Analysis and Application of Hash-based Similarity Estimation Techniques for Biological Sequence Analysis

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät Informatik

von

Henning Timm

Dortmund

2021

Tag der mündlichen Prüfung: 13.01.2022

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

Gutachter:
1. Prof. Dr. Sven Rahmann
2. Prof. Dr. Axel Mosig

Abstract

In Bioinformatics, a large group of problems requires the computation or estimation of sequence similarity. Tasks ranging from the detection of malicious genomic variants, over identifying the structure of different populations of one species, to exploring the capabilities of mixed species communities, all rely on the identification and quantification of similar strings. However, the analysis of biological sequence data has, among many others, three capital challenges: Through the use of high-throughput sequencing the amount of generated data surpasses the time required for detailed analysis, making the identification of worthwhile computations important. Secondly, for most species, no reference genome is available, necessitating the use of alignment-free approaches. Finally, since DNA sequencing relies on the observation of biological processes, the generated data contains errors specific to the sequencing technology. These errors need to be addressed by analysis approaches to avoid confounding biological signals and technological artifacts. Additionally, the explicit computation of sequence similarity remains a computationally expensive endeavor. Through the use of locality sensitive hashing methods, both the efficient estimation of sequence similarity and tolerance against the errors specific to biological data can be achieved.

Locality sensitive hashing (LSH) describes techniques, which generate similar hash values for similar input keys. Common LSH variants rely on MinHash values—the numerically smallest hash values in the input data—to sample an evenly distributed subset of minimizers from the input data that serves as a reduced representation. These so called sketches allow to estimate similarity measures between sequences by employing hash tables, thus circumventing the need for all-vs-all comparison. MinHashing techniques have found wide application within the field of Bioinformatics within recent years, since they do scale well and are able to mitigate many challenges specific to biological data. For example, possible alignment candidates can be identified, which can then be verified by computing alignments only for candidates with high estimated similarity. In this dissertation, I provide an overview of their application and variations across the field of Bioinformatics.

I developed a variant of the winnowing algorithm for local minimizer computation, which is specifically geared to deal with repetitive regions within biological sequences. Repetitive regions can

hamper the efficiency of downstream analysis, for example through introducing a large number of potential alignment positions, all equally likely and therefore not informative. Through compressing redundant information, I can both reduce the size of the hash tables required to save the sketches, as well as reduce the amount of redundant low quality alignment candidates. Analyzing the distribution of segment lengths generated by this approach, I can better judge the size of required data structures, as well as identify hash functions feasible for this technique. My evaluation could verify that simple and fast hash functions, even when using small hash value spaces (hash functions with small codomain), are sufficient to compute compressed minimizers and perform comparable to uniformly randomly chosen hash values. As an application for compressed winnowed minimizers, I outlined an index for a taxonomic protein database using multiple compressed winnowings to identify alignment candidates. To store MinHash values, I present a cache-optimized implementation of a hash table using Hopscotch hashing to resolve collisions.

As a biological application of similarity based analysis, I describe the analysis of double digest restriction site associated DNA sequencing (ddRADseq). This sequencing technique, which relies on cutting the DNA in reproducible positions across individuals of the same species, is commonly used to judge the biological diversity and population structure of organisms without reference genomes. I implemented a simulation software able to model the biological and technological influences of this technology to allow better development and testing of ddRADseq analysis software. Using datasets generated by my software, as well as data obtained from population genetic experiments, I developed an analysis workflow for ddRADseq data, based on the Stacks software. Since the quality of results generated by Stacks strongly depends on how well the used parameters are adapted to the specific dataset, I developed a Snakemake workflow that automates preprocessing tasks while also allowing the automatic exploration of different parameter sets. As part of this workflow, I developed a PCR deduplication approach able to generate consensus reads incorporating the base quality values (as reported by the sequencing device), without performing an alignment first. PCR duplicates are technical artifacts—copies of sequences introduced during sequencing, which can adversely affect analyses relying on sequence abundance—that can be removed while using their redundant information to mitigate sequencing errors. Both ddRADseq analysis and PCR deduplication require sequence clustering approaches, based on sequence similarity. As an outlook, I outline a MinHashing approach that can be used for a faster and more robust clustering, while addressing incomplete digestion and null alleles, two effects specific for ddRADseq that current analysis tools cannot reliably detect.

Acknowledgements

On these pages I would like to thank several people who have helped me with the completion of this thesis.

Special thanks go to my advisor Dr. Sven Rahmann for supporting me throughout my PhD studies and for giving me this opportunity in the first place. Further, I would like to thank Dr. Axel Mosig for serving as second reviewer for this thesis and working through its almost 300 pages. I would also like to thank Dr. Heinrich Müller and Dr. Günter Rudolf for completing my review committee and helping me with preparing my thesis defense.

Additionally, I owe thanks to Dr. Johannes Köster for his extensive support during the writing period of this thesis, in which he went out of his way for me multiple times, and for nudging me towards Rust. Thanks also go to Dr. Florian Leese for serving the role of mentor in my PhD program and to Gundel Jankord and Martina Gentzer for helping me with the administrative hassles of working at a university and doctoral procedures.

I would like to thank all the people I had the pleasure to share an office with: Marianna, Dominik, Nina, Elias, Jens, Nils and Till Schäfer in Dortmund, as well as Christo, Corinna, Bianca, Till Hartmann, David, Felix and Daniela in Essen. I enjoyed the time I was able to spend with you, your input on my academic and recreational projects, and the chance to share the perspectives on the world of such a diverse group of intelligent and driven people. Specifically, I would like to thank Denis, David, and Till H. for their strong and inspiring stances on how to be a better human being as well as a better scientist. Thank you Denis for our collaborative work, which I have enjoyed greatly, as well as for being an excellent partner in learning Japanese and rock climbing. Thank you Bianca for our long conversations about work as well as other topics. Thank you Christo for our interesting discussions about many different facets of computer science, but also for introducing me to a selection of interesting hobbies, and for organizing game nights.

Martina and Hannah gave me a glimpse into the field of biology. Thank you for walking me through the peculiarities of ddRAD sequencing. Finally, I would like to thank Dr. Marcel Martin for asking me to return a copy of Edward Tufte's *Envisioning Information* he kept in his office to the library. I took the liberty to read it before returning and thus impacted much of the typesetting of

this work, my approach to visualization, and to graphic design in general.

Special thanks go to Bianca, Kim, David, Daniela, Till, and Maxi for proofreading this work in parts or as a whole.

I would like to my parents for listening to my problems and success stories alike during my PhD program as well as giving me valuable input from outside my own head. Finally, many thanks are due to my wife Maxi who supported me throughout the whole process with both kind and stern words, support, and understanding.

Contents

1
Introduction

Over the course of the last two decades, DNA sequencing has evolved from a slow, expensive, and highly specialized endeavor to a widely used technology. The advent of Second Generation Sequencing (SGS), which allows to generate large genomic datasets at comparably low costs,[1] opened new fields of application ranging from the targeted analyses of single individuals to metagenomic and population genomic experiments. While the generation of sequencing data exploded, the generation of reference genomes did not keep up. Analyzing sequencing data without a reference genome requires methods that do not rely on the alignment of sequences.

In this thesis, I focus on the analysis of biological sequencing data through similarity based methods. Abstractly, the similarity of two items is quantified by a value between 0 and 1 that is high if the items are closely related under a defined metric. Similarities can be efficiently estimated and, if paired with a hashing approach, can reduce a quadratic all-vs-all comparison to a linear series of hash table accesses. Similarity-based methods can be used to cluster input sequences, but also to identify targets for a more detailed alignment-based analysis.

Since this work spans the realms of computer science and biology, I introduce both biological and mathematical concepts required in the first chapters. The chapter Biological and Mathematical Basics covers prerequisites from the fields of biology, computer science, and mathematics. I placed a special focus on hashing data structures and techniques, which are described in the chapter Hashing in Bioinformatics.

The first content chapter of this work is the chapter Reducing Cache Misses in Hopscotch Hash Tables, which describes a hash table architecture that reduces the number of compulsory cache misses with respect to its reference implementation.

The following chapter Computing and Approximating Resemblance and Containment describes Locality Sensitive Hashing (LSH) approaches and gives an overview of their use in the context of bioinformatics. I focus on the MinHash and Winnowing approaches, which rely on the sampling of items from a set or sequence via numerically small hash values as representative features.

[1] Sboner et al., "The real cost of sequencing: higher than you think!", 2011.

The winnowing technique[2] relies on incorporating locality information into the MinHash value computation by restricting the MinHash computation to a fixed length window. This is especially helpful to find similarities between documents of different sizes, like a DNA read and a reference database. In the chapter Distribution of Minimizer Segment Lengths I introduce compressed winnowing, which reduces the influence of repetitive regions on MinHash value computation, and analyze the distribution of minimizers generated by this technique. As an application example, I describe an index data structure for a protein database using multiple compressed winnowings. Additionally, I explore and evaluate a technique to store MinHash values in hash tables.

As an application of similarity based analyses, I focus on the topic of double digest RAD seq (ddRADseq) in the chapter Analysis of ddRAD Data. This sequencing technique is usually applied for non-model organisms and therefore cannot employ alignment-based methods for analysis. I first describe our data simulation software ddRAGE, which incorporates biological and technological effects specific to ddRADseq data. Subsequently, I present our analysis workflow for ddRADseq data and its evaluation through data simulated by ddRAGE. Rounding up this chapter is a section describing a software I developed for our workflow to remove duplicate reads introduced by Polymerase Chain Reaction (PCR duplicates).

1.1 *Contributing and Collaborative Work*

This section provides a detailed description of cooperations and collaborations contributing to this dissertation. These include published papers, software publications (via Zenodo[3]), and yet unpublished work. Prof. Dr. Sven Rahmann assisted in all stages as advisor.

Two works works found application in multiple chapters of this work: The Bioconda[4] project and the Python package dinopy. I partook in improving the Bioconda software through implementation, the addition of software packages (recipes) to the repository, and by performing administrative tasks for adding and maintaining recipes. Additionally, I assisted in writing and editing the paper. The dinopy Python package[5] for DNA input used for several evaluations in this work was developed by me with assistance from Till Hartmann.

I co-wrote the paper on cost-optimal assignments in multi-way bucketed cuckoo hash tables[6] (introduced in Chapter 3) and provided illustrations. The implementation was performed by Jens Zentgraf and Prof. Dr. Sven Rahmann. The foundation of this paper was the Masters' thesis of Jens Zentgraf, whom I assisted and guided with development and evaluation of the approach, as well as with the implementation of the software.

[2] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003; Roberts et al., "Reducing Storage Requirements for Biological Sequence Comparison", 2004.

[3] `https://zenodo.org`

[4] Grüning et al., "Bioconda: Sustainable and Comprehensive Software Distribution for the Life Sciences", 2018.

[5] Timm and Hartmann, *Dinopy — DNA input and output for Python and Cython*, 2020.

[6] Zentgraf, Timm, and Rahmann, "Cost-optimal Assignment of Elements in Genome-scale Multi-way Bucketed Cuckoo Hash Tables", 2020.

The bit-packed hopscotch hash table (BPHT) described in Chapter 4 was developed by myself with guidance from Prof. Dr. Sven Rahmann. I published all parts of the evaluation workflow,[7] including my implementation of the BPHT itself[8] and the employed tabulation hash functions,[9] via Zenodo.

For the development of the TaxMapper software[10] mentioned in Chapter 6, I helped devising the architecture and implementation of the analysis software, along with providing documentation, installation through the Bioconda repository, and refining the respective section in the paper. The main implementation of the software was performed by Dr. Daniela Beisser.

I developed the computation of expected segment length distributions described in Section 6.4 in cooperation with Dr. Denis Kurz. The structure of the proofs, which are the foundation of my description, were devised by Dr. Denis Kurz and myself. Dr. Denis Kurz and myself collaboratively created the first implementation of the Python program used to compute expected segment length distribution. I further refined this implementation and published it via Zenodo as part of the analysis workflow used for this section.[11]

For the double digest restriction site associated DNA sequencing (ddRADseq) simulation software ddRAGE,[12] I conceptualized the simulation workflow, implemented the software, and wrote the paper. Prof. Dr. Sven Rahmann provided guidance, optimizations, and code review for the implementation. Prof. Dr. Florian Leese, Dr. Hannah Weigand, and Dr. Martina Weiß provided domain knowledge about ddRADseq technology.

I developed the ddRAD analysis workflow[13] described in Section 7.3 in cooperation with Dr. Johannes Köster and Dr. Martina Weiss, who provided both architectural and domain knowledge. The in-house *Gammarus fossarum* dataset analyzed for the evaluation of this workflow was provided by Prof. Dr. Florian Leese and Dr. Martina Weiss. The evaluation workflow used for this section[14] was devised and implemented by myself and has been published via Zenodo.

Dr. Johannes Köster provided the idea and initial structure for the PCR deduplication process presented in Section 7.4. Further development and implementation was executed by myself. I worked on improving the implementation in cooperation with Felix Mölder, who devised an implementation for BAM files based on my implementation for FASTQ files. The evaluation workflow used for this section[15] was devised and implemented by myself and has been published via Zenodo.

For the remainder of this text, I will use the scientific "we".

[7] Timm, *BPHT Evaluation Workflow*, 2020.

[8] Timm, *BPHT Source Code*, 2020.

[9] Timm, *Rust-tab-hash Source Code*, 2020.

[10] Beisser et al., "TaxMapper: An Analysis Tool, Reference Database and Workflow for Metatranscriptome Analysis of Eukaryotic Microorganisms", 2017.

[11] Timm, *Segment Length Analysis Workflow*, 2021.

[12] Timm et al., "ddRAGE: A Data Set Generator to Evaluate ddRADseq Analysis Software", 2018.

[13] Köster and Timm, *snakemake-workflows/rad-seq-stacks*, 2021.

[14] Timm, *Rad-seq-stacks Evaluation Workflow*, 2021.

[15] Timm, *PCR Deduplication Analysis Workflow*, 2021.

2
Biological and Mathematical Basics

From the perspective of a computer scientist, many important states and processes in biology can be described using strings. For example, deoxyribonucleic acid (DNA) is a string over the nucleotide alphabet $\Sigma_{\text{DNA}} := \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Nucleotides of a DNA sequence can be translated into messenger ribonucleic acid (mRNA) which in turn codes for amino acids using substrings of length three (codons) thus creating a new string. This chapter covers the description of biological aspects with mathematical tools. If not noted otherwise, the biological basics presented here are based on the books Brock Biology of Microorganisms[1] and Molecular Biology of the Cell.[2]

2.1 Biological Sequences

Most integral functions of living cells are made possible by proteins. These biological molecules realize structural tasks like cell membranes or cytoskeletons, catalyze (bio-) chemical reactions (enzymes), and perform regulatory tasks (for example by binding to DNA sequences). Hence, in order to build a new cell or maintain the function of an existing one, a plethora of proteins needs to be synthesized. The blueprints to synthesize proteins are encoded in the genome of an organism, consisting of DNA. However, several steps are required to synthesize a protein from a DNA sequence, which are illustrated in Figure 2.1.

2.1.1 Structure of DNA and RNA Sequences

The DNA sequence in the nucleus is present as a DNA double helix, which consists of two complementary strands of nucleobases.

These four nucleobases **A**denine, **C**ytosine, **G**uanine, and **T**hymine,[3] called bases for short, form the lowest level of sequence analyzed in this work. As a shorthand notation, bases are represented by their first letter A, C, G, and T respectively. Chemically, each nucleobase comprises a phosphate sugar backbone (consisting of deoxyribose sugar) that forms the strand structure of DNA molecules and a nitrogenous base which encodes genetic information.

The phosphate backbone allows nucleobases to form strands, i.e. strings or sequences of bases, by forming covalent bonds with

[1] Madigan et al., *Brock Biology of Microorganisms*, 2012.

[2] Alberts et al., *Molecular Biology of the Cell*, 2017.

[3] In this work, items of biological sequences, like bases and amino acids, will be represented using sans-serif monospace font e.g. ACGT.

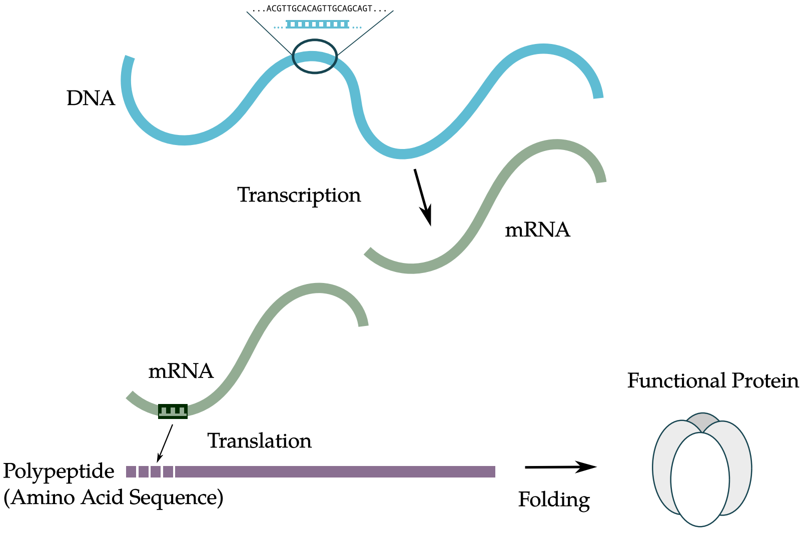


Figure 2.1: DNA is translated into mRNA which is translated into an amino acid sequence that folds itself into a functioning protein.

other phosphate sugars. Specifically, the 3′ carbon atom of a sugar molecule is connected to the 5′ carbon atom of the following sugar molecule with a phosphate group. Here 3′ and 5′ prime denote the position of the carbon atom in the ribose sugar, as illustrated in Figure 2.2. Due to the asymmetric structure of the sugar molecules, there is only one direction in which the sequence can extend and that is on the 3′ end.

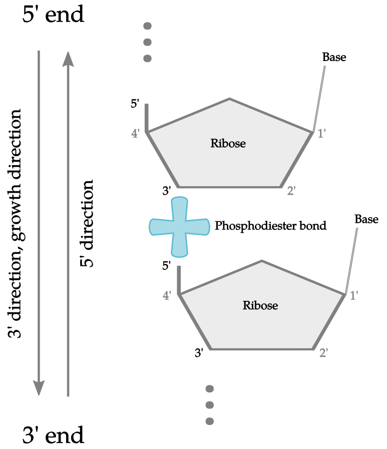


Figure 2.2: Structure of DNA and RNA backbones. Two ribose sugars are joined with a phosphodiester bond, connecting the 3′ carbon atom of the upper to the 5′ carbon atom of the lower sugar ring. Bases are attached to the 1′ carbon atom.

A DNA string T with a length of $|T| = n$ base pairs (bp) can be described as

$$T = (t_i)_{i=0}^{n-1} \qquad t_i \in \Sigma_{\text{DNA}} = \{\, \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T} \,\}\,.$$

We denote the base at position i of T as $T_{[i]}$ and the subsequence from position i to position j (exclusively) as $T_{[i:j]}$. By default, we denote all DNA sequences in 3′ direction.

The distribution of the four bases varies greatly between different organisms. A common metric to describe this is the GC-content (also GC-frequency) of the genome T in question, which is defined as:

$$\mathrm{GC}(T) = \frac{\sum_{i=0}^{|T|-1} \mathrm{gc}(T_{[i]})}{|T|}, \qquad \mathrm{gc}(t) = \begin{cases} 1, & t \in \{\, \mathtt{G}, \mathtt{C} \,\} \\ 0, & t \in \{\, \mathtt{A}, \mathtt{T} \,\} \end{cases} \tag{2.1}$$

Analogously, the AT-content is defined as $\mathrm{AT}(T) = 1 - \mathrm{GC}(T)$. The GC-content varies between $\sim$0.17 and $\sim$0.80, where extreme values are often associated with microorganisms, whereas the range for eukaryotes is smaller.

Each nitrogenous base has a complementary base that it can form hydrogen bonds with. A and T bind to each other with two hydrogen bonds, while C and G bind with three hydrogen bonds.

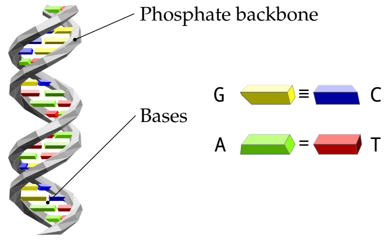


Figure 2.3: Left: DNA double helix, i.e. two complementary DNA strands wound together. Right: Pairs of complementary bases and their binding type. Graphic derived from "SNP model" by David Eccles (gringer) CC BY 4.0

We define the complement function

$$c(t) = \begin{cases} \mathtt{A}, & t = \mathtt{T} \\ \mathtt{T}, & t = \mathtt{A} \\ \mathtt{C}, & t = \mathtt{G} \\ \mathtt{G}, & t = \mathtt{C} \end{cases} \qquad t \in \Sigma_{\text{DNA}} = \{\,\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\,\}$$

which returns the complement of a given base t. A DNA double strand, i.e. a single strand with its bound complement, forms the characteristic double helix structure as illustrated in Figure 2.3. This is the default state in which DNA is present in all eukaryotes (including the human body). Note that the orientation of the complementary strand is reversed.

For each DNA (single) strand T, there exists a reverse complementary strand

$$\overline{T} = \left(c(t_{n-1-i})\right)_{i=0}^{n-1}, \qquad t_i \in T$$

which comprises the complementary bases in reverse orientation, as illustrated in Figure 2.4.

While we focus mostly on the analysis of DNA and protein sequences in this work, Ribonucleic Acid (RNA) sequences perform an important role during protein synthesis. RNA molecules are structurally very similar to DNA, however, their backbone is built from ribose sugar instead of deoxyribose and RNA does not usually form double helices. RNA nucleotides do not contain Thymine but the structurally similar **U**racil, which also binds to Adenine. Due to this, it is possible to build a complementary RNA strand to a DNA single strand.

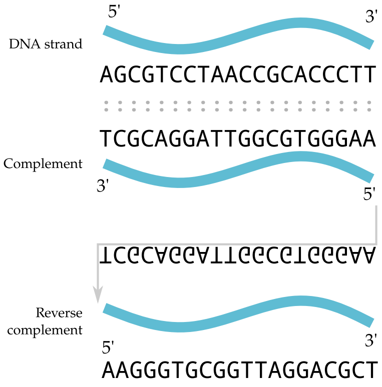


Figure 2.4: A DNA single strand, its complement (read from 3′ to 5′), and reverse complement (complement read from 5′ to 3′).

2.1.2 *Protein Synthesis from DNA*

To synthesize a protein, an RNA copy of the template DNA sequence is passed to a ribosome which assembles the protein from amino acids.

We deliberately skip over which parts of the DNA are actually transcribed, since gene expression, regulation, splicing etc. are beyond the scope of this work.

First, the DNA sequence is transcribed into an RNA molecule. An RNA polymerase molecule binds to a DNA double helix and separates the strands, forming a transcription bubble. Using free nucleotides that are available in the cell, the polymerase constructs a complementary RNA strand to a single DNA strand. This is illustrated in Figure 2.5. The created RNA molecule is called messenger RNA (mRNA) and is transported through the cell to a ribosome for the next step.

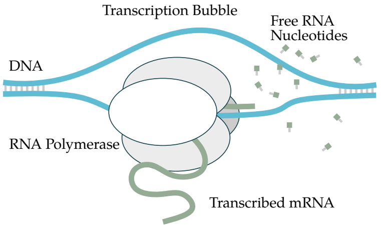


Figure 2.5: Transcription of DNA into mRNA.

In the second step of protein synthesis, the mRNA molecule is translated into an amino acid sequence. This translation takes place in the ribosomes of the cell, which are, depending on the proteins they synthesize, either located in the cytoplasm or affixed to the endoplasmatic reticulum.

During translation, the mRNA molecule is pulled into the ribosome three bases at a time and *read* by the ribosome. Depending on the base triplet, called a codon in this context, the ribosome performs one of three actions:

1. If the start codon AUG (▶) is read and no amino acid sequence is already being assembled, assembly of an amino acid sequence is started.

2. If a stop codon $\in \{\, \texttt{UAG}, \texttt{UGA}, \texttt{UAA} \,\}$ (●) is read and an amino acid sequence is being assembled, finish the assembly and release the amino acid sequence.

3. Otherwise append the amino acid encoded by the codon to the amino acid sequence.

Note that the codons are given as RNA bases. In DNA sequences the Us would be replaced with Ts.

All possible $|\Sigma_{\text{DNA}}|^3 = 64$ codons with their associated amino acids or functions are illustrated in Figure 2.6. This visualization is called the code wheel and was introduced by Bresch and Hausmann in the third edition of their book "Klassische und molekulare Genetik".[4] It is read by following a codon sequence beginning from the center of the sun to the outer layer, where the encoded amino acid is denoted.

Note that not all of the mRNA sequences are translated into amino acids, but only subsequences that fall between a start codon and a stop codon. These subsequences are called open reading frames (ORFs). Parts of the genome that are translated into proteins are called coding regions, as opposed to non-coding regions. Additionally, there are also RNA sequences of which no part is translated into amino acids. While these perform important functions in the cell, including the regulation of gene expression and the transport of amino acids to the ribosome, we focus solely on translated mRNA sequences.

A sequence of amino acids is called a polypeptide and can be described as a string over the alphabet

$$\Sigma_{\text{AA}} := \{\, \texttt{A}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, \texttt{H}, \texttt{I}, \texttt{K}, \texttt{L}, \texttt{M}, \texttt{N}, \texttt{P}, \texttt{Q}, \texttt{R}, \texttt{S}, \texttt{T}, \texttt{V}, \texttt{W}, \texttt{Y} \,\}, |\Sigma_{\texttt{AA}}| = 20$$

which comprises the standard amino acids. However, additional rare amino acids can be formed in some organisms, most of them bacteria. Most prominently Selenocysteine (U) and Pyrrolysine (O), which are chemically similar to Cysteine and Lysine respectively. While Pyrrolysine is extremely rare, Selenocysteine occurs in some proteins in *Escherichia coli*. Hence, the extended amino acid alphabet which contains the rare amino acid is defined as

$$\Sigma_{\text{AA+}} := \Sigma_{\text{AA}} \cup \{\, \texttt{U}, \texttt{O} \,\}$$

and comprises 22 amino acids.

After the translation is terminated by a stop codon, the polypeptide is released from the ribosome. Determined mostly by its amino acid sequence, it then undergoes structural changes, which determine its function in the body.

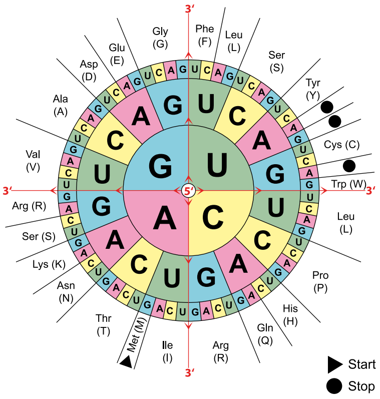


Figure 2.6: Code wheel for all codons of the standard amino acid alphabet Σ_{AA}. The rare amino acid variants U and O are encoded by UGA and UAG respectively. Those codons also encode stop codons. Image released into the public domain by Wikipedia user Mouagip.

[4] Bresch and Hausmann, *Klassische und molekulare Genetik*, 1972.

2.1.3 *Protein Structures and Functions*

A polypeptide does not maintain the linear shape that is has after assembly, but *bends* into shape by interactions between its amino acids. Since each amino acid in Σ_{AA} has different chemical properties, a polypeptide is subject to interactions between its building blocks. Depending on their vicinity and other factors, amino acids interact with each other and bend the polypeptide into a three-dimensional shape in a process called folding. These three-dimensional structures enable the protein to perform a biological function. For example it allows a potassium channel to be integrated into a membrane, transport ions through the membrane and to be opened and closed. There are four levels of organization that are used to describe the structure of a protein:

Primary Structure describes the sequence of amino acids in the protein.

Secondary Structure describes the expression of local structure, namely α-helices and β-sheets. Both comprise of small and local repetitive patterns of amino acids which form hydrogen bonds. These are typically formed, before the three-dimensional shape is adopted.

Tertiary Structure describes the three-dimensional folding of the protein that enables its biological function. A protein folds itself into the state of minimal free energy, called the native state. For some polypeptides, this folding process is supported by chaperone proteins. These guide the folding process and prevent abnormal folding, for example by slowing the folding process. Abnormally folded proteins are useless at best and at worst can cause diseases like Creutzfeld Jakob Disease and Kuru.

Quaternary Structure describes the reversible organization of several proteins into a protein complex. Each subunit of the complex has its own primary, secondary, and tertiary structure. Not all proteins adopt a quaternary structure, but many, including hemoglobin, ion channels, and RNA polymerase do.

Since we will not focus on protein structure, for the sake of brevity we will refer to all polypeptides as proteins.

2.2 *Genomic Mutations*

Mutations are alterations in the genome of an individual or population with respect to a specified *normal state*. These can be introduced by a multitude of factors including exposure to radiation, errors during DNA replication, and certain viruses. There are several ways to classify mutations, including by type, i.e. how they change the genome, and by effect, i.e. what changes they introduce. We will classify mutations by type and describe their possible effects.

