2013. *Nucleic Acids Research*, 41(Database issue):D48–D55, Jan 2013. doi:10.1093/nar/gks1236. (Page 92.)

M. Frommer, L. E. McDonald, D. S. Millar, C. M. Collis, F. Watt, G. W. Grigg, P. L. Molloy, and C. L. Paul. A genomic sequencing protocol that yields a positive display of 5-methylcytosine residues in individual DNA strands. *Proceedings of the National Academy of Sciences of the United States of America*, 89(5):1827–1831, Mar. 1992. (Page 51.)

M. Gardiner-Garden and M. Frommer. CpG islands in vertebrate genomes. *Journal of Molecular Biology*, 196(2):261–282, July 1987. (Pages 50 and 51.)

C. Gilissen, A. Hoischen, H. G. Brunner, and J. A. Veltman. Unlocking Mendelian disease using exome sequencing. *Genome Biology*, 12(9):228, 2011. doi:10.1186/gb-2011-12-9-228. (Page 84.)

C. Gilissen, A. Hoischen, H. G. Brunner, and J. A. Veltman. Disease gene identification strategies for exome sequencing. *European Journal of Human Genetics*, 20(5):490–497, May 2012. doi:10.1038/ejhg.2011.258. (Page 84.)

D. Golan and P. Medvedev. Using state machines to model the Ion Torrent sequencing process and to improve read error rates. *Bioinformatics*, 29(13):i344–i351, July 2013. doi:10.1093/bioinformatics/btt212. (Pages 117 and 120.)

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. (Pages 71, 72 and 73.)

D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, May 1997. (Pages 15, 16, 18 and 19.)

F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7(8):576–577, Aug. 2010. doi:10.1038/nmeth0810-576. (Page 57.)

M. Hafner, P. Landgraf, J. Ludwig, A. Rice, T. Ojo, C. Lin, D. Holoch, C. Lim, and T. Tuschl. Identification of microRNAs and other small regulatory RNAs using cDNA library sequencing. *Methods*, 44(1):3–12, 2008. doi:10.1016/j.ymeth.2007.09.009. (Page 11.)

J. W. Harbour, E. D. O. Roberson, H. Anbunathan, M. D. Onken, L. A. Worley, and A. M. Bowcock. Recurrent mutations at codon 625 of the splicing factor SF3B1 in uveal melanoma. *Nature Genetics*, 45(2):133–135, Feb. 2013. doi:10.1038/ng.2523. (Pages 43, 110, 111, 113 and 134.)

E. Y. Harris, N. Ponts, A. Levchuk, K. L. Roch, and S. Lonardi. BRAT: bisulfite-treated reads analysis tool. *Bioinformatics*, 26(4):572–573, Feb. 2010. doi:10.1093/bioinformatics/btp706. (Page 57.)

E. Y. Harris, N. Ponts, K. G. Le Roch, and S. Lonardi. BRAT-BW: efficient and accurate mapping of bisulfite-treated reads. *Bioinformatics*, 28(13):1795–1796, Jul 2012. doi:10.1093/bioinformatics/bts264. (Page 57.)

D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975. doi:10.1145/360825.360861. (Pages 17 and 26.)

A. Hoischen, B. W. M. van Bon, C. Gilissen, P. Arts, B. van Lier, M. Steehouwer, P. de Vries, R. de Reuver, N. Wieskamp, G. Mortier, K. Devriendt, M. Z. Amorim, N. Revencu, A. Kidd, M. Barbosa, A. Turner, J. Smith, C. Oley, A. Henderson, I. M. Hayes, E. M. Thompson, H. G. Brunner, B. B. A. de Vries, and J. A. Veltman. De novo mutations of SETBP1 cause Schinzel-Giedion syndrome. *Nature Genetics*, 42(6):483–485, June 2010. doi:10.1038/ng.581. (Page 84.)

N. Homer and S. F. Nelson. Improved variant discovery through local re-alignment of short-read next-generation sequencing data using SRMA. *Genome Biology*, 11(10):R99, 2010. doi:10.1186/gb-2010-11-10-r99. (Page 89.)

N. Homer, B. Merriman, and S. F. Nelson. BFAST: an alignment tool for large scale genome resequencing. *PLoS One*, 4(11):e7767, 2009. doi:10.1371/journal.pone.0007767. (Pages 36, 55 and 62.)

Intel. *Intel®64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, Mar. 2012. Order number 325383. (Page 67.)

International HapMap 3 Consortium, D. M. Altshuler, R. A. Gibbs, L. Peltonen, D. M. Altshuler, R. A. Gibbs, L. Peltonen, E. Dermitzakis, S. F. Schaffner, F. Yu, L. Peltonen, E. Dermitzakis, P. E. Bonnen, D. M. Altshuler, R. A. Gibbs, P. I. W. de Bakker, P. Deloukas, S. B. Gabriel, R. Gwilliam, S. Hunt, M. Inouye, X. Jia, A. Palotie, M. Parkin, P. Whittaker, F. Yu, K. Chang, A. Hawes, L. R. Lewis, Y. Ren, et al. Integrating common and rare genetic variation in diverse human populations. *Nature*, 467(7311):52–58, Sept. 2010. doi:10.1038/nature09298. (Pages 86 and 100.)

S. Jünemann, K. Prior, R. Szczepanowski, I. Harks, B. Ehmke, A. Goesmann, J. Stoye, and D. Harmsen. Bacterial community shift in treated periodontitis patients revealed by Ion Torrent 16S rRNA gene amplicon sequencing. *PLoS One*, 7(8):e41606, 2012. doi:10.1371/journal.pone.0041606. (Page 40.)

W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, Apr. 2002. doi:10.1101/gr.229202. (Page 80.)

Y. Kodama, M. Shumway, R. Leinonen, and International Nucleotide Sequence Database Collaboration. The sequence read archive: explosive growth of sequencing data. *Nucleic Acids Research*, 40(Database issue):D54–D56, Jan. 2012. (Page 41.)

J. Köster and S. Rahmann. Snakemake–a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, Oct. 2012. doi:10.1093/bioinformatics/bts480. (Page 87.)

G. M. Landau, U. Vishkin, and R. Nussinov. An efficient string matching algorithm with *k* differences for nucleotide and amino acid sequences. *Nucleic Acids Research*, 14(1):31–46, Jan. 1986. (Page 23.)

E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, R. Funke, D. Gage, K. Harris, A. Heaford, J. Howland, L. Kann, J. Lehoczky, R. LeVine, P. McEwan, K. McKernan, J. Meldrim, J. P. Mesirov, C. Miranda, W. Morris, J. Naylor, C. Raymond, M. Rosetti, R. Santos, A. Sheridan, C. Sougnez, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, Feb 2001. (Page 1.)

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. doi:10.1186/gb-2009-10-3-r25. (Page 57.)

F. Larsen, G. Gundersen, R. Lopez, and H. Prydz. CpG islands as gene markers in the human genome. *Genomics*, 13(4):1095–1107, Aug. 1992. (Page 50.)

D. E. Larson, C. C. Harris, K. Chen, D. C. Koboldt, T. E. Abbott, D. J. Dooling, T. J. Ley, E. R. Mardis, R. K. Wilson, and L. Ding. SomaticSniper: identification of somatic point mutations in whole genome sequencing data. *Bioinformatics*, 28(3):311–317, 2012. doi:10.1093/bioinformatics/btr665. (Page 110.)

R. Leinonen, H. Sugawara, M. Shumway, and International Nucleotide Sequence Database Collaboration. The sequence read archive. *Nucleic Acids Research*, 39(Database issue):D19–D21, Jan. 2011. (Page 41.)

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, Feb. 1966. (Page 16.)