2.2.1 *Single Nucleotide Variations*

The smallest variation of a genome is changing a single base. This is called a single nucleotide variation (SNV) and is the most common kind of mutation.[5] While their effect might seem small, it can be the cause of diseases, like sickle cell disease.[6] Others may increase the probability of cancer onset, given a particular genetic makeup.[7] However, most of the more than 4 million SNVs expected within a human genome with respect to the reference genome do not have negative effects and can be counted towards expected genetic variability.[8]

An SNV in a coding region can have several effects on the genome, depending on the changes it introduces in the base's codon. For an illustration of the effects discussed, refer to Figure 2.7. Consider the codon `UGU`, which encodes the amino acid Cysteine, and several mutations of its last base:

- If the last `U` mutates into `C`, the new codon also codes for Cysteine (`C`) and the resulting protein is identical to the unmutated one. This is called a silent mutation.
- If instead, the `U` mutates into `G`, the codon is now translated into Tryptophan (`W`). This can alter the three-dimensional structure of the protein and influence its biological function. Thus, this is a missense mutation. Depending on the new amino acid, missense mutations are further classified into conservative and non-conservative mutations. In a conservative mutation, the new amino acid has similar properties as the old one, mitigating the mutation's effect on the protein structure.
- If `U` mutates into `A` instead, a new stop codon is introduced that terminates the translation of the mRNA prematurely. The resulting protein is incomplete and cannot function properly. Hence this type of mutation is called a nonsense mutation.
- Finally, if originally there was a stop codon, like `UAG`, and the SNV changed a decisive base, the translation of the mRNA does not stop as expected, causing a readthrough.

A single nucleotide variation that is present in a significant fraction of a population is also called a single nucleotide polymorphism (SNP).

2.2.2 *Insertions and Deletions*

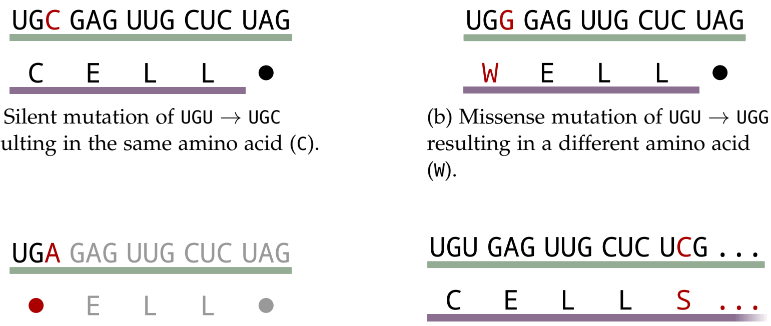
Indel mutations (a portmanteau of insertion and deletion) are genetic variants that either introduce (insertion) one or more new bases, or remove (deletion) one or more bases from the genome.

[5] The 1000 Genomes Project Consortium, "A Global Reference for Human Genetic Variation", 2015.

[6] Rees, Williams, and Gladwin, "Sickle-Cell Disease", 2010.

[7] Levy-Lahad et al., "A Single Nucleotide Polymorphism in the Rad51 Gene Modifies Cancer Risk in BRCA2 but not BRCA1 Carriers", 2001.

[8] The 1000 Genomes Project Consortium, "A Global Reference for Human Genetic Variation", 2015.



(a) Silent mutation of UGU → UGC resulting in the same amino acid (C).

(b) Missense mutation of UGU → UGG resulting in a different amino acid (W).

(c) Nonsense mutation of UGU → UGA resulting in premature termination of the translation.

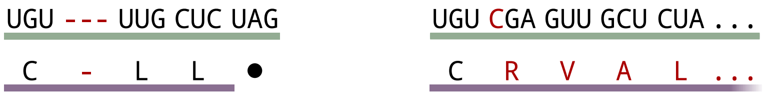
(d) Readthrough mutation of UAG → UCG adding additional amino acids.

Figure 2.7: Effects of different SNVs. The mRNA sequence is drawn in green with its codons on top. The translated polypeptide is drawn in purple with the translated amino acid under the template mRNA codon. Mutated bases in the mRNA as well as different translated amino acids are highlighted in red. Stop codons are denoted as •.

While indels occur less frequently than SNVs,[9] their effect on protein synthesis can be devastating.

If an indel removes (or insertions) a sequence with a length that is a multiple of three, amino acids are missing from the polypeptide, or additional amino acids are present. This is the case, since only complete codons are removed (or added). However, in the case that an indel has a length that is no multiple of three, all subsequent codons are effected.

Consider a deletion of one nucleotide. This alters the codon, since the two remaining bases form a new codon with the first base of the following (old) codon. Such a frameshift passes on through the entire mRNA and is likely to result in a completely different protein. Illustrations of both types of indels can be found in Figure 2.8.



(a) Indel of length 3 not causing a frameshift mutation.

(b) Indel of length 1 affecting all subsequent codons due to frameshift.

Figure 2.8: Different types of indel mutations. As in the figure above mRNA is drawn green, the polypeptide purple, and changes are highlighted red.

2.2.3 *Structural Variations*

Until now we ignored how the genome of an organism is stored. For eukaryote species, including humans, most animals, and plants, the genome is distributed along several DNA molecules, called chromosomes. Each chromosome is a linear DNA molecule that is *wound up* into a space-efficient shape supported by histones and other scaffold proteins. Among other effects, this allows storing the whole genome of an organism in the nucleus of each cell. Chromosomes play an important role in sexual reproduction, where chromosomes from both parents are combined to form a new genome (see Section Ploidy and Zygosity).

Structural variations affect a genome on the chromosome level, i.e. large subsequences (>50 bp) within one chromosome are changed in number (copy number variation, CNV) or location (structural variation, SV)[10]. Among others, the following types

[9] Mills et al., "An Initial Map of Insertion and Deletion (INDEL) Variation in the Human Genome", 2006.

[10] Alkan, Coe, and Eichler, "Genome Structural Variation Discovery and Genotyping", 2011.

of mutations are classified as structural or copy number variations:

Inversions, where a subsequence is replaced by its reverse complement.

Duplications, where one or multiple copies of a subsequence are introduced into the chromosome. A duplication in which these copies occur next to each other is called a tandem duplication.

Translocations, where a subsequence is moved to a different location within the chromosome or onto another chromosome.

Large indels (>50 bp), which behave like the indels described above, are also classified as SVs.

SVs and CNVs have been shown to be very prevalent in the human genome and, due to their size, are responsible for the majority of mutated bases.[11]

2.3 *Ploidy and Zygosity*

As mentioned in Section Structural Variations, many organisms possess more than one copy of each chromosome; they vary in ploidy. Organisms with a single set of chromosomes are called haploid, in contrast to diploid organisms (with two copies) and polyploid organisms (more than two copies). Humans and many other organisms are diploid, while many microorganisms possess greatly differing ploidies. If an organism possesses more than one chromosome set, i.e. has several copies of the same genome, each of these sets can contain different genetic variants. The possible states of a genetic variant, for example the presence of a certain SNP or combination of SNPs, are called alleles. When different alleles in an organism are described, they are denoted with capital letters. Since a variant can either be present or not on both chromosome sets, there can be three combinations of alleles in a diploid organism. These combinations of alleles make up the genotype of the organism. When both chromosomes present the same allele, we speak of a homozygous genotype; a genotype with diverging alleles is heterozygous.

Example: We distinguish two alleles of a genetic variant, one where a deletion removed 3 bases (B) and another without the deletion (A). The three possible genotypes AA, AB (mutation only on one chromosome set), and BB (mutated in both chromosome sets) can each have a different influence on the organism.

Alleles and genotypes are descriptions of biological configurations of an organism. In contrast to this, their effect on the individuals carrying them is called the phenotypic effect.

[11] Sudmant et al., "An Integrated Map of Structural Variation in 2,504 Human Genomes", 2015.

Example: Consider a disease that is linked to the genetic variant described in the previous example. The disease only breaks out for the genotype BB, i.e. if the variant is present on both chromosome sets. There are two phenotypes (*disease present* vs. *disease not present*) that are associated with three genotypes (BB vs. AA and AB).

For polyploid organisms that possess more than two chromosome sets, the number of potential genotypes is higher. However, we will not focus on these and hence omit this topic.

2.3.1 *Allele Frequency*

When several alleles of a genomic site are present in a population, their abundance can give insight into the mechanisms that caused them, their effect on the organism, and their evolutionary dynamics. For example the interplay of genotype and phenotype. Allele frequencies describe how large the fraction of a given population is that carries the allele in question.

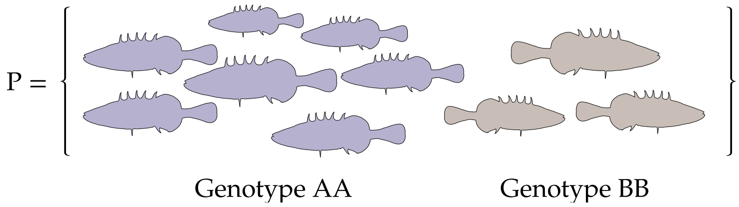


Figure 2.9: Population with $|P| = 10$ individuals of a diploid species. The variant is homozygous for all individuals, i.e. each individual carries either two copies of allele A or allele B.

For example, in the population in Figure 2.9 the allele frequencies of A (🐟) and B (🐟) are:

$$F(A,P) = \frac{7}{|P|} = 0.7 \qquad F(B,P) = \frac{3}{|P|} = 0.3 \tag{2.2}$$

Given a population P of diploid organisms, the frequency of an allele x is computed as:

$$F(x,P) = \frac{1}{|P|} \sum_{i \in P} \begin{cases} 1, & \text{if } i \text{ is } xx \text{ homozygous} \\ \frac{1}{2}, & \text{if } i \text{ is } x\cdot \text{ or } \cdot x \text{ heterozygous} \\ 0, & \text{else} \end{cases} \tag{2.3}$$

In the diploid case, individuals with heterozygous genotypes are counted for both alleles, each time weighted by $\frac{1}{2}$. Note that for polyploid organisms, the number of possible genotypes rises with the number of chromosome copies.

2.4 *Acquisition of Biological Sequences*

Up until now, we offered both a description of the biological sequence as well as a mathematical description. We did not touch on the subject of the acquisition of biological sequences for bioinformatic analysis yet. In this section we describe sequencing technologies, biotechnological workflows that bridge the gap between biological sample and a text representation on our hard drive. Their goal is to provide a bioinformatician with string representations of (most of the time small) DNA fragments, the so called reads.

All currently available sequencing technologies rely on the reconstruction of a DNA single strand. However, the measured parameters as well as the volume, structure, and price of the acquired data vary greatly between different generations of sequencing technologies. The first generation of sequencing technologies (FGS), like the Sanger Chain Termination method,[12] allows for the analysis of one DNA molecule at a time, but provides a high level of accuracy. In contrast to this, second generation sequencing (SGS) introduced massively parallel analysis of short reads ($\leq$1000 bp), like the Sequencing by Synthesis approaches employed by Illumina[13] and 454 Pyrosequencing[14] technologies. However, the massive gain in throughput, and consequently a lower price per sequenced base, came at the cost of a higher error rate. Finally, the current third generation sequencing (TGS) technologies, like Pacific Biosciences (PacBio) Single Molecule Real Time Sequencing and Oxford Nanopore Sequencing technologies like the MinION are able to process longer reads.[15] Additionally, this analysis can be done in real time, but again introduces an even higher error rate.

Currently, SGS is still the most widely used technology, due to its flexibility and availability. While TGS data are very well suited to solve problems that were hard to tackle with FGS and SGS data only, they require the development of new analysis software. For some of these problems hybrid solutions that combine long TGS reads with precise and abundant SGS reads are used. FGS, especially Sanger sequencing, is valued for its high accuracy and is still employed for single gene analysis and to validate results. Due to the specific characteristics of the different kinds of sequencing data, we will take a more detailed look into the different sequencing technologies. A more comprehensive overview of sequencing technology has been presented by Heater and Chain,[16] Goodwin et al.[17] and Shendure et al.[18]

2.4.1 *Structure of DNA and RNA Reads*

All sequencing technologies are restricted in the length of reads they can generate due to the lossyness of the exploited biological processes. Hence, the DNA sequence to analyze is broken up into fragments as part of the analysis. Depending on the employed sequencing technology and its respective fragmentation technology, the length and structure of the analyzed fragments can vary drastically. Additionally, the reads that are generated from the fragments are usually smaller than the fragments themselves and can range from a few tens of bases to several thousands, as illustrated in Table 2.1.

Furthermore, there are two variants of reads that can be acquired, single-end (SE) reads and paired-end (PE) reads. A SE read

[12] Sanger, Nicklen, and Coulson, "DNA Sequencing with Chain-Terminating Inhibitors", 1977.

[13] Turcatti et al., "A New Class of Cleavable Fluorescent Nucleotides: Synthesis and Optimization as Reversible Terminators for Dna Sequencing by Synthesis", 2008.

[14] Margulies et al., "Genome Sequencing in Microfabricated High-Density Picolitre Reactors", 2005.

[15] Shendure et al., "DNA Sequencing at 40: Past, Present and Future", 2017.

[16] Heather and Chain, "The Sequence of Sequencers: The History of Sequencing DNA", 2016.

[17] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.

[18] Shendure et al., "DNA Sequencing at 40: Past, Present and Future", 2017.

Technology	Nr. of reads	Read length (bp)	Type
Sanger	1	700	SE
Illumina (HiSeq X)	3 000 000 000	150	PE
PacBio RSII	55 000	≈ 20 000	SE
ONT MinION	>100 000	≤ 200 000	SE

Table 2.1: Read length and number of reads per sequencing run for some of the most prominent sequencing technologies. A more comprehensive list has been presented by Goodwin, McPherson, and McCombie ("Coming of Age: Ten Years of Next-Generation Sequencing Technologies").

is generated by reading only one end of a fragment, while to generate a PE read, both ends of the fragments are analyzed. Depending on context, forward and reverse read are also called p5 and p7 read respectively In addition to the sequences of both reads in the pair, the length of the fragment and in turn the distance of the two reads in the analyzed genome is also analyzed. This distance is called the insert size of the PE read. The structure of SE and PE reads is illustrated in Figure 2.10.

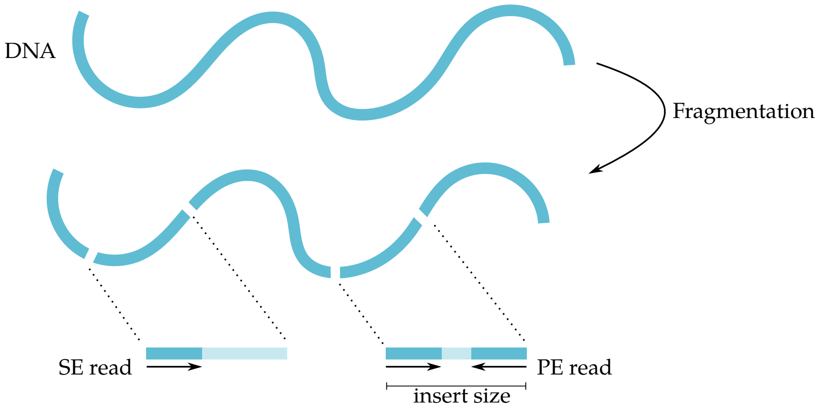


Figure 2.10: Example of a SE read (left) and a PE read (right). The DNA sequence is broken apart by a fragmentation process. Then, either SE reads (from one end) or PE reads (from both ends) can be generated.

2.4.2 *Sequencing Quality and Phred Scores*

As mentioned above, all sequencing approaches exploit biological processes, which are not perfectly effective. While an error that occurs during sequencing can not be corrected within the sequencing process itself, uncertainty about the nucleotide detected can be quantified. Sequencers report a Phred quality score for each base they analyze, i.e. a measure how certain the sequencer is that it called the correct base.

The Phred quality score[19] describes the quality of the base call as

$$Q = -10 \cdot \log_{10}(e) \tag{2.4}$$

where $e = \mathbb{P}(\text{base call was incorrect})$ is the error probability. For example, if the base call was correct with 99.99% certainty, this would result in a Phred score of

$$Q = -10 \cdot \log_{10}(1 - 0.9999) = -10 \cdot (-4) = 40. \tag{2.5}$$

Phred scores are often rounded to the nearest integer and encoded using ASCII characters.

[19] Ewing and Green, "Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities", 1998.

2.4.3 *Polymerase Chain Reaction*

Polymerase chain reaction (PCR) is an important technology for the preparation of samples that is employed by many sequencing technologies. As mentioned above, sequencing technologies observe and analyze the reconstruction of a DNA single strand into a double strand. Since each binding reaction occurs on a very small scale, they are difficult to detect by themselves. SGS sequencing technologies, for example, rely on detecting fluorescent components added to DNA bases, which are too dim to be detected by themselves. To mitigate this, many copies of the DNA molecule to be sequenced are needed to make the signal detectable. However, most of the time we start with only a small sample of DNA molecules and need to amplify it for analysis. This is the realm of PCR, which harnesses the function of polymerase proteins to multiply DNA sequences in a structured process. While several improvements to the PCR process have been introduced, the core concept remains as follows:

Denaturing Complementary DNA strands are separated by heating. This increase in temperature destroys the hydrogen bonds connecting the bases of forwards and reverse strands but keeps the backbone intact.

Primer Annealing Primer sequences that bind to the single strands are introduced. For this, a high abundance of primer sequences is required to prevent separated strands from just rebinding. By lowering the temperature, the primer sequences can bind to the single strands and offer a surface for DNA polymerase to bind to.

Elongation As described in Section Protein Synthesis from DNA (p. 11), DNA polymerase (re-)constructs the complementary strand to the DNA single strands. After this, there are two DNA double strands, each comprising one strand of the template and one reconstructed strand.

The three steps detailed above comprise one PCR cycle and are applied to not one, but many sequences in parallel. If we assume perfect efficiency of this process, after n PCR cycles there are 2^n copies of each template sequence.

During this process, however, several kinds of errors can occur, two of which we will highlight: substitution errors and chimeras. A substitution error describes the case that a single faulty base was incorporated by the DNA polymerase. Since these errors introduce changes that are similar to SNVs (as introduced in Section Single Nucleotide Variations, p. 14), distinguishing them is important to avoid introducing spurious mutations. The second kind of PCR error are chimeric reads[20]. When the elongation step terminates prematurely, this incomplete sequence can bind to another similar (but different) sequence and finish its elongation there. The result-

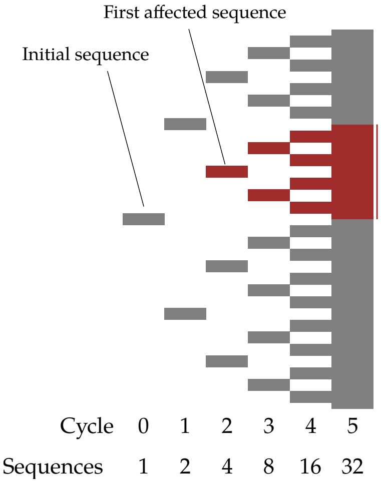


Figure 2.11: Propagation of PCR errors. The error (shown in red) is introduced in PCR cycle 2. After cycle 5, $2^{5-2} = 8$ of the $2^5 = 32$ fragments carry the PCR error (visualized by the red bar on the right side).

[20] Meyerhans, Vartanian, and Wain-Hobson, "DNA Recombination During PCR", 1990; Smyth et al., "Reducing Chimera Formation During PCR Amplification to Ensure Accurate Genotyping", 2010.

ing DNA double strand is a combination of the prefix of one of its parent strands and the suffix of another one.

Note that once an error has been introduced in the PCR process, further copies of the affected strand will all contain this error. Consequently, if exactly one error is introduced in cycle m, there are 2^{n-m} affected reads and $2^n - 2^m$ unaffected reads (see Figure 2.11).

2.4.4 Illumina – Cyclic Reversible Termination

Currently, the most wide spread sequencing technology is the cyclic reversible termination (CRT) approach employed by Illumina devices[21,22]. This SGS approach, which implements a massively parallelized version of chain termination, is able to generate several billion reads of length up to 300 bp. An illustration describing the workflow can be found in Figure 2.12. As described by Goodwin et al.,[23] the core idea of the Illumina CRT sequencing approach is as follows.

After fragmentation, primers are ligated to the DNA fragments which allow them to bind to a specific cluster of probes on a flow cell. These single samples are amplified using bridge amplification, a variant of PCR that is local to a cluster on the flow cell. Consequently, all sequences within a cluster are identical, apart from PCR errors.

After the amplification step is completed, the actual sequencing can commence. Reverse complementary sequences to the primer sequences are added, which allow a polymerase to bind to the strand and initialize reconstructing its complementary sequence. Then, the following three steps are repeated, until all bases have been sequenced:

Add modified nucleotides in which the 3′ group in the ribose is blocked to prevent further elongation. All four nucleotides are contained in the mixture, each of which is labeled with a specific fluorophore. In this step, all sequences present on the flow cell are advanced by one nucleotide. Finally, all unbound nucleotides are removed from the flow cell.

Measure light response of all clusters by exciting the fluorophores with a laser. Since each cluster was elongated with a particular nucleotide, all sequences within one cluster produce the same response. These added light signals can be detected by a camera and provide the sequence information for one nucleotide per cluster.

Reverse chain termination by cleaving off the fluorophore and restoring the 3′ end of the ribose. After washing out the debris, a new cycle can be started.

In each of these three steps, errors can occur which inform the specific error profile of Illumina sequencers. A major problem is

[21] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.

[22] Also known as sequencing by synthesis (SBS), which describes a broader class of SGS technologies that also comprises single nucleotide addition approaches such as Ion Torrent and 454 pyrosequencing.

[23] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.

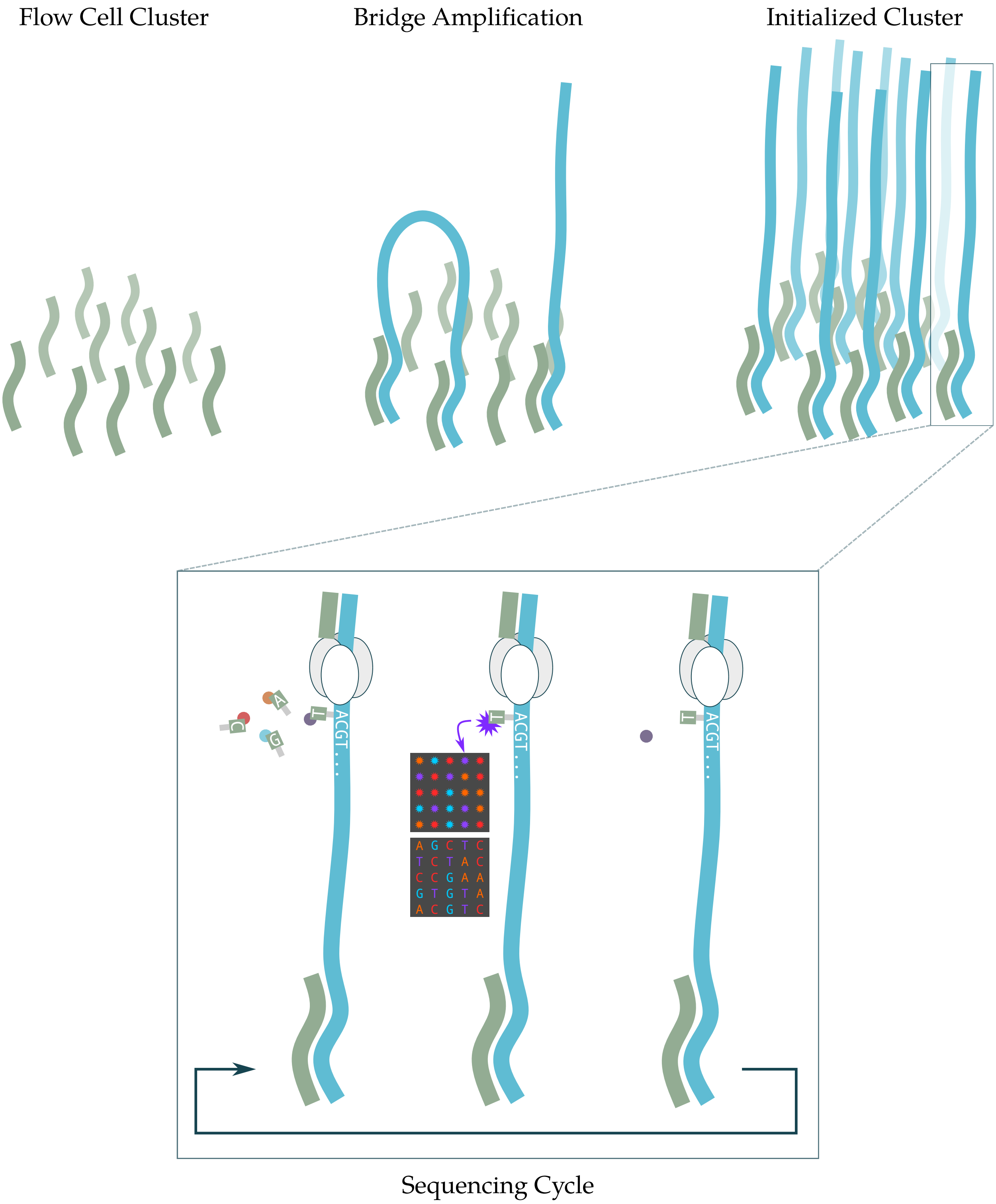


Figure 2.12: Illustration of the Illumina CRT sequencing workflow. The lower box shows the repeating sequencing cycle for one molecule in a cluster. Each cycle determines one base of the analyzed sequence.

strand desynchronization, i.e. some strands within one cluster elongate different positions within their respective sequence. This can occur, for example, when no nucleotide has bound to a strand in a previous step, or when the terminator is not removed correctly. Desynchronized strands produce contradictory light responses for the cluster, which increases the difficulty to call the correct base. Since these errors accumulate, the quality of base calls degrades with each step. Consequently, the error rate of base calls increases towards the end of the read.

The kind of errors introduced by faulty base calls are substitution errors, which are similar to the ones introduced by PCR. However, while substitution errors from PCR affect several reads, sequencing substitution errors only affect single reads.

In total, Illumina sequencers generate substitution errors with an error rate of $0.1 - 1\%$.

2.4.5 *PacBio SMRT and ONT MinION – Single Molecule Sequencing*

In contrast to SGS short reads, TGS approaches are able to yield longer reads, which helps the detection of structural variations. However, these longer reads also come with a higher error rate. This section is based on the more detailed description of these technologies given by Goodwin et al.[24]

The main difference to SGS approaches is, that TGS technologies like Pacific Biosciences' Single Molecule Real Time (SMRT) sequencing and Oxford Nanopore Technologies' (ONT) sequencing do not require PCR amplification and work with single molecules. While both technologies produce long reads with a high error rate, the underlying technologies differ greatly.

SMRT sequencing uses a flow cell with picolitre wells with a transparent bottom (called Zero Mode Waveguide; ZMV). In each well, a polymerase molecule is affixed which is used to synthesize a complementary DNA strand using fluorescent nucleotides. The emitted light signals for each well are recorded in real time using a laser and CCD camera, similar to Illumina SBS. Due to the small size of the ZMW (less than half the wavelength of light, thus reducing interference) and the stationary polymerase, singular light signals can be detected. A unique feature of SMRT sequencing is that analyzed DNA molecules can either be linear or circular. While long linear molecules can be analyzed in single pass mode with a high error rate (10 - 15%), circular molecules can be passed through the polymerase multiple times. This reduces the error rate to $\sim$0.0001, but is only possible for short molecules ($<$3000 bp).

In contrast to all other sequencing approaches, the ONT sequencing does not rely on secondary features, like the incorporation of bases etc. Instead of optical measurements, these devices analyze changes in electric currents. Using a motor protein, a DNA strand is pushed through a pore while the changes in electric cur-

[24] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.

rent generated by the DNA nucleotides passing through the pore is recorded. From the changes in current, q-grams (subsequences of length q) in the sequence can be inferred. While the error rate of ONT sequencing is relatively high with up to 30%, its main advantages are:

- There is no upper limit on the length of analyzed molecules, however in practice very long molecules are still a challenge.[25]
- The MinION device itself is very small, about the size of a USB stick, allowing its use outside of a laboratory.

Another difference to Illumina short reads is that both of these TGS approaches mainly produce indel errors. Additionally, ONT reads also suffer from homopolymer errors, i.e. the length of runs of the same base might not be reported correctly. This is due to the fact that the detected signal does not change within a run of identical bases, which makes it hard to discern the length of a run.[26]

2.4.6 *Specialized Application Sequencing Workflows*

Going a step further from sequencer technologies, sequencing workflows are designed to solve a specific problem or provide a special kind of data. On a very basic level this can be described by *what* is sequenced as opposed to *how* it is sequenced. Two examples, which are relevant for this work are RNA sequencing and RAD sequencing.

(Transcriptomic) RNA sequencing focuses the sequencing endeavor on the RNA in a cell or sample of cells instead of the whole genome.[27] By extracting the mRNA from a cell the analysis can be limited to transcribed sequences of the genome: the transcriptome. Analyzing the transcriptome of an organism yields information about abundance and types of proteins the organism produces, which in turn provide insight into the capabilities of the analyzed species. When working with a protein reference database, a transcriptomic dataset has the advantage that it does not contain reads from non-coding regions. Additionally, transcriptomic RNA sequencing data allows to perform differential expression analysis, where the abundance of a transcript is compared between two groups of samples.

Restriction Site Associated DNA (RAD) sequencing, or RADseq for short, limits the analysis to the cut sites of restriction enzymes. This technique is frequently applied for the analysis of genetic diversity in non-model organisms. There are several variants of RADseq, including basic RADseq[28], 2b-RAD[29], and double digest RADseq (ddRAD)[30], which differ in the type and number of restriction enzymes used. We will discuss ddRAD in detail in Section Analysis of ddRAD Data.

[25] Goodwin et al., "Oxford Nanopore Sequencing, Hybrid Error Correction, and de novo Assembly of a Eukaryotic Genome", 2015.

[26] Rang, Kloosterman, and Ridder, "From Squiggle to Basepair: Computational Approaches for Improving Nanopore Sequencing Read Accuracy", 2018.

[27] Ozsolak and Milos, "RNA Sequencing: Advances, Challenges and Opportunities", 2011.

[28] Davey and Blaxter, "RADSeq: Next-Generation Population Genetics", 2010.

[29] Wang et al., "2b-RAD: a Simple and Flexible Method for Genome-Wide Genotyping", 2012.

[30] Peterson et al., "Double Digest RADseq: an Inexpensive Method for de novo SNP Discovery and Genotyping in Model and Non-Model Species", 2012.

Metagenomic and Metatranscriptomic sequencing describes sequencing genetic material from different species at the same time.[31] As an example, consider sampling water from a pond, which is likely to contain several different species of algae, bacteria, and other microorganisms. The presence and abundance of certain indicator species provides insight into environmental parameters of the habitat.

Since for most species there is no reference genome, metagenomic analyses often use protein databases to infer the capabilities of the sequenced communities in lieu of identifying their precise species. This suggests the use of metatranscriptomics, which, as above, restricts the sequencing to transcribed sequences.

2.4.7 *Reference Genomes and their Bioinformatic Analysis*

As mentioned above, genomes of different individuals of the same species share the majority of their genomic sequences. The small set of differences accounts for intra-species genomic variation, generated by mutations and recombinations through sexual reproduction. Their effects range from different hair and eye colors to the presence or absence of illnesses. Apart from these small but important diversifications, most of the genomic sequence of a species can be sequenced and compared between different individuals to identify shared parts. By assembling results of one or (more likely) several sequencing runs into contiguous sequence fragments (contigs), which can in turn be used to derive the sequence of chromosomes, a reference genome for a species can be created. Using such a reference genome, we can perform queries against it to find out, if data from a different sequencing experiment match this reference.

Creating a reference genome is a time and resource consuming task, that is only performed for a few organisms of interest, including the human genome.[32] Beginning with DNA sequencing data, a huge puzzle needs solving, which determines which reads share overlapping sequences and can be fused into contigs. This problem, called *de novo* assembly, is strongly influenced by the number and length of the available reads. Longer reads increase confidence in the correctness of assembled sequences, as does a high coverage, i.e. a high number of reads that are expected to span a genome position. Moreover, due to intra-species diversity, a reference genome is always a consensus and never fits one individuals genomic setup perfectly. Uncertainty in a determined base is expressed using IUPAC ambiguity codes, while longer variations need to be handled externally. Note that it is also possible to create reference genomes for a set of similar organisms, for example multiple strains of bacteria. These are called pan-genomes.[33]

Using a library of reference genomes, we can answer questions such as: For a given sequenced organism, to which species does it belong? Does it differ significantly from the reference genome and are the detected variations already known? Technologically, these

[31] Bashiardes, Zilberman-Schapira, and Elinav, "Use of Metatranscriptomics in Microbiome Research", 2016.

[32] The 1000 Genomes Project Consortium, "A Global Reference for Human Genetic Variation", 2015.

[33] Tettelin et al., "Genome Analysis of Multiple Pathogenic Isolates of Streptococcus Agalactiae: Implications for the Microbial "Pan-Genome"", 2005.

questions can be solved by *read mapping*, i.e. by assigning each read within the sequencing sample to a likely position in the reference. Depending on several factors, including the length and error rate of a read, as well as its sequence, one or more positions can be found. Finally, the read is aligned to the reference sequence. This means, the differences between read and reference are computed and evaluated using a scoring system, resulting in a mapping quality value. We will describe alignment methods in more detail in the chapter Computing and Approximating Resemblance and Containment.

Given mapped reads, we can identify differences between an organism and the reference, like SNVs or other mutations. For each of the steps detailed above, there are associated file formats commonly used in bioinformatics. Reference genomes are commonly stored as FASTA files, sequenced reads in FASTQ files, aligned reads in SAM (BAM, CRAM) files. Finally, VCF and BCF files describe differences between an organism and the reference genome. We will now describe these file types in detail.

2.4.8 File Formats

Several file formats have been established as de facto standards for biological sequences. We will now showcase the file formats that are relevant for this work, omitting SAM files, which only play a minor role. Regardless of technology, sequenced DNA is mostly stored as text files.

FASTA files, as described by Pearson and Lipman,[34] are used to store DNA and Protein sequence data in plain text. Each line in a FASTA file is either a name line, starting with the character > or a sequence line. A sequence, be it a chromosome, read, or protein, is encoded by a name line followed by one or more sequence lines. Usually, all sequence lines have the same length and are limited to 80 or 100 characters. An example of a FASTA file containing two chromosomes of length 120 and 160:

```
>Chromosome 1
TACATTTCCATAACAGCCATCGTATCATATCAATGTCGACGAGCCCTTGAAAGTCGATTA
GAATTTGCGATTTCCACAGTTACGCGTTTCAATTAGCCCGTTCATAGCATTAGTAAAGCT
>Chromosome 2
CCTGTTCATGTGAACTAAGGTGTAAGGCTTTTTTAGTCTCTGAGTACGCGCCCGGTGCTT
GTCAGGCAACCCCACAGCGACACGACTTATGCTTTCATTGGGGTGGGTTGCTTACTGCTT
ATACTGTGCATGGAATACAGTACGAGGGTCGTGACGAGCT
```

FASTQ files, as described by Cock,[35] also contain sequence data in plain text. In addition to the sequence, they also contain the quality values for each read and are hence used as output by sequencers. A FASTQ file can contain the four possible line types:

[34] Pearson and Lipman, "Improved Tools for Biological Sequence Comparison", 1988.

[35] Cock et al., "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants", 2009.

Name lines starting with `@`, followed by an identifier. This identifier can contain information about the sequencing process, such as the device used to sequence the read and the position on the flow cell.

Sequence lines containing the sequence.

Plus lines starting with `+`, which can contain an additional comment, but are usually empty.

Quality lines containing an ASCII-encoded Phred-score for each nucleotide in the corresponding sequence line.

An exemplary FASTQ file containing two reads with quality values is illustrated below:

```
@Read 1
TACATTTCCATAACAGCCATCGTATCATATCAATGTCGACGAGCCCTTGAAAGTCGATTA
+
3+=%(+6.70:;3(4$+*13=/.%'/$2!66%*/=+(92(=$1!!8.":;:<<8.""4'$
@Read 2
CCTGTTCATGTGAACTAAGGTGTAAGGCTTTTTTAGTCTCTGAGTACGCGCCCGGTGCTT
+
GEC=<=CG?CHG?GE>FHD;=?BD?>?:>>GIG@DG<;<@BICE=AE;E:G:F?E>@CD>
```

There are several formats for the name line, like the Illumina CASAVA format, which are specific to sequencing technologies. Additionally, several encodings for quality values are in use. We will use Illumina 1.8 PHRED+33 encoding, the agreed-upon defacto standard, which maps the the quality scores 0 - 41 to the ASCII range 33 - 74 (`!` - `J`).

Variant Call Format (VCF) files, described by Danecek et al.,[36] contain information about genetic variants. They can be used to describe the variants present in a set of samples with respect to a reference FASTA file. A VCF file contains a header section of lines starting with `##`, which contain meta information and define the entries in the `INFO` column and `FORMAT` column, the latter of which determines the layout of the `SAMPLE` columns. The header is followed by one line per variant, each of which contains the following information:

A chromosome and a position on said chromosome where the variant is located, followed by an identifier, if the variant is present in a variant database like dbSNP.[37] Next the reference sequence and sequences of alternative alleles are listed, followed by the Phred quality score for the presence of such an alternative allele. Finally, the sample fields contain entries that have been defined in the header and specified in the `FORMAT` column. Each of these fields is associated with one sample and can, for example, contain genotypes, haplotypes, and read coverage.

[36] Danecek et al., "The variant call format and VCFtools", 2011.

[37] Sherry et al., "dbSNP: the NCBI Database of Genetic Variation", 2001.

An exemplary VCF file is illustrated below, containing two SNPs (`C>T` and `A>(G or T)`), a deletion (`TG>T`), and an insertion (`AG>AGC`) for two individuals. For illustrative purposes, it contains only a small number of `INFO` and `FORMAT` fields and has been modified to fit the page. In this example, the first SNP is homozygously present (1) in both samples with a coverage of 15 each. The second SNP is homozygously present in two different alleles (1: `A>G`, 2: `A>T`), one in each individual. The deletion is only heterozygously expressed by `Individual1` while both alleles of `Individual2` show the reference allele (0). Finally, the insertion is present heterozygously in both individuals.

```
##fileformat=VCFv4.3
##fileDate=20190326
##reference=file://genome.fasta
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
#CHROM   POS   ID   REF   ALT   QUAL   FILTER   INFO   FORMAT   Individual1   Individual2
1        23    .    C     T     17     PASS     NS=2   GT:DP    1/1:15        1/1:15
1        42    .    A     G,T   23     PASS     NS=2   GT:DP    1/1:19        2/2:18
2        17    .    TG    T     55     PASS     NS=1   GT:DP    0/1:17        0/0:12
3        128   .    G     GC    31     PASS     NS=2   GT:DP    0/1:30        1/0:21
```

2.5 *Bioinformatic Basics*

In the previous sections we have already hinted at the mathematical representation of the biological sequences described there. Here we will aggregate and complete this information, starting with sets and sequences.

A set is an unordered collection of unique items, where the items can be from any domain, for example integers from $\mathbb{N}$ or individuals as in Section Allele Frequency (p. 17). Unordered means that the sets $A := \{1,2,3\}$ and $B := \{3,2,1\}$ are identical and $A = B$ holds. Unique items means that the sets $A := \{1,1,2,3\} = \{1,2,3\}$ and $B := \{1,2,3\}$ are identical and $A = B$ holds. By counting the number of items in a set A we get its size $|A|$. We use $[x] = \{0,\ldots,x-1\}$ to denote a range, i.e. a set of x integer numbers starting with 0. Finally, an empty set is denoted as $A := \{\ \} = \emptyset$

The three most prominent operations that can be performed with two sets are:

The union $A \cup B$ is the set of items that are contained in either A or B.

The intersection $A \cap B$ is the set of items that are contained in both A and B.

The difference $A \setminus B$ (resp. $B \setminus A$) is the set of items that are in A and not in B (in B but not in A).

A variant of sets are multisets, in which items can occur multiple times. This number of occurrences is called the multiplicity of an item. For example, in the multiset $\{1,1,2,3\}$ the item 1 occurs with multiplicity 2.

The smallest item in a set A—its minimum—is denoted as $\min A$. When referring to the set of the k smallest items in A, we use the short hand notation $\min^k A$, which is defined as:

$$\begin{aligned}\min^k A &:= \bigcup_{i=1}^{k} X_i\\ X_1 &= \min A\\ X_2 &= \min(A \setminus X_1)\\ &\dots\\ X_i &= \min((((A \setminus X_1) \setminus X_2) \setminus \dots) \setminus X_i)\end{aligned}$$

The maximum of a set A is denoted as $\max A$ and its k largest items as $\max^k A$ (defined analogously to $\min^k$).

Sequences are also collections of items but, in contrast to sets, sequences are both ordered

$$(1,2,3) =: A \neq B := (3,2,1)$$

and allow repetitions of items

$$(1,1,2,3) =: A \neq B := (1,2,3).$$

Finite sequences of characters chosen from a set of items are also called strings, where the item set is referred to as the alphabet Σ. The size of the alphabet is denoted as $|\Sigma| = \sigma$. For example, 32 bit words are strings of length 32 from the alphabet $\Sigma = \{0,1\}$ (or from Σ^{32} for short), where $\sigma = 2$. An empty sequence, or sequence of length 0, is denoted by the character ϵ:

$$A := \epsilon \qquad |A| = 0$$

The item at position i of a sequence A is $A_{[i]}$ and a subsequence beginning at position i and spanning the positions i to $j-1$ is denoted as $A_{[i:j]}$.

In the following section, the alphabets used in this work are described.

2.5.1 *Biological Alphabets*

As mentioned in the previous sections, different biological sequences stem from different alphabets. We already introduced the DNA alphabet $\Sigma_{\text{DNA}}, \sigma_{\text{DNA}} = 4$, amino acid alphabet $\Sigma_{\text{AA}}, \sigma_{\text{AA}} = 20$, and the extended amino acid alphabet $\Sigma_{\text{AA+}}, \sigma_{\text{AA+}} = 22$. While these alphabets allow the representation of all DNA and protein sequences, additional alphabets that model certain properties of biological sequences are also in use.

The IUPAC alphabet[38] extends the DNA alphabet by characters

[38] Cornish-Bowden, "Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984.", 1985.

Code	Bases	Code	Bases	Code	Bases
A	A	Y	C or T	B	C or G or T
C	C	S	G or C	D	A or G or T
G	G	W	A or T	H	A or C or T
T	T	K	G or T	V	A or C or G
R	A or G	M	A or C	N	any base

Table 2.2: IUPAC code table for DNA sequences. In the table for RNA sequences, all Ts are replaced with Us. Both contain $\sigma = 15$ characters.

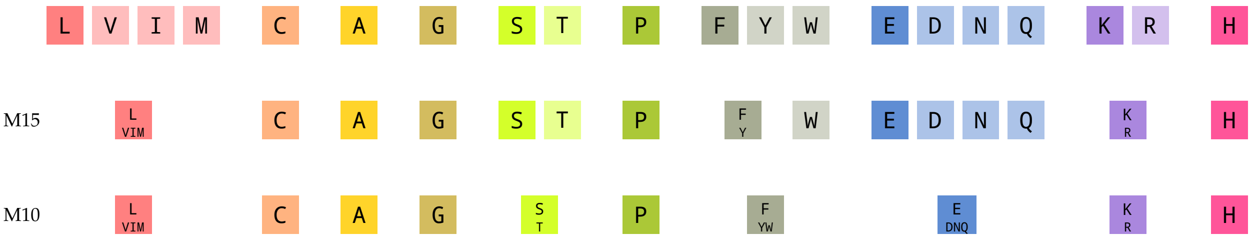


Figure 2.13: Structure of the M15 and M10 protein alphabets. Beginning with the top layer, the full amino acid alphabet, amino acid groups are merged to form smaller alphabets. Merged amino acids are denoted as smaller text within the square of their representative. Note that since the extended amino acids are very rare and chemically similar to their more common counterparts, they are not explicitly mentioned here.

that model more than one possible base, as illustrated in Table 2.2.

Using this alphabet, uncertainty, missing information, and ambiguous patterns can be modeled. For example, uncertain base calls from a sequencer are replaced with Ns and all codons for Q (CAA and CAG) can be expressed as CAR. The IUPAC alphabet Σ_{IUPAC} is an extended alphabet, as it introduces additional information.

In contrast to extended alphabets, reduced alphabets are applied as well. Most notably, for protein sequences the reduced representation alphabets introduced by Murphy et al.,[39] are in common use. The Murphy alphabets reduce Σ_{AA} by grouping amino acids with similar folding properties (see Section Protein Structures and Functions, p. 13), as illustrated in Figure 2.13.

If we need to distinguish between strings from specific alphabets, we denote the employed alphabet as a subscript behind the string. For example:

$$\texttt{ACGT}_{\text{DNA}} \qquad \texttt{ALYN}_{\text{AA}} \qquad \texttt{ALFN}_{\text{M10}}$$

Table 2.3 illustrates the alphabets most important for this work.

2.5.2 *Interaction of DNA and Protein Data*

When working with both DNA and protein sequences, DNA sequences are commonly translated into protein sequences. This is robust against mutations, since, for example all silent mutations still result in the same protein sequence. As mentioned in Section Structure of DNA and RNA Sequences (p. 9), each codon after the start codon is translated into one amino acid. This is modeled by the function

$$\Delta : \Sigma_{\text{DNA}}^{3\ell} \to (\Sigma_{\text{AA}} \cup \{ \blacktriangleright, \bullet \})^{\ell}$$

which translates a nucleotide sequence into its corresponding sequence of amino acids, start and stop codons. A problem that arises especially with SGS short reads is that the position of the start

[39] Murphy, Wallqvist, and Levy, "Simplified Amino Acid Alphabets for Protein Fold Recognition and Implications for Folding", 2000.

Sequence Type		Alphabet		Size
Bits	$\Sigma_{01} :=$	$\{$ 0, 1 $\}$	$\sigma_{01} =$	2
DNA Nucleotides	$\Sigma_{\text{DNA}} :=$	$\{$ A, C, G, T $\}$	$\sigma_{\text{DNA}} =$	4
RNA Nucleotides	$\Sigma_{\text{RNA}} :=$	$\{$ A, C, G, U $\}$	$\sigma_{\text{RNA}} =$	4
IUPAC Nucleotides	$\Sigma_{\text{INT}} :=$	$\{$ A, C, G, T, R, Y, S, W, K, M, B, D, H, V, N $\}$	$\sigma_{\text{INT}} =$	15
Amino Acids	$\Sigma_{\text{AA}} :=$	$\{$ A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y $\}$	$\sigma_{\text{AA}} =$	20
IUPAC Amino Acids	$\Sigma_{\text{IAA}} :=$	$\{$ A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y, ▶ , ● $\}$	$\sigma_{\text{IAA}} =$	22
Extended Amino Acids	$\Sigma_{\text{AA+}} :=$	$\{$ A, C, D, E, F, G, H, I, K, L, M, N, O, P, Q, R, S, T, U, V, W, Y $\}$	$\sigma_{\text{AA+}} =$	22
M10 Amino Acids	$\Sigma_{\text{M10}} :=$	$\{$ A, C, E, F, G, H, K, L, P, S $\}$	$\sigma_{\text{M10}} =$	10
M15 Amino Acids	$\Sigma_{\text{M15}} :=$	$\{$ A, C, D, E, F, G, H, K, L, N, P, Q, S, T, W $\}$	$\sigma_{\text{M15}} =$	15

Table 2.3: Comparison of the alphabets used in this work. In the Murphy alphabets, the extended amino acids are treated as their more common equivalents.

codon and consequently the reading frame is unknown. Hence, the correct offset of the translation can only be guessed by computing the six frame translation (SFT) of a read, as illustrated in Figure 2.14.

For a given read r, its six frame translation is the set

$$\begin{aligned} \text{sft}(r) = \{&\Delta(\phi(r,0)), \Delta(\phi(r,1)), \Delta(\phi(r,2)), \\ &\Delta(\phi(\bar{r},0)), \Delta(\phi(\bar{r},1)), \Delta(\phi(\bar{r},2))\} \end{aligned} \tag{2.6}$$

where the ϕ function

$$\phi(r,i) = r_{[i:s]} \qquad s := 3\left\lfloor \frac{|r|-i}{3} \right\rfloor + i \tag{2.7}$$

provides the reading frame for a sequence r, beginning at position i.

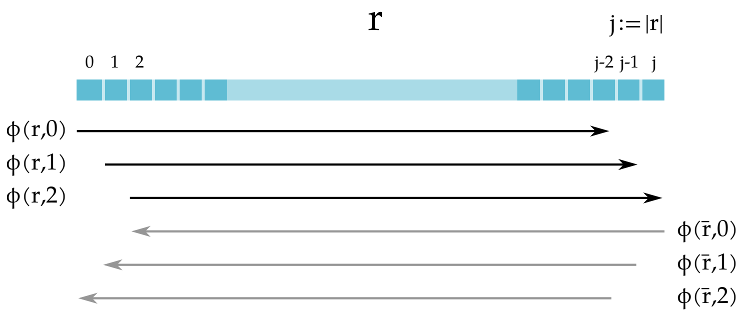


Figure 2.14: Six frame translation of a read r. The six reading frames comprise three forward reading frames (black arrows starting on the left) and three reverse complementary reading frames (gray arrows starting on the right). In this case the read length $j := |r|$ is divisible by 3, so that all translations have the same lengths. If this is not the case, the lengths of the translations can vary.

Since only one of the six reading frames can be present in an analyzed organism, this procedure can introduce amino acid sequences that are not actually produced by the organism. Additionally, some translations can be eliminated before analysis, based on the input data. Especially, when working with transcriptomic data, all sequences are subsequences of ORFs and hence do not contain stop codons (●). Eliminating all sequences that contain ●, can reduce the input data size and prevent the introduction of spurious proteins, as well as reduce the required runtime of analysis software.

For genomic data, sequences in the SFT that contain ● cannot be excluded, since they might contain untranslated regions. But

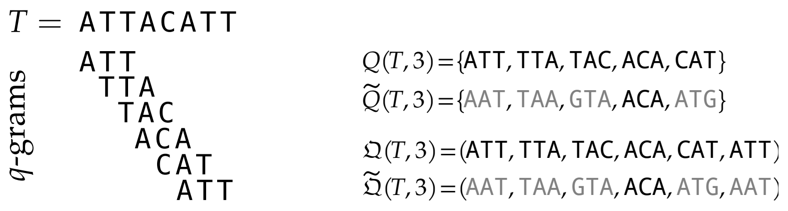

$T =$ ATTACATT
q-grams
ATT
TTA
TAC
ACA
CAT
ATT
$Q(T,3) = \{$ATT, TTA, TAC, ACA, CAT$\}$
$\widetilde{Q}(T,3) = \{$AAT, TAA, GTA, ACA, ATG$\}$
$\mathfrak{Q}(T,3) = ($ATT, TTA, TAC, ACA, CAT, ATT$)$
$\widetilde{\mathfrak{Q}}(T,3) = ($AAT, TAA, GTA, ACA, ATG, AAT$)$

Figure 2.15: The 3-gram set and sequence for the text $T =$ ATTACATT. The 3-gram ATT occurs twice in the text. It is present twice in $\mathfrak{Q}(T,3)$, but only once in $Q(T,3)$. For the canonical set and sequence, 3-grams that were replaced by their reverse complemented are highlighted in gray.

such sequences can be truncated after ● to exclude untranslated sequences.

2.5.3 *q-Gram Sets and Sequences*

For many applications in bioinformatics it is helpful, or even required, to split large strings up into smaller substrings. Among other benefits, this approach reduces the impact of sequencing errors and mutations on text comparison algorithms.

When a string T is split up into all its substrings of length q, these are called the q-grams of T.[40] We define the q-gram set of a string T as:

$$Q(T,q) := \{\, T_{[i:i+q]} \mid i \in [0, |T| - q] \,\} \tag{2.8}$$

This is equivalent to moving a window of size q through the text and emitting all characters covered by this window.

If we need to retain the order of the q-grams, we denote the q-gram sequence of T as:

$$\mathfrak{Q}(T,q) := (g_i)_0^{|T|-q} \quad g_i := T_{[i:i+q]} \tag{2.9}$$

Note that the input text T can be reconstructed from $\mathfrak{Q}(T,q)$, while $Q(T,q)$ lacks both the positional information and the cardinality information required for this task.

Since the orientation of a DNA string with respect to a reference might not be known, canonical q-grams are used to resolve this uncertainty. A canonical q-gram is the minimum of a q-gram and its reverse complement under a given order, e.g. lexicographical order or an integer encoding. When used consistently, canonical q-grams from the reference sequence and the read are identical. Hence we define the canonical q-gram set

$$\widetilde{Q}(T,q) := \{\, \kappa(T_{[i:i+q]}) \mid i \in [0, |T| - q] \,\} \tag{2.10}$$

and canonical q-gram sequence respectively

$$\widetilde{\mathfrak{Q}}(T,q) := (g_i)_0^{|T|-q} \quad g_i := \kappa(T_{[i:i+q]}) \tag{2.11}$$

for a function

$$\kappa(g) := \begin{cases} g & \text{if } g < \bar{g} \\ \bar{g} & \text{else} \end{cases} \tag{2.12}$$

which yields the minimum of a q-gram and its complement. The 3-gram set and sequence of the text $T =$ ATTACATT are illustrated in

[40] Depending on the discipline, other names are common, including: n-grams, k-mers, and w-shingles.

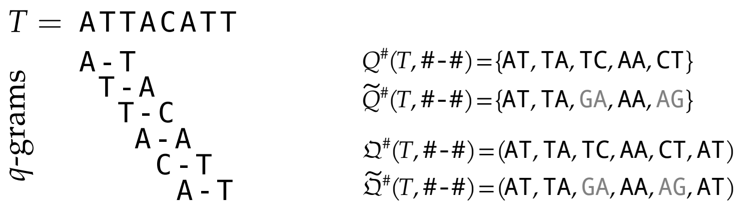

$T =$ ATTACATT
q-grams
A - T
T - A
T - C
A - A
C - T
A - T
$Q^{\#}(T,\texttt{\#-\#}) = \{\texttt{AT}, \texttt{TA}, \texttt{TC}, \texttt{AA}, \texttt{CT}\}$
$\widetilde{Q}^{\#}(T,\texttt{\#-\#}) = \{\texttt{AT}, \texttt{TA}, \texttt{GA}, \texttt{AA}, \texttt{AG}\}$
$\mathfrak{Q}^{\#}(T,\texttt{\#-\#}) = (\texttt{AT}, \texttt{TA}, \texttt{TC}, \texttt{AA}, \texttt{CT}, \texttt{AT})$
$\widetilde{\mathfrak{Q}}^{\#}(T,\texttt{\#-\#}) = (\texttt{AT}, \texttt{TA}, \texttt{GA}, \texttt{AA}, \texttt{AG}, \texttt{AT})$

Figure 2.16: The gapped $(3,2)$-gram set and sequence for the text $T =$ ATTACATT with the shape $\{0,2\} =$ #-#. For the canonical set and sequence, $(3,2)$-grams that were replaced by their reverse complemented are highlighted in gray.

Figure 2.15. Implementing such a function is especially easy if the q-grams are encoded as integer values.

Finally, there are variants of q-grams that change the set of incorporated characters. So far, we looked at solid q-grams where all characters within the q-gram were adjacent in the source text. However, it can be beneficial to use gapped q-grams,[41] which use a shape of care- and don't-care positions to generate the q-gram text.[42] Consider a shape (or mask) $m = \{0,2\} =$ #-#, where # denotes a care-position and - denotes a don't-care position that is moved through the text as with solid q-grams (see Figure 2.16). For each start position only the characters from T that are covered by care-positions are incorporated into the q-gram, the don't-care positions are ignored.

We define the set of gapped (q,k)-grams with a shape m containing k care-positions and a total length of q as

$$Q^{\#}(T,m) := \{\, g_i \mid i \in [0, |T| - q] \,\} \qquad g_i := (T_{[i+m_j]})_{j=0}^{k-1} \tag{2.13}$$

and the sequence of gapped q-grams as

$$\mathfrak{Q}^{\#}(T,m) := (g_i)_0^{|T|-q} \quad g_i := (T_{[i+m_j]})_{j=0}^{k-1} \tag{2.14}$$

Canonical gapped q-gram set $\widetilde{Q}^{\#}(T,m)$ and sequence $\widetilde{\mathfrak{Q}}^{\#}(T,m)$ are defined analogously, using the minimum of the gapped and reverse gapped q-grams.

Gapped q-grams offer resilience to substitution errors and SNVs for sequence analysis. For solid q-grams, one substitution error affects q consecutive q-grams, while a (q,k)-gram is able to bridge this gap.

2.5.4 Memory Efficient Storage of q-Grams

To save memory while allowing easy comparisons, q-grams can be encoded as integer numbers. The most common way to do this is 2-bit encoding, which assigns each base a 2-bit integer value, for example:

$$\texttt{A} := 00_2, \quad \texttt{C} := 01_2, \quad \texttt{G} := 10_2, \quad \texttt{T} := 11_2 \tag{2.15}$$

A q-gram g is then encoded in a $2q$-bit bit-vector where the bit positions $b(g)_{[2i-1]}$ and $b(g)_{[2i]}$ contain the bit encoding of the base $g_{[i]}$. Example:

$$\begin{array}{rccccccl} g = & \texttt{T} & \texttt{T} & \texttt{A} & \texttt{C} & \texttt{G} & & \\ b(g) = & 11_2 & 11_2 & 00_2 & 01_2 & 10_2 & & = 1111000110_2 = 966_{10} \end{array}$$

[41] Burkhardt and Kärkkäinen, "Better Filtering with Gapped q-Grams", 2001.

[42] Depending on discipline, gapped q-grams are also known as spaced seeds.

Blocks of two bits per base are concatenated, forming the bit vector 1111000110_2, which is equivalent to the decimal number 966.

The 2-bit encoding allows saving a DNA sequence $T \in \Sigma_{\text{DNA}}^n$ using $2n$ bits of memory. To model IUPAC sequences, four bits per character are required:

$$\begin{array}{llll} \texttt{A} := 0001_2, & \texttt{C} := 0010_2, & \texttt{G} := 0100_2, & \texttt{T} := 1000_2 \\ \texttt{R} := 0101_2, & \texttt{Y} := 1010_2, & \texttt{S} := 0110_2, & \texttt{W} := 1001_2 \\ \texttt{K} := 1100_2, & \texttt{M} := 0011_2, & \texttt{B} := 1110_2, & \texttt{D} := 1101_2 \\ \texttt{H} := 1011_2, & \texttt{V} := 0111_2, & \texttt{N} := 1111_2 & \end{array} \tag{2.16}$$

Note that the presence of a 1-bit in the 4-bit bit vector for each character from Σ_{IUPAC} represents the possible presence of A, C, G, or T. For example, the base denoted by $\texttt{S} := 0110_2$, can be either C or G. The ability to model ambiguous bases comes at the expense of needing two additional bits per character, giving a 4-bit encoded IUPAC sequence $T \in \Sigma_{\text{IUPAC}}^n$ a memory footprint of $4n$ bits.