H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009. doi:10.1093/bioinformatics/btp324. (Pages 14, 29, 36, 41, 62 and 89.)

H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, Mar. 2010. doi:10.1093/bioinformatics/btp698. (Page 48.)

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, Nov. 2008a. doi:10.1101/gr.078212.108. (Pages 14, 36, 41, 55, 56, 62 and 80.)

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 format and SAMtools. *Bioinformatics*, 25(16):2078–2079, Aug. 2009. (Pages 7, 48, 86, 89 and 97.)

R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, Mar. 2008b. doi:10.1093/bioinformatics/btn025. (Pages 14 and 56.)

R. Lister and J. R. Ecker. Finding the fifth base: genome-wide sequencing of cytosine methylation. *Genome Research*, 19(6):959–966, June 2009. doi:10.1101/gr.083451.108. (Page 51.)

R. Lister, R. C. O'Malley, J. Tonti-Filippini, B. D. Gregory, C. C. Berry, A. H. Millar, and J. R. Ecker. Highly integrated single-base resolution maps of the epigenome in Arabidopsis. *Cell*, 133(3):523–536, May 2008. doi:10.1016/j.cell.2008.03.029. (Pages 51 and 56.)

R. Lister, M. Pelizzola, R. H. Dowen, R. D. Hawkins, G. Hon, J. Tonti-Filippini, J. R. Nery, L. Lee, Z. Ye, Q.-M. Ngo, L. Edsall, J. Antosiewicz-Bourget, R. Stewart, V. Ruotti, A. H. Millar, J. A. Thomson, B. Ren, and J. R. Ecker. Human DNA methylomes at base resolution show widespread epigenomic differences. *Nature*, 462(7271):315–322, Nov. 2009. doi:10.1038/nature08514. (Page 52.)

F. Lysholm, B. Andersson, and B. Persson. FAAST: Flow-space assisted alignment search tool. *BMC Bioinformatics*, 12:293, 2011. doi:10.1186/1471-2105-12-293. (Page 119.)

B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, Mar. 2002. (Page 55.)

T. Magoč and S. L. Salzberg. FLASH: fast length adjustment of short reads to improve genome assemblies. *Bioinformatics*, 27(21):2957–2963, Nov. 2011. doi:10.1093/bioinformatics/btr507. (Page 46.)

M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader, L. A. Bemben, J. Berka, M. S. Braverman, Y.-J. Chen, Z. Chen, S. B. Dewell, L. Du, J. M. Fierro, X. V. Gomes, B. C. Godwin, W. He, S. Helgesen, C. H. Ho, C. H. Ho, G. P. Irzyk, S. C. Jando, M. L. I. Alenquer, T. P. Jarvie, K. B. Jirage, J.-B. Kim, J. R. Knight, J. R. Lanza, J. H. Leamon, S. M. Lefkowitz, M. Lei, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057): 376–380, Sept. 2005. doi:10.1038/nature03959. (Pages 4, 118 and 119.)

T. Marschall, I. G. Costa, S. Canzar, M. Bauer, G. W. Klau, A. Schliep, and A. Schönhuth. CLEVER: clique-enumerating variant finder. *Bioinformatics*, 28(22):2875–2882, Nov. 2012. doi:10.1093/bioinformatics/bts566. (Pages 6 and 112.)

M. Martin. Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet.journal*, 17(1):10–12, 2011. (Page 40.)

M. Martin and S. Rahmann. Aligning flowgrams to DNA sequences. In T. Beißbarth, M. Kollmar, A. Leha, B. Morgenstern, A.-K. Schultz, S. Waack, and E. Wingender, editors, *German Conference on Bioinformatics 2013*, volume 34 of *OpenAccess Series in Informatics (OASIcs)*, pages 125–135, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.GCB.2013.125. (Page 139.)