For (reduced) protein alphabets, no widely established bit-encoding scheme is available. We implemented the following encoding of the Σ_{M15} alphabet for use in our applications on protein sequences.

$$\begin{array}{llll} \texttt{L} := 0001_2, & \texttt{S} := 0101_2, & \texttt{W} := 1001_2, & \texttt{Q} := 1101_2 \\ \texttt{C} := 0010_2, & \texttt{T} := 0110_2, & \texttt{E} := 1010_2, & \texttt{K} := 1110_2 \\ \texttt{A} := 0011_2, & \texttt{P} := 0111_2, & \texttt{D} := 1011_2, & \texttt{H} := 1111_2 \\ \texttt{G} := 0100_2, & \texttt{F} := 1000_2, & \texttt{N} := 1100_2 & \end{array} \tag{2.17}$$

The value 0000_2 is reserved for unknown values.

Another convenient aspect of integer encoded q-grams is that we can use integer hash functions to process them rather than being restricted to hash functions that work on byte streams.

We do not use a specific notation for integer-encoded sequences. Rather, when we need to treat a q-gram as an integer number, we implicitly assume the encoding detailed above. If not noted otherwise, nucleotide sequences are encoded with 2-bit encoding.

2.6 Probability Distributions

In this section, based on the supplementary material to our article on DDRAGE,[43] we describe probability distributions used in this work. A random variable X that is distributed as a probability distribution D is denoted as $X \sim D$.

[43] Timm et al., "ddRAGE: A Data Set Generator to Evaluate ddRADseq Analysis Software", 2018.

2.6.1 Discrete Uniform Distribution (DUD)

From a consecutive range of the n numbers from a to b, where $n = b - a + 1$, all numbers are chosen with the same probability $1/n$. Hence, the probability mass function is constant $\mathbb{P}(X = k) = 1/n$ for all $k \in \{a, a+1, \ldots, b\}$. The mean and variance of the DUD are

$$\mathbb{E}(X) = \frac{a+b}{2} \quad \text{and} \quad \operatorname{Var}(X) = \frac{(b-a+1)^2 - 1}{12} \quad \text{for} \quad X \sim \operatorname{DUD}(a, b). \tag{2.18}$$

2.6.2 *Poisson Distribution (PD)*

A Poisson distribution can be used to model the number of occurrences of independent events with a fixed probability over an interval of time. The distribution uses one parameter λ, which is the expected number of events in the interval. If $X \sim \mathrm{PD}(\lambda)$ for some $\lambda \geq 0$, then the probability mass function is

$$\mathbb{P}(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \qquad k \in \mathbb{N}_0, \tag{2.19}$$

with mean and variance of

$$\mathbb{E}(X) = \lambda \quad \text{and} \quad \mathrm{Var}(X) = \lambda\,. \tag{2.20}$$

2.6.3 *Zero-Truncated Poisson Distribution (ZTPD)*

This variant of the PD only yields values above zero and is equivalent to sampling from a PD, but rejecting and re-drawing all zeros. If $X \sim \mathrm{ZTPD}(\lambda)$, then

$$\mathbb{P}(X = k) = \frac{\lambda^k}{(e^{\lambda} - 1) \cdot k!} \quad \text{for } k \in \mathbb{N}_{\geq 1}\,, \tag{2.21}$$

with mean and variance of

$$\mathbb{E}(X) = \frac{\lambda e^{\lambda}}{e^{\lambda} - 1} \qquad \mathrm{Var}(\mathrm{X}) = \mathbb{E}[\mathrm{X}](1 + \lambda \quad \mathbb{E}[\mathrm{X}]). \tag{2.22}$$

2.6.4 *Binomial Distribution (BD)*

The binomial distribution requires a parameter n that specifies a number of experiments executed and a probability p of each experiment to be successful, independently of the other experiments. If $X \sim \mathrm{BD}(n, p)$, then

$$\mathbb{P}(X = k) = \binom{n}{k} p^k (1 - p)^{n-k} \tag{2.23}$$

with mean and variance of

$$\mathbb{E}(X) = np, \qquad \mathrm{Var}(X) = np(1 - p). \tag{2.24}$$

2.6.5 *Beta Distribution (Beta)*

The beta distribution has two shape parameters $\alpha > 0$ and $\beta > 0$, which control the shape of its left and right tailing respectively on the continuous interval $(0, 1)$.

If $X \sim \mathrm{Beta}(\alpha, \beta)$, then

$$\mathbb{P}(X = x) = \frac{x^{\alpha-1}(1 - x)^{\beta-1}}{B(\alpha, \beta)} \qquad x \in (0, 1), \tag{2.25}$$

where $B(\alpha, \beta)$ is the Beta function, which is defined as

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)} \tag{2.26}$$

where Γ is the Gamma function. The mean and variance of the beta distribution are

$$\mathbb{E}(X) = \frac{\alpha}{\alpha + \beta}, \qquad \mathrm{Var}(X) = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}\,. \tag{2.27}$$

2.6.6 *Beta-Binomial Distribution (BBD)*

The BBD has three parameters: the shape parameters $\alpha > 0$ and $\beta > 0$, which control the shape of left and right tailing respectively, and $n \in \mathbb{N}_0$, which signifies the maximum number of events. A sample from this distribution arises as follows: First, a success probability p is sampled from the beta distribution with parameters α and β. Then X is drawn from a Binomial distribution with parameters n and the random p.[44]

If $X \sim \text{BBD}(\alpha, \beta, n)$, then

$$\mathbb{P}(X = k) = \binom{n}{k} \frac{B(k + \alpha, n - k + \beta)}{B(\alpha, \beta)} \tag{2.28}$$

where $B(\alpha, \beta)$ is the Beta function (see equation 2.26). The mean and variance of a BBD are

$$\mathbb{E}(X) = \frac{n\alpha}{\alpha + \beta}, \qquad \text{Var}(X) = \frac{n\alpha\beta(\alpha + \beta + n)}{(\alpha + \beta)^2(\alpha + \beta + 1)}\,. \tag{2.29}$$

2.7 *Caching*

Many decisions in algorithm engineering are informed by memory latencies and the use of caches to alleviate them. The behavior of memory hierarchies, as well as commonly used caching strategies, have been thoroughly described by Drepper,[45] and we will only sum up the aspects most important for this work. This section is based solely on Drepper's work and should only serve to provide some level of intuition to discern cache-friendly and cache-unfriendly operations. Furthermore, we omit the concept of tiered cache architectures for the sake of simplicity and just refer to all cache layers as *the cache*.

In order to work with any form of data, a processor needs to load it into processor registers, which are small both in number and memory capacity. Loading data from main memory, or even from a HDD, is orders of magnitude slower than the processors clock cycles.[46] This can lead to the processor waiting for data to arrive for several hundred cycles, while no actual work can be performed. As a rule of thumb, contemporary memory can either be large or fast, but not both.

To reduce the time a processor spends in wait state, modern processor architectures maintain a tiered architecture of small and fast memory units, so called *caches*. When the processor reads data, say a 32-bit integer, from memory for the first time, this integer value and its neighborhood of values in memory are copied to a cache (refer to Figure 2.17 for an illustration).

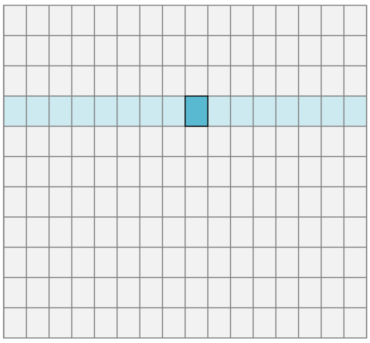


Figure 2.17: Visualization of a computer's main memory. Each cell represents a 32-bit integer value, aligned into cache lines of $16 \cdot 4$ bytes $= 64$ bytes. When the value 🟦 is requested, its whole cache line ▭ is loaded into the cache.

Subsequently, this integer can be accessed quickly from the fast cache memory. Since caches are small, old values are removed and overwritten if space is required for newer entries.

The smallest amount of data that caches work with is called a *cache line*, which is a contiguous slice of (main) memory. Cache line size can vary between processor architectures, however, typical

[44] Johnson, Kemp, and Kotz, *Univariate Discrete Distributions*, 2005.

[45] Drepper, *What Every Programmer Should Know About Memory*, 2007.

[46] Loading data from main memory takes $\sim$240 cycles, accessing a register ≤ 1 cycle.

sizes vary around 64 bytes per cache line, i.e. 16 32-bit integers or 8 64-bit integers. If the processor tries to access memory, it queries the cache if the desired cache line is already present in cache and can be accessed quickly. This case is called a *cache hit*. Accessing an item that is not yet contained in cache is called a *cache miss* and triggers loading its containing line into cache. Cache misses that arise when data is accessed for the first time are called *compulsory cache misses*.

The amount at which caching improves the performance of a program depends on the kind of memory accesses this program performs. We denote operations that benefit strongly from caching as *cache-friendly* in contrast to *cache-unfriendly* operations which cannot benefit from caching. For good cache performance, the following three aspects are crucial:

Memory locality : If the accessed data is saved contiguously in memory, there is a higher chance that a subsequent access can benefit from an already cached line.

Temporal locality : Since with each cache miss an old cache line is overwritten, cached data only remains relevant for a limited time. A typical L1 cache holds about 2^{10} cache lines.

Predictability : If the access pattern is predictable, compilers and processors can optimize the generated and executed machine code to better leverage caching effects (prefetching).

Consider an integer array $a = (a_i)_{i=0}^{n}$, $n \gg |\text{Cache}|$. Iterating through a using a for-loop is very cache-friendly, since all entries are contained within contiguous cache lines (memory local), which are accessed one after another (temporally local) in a linear fashion (predictable). On the other end of the spectrum, randomly indexing of such an array is cache-unfriendly, since lines in the cache only have a slim chance to be used again before being overwritten.

Another advantage of predictable access patterns is that it allows compiler and processor to *prefetch* cache lines. This means preemptively loading cache lines that have not been accessed yet. Processors can detect linear access patterns of contiguous (see Figure 2.18) or spaced cache lines (see Figure 2.19) and automatically prefetch upcoming lines (*hardware prefetching*). For more complex patterns, *software prefetching* can be used to explicitly prefetch cache lines. This can either be employed by the compiler or manually by the user.

When designing algorithms, taking caching effects into account can greatly increase performance. However, this is not trivially possible for all problems, especially if the core idea of the algorithm requires random accesses, as for example for hash tables.[47]

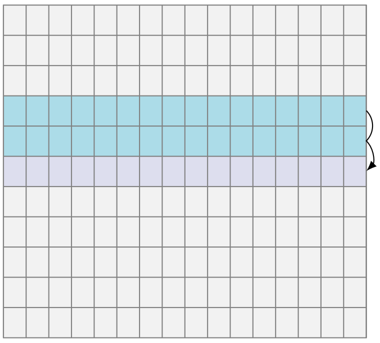


Figure 2.18: Illustration of hardware prefetching for linear accesses. After the first two lines 🟦 have been accessed, the third line 🟪 can be prefetched.

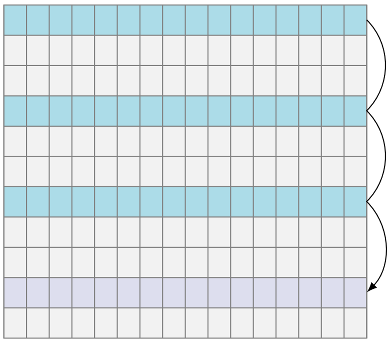


Figure 2.19: Illustration of hardware prefetching for spaced linear accesses. Spaced accesses require an additional access to be detected. After the first three lines 🟦 have been accessed, the fourth line 🟪 can be prefetched.

[47] Heileman and Luo, "How Caching Affects Hashing", 2005.

3
Hashing in Bioinformatics

Hashing has a wide range of applications in virtually all fields of computer science. Abstractly speaking, a hash function takes items from a (potentially infinite) set of possible items and translates them into integer values. These computed hash values can for example be used to efficiently compare items, since integer comparisons are more efficient than, say, string comparisons. One application of this is the comparison of files using checksums. However, the main use of hash values is to enter the items into a hash table, from which they can be efficiently retrieved, using the hash values as addresses. Hash tables are employed in virtually all fields of bioinformatics, ranging from text indexing problems and q-gram counting to more technical applications like the association of chromosome names to sequences.

As we will highlight in this chapter, the implementation of a hash table does not comprise solely of a hash function, but also requires choosing an underlying data structure and a collision resolution strategy. Furthermore, several applications of hashing require different properties of hash functions to uphold their expected performance.

3.1 Hash Functions

We define a hash function $h : \mathcal{D} \rightarrow \mathcal{C}$ as a function that assigns items from the input set (domain) $\mathcal{D}$ to integer values from a target set (codomain) $\mathcal{C} \subset \mathbb{N}_0$. For a given item $x \in \mathcal{D}$, we call $h(x) \in \mathcal{C}$ the hash value of x. An Illustration of such a function can be found in Figure 3.1. Usually, the codomain is smaller than the domain, so that there are less possible hash values than items (keys) in $\mathcal{D}$. Hash functions mostly are one-way functions, i.e. for a given hash value

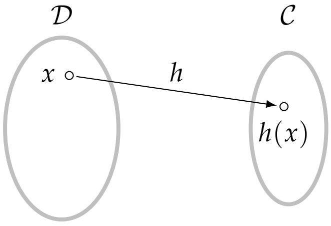


Figure 3.1: Illustration of a hash function h, mapping an item x from the domain to a hash value $h(x)$ in the codomain.

$h(x)$ it is impossible (or computationally infeasible) to retrieve x. At this point we do not consider any restriction for the items of $\mathcal{D}$. As a shorthand, we define a *general purpose hash function* as a function that maps arbitrary items to an integer range $[|\mathcal{C}|]$. Further, we define a family of hash functions as a set $\mathcal{H} := \{h_p\}$, where p is a parameter allowing the generation of different hash functions.

If two different keys $x_0, x_1 \in \mathcal{D},\ x_0 \neq x_0$ receive the same hash value $h(x_0) = h(x_1)$ from a given hash function h, these items cause a collision (see Figure 3.2). Collision properties of a hash function are one of the most crucial factors which determine their potential use. For most applications, collision resolution strategies are employed, which we will highlight later in this chapter.

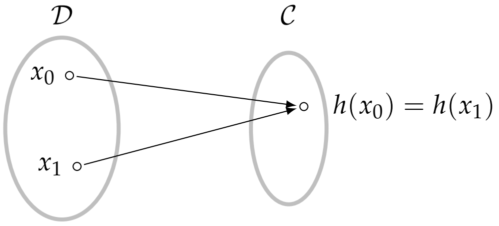


Figure 3.2: Illustration of a collision in hash function h. Both items x_0 and x_1 are mapped onto the same hash value $h(x_0) = h(x_1)$.

In the following sections, we will describe collision properties of hash functions as well as other aspects that inform the choice of a hash function for a specific application.

3.1.1 Universality, Independence, and Min-Wise Independence

The concept of universal hashing was introduced by Carter and Wegman[1] and describes using randomly chosen hash functions from a family of hash functions. Universal hash functions are employed where different hash values for the same keys need to be computed, for example for MinHashing.[2] To be feasible for this and other applications, hash functions chosen from a family are classified by the following properties:

Universal: Hash values are uniformly distributed over $\mathcal{C}$.

Independent: For a set of items chosen from $\mathcal{D}$, their hash values behave like independent random variables.

Min-Wise Independent: For a set of items chosen from $\mathcal{D}$, all have the same probability to receive the numerically smallest hash value.

The most basic constraint is universality. To be universal, a hash function family needs to satisfy Definition 3.1.1.

Definition 3.1.1. *A family $\mathcal{H}$ of hash functions is (truly)* ***universal****, if for a hash function h, chosen uniformly at random from $\mathcal{H}$, the following condition holds:*

$$\mathbb{P}(h(x) = i) \leq \frac{1}{|\mathcal{C}|} \quad x \in \mathcal{D}, i \in \mathcal{C}$$

[1] Carter and Wegman, "Universal Classes of Hash Functions", 1979.

[2] See Section Computing and Approximating Resemblance and Containment, p. 89.

For any key chosen from $\mathcal{D}$, the chance of it hashing to a specific hash value i using a randomly selected hash function $h \in \mathcal{H}$ has to be at most $\frac{1}{|\mathcal{C}|}$. This allows expressing constraints for the collision probabilities, depending on the number of observed keys. The probability of two keys colliding when hashed by a universal hash function is at most $\frac{1}{|\mathcal{C}|}$. If Definition 3.1.1 is satisfied, let $i \in |\mathcal{C}|$ be (fixed) hash value, the probability to choose the same hash value for another key is at most $\frac{1}{|\mathcal{C}|}$. This property is also called weak universality.[3]

Definition 3.1.2. *A family $\mathcal{H}$ of hash functions is **weakly universal**, if for a hash function h, chosen uniformly at random from $\mathcal{H}$, the following condition holds:*

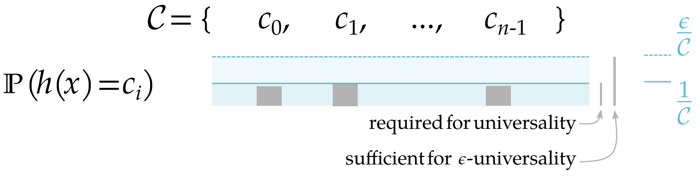
$$\mathbb{P}(h(x_0) = h(x_1)) \leq \frac{1}{|\mathcal{C}|} \quad x_0 \in \mathcal{D}, x_1 \in \mathcal{D}, x_0 \neq x_1$$

A relaxed version of universality is ϵ-almost universality,[4] which allows hash values to collide with an error factor ϵ, as described in Definition 3.1.3. Thorup[5] points out that this is sufficient for many applications.

Definition 3.1.3. *A family $\mathcal{H}$ of hash functions is **ϵ-almost universal** for $\epsilon \in \mathbb{R}_+, \epsilon > 1$, if for a hash function h, chosen uniformly at random from $\mathcal{H}$, the following condition holds:*

$$\mathbb{P}(h(x) = i) \leq \frac{\epsilon}{|\mathcal{C}|} \quad x \in \mathcal{D}, i \in \mathcal{C},$$

The difference between true universality and ϵ-almost universality is illustrated in Figure 3.3.



$\mathcal{C}=\{ \quad c_0, \quad c_1, \quad \ldots, \quad c_{n\text{-}1} \quad \}$
$\mathbb{P}(h(x)=c_i)$
required for universality
sufficient for ϵ-universality
$\frac{\epsilon}{\mathcal{C}}$
$\frac{1}{\mathcal{C}}$

Figure 3.3: Illustration of the difference between true and ϵ-almost universality. The probability of an element c_i being chosen as hash value $h(x)$ is symbolized by the height of the gray blocks. For true universality, all items from $\mathcal{C}$ must have a probability of at most $1/|\mathcal{C}|$ to be a hash value for x, i.e. fall beneath the solid line. For ϵ-almost universality, the probabilities are allowed to fall below the dashed line (a multiple of $1/|\mathcal{C}|$).

In addition to collision probabilities of items, another important property is the distribution of hash values. Consider the hash function family $\mathcal{H} := \{ h \}$ containing only the identity hash function $h : \mathcal{D} \to \mathcal{D}$ with $h(x) = x$. This family is weakly universal, since any two distinct keys x and x' have a probability of $0 \leq |\mathcal{D}|$ which is also the codomain of h in this case. While keys do not collide when hashed with h, the distribution of hash values for different keys throughout the table is not well distributed: Its hash values are not independent.

Wegman and Carter[6] define k-independent[7] hash functions families that uniformly distribute the hash values of k different items throughout $\mathcal{C}$.

[3] Broder et al., "Min-Wise Independent Permutations", 1998.

[4] Stinson, "Universal Hashing and Authentication Codes", 1994.

[5] Thorup, "High Speed Hashing for Integers and Strings", 2015.

[6] Wegman and Carter, "New Hash Functions and Their Use in Authentication and Set Equality", 1981.

[7] Wegman et al. use the term strongly universal$_k$ instead of k-independent. The names k-strongly universal and k-wise independent are also used.

Definition 3.1.4. *A family $\mathcal{H}$ of hash functions is k-**independent**, if for a hash function h, chosen uniformly at random from $\mathcal{H}$, the following condition holds:*

$$\mathbb{P}(\bigwedge_{j=0}^{k-1} h(x_j) = i_j) = \frac{1}{|\mathcal{C}|^k} \quad (x_0, \dots, x_{k-1}) \in \mathcal{D}^k, (i_0, \dots, i_{k-1}) \in \mathcal{C}^k$$

If Definition 3.1.4 holds, hash values from h are uniformly distributed within $\mathcal{C}$. This property is required by several collision resolution strategies, as we will detail in one of the following sections.

Analogously to ϵ-almost universality, a relaxed variant of k-independence was introduced by Siegel.[8]

Definition 3.1.5. *A family $\mathcal{H}$ of hash functions is (ϵ, k)-**independent** for an error term $\epsilon > 1$, if for a hash function h, chosen uniformly at random from $\mathcal{H}$, the following condition holds:*

$$\mathbb{P}(\bigwedge_{j=0}^{k-1} h(x_j) = i_j) \leq \frac{\epsilon}{|\mathcal{C}|^k} \quad (x_0, \dots, x_{k-1}) \in \mathcal{D}^k, (i_0, \dots, i_{k-1}) \in \mathcal{C}^k$$

The difference between k-independence and (ϵ, k)-independence is illustrated in Figure 3.4. Note, however, that all $(1, k)$-independent hash function are also k-independent (see Corollary 3.1.6).

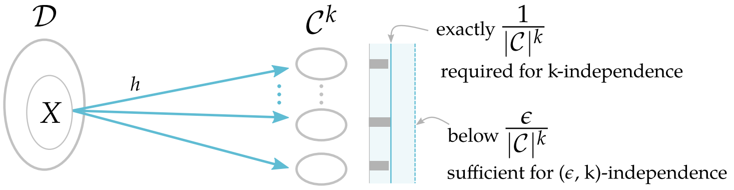

$\mathcal{D}$
$\mathcal{C}^k$
h
X
exactly $\frac{1}{|\mathcal{C}|^k}$
required for k-independence
below $\frac{\epsilon}{|\mathcal{C}|^k}$
sufficient for (ϵ, k)-independence

Figure 3.4: Illustration of the difference between k-independence and (ϵ, k)-independence. The probability of set X of keys being mapped to a tuple from $\mathcal{C}^k$ (column of ellipses in the center) by a randomly chosen hash function h (teal arrows) is illustrated by the gray boxes on the right. For k-independence, all tuples must have exactly a probability of $1/|\mathcal{C}|^k$, i.e. fall onto the teal line. For (ϵ, k)-independence, the probabilities are allowed to fall below the dashed line (a multiple of $1/|\mathcal{C}|^k$).

Corollary 3.1.6. *A k-independent family of hash functions is also $(1, k)$-independent and vice versa.*

Proof. Choose $\epsilon = 1$ for Definition 3.1.5. □

For use in many applications, like the computation of hash table addresses, ϵ-almost universal and (ϵ, k)-independent hash function are sufficient. Some examples of this will be highlighted in the Section Collision Resolution (p. 56). However, there are use cases that require stronger properties.

As mentioned above, one such use case is MinHashing, which we describe in detail in Chapter Computing and Approximating Resemblance and Containment (p. 89). MinHashing[9] uses minimal (as in smallest) hash values of sets of keys to estimate their similarity. This approach requires that minimal hash values are sampled uniformly from these sets using min-wise independent universal hash functions. Under a hash function from a min-wise independent family, all keys from a set $X \subseteq \mathcal{U}$ have the same probability to receive the minimal hash value.

[8] Siegel, "On Universal Classes of Extremely Random Constant-Time Hash Functions", 2004.

[9] Broder, "On the Resemblance and Containment of Documents", 1997.

Definition 3.1.7. *A family $\mathcal{H}$ of hash functions is **exactly min-wise independent** if for a hash function h chosen uniformly at random from $\mathcal{H}$, any subset $X \subseteq \mathcal{U}$, and any $x \in X$ the following condition holds:*

$$\mathbb{P}(\min\{\,X'\,\} = h(x)) = \frac{1}{|X|} \qquad X' := \{\,h(x) \mid x \in X\,\}$$

In other words, under a hash function from $\mathcal{H}$, all items in an input set, for example a set of q-grams, receive the minimal hash value with the exact same probability $\frac{1}{|X|}$. As Broder et al.[10] pointed out, designing such a function is practically unfeasible, as the size of exactly min-wise independent permutations grow exponentially with the size of $\mathcal{U}$. For practical applications, it is sufficient that items receive the minimal hash value with *almost* equal probability:

Definition 3.1.8. *A family $\mathcal{H}$ of hash functions is **ϵ-approximately min-wise independent** if for a hash function h chosen uniformly at random from $\mathcal{H}$, any subset $X \subseteq \mathcal{U}$, any $x \in X$, and an error ϵ the following condition holds:*

$$\left|\mathbb{P}(\min\{\,X'\,\} = h(x)) - \frac{1}{|X|}\right| \leq \frac{\epsilon}{|X|} \qquad X' = \{\,h(x) \mid x \in X\,\}$$

This allows keys to be chosen with a probability that deviates from a uniform distribution by at most ϵ, as illustrated in Figure 3.5. Since ϵ-approximately min-wise independent hash functions are used for all practical purposes, we use the shorter name ϵ-min-wise independent.

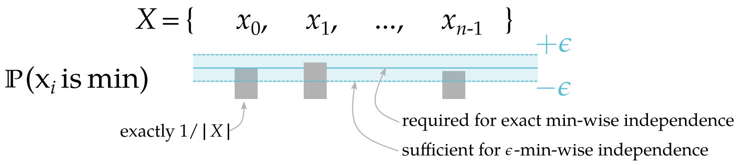

$X = \{ \quad x_0, \quad x_1, \quad ..., \quad x_{n\text{-}1} \quad \}$
$+\epsilon$
$\mathbb{P}(x_i \text{ is min})$
$-\epsilon$
required for exact min-wise independence
exactly $1/|X|$
sufficient for ϵ-min-wise independence

Figure 3.5: Illustration of the difference between exact and ϵ-min-wise independence. Probabilities are symbolized by the height of the gray blocks. For exact min-wise independence, all items from X must have the exact same probability ($\frac{1}{|X|}$) to be selected as minimal hash value, i.e. fall on the solid line. For ϵ-min-wise independence, the probabilities are allowed to fall between the dashed lines.

While the properties detailed above are crucial for this work, hash function can be classified into a myriad of other groups.

3.1.2 A Taxonomy of Hash Functions

In addition to the properties detailed above, hash functions are required to possess certain properties depending on their application, some of which are mutually exclusive. Consequently, it is not easily possible to define what an objectively *good* hash function is without a given application. In this section we give an overview of specific use cases, introduce classes that have strong restrictions, and point out which are used in a bioinformatics context.

Most notably, *cryptographic* and *non-cryptographic* hash functions are distinguished. A cryptographic hash function is required to be

[10] Broder et al., "Min-Wise Independent Permutations", 1998.

preimage-resistant (for a given hash value $h(x)$ it is computationally hard to find x or another key x' so that $h(x') = h(x)$) and *collision-resistant* (for a given key x it is computationally hard to find another key x' with the same hash value $h(x) = h(x')$)[11]. Cryptographic hash functions, like MD5,[12] and the SHA-2[13] and SHA-3[14] families, are commonly used to validate the identity of files or store user passwords in a database[15]. In most bioinformatics applications however, hashes are not required to be cryptographic.

Avalanching describes how different the hash values of similar, but different keys are under a given hash function. For example, if one bit is flipped in the input, how many bits change in the hash value? The highest amount of avalanching, described by the strict avalanche criterion,[16] requires each output bit to flip with a probability of 0.5. Hash functions with low avalanching allow to guess similar hash values and are more likely to be less cryptographically secure.

On the other end of the spectrum, two way hash functions allow reversing the hashing process, i.e. they allow to reconstruct keys from hashes. For a *reversible* (or *two-way*) hash function h there exists an inverse function h^{-1}, so that $h^{-1}(h(x)) = x$. While this class of functions is obviously unsuitable for cryptographic purposes, it has many applications in bioinformatics. These include space efficient storage of q-grams by integer encoding and comparisons within a single processor cycle.

Rolling hash functions work on input sequences and compute a new hash value based on the previous hash value. This is only possible if the hashed items have a high amount of locality. For example in a q-gram sequence two adjacent items share $q-1$ bases. By eliminating the need to reread the whole q-gram and *reuse* the $q-1$ known bases, the amount of memory accesses is reduced.

Finally, hash functions can be classified by their input alphabet, most notably integer and string hash functions. Integer hash functions typically use the domains $\mathcal{D} = [2^{32}]$ or $\mathcal{D} = [2^{64}]$ based on the integer data types used for their implementation. Consequently, they have a fixed input size. On the other hand, hash functions working on strings (or their representations) can have variable input lengths: for example the set of all nucleotide sequences $\mathcal{D} = \Sigma^*_{\text{DNA}}$ or the set of all byte strings $\mathcal{D} = [2^8]^*$. Especially hash functions used for file fingerprinting, like the MD5 and SHA, need to deal with files of arbitrary lengths. This requires them to read from an input and incorporate all parts into a hash value.

3.1.3 Hash Functions for Use in Bioinformatics

We will now describe hash functions commonly used with biological sequences. These include the integer encoding described in Section Memory Efficient Storage of q-Grams (p. 33), which can be used to convert strings from arbitrary alphabets into integer numbers, followed by integer hash functions that provide theoretical

[11] Rogaway and Shrimpton, "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance", 2004.

[12] Rivest, *The MD5 Message-Digest Algorithm*, 1992.

[13] Penard and Werkhoven, "On the Secure Hash Algorithm Family", 2008.

[14] Bertoni et al., "Keccak Sponge Function Family Main Document", 2009.

[15] Note that MD5 has been successfully attacked and is no longer considered safe for cryptographic use.

[16] Webster and Tavares, "On the Design of S-Boxes", 1986.

guarantees required by hashing techniques like MinHashing. Finally, we will briefly describe some widely used general purpose hash functions.

Integer Encoding allows mapping strings onto an integer range. Hence, integer encoding of sequences can be interpreted as a hash function

$$h_{\text{enc}} : \Sigma^n \to \sigma^n \tag{3.1}$$

for a given alphabet Σ of size σ.

Since each hash value encodes exactly one input string, integer encoding is a reversible hash function. This approach can be implemented as a rolling hash, as shown in Listing 2 in the appendix (p. 235) for the special case of Σ_{DNA}. Additionally, for this work we also implemented integer encoding for the Σ_{M10} and Σ_{M15} reduced protein alphabets.

The $\mathcal{H}^{lin}$ family of hash functions, introduced by Dietzfelbinger,[17] is a universal integer hash function. For a given integer domain $\mathcal{D}, |\mathcal{D}| = d$, integer codomain $\mathcal{C}, |\mathcal{C}| = c$, and an integer value $k \geq d$ it is defined as follows:

$$\begin{aligned} \mathcal{H}^{\text{lin}}_{d,c,r} &:= \{\, h_{a,b} \mid a, b \in [r] \,\} \qquad r = |\mathcal{R}| = k \cdot |\mathcal{C}| \\ h_{a,b}(x) &:= (ax + b \mod r) \operatorname{div} k \qquad x \in \mathcal{D},\ k \in \mathbb{N} \geq |\mathcal{D}| \end{aligned} \tag{3.2}$$

Recall, that $[n] := \{\, 0, \ldots, n-1 \,\}$

The integer set $\mathcal{R}$ is an intermediate domain of size at least d^2. To generate a hash function from an $\mathcal{H}^{\text{lin}}_{d,c,r}$ family, two random values a, b are chosen from the intermediate domain. Using these values, the input value is projected into $\mathcal{R}$ and subsequently mapped into the codomain, as illustrated in Figure 3.6.

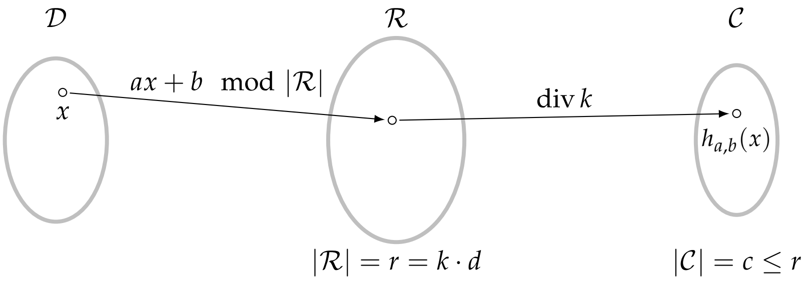

$\mathcal{D}$
$\mathcal{R}$
$\mathcal{C}$
$ax + b \mod |\mathcal{R}|$
$\operatorname{div} k$
x
$h_{a,b}(x)$
$|\mathcal{R}| = r = k \cdot d$
$|\mathcal{C}| = c \leq r$

Figure 3.6: Illustration of the $\mathcal{H}^{\text{lin}}_{d,c,r}$ family of hash functions, where d, c, and r denote the sizes of the domain, codomain and intermediate domain respectively. The hash value $h_{a,b}(x)$ is computed by mapping x into the larger intermediate (integer) domain and then mapping this intermadiate value onto $\mathcal{C}$. In practice, $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{C}$ are usually chosen as powers of 2 so that the second mapping can be performed using a bit shift operation.

Choosing the sizes of domain, codomain, and intermediate domain as powers of 2, allows an efficient implementation, since modulo and div operations can be realized using bitwise operations. An implementation of the $\mathcal{H}^{\text{lin}}_{2^{64},2^{64},2^{128}}$ universal hash function, which is able to generate 64-bit hash values from 2-bit encoded 32-grams is shown in Listing 3 in the appendix (p.236).

Note, that the $\mathcal{H}^{\text{lin}}$ family of hash functions is closely related to the Multiply-Shift hash family[18]. As shown by Thorup,[19] the $\mathcal{H}^{\text{lin}}$ family offers stronger guarantees (2-independence, see below) at

[17] Dietzfelbinger, "Universal Hashing and k-Wise Independent Random Variables via Integer Arithmetic without Primes", 1996.

[18] $\mathcal{H}^{\text{lin}}$ is sometimes also referred to as Multiply-Add-Shift hashing, if d, c, and r are powers of 2.

[19] Thorup, "High Speed Hashing for Integers and Strings", 2015.

minor additional computational costs. Hence, we omit a detailed discussion of Multiply-Shift and refer to the paper.[20]

While fast, $\mathcal{H}^{\text{lin}}$ hash functions are not very well suited for use with MinHashing, since they are not min-wise independent. We can show that both $\mathcal{H}^{\text{lin}}_{2^{64},2^{64},2^{128}}$ and $\mathcal{H}^{\text{lin}}_{2^{32},2^{32},2^{64}}$ are 2-independent[21]:

Theorem 3.1.9. *The $\mathcal{H}^{lin}_{2^{64},2^{64},2^{128}}$ hash family is 2-independent.*

Proof. As described by Dietzfelbinger,[22] a family of $\mathcal{H}^{\text{lin}}$ hash functions is $(1,2)$-independent, if:

- d, c, k, and r are all powers of 2: Which is obviously the case for $d = 2^{64}$, $c = 2^{64}$, $r = 2^{128}$, and $k = \frac{r}{c} = 2^{64}$.
- $k \geq \frac{d}{2}$: Which holds, for $k = 2^{64} \geq 2^{63} = \frac{2^{64}}{2}$.

A $(1,2)$-independent hash function is 2-independent per Corollary 3.1.6. □

Theorem 3.1.10. *The $\mathcal{H}^{lin}_{2^{32},2^{32},2^{64}}$ hash family is 2-independent.*

Proof. Analogous to the proof for $\mathcal{H}^{\text{lin}}_{2^{64},2^{64},2^{128}}$. □

Subfamilies of $\mathcal{H}^{\text{lin}}$ where d, c, and r are all powers of 2 can be efficiently implemented by replacing the modulo operations with bit shift operations. Hash functions created this way are also referred to as Multiply-Shift-Add hash functions.

Tabulation Hashing Based on Zobrist hashing,[23] which was devised to encode game states for board games like chess or Go, tabulation hashing has recently been developed and analyzed for use with MinHashing.[24] Tabulation hashing and its variants use a table

$$\Theta = (\theta_{i,j}), \quad i \in [n], \quad j \in [u^{\frac{1}{n}}]$$

initialized with random integers, from which values are selected and processed using bitwise exclusive-OR operations (XOR, denoted by $\oplus$). To index the table, keys $k \in \mathcal{U} = [u]$, where u is a power of 2, are split into n chunks $(k_0, \ldots, k_{n-1})$ of $\log_2 u^{\frac{1}{n}}$ bits. While a fixed size table restricts tabulation hashing to fixed length keys, this design is an important performance consideration. A small table can be kept within one cache line by the processor, allowing fast access to the table entries.

For simple tabulation hashing,[25] the hash value $h(k)$ for a key $k = (k_0, \ldots, k_{n-1})$ is computed as:

$$h(k) = \bigoplus_{i \in [n]} \theta_{i,k_i}$$

Each chunk of the key is used to pick an entry of Θ, which are processed using XOR operations as illustrated in Figure 3.7 (a). An implementation of this hash function is given in Listing 4 (p.237).

Twisted tabulation hashing[26] modifies the last step of simple tabulation hashing to incorporate not only the last chunk, but also the

[20] Dietzfelbinger et al., "A Reliable Randomized Algorithm for the Closest-Pair Problem", 1997.

[21] Note that Dietzfelbinger uses the name (c,l)-universal for this concept.

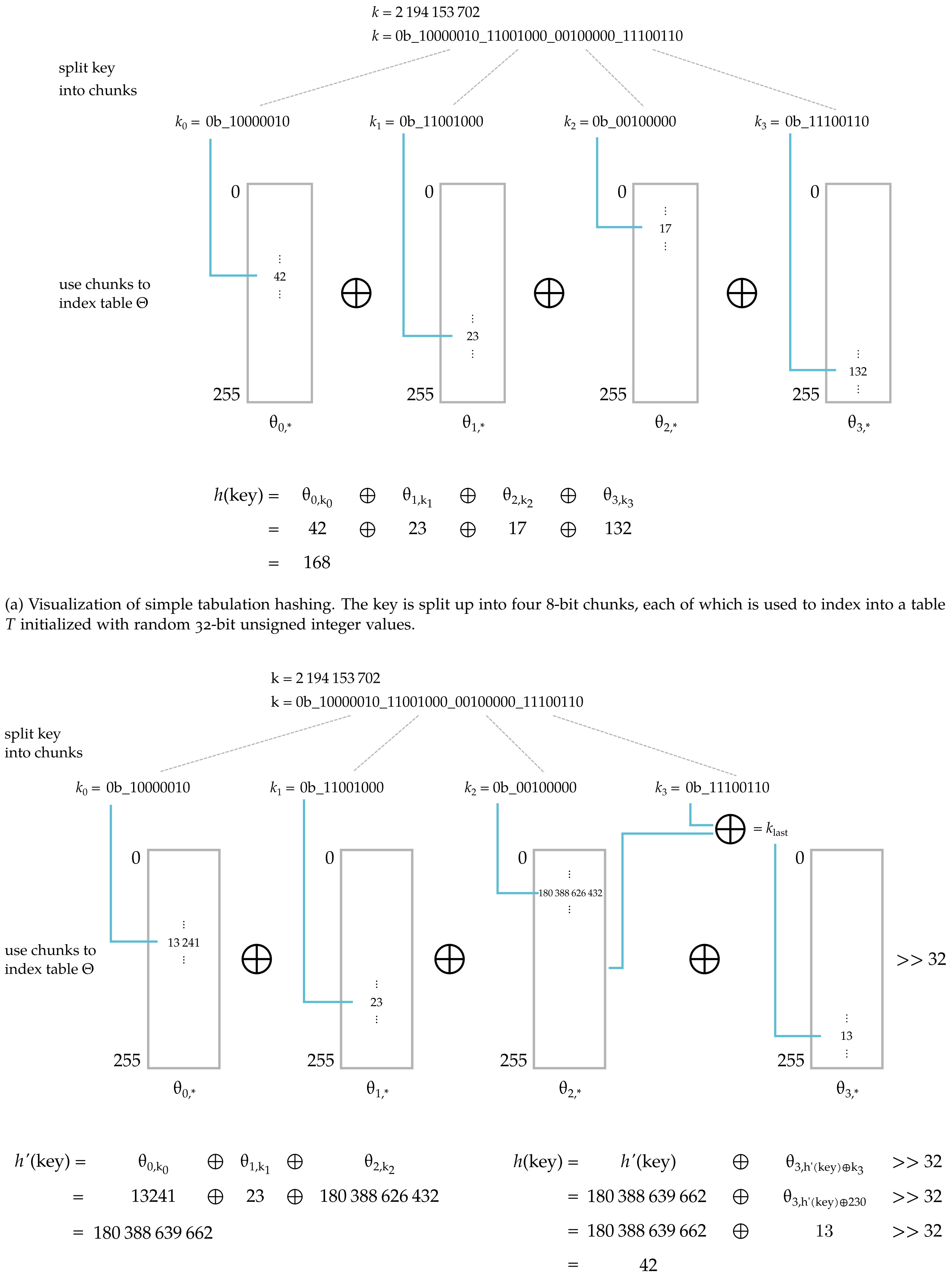
[22] Dietzfelbinger, "Universal Hashing and k-Wise Independent Random Variables via Integer Arithmetic without Primes", 1996.

[23] Zobrist, *A New Hashing Method With Application for Game Playing*, 1970.

[24] Dahlgaard and Thorup, "Approximately Minwise Independence with Twisted Tabulation", 2014; Pătraşcu and Thorup, "The Power of Simple Tabulation Hashing", 2011; Pătraşcu and Thorup, "Twisted Tabulation Hashing", 2013.

[25] Pătraşcu and Thorup, "The Power of Simple Tabulation Hashing", 2011.

[26] Pătraşcu and Thorup, "Twisted Tabulation Hashing", 2013.



k = 2 194 153 702
k = 0b_10000010_11001000_00100000_11100110
split key into chunks
k_0 = 0b_10000010
k_1 = 0b_11001000
k_2 = 0b_00100000
k_3 = 0b_11100110
use chunks to index table Θ
0
255
42
23
17
132
$\theta_{0,*}$
$\theta_{1,*}$
$\theta_{2,*}$
$\theta_{3,*}$
$h(\text{key}) = \theta_{0,k_0} \oplus \theta_{1,k_1} \oplus \theta_{2,k_2} \oplus \theta_{3,k_3}$
$= 42 \oplus 23 \oplus 17 \oplus 132$
$= 168$
k = 2 194 153 702
k = 0b_10000010_11001000_00100000_11100110
split key into chunks
k_0 = 0b_10000010
k_1 = 0b_11001000
k_2 = 0b_00100000
k_3 = 0b_11100110
$= k_{\text{last}}$
use chunks to index table Θ
0
255
13 241
23
180 388 626 432
13
>> 32
$\theta_{0,*}$
$\theta_{1,*}$
$\theta_{2,*}$
$\theta_{3,*}$
$h'(\text{key}) = \theta_{0,k_0} \oplus \theta_{1,k_1} \oplus \theta_{2,k_2}$
$= 13241 \oplus 23 \oplus 180\,388\,626\,432$
$= 180\,388\,639\,662$
$h(\text{key}) = h'(\text{key}) \oplus \theta_{3,h'(\text{key})\oplus k_3} >> 32$
$= 180\,388\,639\,662 \oplus \theta_{3,h'(\text{key})\oplus 230} >> 32$
$= 180\,388\,639\,662 \oplus 13 >> 32$
$= 42$

(a) Visualization of simple tabulation hashing. The key is split up into four 8-bit chunks, each of which is used to index into a table T initialized with random 32-bit unsigned integer values.

(b) Visualization of twisted tabulation hashing. In contrast to simple tabulation hashing, the entries of T are 64-bit values and the last address is computed using the preliminary hash value h' and k_3. To receive a 32-bit hash value, the lower 32 bits are shifted out.

Figure 3.7: Visualization of simple (a) and twisted tabulation hashing (b) with $n = 4$ chunks. Gray blocks represent vectors comprising the index table Θ. Teal lines indicate at which position said vector $\theta_{i,*}$ is accessed.

preliminary hash value $h'(k)$ computed from all previous chunks. Additionally, twisted tabulation hashing extends the length of the entries in Θ and uses only the upper bits of the resulting hash value. For 32-bit twisted tabulation hashing, the hash value $h(k)$ for a key $k = (k_0, \dots, k_{n-1})$ is computed as

$$\begin{aligned} h(k) &= \left(h'(k) \oplus \theta_{n-1, h'(k) \oplus k_{n-1}}\right) \gg 32 \\ h'(k) &= \bigoplus_{i \in [n-1]} \theta_{i, k_i} \end{aligned}$$

where k and $h(k)$ are 32-bit integers and the entries of Θ are 64-bit integers. The "$\gg x$" operator denotes a bit-wise right shift by x bits. This is illustrated in Figure 3.7 (b). Our Rust implementation of simple and twisted tabulation hashing[27] can be found on GitHub[28] and the Rust package repository `crates.io`[29]. An excerpt of this implementation is shown in Listing 4 in the appendix (p.237).

While both simple and twisted tabulation hashing are 3-independent,[30] twisted tabulation hashing offers higher approximately min-wise independence. Simple tabulation hashing is $\mathcal{O}(\frac{1}{k^{1/n}})$-approximately min-wise independent,[31] where k is the size of the selected subset[32] and n is the number of chunks. This yields a higher bias for small set sizes. Twisted tabulation hashing, on the other hand, is $\mathcal{O}(\frac{1}{u^{1/n}})$-approximately min-wise independent,[33] where $u = |\mathcal{C}|$ is the size of the hash functions codomain, which is independent of the set size.

General purpose hash functions (GPHFs) are used more for their convenience and speed than for their theoretical properties. There are several such hash functions that are used in bioinformatics applications, including Murmur Hash,[34] Cityhash,[35] and xxHash.[36] All three of these non-cryptographic hash functions capable of hashing byte sequences and are able to generate 32-bit hash values. Bigger hash values are also supported, using slight variations of their respective implementations: For `mmh3` a version for 128-bit hash values exists and xxHash can generate 64-bit hashes. Cityhash has implementations for 64-bit, 128-bit, and 256-bit hash values. Furthermore, all of them have been analyzed and benchmarked with the SMHasher benchmark suite for non-cryptographical hash functions.[37] While almost all claims made in this section would hold for all mentioned GPHFs, we will focus our explanation on `mmh3` due to its brief and elegant implementation, its use in several bioinformatics applications like Mash[38] and Mashmap,[39] and its wide use in general.[40]

While there are architecture specific variants of the `mmh3` hash function family, their main notion is similar. All of them can use an integer seed value, which allows the generation of hash functions, making `mmh3` behave like a universal hash function for practical

[27] Timm, *Rust-tab-hash Source Code*, 2020.

[28] `https://github.com/HenningTimm/rust-tab-hash`

[29] `https://crates.io/crates/tab-hash`

[30] Pătraşcu and Thorup, "Twisted Tabulation Hashing", 2013.

[31] Pătraşcu and Thorup, "The Power of Simple Tabulation Hashing", 2011.

[32] This is equivalent to the number of hash functions used in MinHashing.

[33] Dahlgaard and Thorup, "Approximately Minwise Independence with Twisted Tabulation", 2014.

[34] Appleby, *Murmurhash3*, 2016.

[35] Pike and Alakuijala, *Introducing CityHash*, 2011.

[36] Collet, *xxHash–Extremely Fast Hash Algorithm*, 2016.

[37] Appleby, *SMHasher*, 2016.

[38] Ondov et al., "Mash: Fast Genome and Metagenome Distance Estimation using MinHash", 2016.

[39] Jain et al., "A Fast Approximate Algorithm for Mapping Long Reads to Large Reference Databases", 2017.

[40] Richter, Alvarez, and Dittrich, "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing", 2015.

```
pub fn fmix_32(hash: u32) -> u32 {
    let mut hash = hash;
    hash ^= hash >> 16;
    hash *= 0x_85_eb_ca_6b;
    hash ^= hash >> 13;
    hash *= 0x_c2_b2_ae_35;
    hash ^= hash >> 16;
    hash
}
```

Listing 1: Rust implementation of the `fmix32` hash mixing function.

purposes:

$$\begin{aligned} \mathcal{H}_{32}^{\text{mmh3}} &:= \{\, h_s \mid s \in [2^{32}] \,\} \qquad && h_s : [2^8]^* \to [2^{32}] \\ \mathcal{H}_{128}^{\text{mmh3}} &:= \{\, h_s \mid s \in [2^{32}] \,\} \qquad && h_s : [2^8]^* \to [2^{128}] \end{aligned} \tag{3.3}$$

However, while `mmh3` performs well in practice, even in the context of MinHashing[41], the function has no proven theoretical guarantees.[42]

Since all `mmh3` variants behave similarly, we will focus on the 32-bit version for this section. As input, the hash function requires a 32-bit seed value and sequence of bytes (i.e. 256-bit numbers). The `mmh3` algorithm comprises four steps:

- Initialize the hash value with the seed.
- Read input in chunks of 32 bits and factor it into the current intermediate hash value. Each block is multiplied by a big constant, circularly shifted and XOR-ed with the current hash value.
- Process all remaining bytes that did not fit into a complete chunk.
- Finalize intermediate hash value using a hash mixing step (`fmix32`) using multiplication with big constants and bit shifts.

Note that the hash mixing step can by itself be interpreted as a non-universal integer hash function $h_{32}^{\text{fmix}} : [2^{32}] \to [2^{32}]$. For the 128-bit variant, h_{32}^{fmix} is applied separately to the four 32-bit blocks of the hash value. A Rust implementation of the `fmix32` function is illustrated in Listing 1, while the C++ implementation of both `mmh3` variants can be found in the SMHasher GitHub Repository[43].

3.2 *Hash Tables*

As mentioned in the introduction of this chapter, one of the main applications of hash functions is addressing hash tables. A hash table is a data structure that allows membership queries and retrieval operations in expected constant time. To achieve this, items are stored at a position computed by a hash function, as illustrated in Figure 3.8.

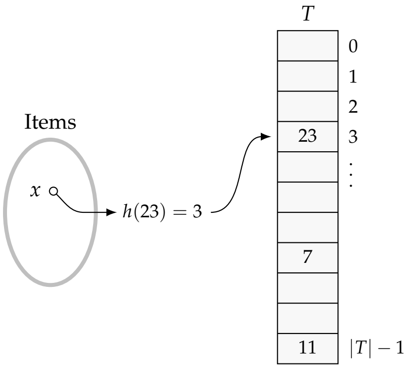


Figure 3.8: A basic (value store type) hash table that contains the entries 23, 7, and 11 at their respective hash addresses. The items 7 and 11 have already been inserted and 23 is currently being inserted, as denoted by the arrows. Hash addresses of the array T are shown at the right of T.

[41] See MinHash, p. 99.

[42] Thorup, "Fast and Powerful Hashing Using Tabulation", 2017; Richter, Alvarez, and Dittrich, "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing", 2015.

[43] `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`

Items entered in the table are stored in an underlying data structure T, usually implemented using an array or a list, which can be indexed (or addressed) using integer values. We refer to inserted items as entries. To enter an item x into a hash table, its address is computed using the hash value $h(x)$ from a hash function h and x is saved at the index $h(x)$ in T: $T[h(x)] = x$. Entries of T are also called (hash) slots, or hash positions. We call the ratio of empty slots to total available slots

$$\rho = \frac{\text{\# empty slots}}{\text{\# total slots}}$$

the fill rate (or load) of a hash table. We describe the number of possible hash addresses of T as $\mathcal{T} = [|T|]$, where $|T|$ is the size of the underlying array. Note that, while a hash address can be the unmodified result of a hash function, often the hash address is derived. Most commonly, if $|T| < |\mathcal{C}|$, hash values are usually truncated using the modulo function. If we need to explicitly distinguish between a hash value and a derived address we use $h(x)$ for the hash value of x and the address function $a : \mathcal{C} \to \mathcal{T}$ for the derived hash address $a(x)$. We also use a as a shorthand notation for a specific hash address.

To retrieve x, its address is computed and the table is queried at position $h(x)$. If the position $T[h(x)]$ is empty, x was not in the table, otherwise x or an item with a colliding hash value was entered. In the latter case, to be certain x was contained in the table, and not some item $x' \neq x$ with $h(x) = h(x')$, a collision handling strategy is required. This chapter focuses on the implementations of different kinds of hash tables, terminology, and collision resolution strategies.

There are many possible ways to implement a hash table, differing greatly in expected run times and memory requirements. We define basic hash table operations, which we use to describe implementations:

Insert(x) Put a new item x into the hash table. If a collision occurs, it needs to be handled here.

Contains(x) Perform a membership query. Check if the item x is present in the hash table. If so, return `true`.

Search(x) Return all values stored for the item x. Depending on the specific implementation, this can either be one value (e.g. when used as a counter) or multiple values (e.g. when used as a dictionary).

Delete(x) Remove item from the hash table and assert, that all remaining items can still be found. This is not possible with all collision resolution strategies.

Resize Increase the size of the underlying data structure to accommodate more items. This is required by dynamically sized hash tables, when not enough empty entries remain.

The run time of these operations depends on the collision resolution strategy used by the hash table and we will discuss them while presenting these strategies later in this chapter.

As with hash functions, hash tables can be classified using several properties. For hash tables, these include how they behave in memory, what kind of items they store, and how they deal with collisions.

3.2.1 *Static and Dynamic Hash Tables*

All hash tables need to store their entries in an underlying data structure. In most cases, this is some variation of an array, into which items are inserted at a position determined by the hash function. However, there are several possible layouts. Most prominently, we distinguish between two types of hash tables in this work: Static and dynamic hash tables. A static hash table has a constant number of possible entries, called slots, while the number of slots in a dynamic hash table can be adjusted at run time.

Dynamic hash tables are familiar to anyone using high-level programming languages. For example the Python dictionary, Rust and C++ hash maps, and Java Hashtable are all dynamic hash tables. The main advantage of this class of hash tables is their ability to maintain a low memory footprint for small input datasets, but to be able to grow with the data. This is achieved by resizing the table, once a certain fill rate or a predefined number of collisions has occurred. When computing the hash address, the hash value is mapped onto the number of slots available in the table at the given size. Usually this is performed by choosing a table size $|T|$ and deriving hash addresses $a(x) = h(x) \mod |T|$ from the hash value using modulo operations[44]. When the table is to be resized, choose a new $|T'| > |T|$ and recompute all hash addresses using $a(x) = h(x) \mod |T'|$. However, this comes at the disadvantage that either all hash values need to be recomputed in a process called rehashing, or the hash values need to be stored and thereby increase the memory footprint.

Static hash tables on the other hand, do not allow resizing. While they require choosing a reasonable size at creation, static hash tables provide better control over the memory footprint than their dynamic counterparts. This is especially powerful, when the number of entries is known beforehand, or if guarantees about the fill rate or the memory usage are required. Additionally, the hash function employed can be better tailored to the hash table, since the codomain of the hash function can be chosen to correlate with the number of slots. A downside of static hash tables is that they cannot adapt to unexpected increases in fill rate. With high fill rates, collisions become much more likely and cannot be compensated as easily. Depending on the collision resolution strategy this can either

[44] The choice of $|T|$ is crucial to prevent introducing collisions. Prime numbers are optimal (see Knuth, *The Art of Computer Programming: Sorting and Searching*, 1997, Chapter 6.4), but in practice powers of two are often used.

result in performance drops or the table is simply unable to insert certain keys. Note, that just using a static sized hash table does not guarantee a static amount of memory usage, unless a suitable collision resolution strategy is also employed.

3.2.2 *Key-Value Stores and Value Stores*

Another important distinction is what is actually stored in the hash table. Depending on the task a hash table should perform, two designs are possible: A key-value store, in which a key-value pair is inserted based on the hash value of a provided key, and a value store in which the key is not explicitly stored. These two design are illustrated in Figure 3.9 and Figure 3.10 respectively.

Key-value stores (KVS): A key-value store hash table saves the key used to compute the hash address as well as an item that should be associated with this key.

Consider a tuple (or key-value pair) $x = (k, v)$ consisting of a key $k \in \mathcal{K}$ and a payload item $v \in \mathcal{V}$. When entering x into the hash table, the hash address at which x is stored, i.e. the index in the underlying array, is computed using only k. Hence, we store $T[h(k)] = x$ in the table. Retrieving values works analogously, by querying $T[h(k)]$ to receive x.

Such a key-value pair can, for example, be a q-gram used as key and a genomic position as payload.

This kind of hash table allows retrieving a value for a given key (SEARCH) as well as performing membership queries (CONTAINS) and removing elements (DELETE). For use as a dynamic hash table, during the RESIZE step, keys can be extracted from the stored tuples and items can be rehashed. However, this comes at the cost of a high memory footprint, since both key and value need to be saved.

Note that key and value can be identical, as in the initial hash table example shown in Figure 3.8. The address is computed using $h(k)$, and k is inserted into the table: $T[h(k)] = k$. This strategy results in a lower memory footprint than the traditional KVS model described above, but also limits its usability, as it is not fit to implement dictionary type data structures.

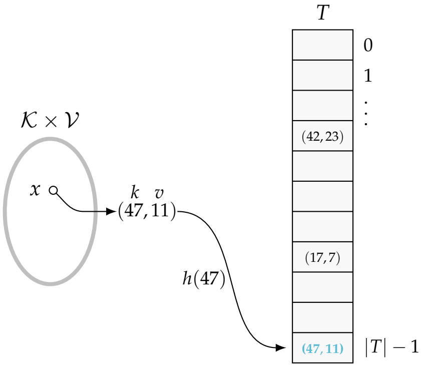


Figure 3.9: Illustration of a key value store. An item $x = (47, 11)$ where 47 is the key and 11 is the payload item, is inserted into a hash table T. The hash address for x is computed as $h(k) = |T| - 1$ and is therefore inserted at $T[|T| - 1]$.

Value Stores (VS): In a value-store hash table, the key generating the address is not stored. Instead, only the payload (value) associated with the key is saved. For example, after hashing a q-gram we might want to store the position at which it is located within a genome. Consequently, keys used to insert into a value store cannot be easily retrieved.

VS hash tables can only perform a subset of the operations KVS hash tables can. Most importantly, VS tables cannot trivially support RESIZE, since the keys used to enter the values are no longer known. However, there are ways to circumvent this limitation detailed below. The same holds for the DELETE operation. For CONTAINS and SEARCH, a VS table cannot validate the identity of a query and might yield false positive results. This is the case for colliding items $h(x) = h(x'), x \neq x'$: If x is not in the table, but x' is, Contains(x) would falsely return `true`.

The main advantage of VS tables is their memory efficiency. Since keys do not need to be explicitly stored, the memory footprint of a VS table is about half that of a KVS table. This makes them suitable for scenarios where neither validation of keys nor resizing the hash table is required.

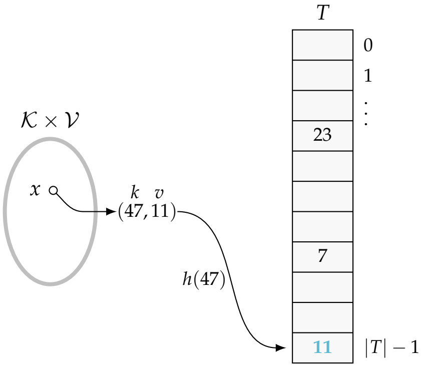


Figure 3.10: Illustration of a value store. An item $x = (47, 11)$ is inserted into a hash table T. The hash address for x is computed as $h(k) = |T| - 1$ and therefore the value 11 is saved at $T[|T| - 1]$.