M. Martin, L. Maßhöfer, P. Temming, S. Rahmann, C. Metz, N. Bornfeld, J. van de Nes, L. Klein-Hitpass, A. G. Hinnebusch, B. Horsthemke, D. R. Lohmann, and M. Zeschnigk. Exome sequencing identifies recurrent somatic mutations in EIF1AX and SF3B1 in uveal melanoma with disomy 3. *Nature Genetics*, 45(8):933–936, Aug. 2013. doi:10.1038/ng.2674. (Pages 2, 83, 109, 134 and 139.)

A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20 (9):1297–1303, Sept. 2010. doi:10.1101/gr.107524.110. (Pages 89 and 91.)

A. Meissner, T. S. Mikkelsen, H. Gu, M. Wernig, J. Hanna, A. Sivachenko, X. Zhang, B. E. Bernstein, C. Nusbaum, D. B. Jaffe, A. Gnirke, R. Jaenisch, and E. S. Lander. Genome-scale DNA methylation maps of pluripotent and differentiated cells. *Nature*, 454(7205):766–770, Aug. 2008. doi:10.1038/nature07107. (Page 51.)

B. Merriman, Ion Torrent R&D Team, and J. M. Rothberg. Progress in ion torrent semiconductor chip based sequencing. *Electrophoresis*, 33(23):3397–3417, Dec. 2012. doi:10.1002/elps.201200424. (Page 4.)

T. Müller, S. Rahmann, and M. Rehmsmeier. Non-symmetric score matrices and the detection of homologous transmembrane proteins. *Bioinformatics*, 17(Suppl. 1):S182–S189, 2001. (Page 127.)

G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46:395–415, May 1999. doi:10.1145/316542.316550. (Pages 27 and 45.)

K. Nakamura, T. Oshima, T. Morimoto, S. Ikeda, H. Yoshikawa, Y. Shiwa, S. Ishikawa, M. C. Linak, A. Hirai, H. Takahashi, M. Altaf-Ul-Amin, N. Ogasawara, and S. Kanaya. Sequence-specific error profile of Illumina sequencers. *Nucleic Acids Research*, 39(13):e90, July 2011. doi:10.1093/nar/gkr344. (Page 105.)

G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, New York, NY, USA, 2002. (Page 45.)

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, Mar. 1970. (Page 119.)

P. C. Ng and S. Henikoff. Predicting deleterious amino acid substitutions. *Genome Research*, 11 (5):863–874, May 2001. doi:10.1101/gr.176601. (Page 102.)

P. C. Ng and S. Henikoff. SIFT: Predicting amino acid changes that affect protein function. *Nucleic Acids Research*, 31(13):3812–3814, July 2003. doi:10.1093/nar/gkg509. (Page 102.)

S. B. Ng, E. H. Turner, P. D. Robertson, S. D. Flygare, A. W. Bigham, C. Lee, T. Shaffer, M. Wong, A. Bhattacharjee, E. E. Eichler, M. Bamshad, D. A. Nickerson, and J. Shendure. Targeted capture and massively parallel sequencing of 12 human exomes. *Nature*, 461(7261):272–276, Sept. 2009. doi:10.1038/nature08250. (Pages 84, 103 and 105.)

S. B. Ng, K. J. Buckingham, C. Lee, A. W. Bigham, H. K. Tabor, K. M. Dent, C. D. Huff, P. T. Shannon, E. W. Jabs, D. A. Nickerson, J. Shendure, and M. J. Bamshad. Exome sequencing identifies the cause of a mendelian disorder. *Nature Genetics*, 42(1):30–35, Jan. 2010. doi:10.1038/ng.499. (Pages 84 and 86.)

W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8):2444–2448, Apr. 1988. (Page 7.)

M. F. Polz and C. M. Cavanaugh. Bias in template-to-product ratios in multitemplate PCR. *Applied and Environmental Microbiology*, 64(10):3724–3730, Oct. 1998. (Page 90.)