Given these two designs, KVS are clearly more flexible which is indicated by their presence in virtually all modern programming languages. VS tables allow a more concise and memory efficient implementation, which can be required for applications that need to handle large amounts of data. Within some limits, VS tables can retain some of the properties normally limited to KVS. This requires using a two-way hash function so that hash keys can be recomputed from the hash address and in turn retains all advantages of a KVS table. This comes with two downsides:

1. Using a two-way hash function severely limits the choices of possible (universal) hash functions. Consequently, it might be hard to find a suitable hash function with the desired properties.

2. To be unique, a two-way hash function needs to maintain $|\mathcal{D}| \leq |\mathcal{C}|$. This limits the amount of possible keys, since with growing $\mathcal{D}$ the size of the hash table also needs to grow.

The second restriction can be dealt with by using quotienting, i.e. by storing a part of the hash value within the hash table entries.

3.2.3 Quotienting and Packing

As outlined by Knuth[45] the quotienting technique describes saving a part of the hash value as part of the hash table entry. This allows choosing a smaller hash table without restricting the codomain of the hash function. As above, starting from a tuple $x = (k, v)$ where $k \in \mathcal{D}$, we compute a hash value $h(x)$ to use as hash address for storing v. However, we do not use $h(x) = y$ as hash address directly, but using a function

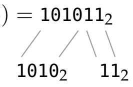
$$\delta(y) : \mathcal{C} \to (\mathcal{A} \times \mathcal{R})$$

we derive two values: an address $a \in \mathcal{A}$ and a remainder[46] $r \in \mathcal{R}$. Given these values, the original hash value y (or even the key x) can be restored. An example for such a function is to interpret the hash value as a bit vector and split up the bits (see Figure 3.11) at a specific point.

$$h(x) = x$$

$$\delta(y) = (\underset{a}{y \mod 4}, \underset{r}{\lfloor \tfrac{y}{4} \rfloor})$$

$$h(43) = \texttt{101011}_2$$



$$a = \texttt{101011}_2 = 3$$

$$r = \texttt{101011}_2 = 10$$

Figure 3.11: Illustration of quotienting using the identity hash function and 6-bit values. All numbers without the subscript$_2$ suffix are base 10 numbers. The address (a) is computed by using the highest four bits of the hash value y, while the remainder (r) comprises the lowest two bits.

Note that given an identity hash function or a translation, like integer encoding, x can be restored. For most non-trivial hash functions, collisions between keys cannot be resolved and hence only the original hash value $y = h(x)$ can be restored. This is enough

[45] Knuth, *The Art of Computer Programming: Sorting and Searching*, 1997.

[46] Depending on context also called a fingerprint. We avoid that term since it is also used in the context of locality sensitive hashing.

to allow resizing for use with dynamic hash tables, if the modulo scheme described above is used, since new hash values can be derived from the original value.

To restore $y = h(x)$, we need an inverse function

$$\delta^{-1} : (\mathcal{A} \times \mathcal{R}) \to \mathcal{C},$$

so that for $\delta(y) = (a, r)$ we have $\delta^{-1}(a, r) = y$. For the example given above, this would mean appending the two bit vectors, as shown in Figure 3.12.

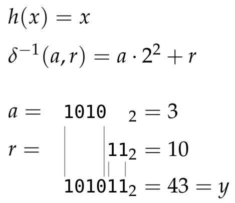


Figure 3.12: Illustration of inverse function to restore hash values from quotiented hash addresses using the identity hash function and 6-bit values. All numbers without the subscript$_2$ suffix are base 10 numbers. The original hash value $y = h(x)$ can be restored by concatenating the bit vectors, i.e. shifting a two bits to the left (multiply by 2^2) and then add r.

Since a can be easily derived from the position in T, only r needs to be stored within the table in one of two ways:

- As part of a tuple (r, v), as a separate number, or
- encoded as part of the value.

The former option is straight forward, but comes at the cost of memory efficiency due to fixed size data types. During implementation, both remainder and value are likely stored as integer variables, like `u8`, `u32`, or `u64`. Especially when the remainder is small, like the two bits illustrated in Figures 3.11 and 3.12, some bits of the saved integer number might not be used. Even when using an 8 bit integer, 75% of the bits are never used. While this can be mitigated with strategies like using word packing, these introduce more computational overhead.

However, it is possible that the values to be stored suffer from the same effect, i.e. the possible values do not fully saturate the data type (for example 32-bit integers). In this case, the remainder can be encoded as part of a stored value v' using a technique similar to the inverse quotienting function detailed above. By limiting the possible values v to $[2^{m-p}]$, where 2^m is the maximal size of the data type used for hash table entries and p is the maximal length of the remainder in bits, we can define a packing function:

$$\begin{aligned} \psi &: (\mathcal{V} \times \mathcal{R}) \to [2^m] \\ \psi(v, r) &= r \cdot 2^{m-p} + v \end{aligned} \tag{3.4}$$

This packing function ψ encodes the remainder in the two highest bits of the result and fills the lower 30 bits with the value. Note that

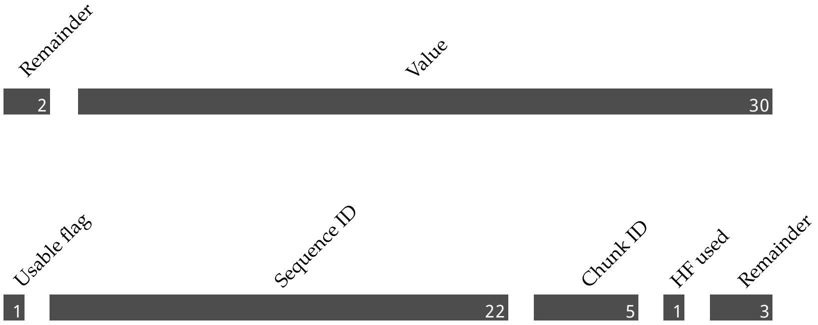


Figure 3.13: Two possible packings of 32-bit integers. The number of bits allocated to a segment of the resulting value is denoted by white numbers. Displayed are the result of the packing function described in Equation 3.4 (top) and of a more complex packing function (bottom, encoding four different values in addition to the remainder).

this is only one possible function to encode the remainder. Using this packing technique, additional information can also be added to the value in more elaborate schemes (see Figure 3.13). In other words, a tuple $v = (v_0, \ldots, v_n)$ of values and the remainder can be combined into a single value for efficient storage in the table.

When used for quotienting, such a packing scheme requires a tradeoff between the size of the remainder (bigger remainders allow the use of smaller hash tables) and the size of the payload, i.e. the size of all encoded values.

For some use cases it is possible to further reduce the memory footprint of the table by not saving the complete remainder. This of course prevents restoring the original hash value, since irresolvable collisions are introduced. Nonetheless, if exact verification is not required, this approach can be used to maintain a larger payload to remainder ratio.

3.2.4 Bloom Filter

A special kind of hash table used to answer membership queries are bloom filters.[47]

Definition 3.2.1. *A **Bloom Filter** $\mathcal{B}$ with error rate p, comprises a bit array with n bits (initially all 0) and k hash functions. If a key x is contained in $\mathcal{B}$, the bits $\mathcal{B}_{[i]}, i \in \{ h_j(x) \mid j \in 1, \ldots, k \}$ are set to 1.*

We denote the bloom filter that contains all items from a set A as $\mathcal{B}(A)$.

To insert a key x_0, hash values are computed using each hash functions and the respective bits are set to 1. When querying a bloom filter with a key x_1, the bit positions of x_1 are computed and used to index the bit array. If all entries for x_1 are 1, i.e. if

$$\mathcal{B}_{[h_1(x_1)]} = 1 \wedge \mathcal{B}_{[h_2(x_1)]} = 1 \wedge \cdots \wedge \mathcal{B}_{[h_k(x_1)]} = 1$$

holds, x_1 is contained in $\mathcal{B}$. An example for this can be found in Figure 3.14.

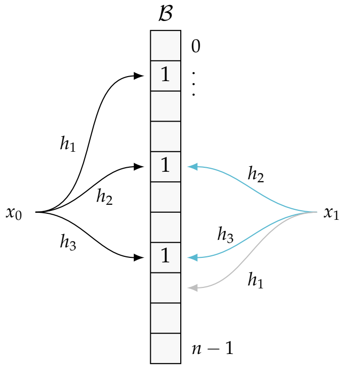


Figure 3.14: Illustration of a bloom filter with three hash functions containing the key x_0. Hash addresses computed by the hash functions h_1, h_2, and h_3 are denoted by arrows; 0-bits are denoted by empty slots. The key x_1 is not contained in $\mathcal{B}$. While the bits for $h_2(x_1)$ and $h_3(x_1)$ have been set to 1 (teal arrows) by x_0, h_1 points to an empty slot (light gray arrow).

There is a probability p, that $x_1 \in \mathcal{B}$ is a false positive, i.e. x_1 is not actually contained in $\mathcal{B}$. This occurs, if all relevant bits for x_1 have been set by other keys, which depends on the size of the bit array, the number of inserted keys and the number of hash functions used. For a bloom filter with n bits, k hash functions, and m inserted elements, the false positive rate

$$p = \left(1 - \left(1 - \frac{1}{n}\right)^{mk}\right) \approx 1 - e^{-km/n}$$

is minimized by choosing $k = \ln 2 \cdot (n/m)$ hash functions.[48]

There are several variants of bloom filters available which improve on different aspects of the concept. Counting bloom filters, for example, maintain a counter for each slots to count occurrences of keys[49].

[47] Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors", 1970.

[48] Bonomi et al., "An Improved Construction for Counting Bloom Filters", 2006.

[49] Fan et al., "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", 2000.

3.3 Collision Resolution

Until now, we omitted details about collision resolution strategies, which are as crucial to an efficient hashing algorithm as a good hash function. As mentioned above, the term collision describes two different keys $x_0 \neq x_1$ which receive the same hash value $h(x_0) = h(x_1)$. When using the hash value as hash address directly, these keys would map to the same slot of the underlying table T. Collision resolution describes strategies to resolve these cases, which can be categorized into three classes:

- Perfect hashing, which prevents collisions in the first place by designing a hash function tailored to a specific input set.
- Closed addressing, which manages a data structure for each slot of T that can take up several values.
- Open addressing, which allows placing entries in one of multiple positions within T.

The collision resolution strategy is the main influencing factor on the run time of a hash table, both asymptotically and practically. For example variants of cuckoo hashing offer SEARCH in $\mathcal{O}(1)$ time, but can generate a large number of cache misses, due to their use of multiple hash functions.[50] Combinations of these strategies are also possible. While perfect hashing is restricted to very specific use cases, combinations of closed and open addressing strategies are used for example with (k, p)-cuckoo hashing.

Deciding on a collision resolution strategy requires careful consideration of the following questions:

- How much memory is available?
- Are run time or cache efficiency restrictions required?
- Should the hash table support deletions?
- Are there domain specific restrictions? For example: Are unresolvable collisions acceptable or not?

3.3.1 Collision Prevention: Perfect Hashing

In contrast to other collision resolution strategies, the main notion of perfect hashing[51] is to construct a hash function that does not collide for a given static set of keys. In other words: For a given static set S of keys, find a hash function that injectively maps the keys to an integer range $\mathcal{T}$, which guarantees that search queries can be completed in worst case run time $\mathcal{O}(1)$. Since many biological problems can have large possible key sets, we will only briefly describe this technique for completeness' sake.

Perfect hashing is described well by Cormen et al.,[52] who give an example of a perfect hash function using universal hash functions (cf. Universality, Independence, and Min-Wise Independence,

[50] Erlingsson, Manasse, and McSherry, "A Cool and Practical Alternative to Traditional Hash Tables", 2006.

[51] Fredman, Komlós, and Szemerédi, "Storing a Sparse Table with O(1) Worst Case Access Time", 1984.

[52] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.5.

p. 40) and two nested hash tables. Since constructing these hash tables can be memory intensive[53], minimal perfect hash function (MPHFs), strive to reduce the memory footprint of perfect hash functions.

A recent example of a minimal perfect hash function (MPHF) has been given by Limasset et al.[54] Their BBHash algorithm uses a series of cascading bit arrays A_i with corresponding hash functions h_i. At the beginning, all keys $k \in S$ are hashed into A_0 using h_0, with an additional temporary array used to count collisions. Each slot of A_0 that received a unique key is marked with a 1, all other entries with 0. After this step, a new and smaller array A_1, with a new hash function h_1, is created and the procedure is repeated. When all keys have been placed, all arrays $A_0, A_1, \dots, A_{\text{last}}$ are concatenated into an array A, as illustrated in Figure 3.15. Using A, the hash value of k can be computed as the rank of its corresponding 1 entry in A (i.e. the number of 1-bits before its own entry).

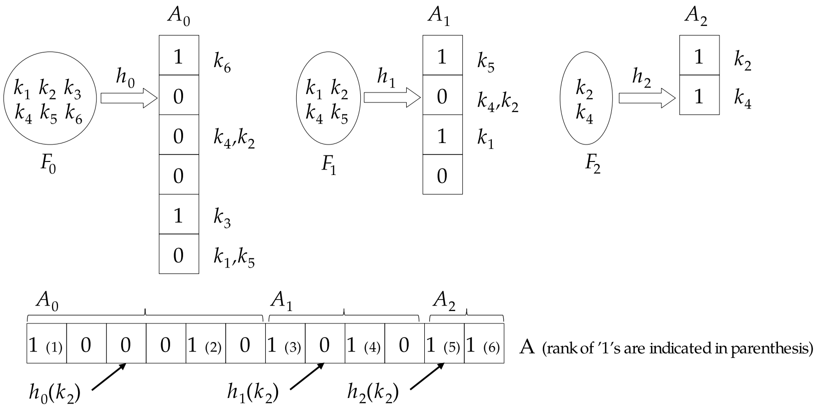


Figure 3.15: Illustration of the BBHash algorithm, showing a MPHF for the key set $S = F_0 = \{k_1, \dots, k_6\}$. The keys k_1, k_2, k_4, k_5 could not be uniquely mapped to A_0, resulting in 0-entries at positions $A_0[2]$ and $A_0[5]$, and are inserted into A_1 in the next step. At the bottom, the combined array A is shown. To find the hash value for k_2, the unique 1-bit for k_2 is found at position $A[10]$ and the rank for position 10 (shown in braces) is computed. For this MPHF, the hash value for k_2 is $h(k_2) = 5$. This figure has been derived from Figure 1 in the paper of (Limasset et al., "Fast and Scalable Minimal Perfect Hashing for Massive Key Sets", 2017), released under CC-BY 3.0 license.

While worst case constant time access is tempting, this technique requires that all keys are known prior to computation. This can be a very limiting factor, since if unexpected keys are introduced at a later time, they either need to be removed with a set query test or they introduce false positives via collisions. It is possible, to compute a MPHF for the whole set of possible keys, i.e. all q-grams, but this can require an enormous amount of memory. For example computing a MPHF for the set of all 19-grams, which is the largest possible with current technology,[55] requires about 600 GB of RAM and 36 hours.[56] When using longer q-grams with the possibility of unexpected new keys, more flexible approaches are required.

3.3.2 *Collision Resolution by Closed Addressing*

Closed addressing describes all techniques that store more than one item in a slots of the array T using a secondary data structure. The most notable, and most basic, implementation of this pattern is to make each slot of the hash table a linear list of items. When a new key x is inserted, create a new list with one item and insert it into T: $T[h(x)] = [x]$.[57] If a collision occurs, i.e. if for a given slot

[53] The second order hash tables proposed in Cormen's example are of size $|T| = k^2$, where k is the number of keys inserted into the second order table.

[54] Limasset et al., "Fast and Scalable Minimal Perfect Hashing for Massive Key Sets", 2017.

[55] Limasset et al., "Fast and Scalable Minimal Perfect Hashing for Massive Key Sets", 2017.

[56] Note, that while this case could also be solved by integer encoding, other key sets might not be easily enumerable.

[57] Here we use square brackets to denote a list data structure. $[\,]$ denotes an empty list, $[x]$ a list with only the item x, and $[x_0, x_1, x_2]$ a list with three items.

there already is a list, append the item to said list. A sequence of colliding items $(x_i)_{i=0}^{n}$, $h(x_i) = a$ would result in the list $T[a] = [x_0, \ldots, x_n]$. Through this, colliding items form chains in each slot, lending this technique its name: *collision by chaining*. This behavior is apparent in Figure 3.16.

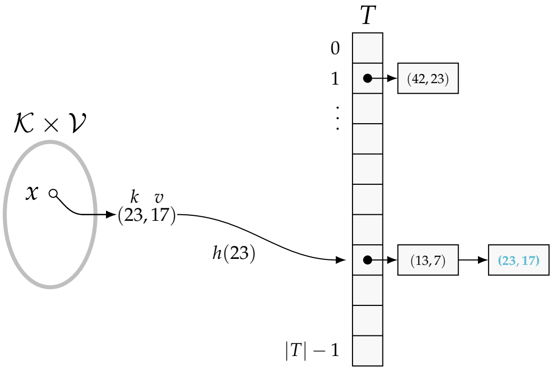


Figure 3.16: Illustration of a hash table T with collision by chaining. Most buckets are empty. In bucket 1, the pair $(42, 23)$ is stored in a list with one element. The item $(23, 17)$ currently being inserted hashes to the same bucket as $(13, 7)$ and is appended to this buckets list.

The main downside of using a linear list is the worst case runtime, as described by Cormen.[58] When all items are hashed to the same slot, the hash table offers no improvement over searching in a linear list. For this unlikely case the worst case run time is $\mathcal{O}(n)$, where n is the number of inserted items. When a universal hash function is used, both successful and unsuccessful SEARCH operations can be decided in $\mathcal{O}(\rho)$ time, where ρ is the hash table load. Since for a hash table using collision by chaining with linear lists there is no maximal number of entries in the table, the table load ρ is defined as the average length of the lists. Using universal hash function[59], the hash table load is expected to be

$$\rho = \frac{n}{|T|}$$

where n is the number of items and $|T|$ is the number of slots. Both INSERT and DELETE can be performed in $\mathcal{O}(1)$ time, if the lists are doubly linked.[60]

Other variants of closed addressing use different data structures within the slots, offering different run time properties. However, the employed data structure always influences the INSERT and DELETE operations. If, for example, a (fully balanced) binary search tree is used in each slot, the worst case runtime for SEARCH is reduced to $\mathcal{O}(\log n)$, while INSERT and DELETE also take $\mathcal{O}(\log n)$ time.[61]

To retain worst case constant time access, the size of the secondary data structure can be restricted. An example of this would be a paged hash table, where each slot contains an array, called a page (or bucket), with a constant number p of slots. While this reduces each of the INSERT, SEARCH, and DELETE operations to a constant run time of at most $\mathcal{O}(p)$, it introduces a new problem: Pages can be full. Unless we are willing to accept that the table can be full and we cannot insert new items, we either need to choose p so that this event is unlikely or employ another collision resolution

[58] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.2.

[59] Cf. Universality, Independence, and Min-Wise Independence

[60] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.2.

[61] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 12.3.

Strategy	Insert	Search	Delete
		Worst Case	
(Doubly) Linked List	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binary Search Tree	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Static Pages	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$
		Average Case	
(Doubly) Linked List	$\mathcal{O}(1)$	$\mathcal{O}(\rho)$	$\mathcal{O}(\rho)$
Binary Search Tree	$\mathcal{O}(\log \rho)$	$\mathcal{O}(\log \rho)$	$\mathcal{O}(\log \rho)$
Static Pages	$\mathcal{O}(\rho)$	$\mathcal{O}(\rho)$	$\mathcal{O}(\rho)$

Table 3.1: Runtimes for closed addressing collision resolution. Here the parameter ρ denotes the expected number of items per slot. We omit the Contains operation, which is identical to Search and the Resize operation, which depends on the Insert operation.

strategy. However, making collisions unlikely requires $p \geq \rho$, which would be less efficient then directly using lists. The advantage of this strategy is a guaranteed memory footprint of $p \cdot |T|$ times the size of the saved data type.

Since the performance of closed addressing strategies depends on the size of the secondary data structures, it can be influenced by performing a Resize. By increasing the number of available slots ($|T|$), while maintaining a constant number of already inserted keys (n), the load factor is reduced, resulting in better expected performance. The performance of a Resize, again, depends on the used secondary data structure.

Another downside of collision resolution by closed addressing is its poor cache performance due to dynamic memory allocation. Since the secondary data structures can not generally assure that colliding items uphold memory locality, additional cache misses might occur, when resolving collisions. Broder and Mitzenmacher[62] proposed an implementation using static pages, which offers better cache performance, but requires a costly Resize, once a page is full. Askitis described a variant of collision by chaining, called array hash, which replaces lists with resizable arrays.[63] If an array is resized, for example grown to insert a new item, it is reallocated to remain memory local. While this approach is cache efficient, it sacrifices efficiency of Insert and Delete operations.

Collision resolution by open addressing does not allow expected constant time access for Search operations, as illustrated in Table 3.1, but offers other advantages: Inserting elements is always possible (for dynamic secondary data structures) and the number of slots can be chosen smaller than with other strategies, since each slot can contain multiple items. As noted above, combining closed address and open address strategies is possible, as we will address later in this section.

3.3.3 Collision Resolution by Open Addressing

In contrast to closed addressing, which keeps all colliding keys in one place, open addressing strategies redistribute colliding keys

[62] Broder and Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups", 2001.

[63] Askitis, "Fast and Compact Hash Tables for Integer Keys", 2009.

throughout the hash table. If a collision occurs, a sequence of alternative addresses, called a *probe sequence*, is computed, until a free slot is found.[64] We will describe the basic probing strategies linear probing, quadratic probing, and double hashing as described by Cormen et al.,[65] followed by cuckoo hashing and hopscotch hashing, which guarantee constant time lookups.

Linear Probing: The most basic open addressing strategy is linear probing, where in case of a collision, we evaluate the following slots until:

- A free slot is found, and we can insert the key, or
- we have evaluated all slots without finding an empty one and terminate.

For two colliding keys $x, x' \in \mathcal{D}$ $h(x) = h(x') = a$ and a hash table T, where $T[a] = x$ has already been inserted, we evaluate the slots

$$a' = a + 1,\, a' = a + 2,\, \ldots,\, a' = a + |T| - 1 \mod |T|$$

until we find an empty slot and insert $T[a'] = x'$. An example for such a probe sequence is illustrated in Figure 3.17.

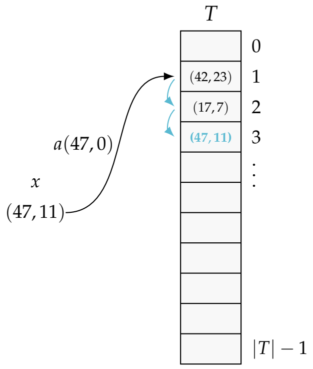


Figure 3.17: Illustration of collision resolution by linear probing. The item $x = (47, 11)$ receives the address $a(x, 0) = 1$, which is already filled. Following the probing sequence (shown as teal arrows) $a(x, 1) = 2$, which again is a filled bucket, to $a(x, 2) = 3$, we find an empty bucket to insert x.

More formally, we use an address function

$$a(x, i) = (h(x) + i) \mod |T| \tag{3.5}$$

to compute hash addresses for a key $x \in \mathcal{D}$ and a probe number $i \in [|T|]$. And during the INSERT operation we insert $T[a(x, i)]$ in the first free slot.

When retrieving an item using SEARCH (x), we follow the same probe sequence, checking all items along the way. Assuming the hash table does not support the DELETE operation, the query can be terminated if either:

- the item was found, or
- we find an empty slot.

Encountering an empty slot means that the probe sequence was exhausted, without encountering x.

Linear probing is attractive due to its simplicity and cache efficiency. Since colliding keys are stored close in memory, a (pre-) fetched cache line is likely to contain all colliding items. The downside of this is, that if two keys with similar hash addresses receive many collisions, their probe sequences can overlap and the resulting *primary clustering*[66] of hash values negatively affects SEARCH times. In the worst case, both INSERT and SEARCH need to evaluate $\mathcal{O}(n)$ slots, one for each inserted item, which again demotes the hash table to a linear list. However, Pătraşcu and Thorup have shown that expected constant time performance can be achieved by using a 5-independent hash function.[67] Variants of this approach change the step size, to avoid its susceptibility to clustering.

[64] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.4.

[65] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.4.

[66] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.4.

[67] Pătraşcu and Thorup, "On the k-Independence Required by Linear Probing and Minwise Independence", 2010.

Quadratic Probing: One way to avoid primary clustering is to increase the step size beyond a linear step size,[68] as illustrated in Figure 3.18. The address function for the quadratic probing strategy combines a linear and a quadratic component, both weighted with a constant value respectively:

$$a(x,i) = (h(x) + c_1 i + c_2 i^2) \mod |T| \tag{3.6}$$

This approach ensures that the probe sequences for different keys diverge, the further we follow them. However, items receiving an identical first address $a(x,0) = a(x',0)$ also have identical probe sequences, resulting in *secondary clustering*. Also, note that the constants c_1 and c_2 need to be chosen carefully to ensure that the whole hash table can be reached.[69]

While preventing primary clustering, quickly moving away from the initial hash address sacrifices memory locality and in turn cache efficiency. Worst case INSERT and SEARCH remain $\mathcal{O}(n)$, if c_1 and c_2 are chosen accordingly.

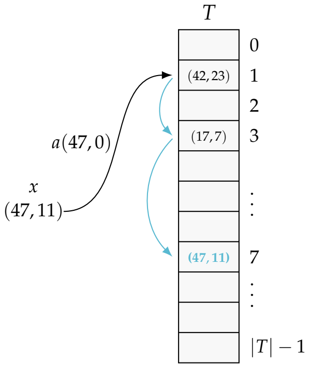


Figure 3.18: Illustration of collision resolution by quadratic probing using the constants $c_1 = c_2 = 1$. The item $x = (47,11)$ receives the address $a(x,0) = 1$, which is already filled. Following the probing sequence (shown as teal arrows) $a(x,1) = 1 + (1 \cdot 1) + (1 \cdot 1^2) = 3$, which again is a filled bucket, to $a(x,2) = 1 + (1 \cdot 2) + (1 \cdot 2^2) = 7$, we find an empty bucket to insert x.

Double Hashing: To avoid secondary clustering, double hashing employs a second hash function to determine a step size for a linear probe. This is similar to choosing $c_2 = 0$ and c_1 depending on x using quadratic probing: Using the address function

$$a(x,i) = (h(x) + i \cdot h'(x)) \mod |T| \tag{3.7}$$

where h' is a hash function with the same domain as h. An example for double hashing is given in Figure 3.19. For optimal results, all items from the codomain should be relative prime to the size of T.[70]

As for the other probing strategies, the worst case time for both INSERT and SEARCH is $\mathcal{O}(n)$. However, it is also possible to achieve constant time access using open addressing.

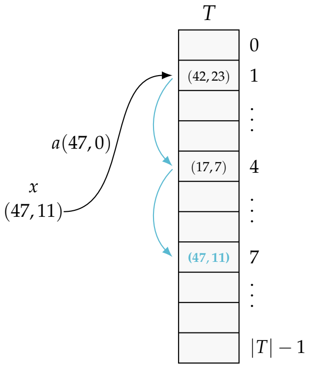


Figure 3.19: Illustration of collision resolution by double hashing. The step size, determined by the seconary hash function is $h'(x) = 3$. The item $x = (47,11)$ receives the address $a(x,0) = 1$, which is already filled. Following the probing sequence (teal arrows) $a(x,1) = 1 + 1 \cdot h'(x) = 4$, which again is a filled bucket, to $a(x,2) = 1 + 2 \cdot h'(x) = 7$, we find an empty bucket to insert x.

Cuckoo Hashing Basic cuckoo hashing, as described by Pagh and Rodler,[71] introduced the idea to remove items from the hash table during collision resolution. Like the bird lending its name to this data structure puts its own eggs into other birds' nests, with cuckoo hashing the INSERT operation removes an already entered item and places it elsewhere. Using two hash tables T_1 and T_2 and corresponding hash functions h_1 and h_2, an item x can be placed either at $T_1[h_1(x)]$ or at $T_2[h_2(x)]$. If the respective slot is already full in T_1, the item y that is currently occupying the slot is removed from the table, and $T_1[h_1(x)] = x$ is inserted. To assure that the now unplaced item y is still present in the table, we try to insert it into T_2 using its second hash function. We test if $T_2[h_2(y)]$ is empty, and repeat as above: Insert y into T_2, if another item z was pushed out for this, try inserting it with its other hash function. This continues until either:

- an empty slot is discovered, or
- a cycle of replacements is detected.

[68] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.4.

[69] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.4.

[70] Cormen et al., *Introduction to Algorithms*, 2009, Chapter 11.4.

[71] Pagh and Rodler, "Cuckoo Hashing", 2004.

If no insertion could be performed, the hash table needs to be rehashed. In practice, detecting cycles is hard, and this is solved using a maximum number of lookups after which the insertion process is terminated. The probability of cycles depends on the properties of the used hash functions. Both h_1 and h_2 need to be at least $\Theta(\lg |T|)$-independent and have to be chosen independently, however this bound is not tight.[72] Cohen and Kane have shown, that 5-independence (alone) is not sufficient.[73]

The main advantage of a cuckoo hash table is that it can perform lookups in worst case constant time. While the INSERT operation, as described above, can create cycles which grow with higher table loads, an item can only be present at exactly two positions: at $T_1[h_1(x)]$ or at $T_2[h_2(x)]$. Hence, for a SEARCH operation, exactly two positions within the tables need to be queried, but this comes at the cost of memory efficiency. Classical cuckoo hash tables are only efficient below a table load of $\rho < 0.5$, however, a large collection of variants of cuckoo hash tables address this issue.