G. Prescher, N. Bornfeld, H. Hirche, B. Horsthemke, K. H. Jöckel, and R. Becher. Prognostic implications of monosomy 3 in uveal melanoma. *Lancet*, 347(9010):1222–1225, May 1996. (Page 110.)

C. Quince, A. Lanzén, T. P. Curtis, R. J. Davenport, N. Hall, I. M. Head, L. F. Read, and W. T. Sloan. Accurate determination of microbial diversity from 454 pyrosequencing data. *Nature Methods*, 6(9):639–641, Sept. 2009. doi:10.1038/nmeth.1361. (Pages 119 and 120.)

A. R. Quinlan and I. M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, Mar. 2010. doi:10.1093/bioinformatics/btq033. (Page 92.)

S. Rahmann, M. Martin, J. H. Schulte, J. Köster, T. Marschall, and A. Schramm. Identifying transcriptional miRNA biomarkers by integrating high-throughput sequencing and real-time

PCR data. *Methods*, 59(1):154–163, Jan. 2013. doi:10.1016/j.ymeth.2012.10.005. (Pages 2, 31, 32, 43 and 139.)

K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all epsilon-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, Mar. 2006. doi:10.1089/cmb.2006.13.296. (Pages 23, 55 and 81.)

P. Rice, I. Longden, and A. Bleasby. EMBOSS: the European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, June 2000. (Page 14.)

G. Rizk and D. Lavenier. GASSST: Global alignment short sequence search tool. *Bioinformatics*, Aug. 2010. doi:10.1093/bioinformatics/btq485. (Page 56.)

M. Ronaghi, S. Karamohamed, B. Pettersson, M. Uhlén, and P. Nyrén. Real-time DNA sequencing using detection of pyrophosphate release. *Analytical Biochemistry*, 242(1):84–89, Nov. 1996. doi:10.1006/abio.1996.0432. (Page 4.)

F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the United States of America*, 74(12):5463–5467, Dec. 1977. (Page 1.)

G. W. E. Santen, E. Aten, Y. Sun, R. Almomani, C. Gilissen, M. Nielsen, S. G. Kant, I. N. Snoeck, E. A. J. Peeters, Y. Hilhorst-Hofstee, M. W. Wessels, N. S. den Hollander, C. A. L. Ruivenkamp, G.-J. B. van Ommen, M. H. Breuning, J. T. den Dunnen, A. van Haeringen, and M. Kriek. Mutations in SWI/SNF chromatin remodeling complex gene ARID1B cause Coffin-Siris syndrome. *Nature Genetics*, 44(4):379–380, 2012. doi:10.1038/ng.2217. (Pages 84, 109 and 134.)

S. Scherer. *A Short Guide to the Human Genome*. Cold Spring Harbor Laboratory Press, 2008. (Pages 50 and 112.)

J. Schiemer. Illumina TruSeq DNA adapters de-mystified, 2011. Tufts University Core Facility. (Page 11.)

A. Schramm, B. Schowe, K. Fielitz, M. Heilmann, M. Martin, T. Marschall, J. Köster, J. Vandesompele, J. Vermeulen, K. de Preter, J. Koster, R. Versteeg, R. Noguera, F. Speleman, S. Rahmann, A. Eggert, K. Morik, and J. H. Schulte. Exon-level expression analyses identify MYCN and NTRK1 as major determinants of alternative exon usage and robustly predict primary neuroblastoma outcome. *British Journal of Cancer*, 107(8):1409–1417, Oct. 2012. doi:10.1038/bjc.2012.391. (Page 2.)

J. H. Schulte, T. Marschall, M. Martin, P. Rosenstiel, P. Mestdagh, S. Schlierf, T. Thor, J. Vandesompele, A. Eggert, S. Schreiber, S. Rahmann, and A. Schramm. Deep sequencing reveals differential expression of microRNAs in favorable versus unfavorable neuroblastoma. *Nucleic Acids Research*, 38(17):5919–5928, Sept. 2010. doi:10.1093/nar/gkq342. (Pages 2, 15, 43 and 139.)