Cuckoo hash tables can use only one table T and both hash functions h_1 and h_2 map to the same codomain, i.e. the address space of T. Pushed out elements are reentered at a different position of the table.[74]

Another popular modification reduces the number of replacement chains by modifying the INSERT operation. Instead of pushing out the item y in the case that $T[h_1(x)] = y$, as described in the example described above, we first use the second hash function to try to place x. If $T[h_2(x)]$ is still free, x is inserted, otherwise one of the two elements is pushed out.

Cuckoo hashing has also be generalized to work with a larger number of hash functions and consequently more possible hash positions for each item.[75] This technique is called k-way or k-ary cuckoo hashing.

Bucketized (or paged) cuckoo hashing[76] combines open and closed addressing strategies. Each slot of T, in this case often called a page[77], can hold a fixed number of items, similar to collision by chaining with fixed size arrays. Items are inserted until a page is full. Once a page is full and is to receive another item, an item currently residing in this page is selected (for example chosen at random) and pushed out. This increases the number of collisions that can be tolerated, before rehashing is required, but again comes at the cost of a larger memory footprint. A modification of bucketized cuckoo hashing which uses overlapping pages and can achieve even higher load factors.[78]

Cuckoo hash tables can be augmented to be more robust against singular colliding items. Consider a set of three keys x_0, x_1, x_2, two (deliberately malevolent) hash functions h_1, h_2 with

$$\begin{aligned} h_1(x_0) &= a & h_2(x_0) &= b \\ h_1(x_1) &= b & h_2(x_1) &= a \\ h_1(x_2) &= b & h_2(x_2) &= a \end{aligned}$$

[72] Cohen and Kane, "Bounds on the Independence Required for Cuckoo Hashing", 2009.

[73] Cohen and Kane, "Bounds on the Independence Required for Cuckoo Hashing", 2009.

[74] Pagh and Rodler, "Cuckoo Hashing", 2004; Drmota and Kutzelnigg, "A Precise Analysis of Cuckoo Hashing", 2012.

[75] Fotakis et al., "Space Efficient Hash Tables with Worst Case Constant Access Time", 2003.

[76] Dietzfelbinger and Weidling, "Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins", 2007.

[77] These are also commonly referred to as buckets. We avoid using the term bucket here to avoid confusion with *normal* hash table slots.

[78] Walzer, "Load Thresholds for Cuckoo Hashing with Overlapping Blocks", 2018.

and a cuckoo hash table with one array, where a and b are addresses within T. When x_0 and x_1 are already in the table, x_2 cannot be entered without rehashing. While the number of these malevolent cases is low, given a good choice of hash functions, a small amount of keys can force a RESIZE operation. To avoid this, a secondary data structure known as a stash can be used.[79] A stash can, for example, be a linked list, which collects all otherwise unplacable items. If an item cannot be inserted into T, it is put into the stash instead. While a stash allows to achieve higher table loads, it means sacrificing worst case constant access time, since we might be forced to search the stash.

Finally, all strategies described above can be combined. Most notably, the combination of k-ary with bucketized cuckoo hashing has proven to achieve high load factors while retaining worst case constant access time.[80] We denote this combination as (k, p)-cuckoo hashing, where k denotes the number of hash functions used and p describes the number of items per page. Usually these hash tables are implemented using only one table. The achievable load factors for both k-ary cuckoo hashing and bucketized cuckoo hashing quickly rise above $\rho = 0.5$ of basic $(2, 1)$-cuckoo hashing, as shown by Walzer[81]. However, when combining these techniques, valid load factors quickly approach $\rho \approx 1$, even for small numbers of k and p.

We explored optimal assignments to a $(3, p)$-cuckoo hash table by computing a cost-optimal matching.[82] Using three hash functions to place entries results in up to three accesses into T, each of which can be expected to cause a cache miss. By increasing the fraction of entries placed using their first or second hash function, we can reduce the amount of cache misses required for CONTAINS and SEARCH operation.

We define two vertex sets of a bipartite graph, the first containing pages, the second comprising items to insert. Hash functions, or more precisely the pages into which an item can be placed, are represented by weighted edges. Thus, for each item x there are three edges pointing to three pages that x can be assigned to. Using the number of hash functions as costs, e.g. the first hash function receives cost 1, the third cost 3, we compute the optimal hash function to store each item by finding a cost-minimal path from all empty pages to all items we want to insert. Such an augmenting path, which we compute using a combination of the Bellman-Ford and Hopcroft-Karp algorithms, represents a chain of replacements required to insert the new item with minimal costs. An example for such paths for one item can be found in Figure 3.20. A replacement constitutes removing an already inserted item from a page and reinserting it using a more expensive hash function, to make room for another entry. For more details refer to our paper[83] describing the approach in detail.

[79] Kirsch, Mitzenmacher, and Wieder, "More Robust Hashing: Cuckoo Hashing With a Stash", 2010.

[80] Erlingsson, Manasse, and McSherry, "A Cool and Practical Alternative to Traditional Hash Tables", 2006.

[81] Walzer, "Load Thresholds for Cuckoo Hashing with Overlapping Blocks", 2018, Table1.

[82] Zentgraf, Timm, and Rahmann, "Cost-optimal Assignment of Elements in Genome-scale Multi-way Bucketed Cuckoo Hash Tables", 2020.

[83] Zentgraf, Timm, and Rahmann, "Cost-optimal Assignment of Elements in Genome-scale Multi-way Bucketed Cuckoo Hash Tables", 2020.

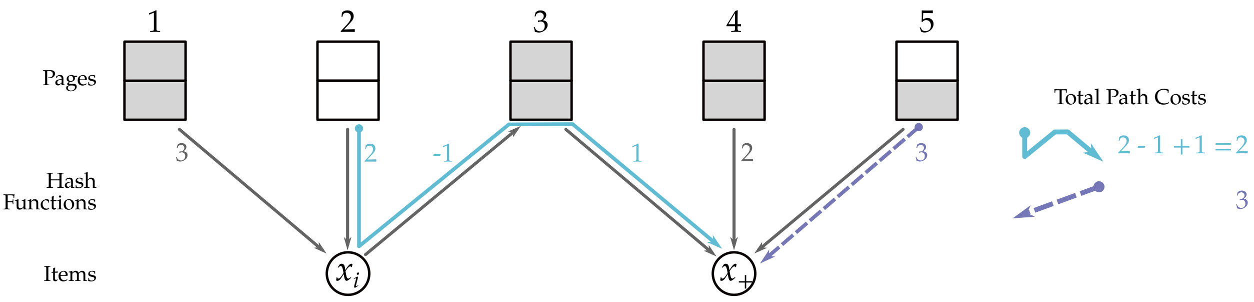


Figure 3.20: There are two possible paths to insert the new item x_+: Inserting it into page 5 at a cost of 3, or moving x_i from its first choice (page 3) to its second choice (page 2) at costs $-1 + 2$ and inserting x_+ into page 3 at cost 1. The teal path is cost-optimal and is therefore selected to insert x_+. This illustration is derived from Figure 2 of our publication: Zentgraf, Timm, and Rahmann, "Cost-optimal Assignment of Elements in Genome-scale Multi-way Bucketed Cuckoo Hash Tables", 2020.

Hopscotch Hashing: Continuing collision resolution by reallocation of keys, the core idea of hopscotch hashing[84] is that an entry x with hash address $h(x) = a$ is guaranteed to be stored in one of the slots $[a, a + H - 1]$ for a constant value of H. To assure this property, entries in the hash table can be moved forward inside their size H neighborhood to make room for other colliding items.

The inserted values are stored in a data array, that contains C entries, where C is the codomain size of the hash function used. In addition to this, we have to maintain a data structure that keeps track of where values have been moved. This is realized with a bitmap array, which contains an H bit vector for each position in the data array.[85]

When inserting an element x with $h(x) = a$, there are three possible options:

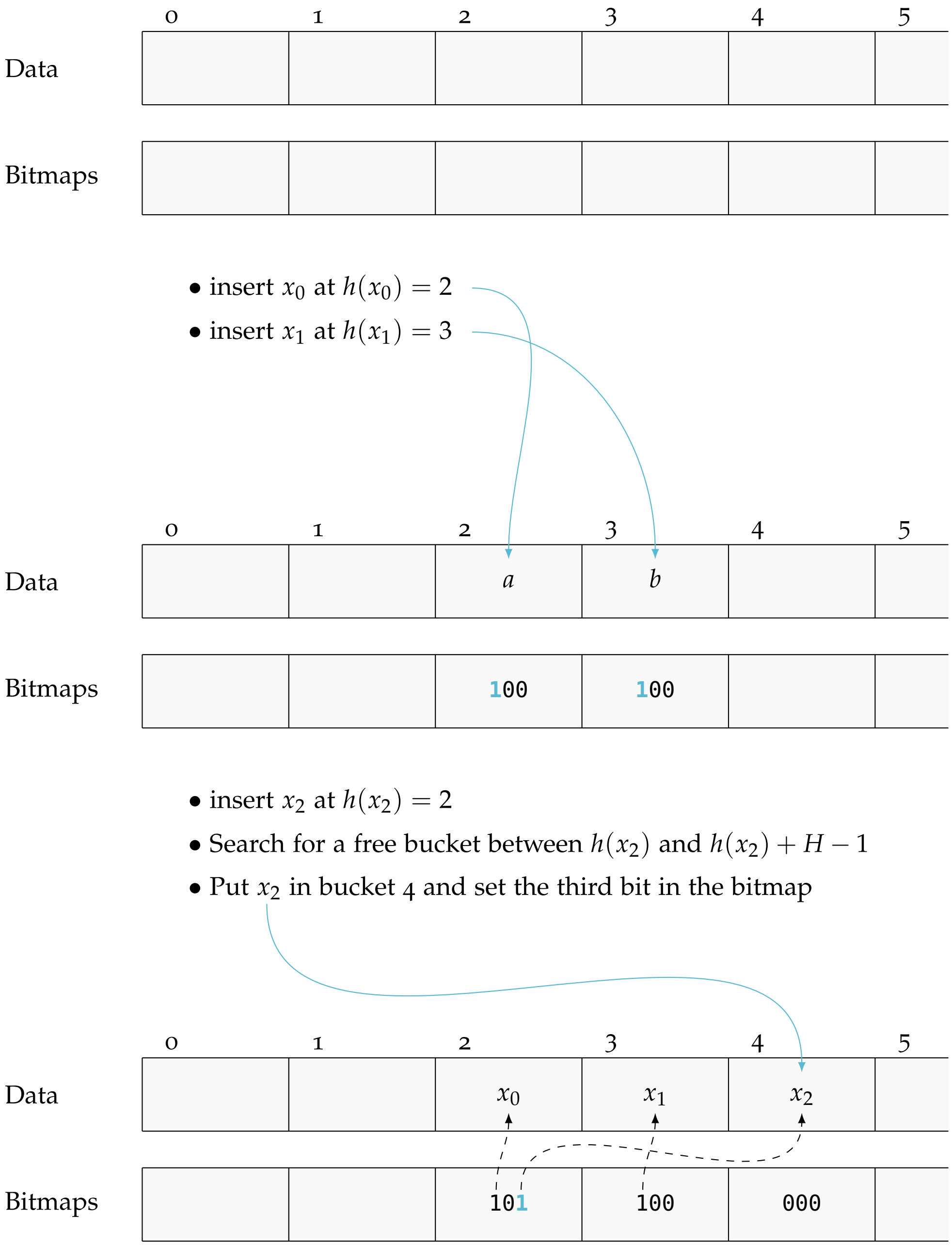
1. The slot a is empty: Insert x into the table, set the first bit in the bitmap of position i.
2. The slot a is full, but an empty slot b with $a < b <= a + H - 1$ exists. Insert x at position b and set the $(b - a)$-th bit of the bitmap of a.
3. The slot a is full and its hopscotch neighborhood is full as well, i.e. the next empty slot b is at a position that is more than H slots away.
   - Find an item that can be moved into position b to free up a slot closer to a.
   - If no such item exists: RESIZE
   - If such an item exists: Move it to position b (updating bitmaps for both positions) and take the freed up position as the new b.
   - Repeat from above (find a new element to move) until either the items can be inserted, or a RESIZE is required.

Through this, empty slots skip towards the insert position, lending this hashing strategy its name derived from the popular children's game. The structure of a hopscotch hash table with three keys inserted and $H = 3$ is illustrated in Figure 3.21.

When checking if a value x is in the hash table, first the bitmap array is queried at position $h(x) = a$ to get all possible positions

[84] Herlihy, Shavit, and Tzafrir, "Hopscotch Hashing", 2008.

[85] This can be implemented using integer numbers to save memory. This approach is used by the implementation in libhhash.


0 1 2 3 4 5
Data
Bitmaps
• insert x_0 at $h(x_0) = 2$
• insert x_1 at $h(x_1) = 3$
0 1 2 3 4 5
Data
a
b
Bitmaps
100
100
• insert x_2 at $h(x_2) = 2$
• Search for a free bucket between $h(x_2)$ and $h(x_2) + H - 1$
• Put x_2 in bucket 4 and set the third bit in the bitmap
0 1 2 3 4 5
Data
x_0
x_1
x_2
Bitmaps
101
100
000

The dashed lines visualize the information encoded in the auxiliary array. keys with hash value 2 can be found in bucket 2 (denoted by the first 1-bit) and bucket 4 (denoted by the 1-bit at position 3), while bucket 3 does not contain a key with hash value 2.

Figure 3.21: Illustration of a hopscotch hash table with $H = 3$. Three insert operations with the keys x_0, x_1, and x_2 with the respective hash values 2, 3, and 2 are performed.

Strategy	INSERT	SEARCH	DELETE
	Worst Case		
Linear probing	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Quadratic probing	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Double hashing	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Classic cuckoo hashing	$\mathcal{O}(n)$	$\mathcal{O}(2)$	$\mathcal{O}(2)$
(k,p)-cuckoo hashing	$\mathcal{O}(n)$	$\mathcal{O}(kp)$	$\mathcal{O}(kp)$
Hopscotch hashing	$\mathcal{O}(n)$	$\mathcal{O}(H)$	$\mathcal{O}(H)$
	Average Case		
Linear probing (5-ind.)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Classic cuckoo hashing	$\mathcal{O}(1)$	$\mathcal{O}(2)$	$\mathcal{O}(2)$
(k,p)-cuckoo hashing	$\mathcal{O}(1)$	$\mathcal{O}(kp)$	$\mathcal{O}(kp)$
Hopscotch hashing	$\mathcal{O}(1)$	$\mathcal{O}(\alpha)$	$\mathcal{O}(\alpha)$

Table 3.2: Runtimes for open addressing collision resolution. Linear probing can achieve expected constant time INSERT and SEARCH operations when using a 5-independent hash function. For cuckoo hashing, k and p are constants denoting the number of hash functions and slots per page respectively. Constant times for INSERT operations (AC) for cuckoo hashing are amortized times. The constant H for hopscotch hashing is the number of successive slots an item can be stored in. For the average case of hopscotch hashing, $\alpha = 1 + ((e^{2\rho} - 1 - 2\rho)/4)$ is the expected number of items per bucket, which is expected to be constant. We omit the CONTAINS operation, which is identical to SEARCH and the RESIZE operation, which depends on the INSERT operation.

of x in the data array. Then, all positions in the data array pointed to by the bitmap are accessed and evaluated. Due to this design, a SEARCH call is expected to cause exactly two cache misses. The first, when accessing the bit vector to get the neighborhood information, followed by the access into the storage array, which causes the second access to an uncached page. Since all positions in which the value can reside are within the $[a, a + H - 1]$, all possible positions fall into the same cache line. As long as $w \cdot H \leq L$ holds, where w is the size of one entry in the hash table (in bits) and L is the cache line size (in bits).

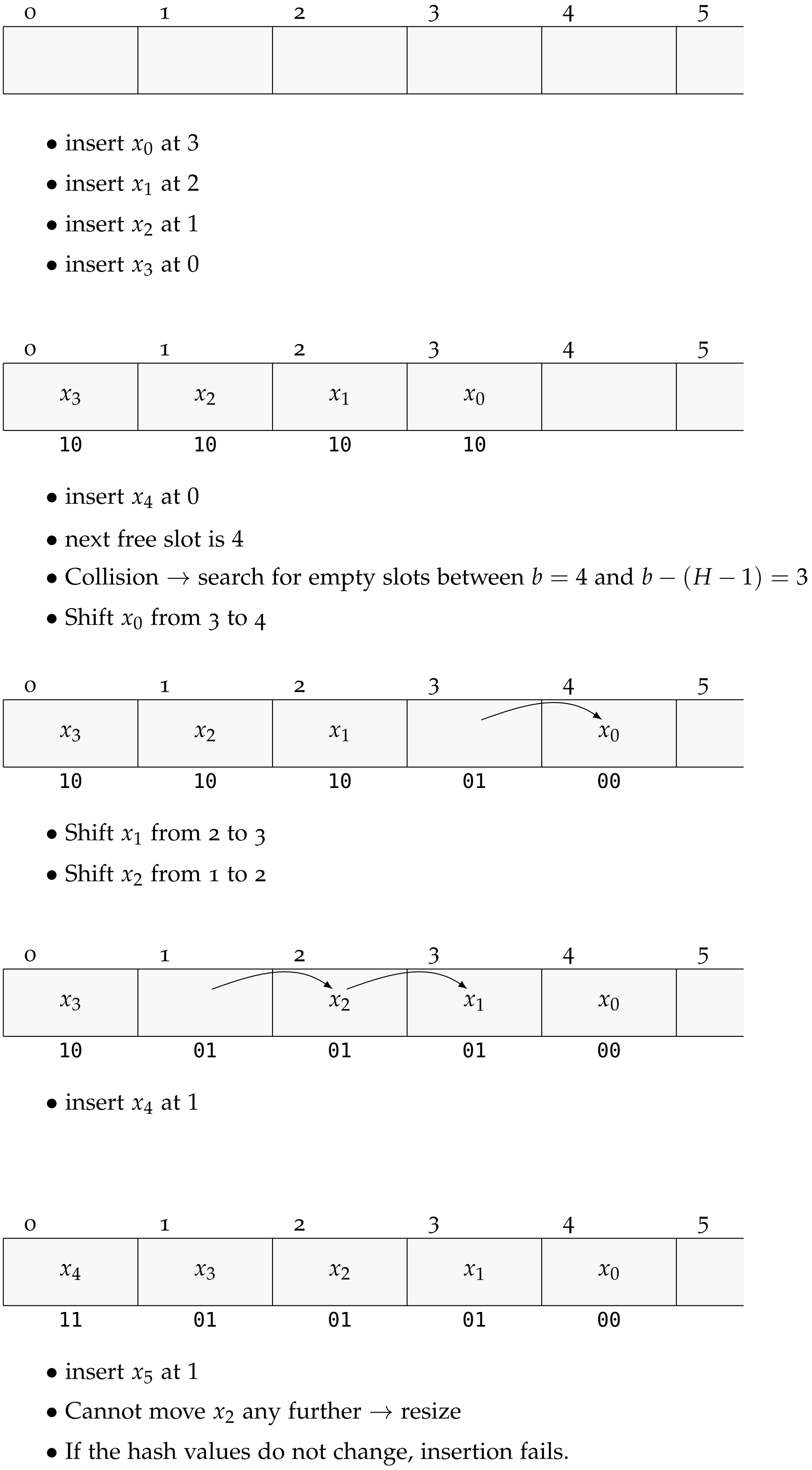
Asymptotically, SEARCH and DELETE operations on a hopscotch hash table can be performed in $O(H)$ time, since there are only H possible positions at which a query item can be located. While the INSERT operation requires can create a large chain of displacements, and finally trigger a RESIZE, the expected time required for INSERT operations is constant.[86] As linear probing, hopscotch hashing is affected by primary clustering, which can ultimately result in several sequential resizes. An example for this is illustrated in Figure 3.22.

3.3.4 Runtimes of Open Addressing Strategies

In conclusion, open addressing collision resolution offers constant time queries, similar to perfect hashing. Runtimes for all discussed strategies are aggregated in Table 3.2.

Deciding for a strategy is a tradeoff that needs to be informed by the specific use case. Cuckoo hashing, especially the $(k,p)-$cuckoo hashing variant, can offer constant time access, but with a rising number of possible cache misses, as k rises. Hopscotch hashing guarantees finding an item with exactly two cache misses, however this comes at the downside of being susceptible to primary clustering.

[86] Herlihy, Shavit, and Tzafrir, "Hopscotch Hashing", 2008, Lemma 3.5.



0	1	2	3	4	5

- insert x_0 at 3
- insert x_1 at 2
- insert x_2 at 1
- insert x_3 at 0

0	1	2	3	4	5
x_3	x_2	x_1	x_0		
10	10	10	10		

- insert x_4 at 0
- next free slot is 4
- Collision → search for empty slots between $b = 4$ and $b - (H - 1) = 3$
- Shift x_0 from 3 to 4

0	1	2	3	4	5
x_3	x_2	x_1		x_0	
10	10	10	01	00	

- Shift x_1 from 2 to 3
- Shift x_2 from 1 to 2

0	1	2	3	4	5
x_3		x_2	x_1	x_0	
10	01	01	01	00	

- insert x_4 at 1

0	1	2	3	4	5
x_4	x_3	x_2	x_1	x_0	
11	01	01	01	00	

- insert x_5 at 1
- Cannot move x_2 any further → resize
- If the hash values do not change, insertion fails.

Figure 3.22: Example for failure to insert with hopscotch hashing due to skewed hash values. Each value can be put in one of $H = 2$ bins, i.e. in this example a hash value can be moved at most one bin to the right. The content of the auxiliary array for each bin is written below the bin.

In Bioinformatics, hash tables have many applications that range from strong seeds for seed-and-extend alignment algorithms, q-gram counting, set cardinality estimation, any many additional tasks so that a complete discussion falls outside the scope of this work. Instead, we will focus on the use of hash tables for sequence similarity estimation. Contrary to the notion we held up during this chapter, collisions may be beneficial, and can harnessed to determine how similar two sequences are. The chapter Computing and Approximating Resemblance and Containment (p. 89) describes the concept of locality sensitive hashing (LSH), i.e. the construction of hash functions that return identical, or similar, hash values for similar and identical keys. In the following chapter, we describe a KVS hash table we developed based on hopscotch hashing, while also reducing the number of cache misses per SEARCH operation to one.

4
Reducing Cache Misses in Hopscotch Hash Tables

Previously, we described hopscotch hashing[1] as a method for open addressing collision resolution (cf. Section 3.3.3). One of the main advantages of hopscotch hashing is its cache friendliness during collision resolution. A hopscotch hash table comprises a data array, in which values or key-value pairs are stored together with a bitmap array managing the positions of colliding entries. Keys are stored either at their target address or in the following $H - 1$ slots, which have a high chance to reside in the same cache line. Accessing this array can be expected to cause a compulsory cache miss. However, a second compulsory cache miss is caused by indexing the bitmap array to identify moved entries. We developed a bit-packed hopscotch hash table (BPHT) that uses bit-packing in combination with quotienting to allow SEARCH operations with as little as one compulsory cache miss.

While other implementations[2] also interleave the hop information with data information, for example through the use of C-structs as hash table entries, they allocate a fixed value of up to 64 hop bits. The implementation proposed by Kourtis et al.[3] utilizes a quotienting approach similar to our architecture, however, they do not leverage the power of bit-packing and cannot perform in-place resizing.

4.1 Hash Table Architecture

To eliminate the second cache miss, we save the information of the bitmap array in the data array alongside the stored values. We realize this, while maintaining a resizable architecture, by employing two techniques: quotienting and bit-packing (cf. Quotienting and Packing, p. 53). Through quotienting, we avoid explicitly saving keys and are still able to perform RESIZE operations. Bit-packing allows us to encode different information, namely value, quotienting remainder, and collision information, into one integer value. However, this imposes a restriction on the kind of values we can store.

The fundamental data structure of a BPHT is an integer array T of size $|T| = |\mathcal{A}| + (H - 1)$ where $\mathcal{A} = [2^{\mathfrak{a}}]$, $\mathfrak{a} \in \mathbb{N}$ is the address space of the quotienting function and H is the hopscotch neighbor-

[1] Herlihy, Shavit, and Tzafrir, "Hopscotch Hashing", 2008.

[2] Kim and Kim, "Enhanced Chained and Cuckoo Hashing Methods for Multi-core CPUs", 2014; Elbayoumi, "Strategies for Quality and Performance Improvement of Hardware Verification and Synthesis Algorithms", 2014.

[3] Kourtis, Ioannou, and Koltsidas, "Reaping the Performance of Fast NVM Storage with uDepot", 2019.

hood size.[4] This array comprises A addresses to which entries can be assigned and $H-1$ additional slots at the end of the array that serve as hopscotch neighborhood for the last $H-1$ addresses. Restricting hash table sizes to powers of 2 allows us to implement an efficient quotienting strategy and efficient Resize operation. While the collision properties of this approach are not optimal,[5] we value the speedup gained through efficient address computation higher. We describe the modifications required for arbitrary hash table sizes in the discussion. Additionally, the possible sizes of our hash table depend on the amount of bits available to store remainders. We discuss the used quotienting technique in more detail below, since it interacts with caching behavior and architecture of Resize operations.

We use an array of 64-bit (unsigned) integer values to realize T and we require the codomain of the hash function $h : \mathcal{D} \to \mathcal{C}$ to be $\mathcal{C} = [2^{32}]$. This is not a hard requirement, as we will describe in the discussion, but simplifies the architecture for both description and implementation.

4.1.1 Bit-Packing

An empty BPHT consists of a 64-bit array initialized with zeros. Into each entry of T we encode three different kinds of information:

- The payload of the entry, i.e. the value that is stored in the hash table.
- The remainder of the quotienting function used to resolve soft collisions.
- H bits used for collision resolution (both soft and hard collisions), which are saved in the bitmap array of basic hopscotch hashing. We will refer to these as hop bits from now on.

Recall that two entries that are assigned to the same slot but are still distinguishable through their remainders are called soft collisions, while different entries stored for the same address and remainders are called hard collisions.

We allocate 32 bits to the payload value and the remaining 32 bits to administrative bits, namely remainder bits and hop bits. Note that for specific use cases, fewer bits might be required for the value. For example for xenograft sorting, where q-grams are assigned to a host genome, a donor genome, or both, we only need to assign a q-gram to one of three groups and hence only require two value bits. While a variable number of value bits allows using more administrative bits and therefore higher values of H or a longer remainder, we restrict this implementation to a fixed number of 32 value bits to simplify its description.

A visualization of this bit-packing layout can be found in Figure 4.1. Payload data is stored in the high 32 bits of the 64-bit value (bits 32 to 64). Hop bits are stored in reverse order in the lower H

[4] Notice, that we denote the set of addresses as $\mathcal{A} = \{0, \ldots, 2^a - 1\}$ which contains $A = 2^a = |\mathcal{A}|$ different addresses.

[5] Knuth, *The Art of Computer Programming: Sorting and Searching*, 1997, Chapter 6.4.

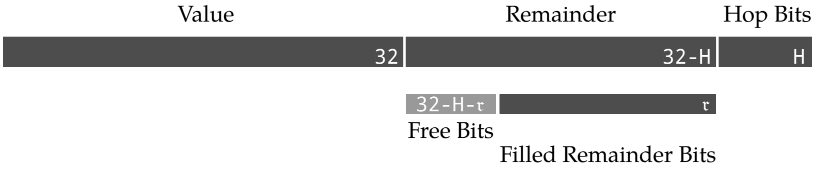
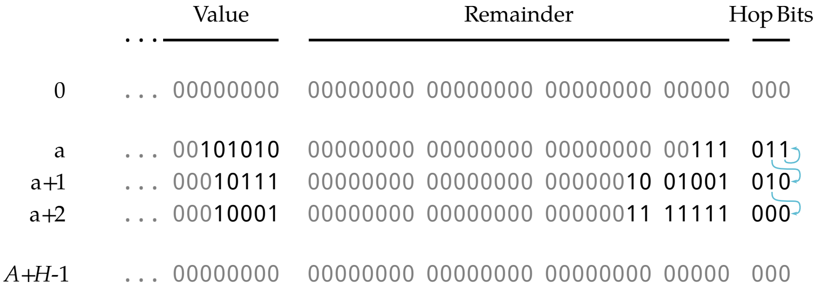


Figure 4.1: Bit-packing layout of 64-bit entries of a BPHT. The number of bits allocated to the value (payload) and hop bits are constant, while the number of remainder bits depends on the hash table size and is decrease when a Resize operation is performed.



	Value	Remainder	Hop Bits
0	. . . 00000000	00000000 00000000 00000000 00000	000
a	. . . 00101010	00000000 00000000 00000000 00111	011
a+1	. . . 00010111	00000000 00000000 00000010 01001	010
a+2	. . . 00010001	00000000 00000000 00000011 11111	000
A+*H*-1	. . . 00000000	00000000 00000000 00000000 00000	000

Figure 4.2: Example of a BPHT with three inserted elements. Empty slots except for the first and last slot are not shown. Leading runs of zeros are shown in gray to increase readability. Into this table the value 101010_2 was inserted at address a with remainder $r = 111_2$. The value 101111_2 was inserted with a and $r = 10\,01001_2$ and stored in slot $a + 1$. Finally, 10001_2 was inserted with address $a + 1$ and $r = 11\,11111_2$ and stored in slot $a + 2$. The relative positions of entries to their address is denoted by teal arrows starting from their respective hop bit.

bits. The rightmost (least significant) bit denotes the slot itself, the bit at position 1 points to the following slot. With H bits assigned to hop bits, the remaining $32 - H$ bits hold remainder information of the quotienting function. Depending on the size of the remainder space $\mathcal{R}$ of the quotienting function, the number of remainder bits $\mathfrak{r}$ can vary between 0 for $A = 2^{32}$ and $32 - H$ for $A = 2^H$. Remainder bits populate the bit-positions H to $H + \mathfrak{r}$, where $\mathfrak{r} = \log_2 R$ is the number of bits required to represent the remainder space of the quotienting function. An example of a BPHT with $H = 3$, which illustrates quotienting remainders saved using the bit-packing scheme described above, is shown in Figure 4.2.