P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980. (Page 16.)

S. T. Sherry, M. H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin. dbSNP: the NCBI database of genetic variation. *Nucleic Acids Research*, 29(1):308–311, Jan. 2001. (Page 85.)

A. D. Smith, Z. Xuan, and M. Q. Zhang. Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, 9:128, 2008. doi:10.1186/1471-2105-9-128. (Page 57.)

A. D. Smith, W.-Y. Chung, E. Hodges, J. Kendall, G. Hannon, J. Hicks, Z. Xuan, and M. Q. Zhang. Updates to the RMAP short-read mapping software. *Bioinformatics*, 25(21):2841–2842, Nov. 2009. doi:10.1093/bioinformatics/btp533. (Page 57.)

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, Mar. 1981. (Pages 18 and 119.)

D. Takai and P. A. Jones. Comprehensive analysis of CpG islands in human chromosomes 21 and 22. *Proceedings of the National Academy of Sciences of the United States of America*, 99(6): 3740–3745, Mar. 2002. doi:10.1073/pnas.052410099. (Page 51.)

The 1000 Genomes Project Consortium, R. M. Durbin, G. R. Abecasis, D. L. Altshuler, A. Auton, L. D. Brooks, R. M. Durbin, R. A. Gibbs, M. E. Hurles, and G. A. McVean. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, Oct. 2010. doi:10.1038/nature09534. (Pages 94 and 112.)

H. Thorvaldsdóttir, J. T. Robinson, and J. P. Mesirov. Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in Bioinformatics*, Apr. 2012. doi:10.1093/bib/bbs017. (Pages 90 and 91.)

F. Tschentscher, J. Hüsing, T. Hölter, E. Kruse, I. G. Dresen, K.-H. Jöckel, G. Anastassiou, H. Schilling, N. Bornfeld, B. Horsthemke, D. R. Lohmann, and M. Zeschnigk. Tumor classification based on gene expression profiling shows that uveal melanomas with and without monosomy 3 represent two distinct entities. *Cancer Research*, 63(10):2578–2584, May 2003. (Page 109.)

Y. Tsurusaki, N. Okamoto, H. Ohashi, T. Kosho, Y. Imai, Y. Hibi-Ko, T. Kaname, K. Naritomi, H. Kawame, K. Wakui, Y. Fukushima, T. Homma, M. Kato, Y. Hiraki, T. Yamagata, S. Yano, S. Mizuno, S. Sakazume, T. Ishii, T. Nagai, M. Shiina, K. Ogata, T. Ohta, N. Niikawa, S. Miyatake, I. Okada, T. Mizuguchi, H. Doi, H. Saitsu, N. Miyake, et al. Mutations affecting components of the SWI/SNF complex cause Coffin-Siris syndrome. *Nature Genetics*, 44(4):376–378, Apr. 2012. doi:10.1038/ng.2219. (Page 109.)

E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132–137, 1985. (Pages 26 and 28.)

V. Vacic, H. Jin, J.-K. Zhu, and S. Lonardi. A probabilistic method for small RNA flowgram matching. *Pacific Symposium on Biocomputing*, pages 75–86, 2008. (Pages 118, 120, 121, 126 and 131.)

H. van Bakel, J. M. Stout, A. G. Cote, C. M. Tallon, A. G. Sharpe, T. R. Hughes, and J. E. Page. The draft genome and transcriptome of Cannabis sativa. *Genome Biology*, 12(10):R102, 2011. doi:10.1186/gb-2011-12-10-r102. (Page 40.)

C. Vesely, S. Tauber, F. J. Sedlazeck, A. von Haeseler, and M. F. Jantsch. Adenosine deaminases that act on RNA induce reproducible changes in abundance and sequence of embryonic miRNAs. *Genome Research*, 22(8):1468–1476, Aug. 2012. doi:10.1101/gr.133025.111. (Page 40.)

C. Voigt, A. Mégarbané, K. Neveling, J. C. Czeschik, B. Albrecht, B. Callewaert, F. von Deimling, A. Hehr, M. F. Smeland, R. König, A. Kuechler, C. Marcelis, M. Puiu, W. Reardon, H. M. F. R. Stensland, B. Schweiger, M. Steehouwer, C. Teller, M. Martin, S. Rahmann, U. Hehr, H. G. Brunner, H.-J. Lüdecke, and D. Wieczorek. Oto-facial syndrome and esophageal atresia, intellectual disability and zygomatic anomalies – expanding the phenotypes associated with EFTUD2 mutations. *Orphanet Journal of Rare Diseases*, 8(110), 2013. doi:10.1186/1750-1172-8-110. (Pages 83 and 134.)

M. Weber, J. J. Davies, D. Wittig, E. J. Oakeley, M. Haase, W. L. Lam, and D. Schübeler. Chromosome-wide and promoter-specific analyses identify sites of differential DNA methylation in normal and transformed human cells. *Nature Genetics*, 37(8):853–862, Aug. 2005. doi:10.1038/ng1598. (Page 51.)

D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert. RazerS–fast read mapping with sensitivity control. *Genome Research*, 19(9):1646–1654, Sept. 2009. doi:10.1101/gr.088823.108. (Pages 55 and 81.)

S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10): 83–91, 1992. (Page 45.)

Y. Xi and W. Li. BSMAP: whole genome bisulfite sequence MAPping program. *BMC Bioinformatics*, 10:232, 2009. doi:10.1186/1471-2105-10-232. (Pages 56 and 66.)

Y. Xi, C. Bock, F. Müller, D. Sun, A. Meissner, and W. Li. RRBSMAP: a fast, accurate and user-friendly alignment tool for reduced representation bisulfite sequencing. *Bioinformatics*, 28(3): 430–432, Feb. 2012. doi:10.1093/bioinformatics/btr668. (Page 57.)

K. Ye, M. H. Schulz, Q. Long, R. Apweiler, and Z. Ning. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871, Nov. 2009. doi:10.1093/bioinformatics/btp394. (Page 112.)

M. Zeschnigk, M. Martin, G. Betzl, A. Kalbe, C. Sirsch, K. Buiting, S. Gross, E. Fritzilas, B. Frey, S. Rahmann, and B. Horsthemke. Massive parallel bisulfite sequencing of CG-rich DNA fragments reveals that methylation of many X-chromosomal CpG islands in female blood DNA is incomplete. *Human Molecular Genetics*, 18(8):1439–1448, Apr. 2009. doi:10.1093/hmg/ddp054. (Pages 12, 15, 52, 56, 70, 79, 80, 134, 135, 137 and 139.)

M. Q. Zhang. Statistical features of human exons and their flanking regions. *Human Molecular Genetics*, 7(5):919–932, May 1998. (Page 104.)

J. Zhao, T. K. Ohsumi, J. T. Kung, Y. Ogawa, D. J. Grau, K. Sarma, J. J. Song, R. E. Kingston, M. Borowsky, and J. T. Lee. Genome-wide identification of polycomb-associated RNAs by RIP-seq. *Molecular Cell*, 40(6):939–953, Dec. 2010. doi:10.1016/j.molcel.2010.12.011. (Page 12.)

M. J. Ziller, F. Müller, J. Liao, Y. Zhang, H. Gu, C. Bock, P. Boyle, C. B. Epstein, B. E. Bernstein, T. Lengauer, A. Gnirke, and A. Meissner. Genomic distribution and inter-sample variation of non-CpG methylation across human cell types. *PLoS Genetics*, 7(12):e1002389, Dec. 2011. doi:10.1371/journal.pgen.1002389. (Page 56.)