When we access entries in T, we can unpack the three parts of an entry using efficient bitwise-AND and bit shift operations. To extract remainder bits, we maintain a bit-mask that depends on the size of $\mathcal{R}$. Note that entries that comprise both the value $v = 0$ and the remainder $r = 0$ are distinguishable from unfilled slots through hop bits. For such a zero entry, there is a slot in T which contains a hop bit pointing to said entry.

We do not explicitly save the key in a BPHT to reduce the memory footprint of the hash table. To perform Resize operations, we only need to restore the initial hash value, as we will show below. For a hash table that allows to retrieve keys in addition to values we can use an invertible hash function.

4.1.2 *Quotienting*

When using quotienting in a hash table of size $2^{\mathfrak{a}}$, we interpret hash values as a bit vector that can be split into address and remainder. The address is used to select a slot in the array T where the entry is stored. Remainders are used to resolve soft collisions, i.e. to distinguish entries that received the same address from different hash values. Using a quotienting function

$$\delta(y) : \mathcal{C} \to (\mathcal{A} \times \mathcal{R})$$

where $\mathcal{A}$ is the address space of the hash table, $\mathcal{R}$ is the space of remainder values, and y is a hash value, we denote addresses as $a \in \mathcal{A}$ and remainders as $r \in \mathcal{R}$.

The number of remainder bits and hop bits as well as the size of the hash table's address space are interdependent. We required above, that both $A = [2^{\mathfrak{a}}] = |\mathcal{A}|$ and $R = [2^{\mathfrak{r}}] = |\mathcal{R}|$ are powers of 2, and $\mathfrak{a} + \mathfrak{r} = 32$. Since only 32 bits out of the 64 bits available per slot remain, we have to choose R and H so that $\mathfrak{r} + H \leq 32$ holds. In other words, the number of remainder bits and hop bits need to be encodable into 32 bits through bit-packing, which leaves us with a discrete space of possible hash table configurations.

To explore an extreme case as an example for this, consider a very small hash table of size $A = 2^5$. When using quotienting with a small hash table, many bits of the hash value are assigned to the remainder. In this case, $\mathfrak{r} = 32 - 5 = 27$ bits are required, leaving only 5 of the administrative bits for hop bits, restricting H to be ≤ 5.

We employ the quotienting function $\delta_{\mathfrak{a}}$ which uses the highest $\mathfrak{a} = \log_2 A$ bits of a 32-bit hash value as address and the low $\mathfrak{r} = \log_2 R$ bits as remainder.

Definition 4.1.1. *High-bit address quotienting: Given a 32-bit hash value $y \in \mathcal{C}, \mathcal{C} := [2^{32}]$ and a hash table with size $A \in 2^{\mathfrak{a}}, \mathfrak{a} \in [H, 32]$, we define the* ***High-bit address quotienting function*** $\delta_{\mathfrak{a}}(y) : [2^{32}] \to ([2^{\mathfrak{a}}], [2^{\mathfrak{r}}])$ *as*

$$\delta_{\mathfrak{a}}(y) := \left(\frac{y}{A}, \, y \bmod A\right)$$

The corresponding inverse quotienting function joins the bit vectors again.

Definition 4.1.2. *Inverse high-bit address quotienting: Given an address remainder pair (a, r) and a hash table with size $A \in 2^{\mathfrak{a}}, \mathfrak{a} \in [H, 32]$, we define the* ***Inverse high-bit address quotienting function*** $\delta_{\mathfrak{a}}^{-1}(a, r) : ([2^{\mathfrak{a}}], [2^{\mathfrak{r}}]) \to [2^{32}]$ *as*

$$\delta_{\mathfrak{a}}^{-1}(a, r) := a \cdot 2^{\mathfrak{r}} + r$$

Figure 4.2 illustrates quotienting remainders saved using the bit-packing scheme described above.

4.2 Hash Table Operations

The bit-packing and quotienting strategies require additional computations for all hash table operations.

For the handling of hard collisions, i.e. key-value pairs entered with identical address and remainder, we describe two possible modes. Hard collisions can be handled as usual for hopscotch hashing, by storing hard colliding keys in the hopscotch neighborhood (normal mode). We also describe a Replacing Insert, which replaces the value already contained in the table with another entry (counting mode). This behavior can, for example, be used to implement a q-gram counter, where each insert increments an integer counter stored in the value bits. Note, that soft collisions are still handled as usual for hopscotch hashing in both cases.

Contains *and* Search When performing a Search operation with a key y, we compute address a and remainder r using the quotienting function $(a, r) = \delta_{\mathfrak{a}}(y)$. All entries in $T_{[a:a+H]}$ are candidates to contain values for y. However, only values in slots $T_i\ i \in [0, H-1]$ that also satisfy the following conditions actually contain values for y:

1. The hop bits of $T_{[a]}$ indicate that $T_{[i]}$ is filled with a value for y.

2. The remainder value of $T_{[i]}$ is identical to the remainder r of the key (soft collision resolution).

In contrast to basic hopscotch hashing, we need to verify for each entry referenced by hop bits of $T_{[a]}$ that their remainders are also identical. For Contains operations, we return true if one matching remainder was found. Depending on the hard collision handling strategy of the BPHT, a Search operation can either return the first entry (replacement strategy), or all up to H entries (chaining strategy).

Insert *and* Replacing Insert For Insert operations with a key y, we search for the closest empty slot b using linear probing as with basic hopscotch hashing. If b is within $H-1$ slots of the address a computed for hash value y, we place the entry in the table and set the corresponding hop bit in slot $T_{[a]}$. Otherwise, we proceed to move empty slots towards a. When moving entries to different slots during this process, we modify the bit-packed hop bits inside T and move both value and remainder to the new slot. For an entry with address c stored in slot $c+o$ that moved from slot $T_{[c+o]}$ to $T_{[b]}$, we need to move value and remainder from $T_{[c+o]}$ to $T_{[b]}$, while the hop bits of $T_{[c]}$ need to be changed. If no free slot can be found, a Resize operation is performed.

For a Replacing Insert operation with key y, we perform a Search operation as described above. If y is found in the table, we retain hop bits and remainder and replace the value in the target slot. Otherwise, if no entry for y was found, we proceed as with Insert.

Delete Similarly, for Delete operations we first identify the position at which the entry is stored relative to its address using the hop bits of $T_{[a]}$. After removing the entry from its assigned slot, say $T_{[b]}$, by replacing both value and remainder in $T_{[b]}$ with zero, we also unset the hop bit pointing to b from the hop bits in $T_{[a]}$.

RESIZE A RESIZE is performed, when a key needs to be inserted at a position with no free slots and additionally, no free slot can be shifted towards the target address a. In this case, a RESIZE increases the number of slots in T to make additional slots available. This requires updating addresses and remainders of all entries already in the table, since the sizes of $\mathcal{A}$ and $\mathcal{R}$ change. However, RESIZE operations using quotienting can be performed without rehashing keys. Since we restricted hash table sizes to powers of 2, we receive new values $\mathfrak{a}' = \mathfrak{a} + 1$ and $\mathfrak{r}' = \mathfrak{r} - 1$ so that a hash table with $A + H - 1$ slots has $A' = 2A + H - 1$ slots after one RESIZE step. Intuitively, one bit is removed from the remainder stored in T and used as an additional (least significant) address bit.

Our hash table implementation is able to perform in-place RESIZE operations on the extended table with $64H$ bits additional memory to increase the size of the hash table.

When migrating from the quotienting function $\delta_{\mathfrak{a}}$ to $\delta_{\mathfrak{a}+1}$, the most significant bit (MSB) of the old remainder r becomes the least significant bit (LSB) of the new address a'. Thus, the entry can be moved only to two possible slots, depending on the MSB of r:

$$a' = 2a \text{ if the MSB of } r \text{ is } 0$$

$$a' = 2a + 1 \text{ if the MSB of } r \text{ is } 1$$

The resized hash table can be interpreted as interspersed with new empty slots, as illustrated in Figure 4.3.

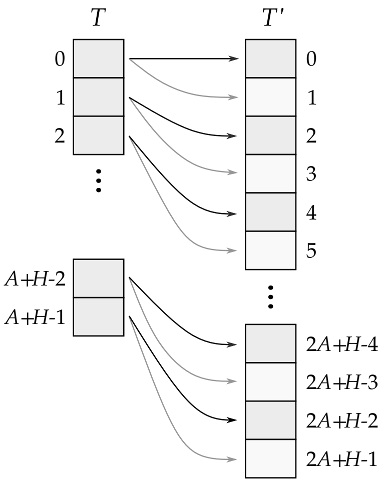


Figure 4.3: Example of the resized BPHT array. The left column shows the array before resizing, the right column after resizing. Entries from slot a can either be assigned to slot $a' = 2a$ if the bit added to a' was 0 (black arrows) or $a' = 2a + 1$ if the bit was 1 (light gray arrows). Note that the $H - 1$ additional slots are not doubled.

The RESIZE operation is performed according to Algorithm 1. First, we increase the size of T from $A + H - 1$ slots to $2A + H - 1$ slots, keeping the old entries in the lower half of T at their old addresses. Beginning from slot $A - 1$—the highest possible address before the resize—and iterating towards lower addresses we remove all up to H entries for each slot, recompute a' and r', and reinsert them at the new address. Until we reach slot $2H - 1$, the newly inserted entries cannot collide with old entries still in the table. For each slot, we remove and reinsert them at their new addresses.

Theorem 4.2.1. *An item inserted into a resizing BPHT T at address $a' > H - 1$ cannot be stored in a slot that still contains old (not rehashed) entries.*

Proof. Let a be the highest unshifted address in a BPHT T during a resize. After removing all entries for a from the table, the highest non-shifted entry in T can be stored at position $(a - 1) + H - 1$, which is the highest colliding entry for address $a - 1$. To not collide with a not yet updated entry, the new address a', which is either $2a$ or $2a + 1$, has to be larger than this. For the smallest possible new address $a' = 2a$ we see:

$$\begin{aligned} & a' > (a-1) + H - 1 \\ \Leftrightarrow \quad & 2a > a + H - 2 \\ \Leftrightarrow \quad & a > H - 2 \end{aligned}$$

Algorithm 1: In-place RESIZE of a BPHT. The last $H-1$ entries need to be handled separety to avoid collisions between reinserted updated entries and non-updated entries in the Hopscotch neighborhood of lower addresses.

```
Input:
• BPHT $T$, with $A + H - 1$ slots and $A = [2^{\mathfrak{a}}]$
• An empty vector $t$ holding up to $H$ 64-bit values
Note:
▷ Entries e of a BPHT comprise an address a, a remainder r and a value v

Increase size of $T$ from $A + H - 1$ to $2A + H - 1$ slots;

▷ Update all non-colliding entries
for $a \leftarrow A$ to $H - 1$ do
    Extract all entries $e := (a, r, v)$ with address $a$ from $T$ into the vector $t$;
    foreach Entry $e \in t$ do
        Restore the hash value $y = \delta_{\mathfrak{a}}^{-1}(a, r)$;
        Compute new address and remainder $a', r' = \delta_{\mathfrak{a}+1}(y)$;
        Insert $v$ into $T$ at address $a'$ with remainder $r'$;

▷ Update the last $H - 1$ entries
for $a \leftarrow H - 1$ to $0$ do
    Extract all entries $e := (a, r, v)$ with address $a$ from $T$ into the vector $t$;
foreach Entry $e \in t$ do
    Restore the hash value $y = \delta_{\mathfrak{a}}^{-1}(a, r)$;
    Compute new address and remainder $a', r' = \delta_{\mathfrak{a}+1}(y)$;
    Insert $v$ into $T$ at address $a'$ with remainder $r'$;
```

As long as the original address a was at least $H - 1$, the new address a' is larger than the highest possible slot for address $a - 1$. □

Depending on the nature of the keys in T, it is possible that one RESIZE does not allow the new entry to be inserted. When the neighborhood of the slot into which the new entry is to be inserted contains only entries with identical highest remainder bit (identical to the key to be inserted) this collision is not resolved. A similar behavior is illustrated in Figure 3.22, however using quotienting, consecutive RESIZE operations move more remainder bits into the address and can prevent endless resizing. In this case, a series of RESIZE operations need to be performed, until at least one slot is freed for the new entry. To avoid multiple resizing, it is possible to evaluate the entries of the offending address and compute the required address space size to resolve the conflict. However, since entries from other addresses which are moved to different slots during a resize can also free up another slot that resolves this conflict, this strategy is only required for the case that all possible slots for a are filled with entries for a.

4.3 *Cache Efficiency*

Our architecture retains the cache-friendliness of hopscotch hashing while also reducing the number of compulsory cache misses during lookups. Most operations on a BPHT are linear accesses to a contiguous array with one compulsory cache miss. Consecutive array accesses can be automatically detected by the processor and can hence benefit from hardware prefetching.[6]

Search, Contains, and Delete operations need to evaluate at most H positions in the array. Depending on the choice of H and the size of a cache line L (in bytes), the probability that all entries for one address are contained in one cache line is

$$\mathbb{P}(H \text{ entries in cache line}) = \frac{\max\{\,0, \ell - (H-1)\,\}}{\ell}$$

where $\ell = L/8$ is the number of 64-bit integers fitting into one cache line. However, since hopscotch neighborhoods overlap, the fact that one neighborhood is perfectly cache aligned means that the following neighborhoods have a higher chance to be unaligned.

Insert operations can be more expensive, since the array might be traversed twice, one time forward during linear probing to find an empty slot and the second time in reverse when empty slots are moved. However, through temporal locality of these accesses, cache lines containing the entries can be expected to still be present at a high cache level for the reverse pass.

Resize comprises one iteration of linear array accesses to extract and reinsert keys. The accesses that occur during reinsertion are not guaranteed to be in a pattern predictable by hardware prefetching. However, for high table loads, it is possible that a linear pattern that can exploit hardware prefetching emerges.

4.4 *Hash Functions*

We described above that using the quotienting function $\delta_{\mathfrak{a}}$, we do not require rehashing keys for resizing. To allow restoring keys from hash values, we employ the invertible multiplicative hash function family

$$\mathcal{H}^{\text{im}} := \{\, h^{\text{im}}_{m,A} \mid m \in \mathbb{N},\ m \equiv 1 \mod 2,\ A \in \mathbb{N}\,\}$$
$$h^{\text{im}}_{m,A}(x) := xm \mod A$$

for an uneven multiplier m and a hash table address range of $\mathcal{A} = [A]$. This hash function allows restoring the initial key from a hash value by multiplying with the multiplicative inverse:

$$h^{\text{im}-1}_{m,A}(x) = xn \mod A \qquad \text{with} \quad n := m^{\frac{A}{2}-1} \mod A$$

The multiplicative inverse n only needs to be computed once for each hash function using modular exponentiation. This hash function has the additional advantage that it is fast to compute. It does,

[6] Drepper, *What Every Programmer Should Know About Memory*, 2007.

however, not offer any statistical guarantees about its collision behavior.

Using a non-invertible universal or independent hash function forfeits the option to restore keys for the benefit of guaranteed collision behavior, which can manifest in higher fill rates. This might also result in a longer computation time for hash function evaluation. We evaluate several different hash functions described in the section Hash Functions for Use in Bioinformatics (p. 44).

4.5 *Evaluation*

To evaluate our hash table architecture, we implemented a BPHT in the Rust programming language.[7] This implementation can also be found on GitHub under the open source MIT License.[8] The workflow used to perform the evaluations and to generate the plots shown in this chapter[9] is contained in a different repository.[10] Our hash table implements two possible behaviors to resolve hard collisions: If not noted otherwise, hard collisions are handled by hopscotch hashing as described above. When used in counting mode, a BPHT contains an integer counter per slot that is increased by 1 each time a hard colliding key is inserted.

4.5.1 *Fill Rates*

The achievable fill rates of a BPHT depend on several aspects: The number of available hash table slots $A = 2^{\mathfrak{a}}$ and the size of the hopscotch neighborhood H, both in relation to the number of inserted keys, are major influencing factors. Additionally, the used hash function and the number of collisions can affect the fill rates.

To show the achievable fill rate of BPHTs in an optimal environment, we evaluated sets of randomly chosen unique keys to rule out the influence of different hash functions and to prevent hard collisions. Note that for all hash table sizes with $\mathfrak{a} < 32$, soft collisions still occur. We sampled fractions of all possible 2^{32} keys relative to the size of the hash table for each combination of hash table size and hopscotch neighborhood. These sets of $\{ 0.05A, 0.1A, \ldots, 0.95A, A \}$ keys were inserted into the BPHT with fixed size that was not able to resize. Each combination was evaluated 10 times, each time with a new set of keys. At the point when an unresolvable collision occurred, i.e. no room could be found for an entry without resizing, we assigned these items to a stash. In a productive context, stashed items would need to be saved in a secondary data structure.

Figure 4.4 shows the achieved fill rates of BPHTs. It can be seen, that fill rates rise linearly with the number of inserted keys, until a point depending on the value of $\mathfrak{a}$, i.e. the table size. After this break point, keys are started to be assigned to the stash and the possible fill rates of the tables drop. For smaller table sizes, the amount of possible soft collisions is increased, since an additional

[7] Timm, *BPHT Source Code*, 2020.

[8] `https://github.com/HenningTimm/bpht`

[9] Timm, *BPHT Evaluation Workflow*, 2020.

[10] `https://github.com/HenningTimm/bpht_evaluation_workflow`

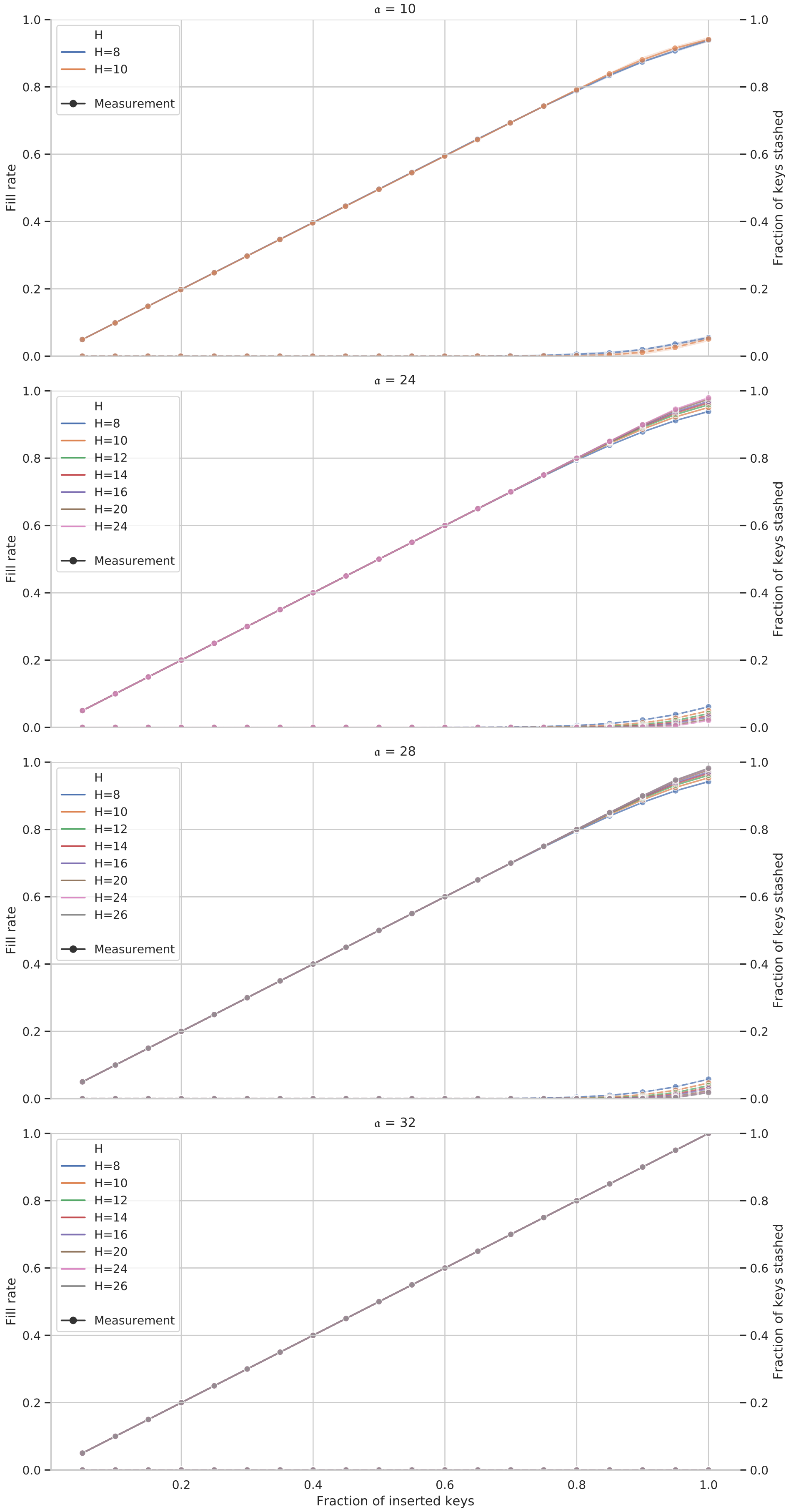


Figure 4.4: Fill rates achievable by a BPHT. Different hash table sizes—denoted as different values of $\mathfrak{a}$—are illustrated along the rows of the plot. Circles denote the mean achieved fill rate of 10 runs for each measurements, with lines connecting the dots for better readability. Circles connected by dashed lines in the lower right quadrant of the plots denote the fraction of stashed keys. Note that no all combinations of $\mathfrak{a}$ and H are possible, due to the restriction that the $32 - \mathfrak{a}$ remainder bits and H hop bits are stored in one 32-bit integer value.

Genome	Length (in bp)	Source
Myxococcus xanthus	9 139 763	Ensemble Genomes, *M. xanthus DK 1622*, 2018
Plasmodium falciparum (Malaria)	23 268 702	Wellcome Sanger Institute, *P. falciparum clone E5 Version 1*, 2017
Humulus lupulus (Hops)	1 812 501 705	Natsume et al., "The Draft Genome of Hop (Humulus lupulus), an Essence for Brewing", 2015
Homo sapiens HG38	3 209 286 105	NCBI, *Genome Reference Consortium Human Build 38*, 2019

Table 4.1: Analyzed reference and draft genomes, sorted by total genome length. Note that the amount of q-grams and therefore the number of keys generated from each genome is roughly equivalent to the genome length.

amount of keys can be assigned to the same slot through quotienting. More precisely, for a hash table with $A = 2^{\mathfrak{a}}$ slots, there are $32 - \mathfrak{a}$ potential soft colliding keys per slot.

Across all parameter combinations, higher values of H are associated with both higher fill rates and a lower fraction of stashed keys. This is due to the fact that occurring (soft) collisions are resolved by storing colliding keys in the hopscotch neighborhood for each slot. However, in contrast to the neighborhood, which is restricted to a small constant value H, the number of possible colliding keys doubles with each additional remainder bit. Large values of H allow to store more keys in the neighborhood of each slot which in turn do not need to be stashed.

4.5.2 *Filling a BPHT*

To evaluate the time required for inserting items in relation to parameter choices for a BPHT, we used our hash table to count q-gram occurrences in four reference genomes. The evaluated genomes are shown in Table 4.1. We selected a hash table of size $\mathfrak{a} = 31$ for all genomes to remove the influence of RESIZE operations. This is large enough to accommodate the $\sim 1.5 \cdot 10^9$ distinct 16-grams present in hg38, the largest reference genome we evaluated. All experiments were performed using a single thread on a compute server with two AMD EPYC 7452 32-Core Processors, with 30 threads running in parallel.

For all genomes, we computed q-grams, hashed them and inserted them into the BPHT. We evaluated values $q = 11$, $q = 15$, and $q = 16$, with 16-grams being the maximal capacity of our current BPHT implementation. Since 16-grams require 32 bits when encoded using 2-bit encoding, higher values would result in more possible keys than the table can store. A way to mitigate this is described in the discussion.

We generated all 2-bit encodable q-grams for each genome, skipping all q-grams containing characters from $\Sigma_{\text{IUPAC}} \setminus \Sigma_{\text{DNA}}$ (i.e. characters that cannot be represented in 2-bit encoding). Subsequently, these q-grams were hashed with randomly chosen hash

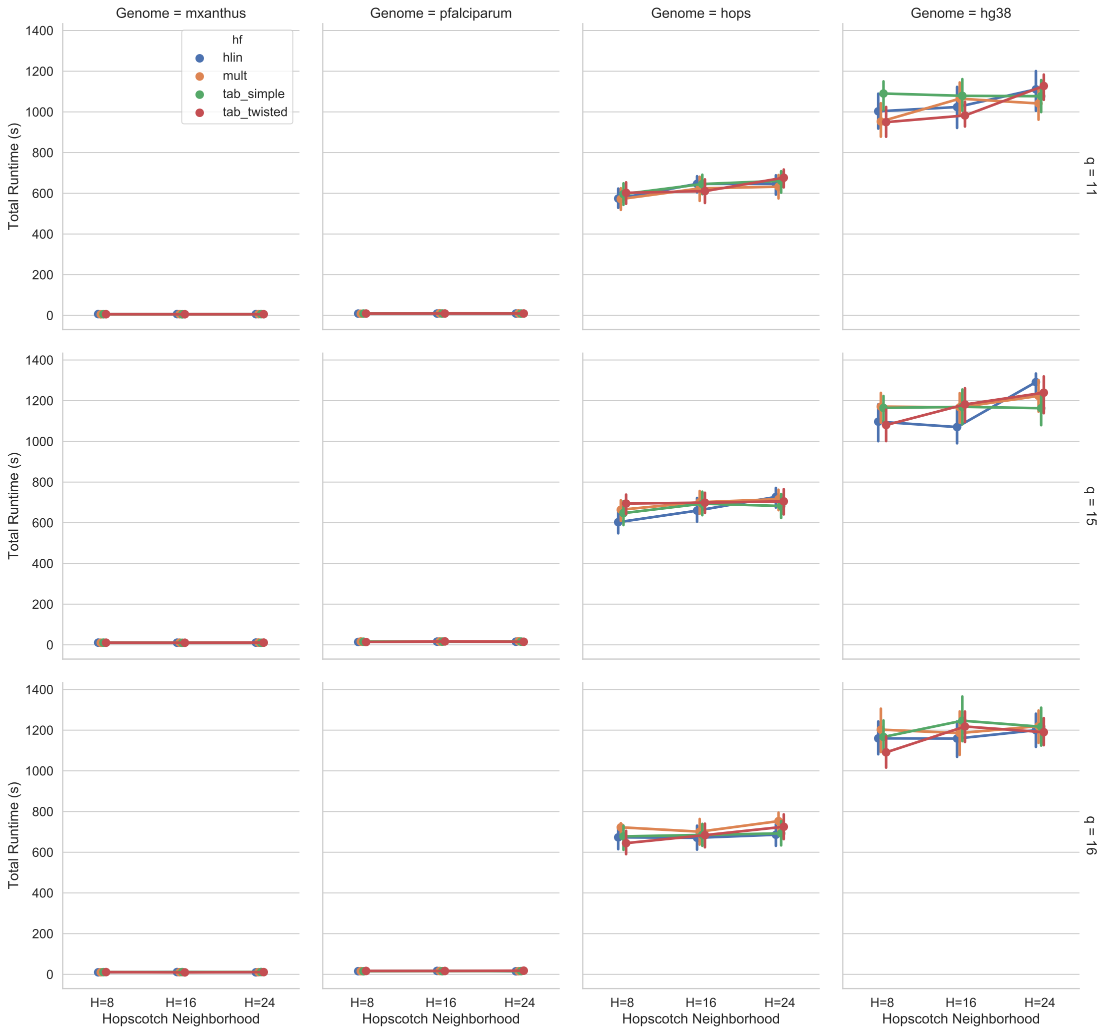


Figure 4.5: Point plot illustrating the insert time required to fill a BPHT with all q-grams (values of q increase by row) from the given genome (column). Each column within a subplot shows three different sizes of the hopscotch neighborhood H. Each point denotes the mean run time of 10 runs, errors denoted as vertical bars.

functions from the $\mathcal{H}^{\text{lin}}_{2^{32},2^{32},2^{64}}$, 32-bit simple and twisted tabulation hashing, as well as the invertible multiplicative hash function family $\mathcal{H}^{\text{im}}$ described earlier in this chapter (p. 76). In figure captions, we refer to these hash functions as *hlin*, *tab_simple*, *tab_twisted*, and *inv-mult* respectively. For a detailed description of these hash functions, see chapter Hash Functions for Use in Bioinformatics (p. 44f).

We used a BPHT in counting mode, where the value stored for each key is increased each time the key is inserted into the table using a Replacing Insert. Consequently, each q-gram is present in the table at most once, with a value indicating how often it was encountered in the sequence. Search operations yield the count value for the query key. The run times shown in this section are wall clock times for reading in the input file, generating and hashing q-grams, and inserting them into the BPHT. Initialization of the BPHT was not timed.

Figure 4.5 illustrates the total time required to count all q-gram occurrences in the respective genomes. For the smaller two genomes—*M. xanthus* and *P. falciparum*—with a respective mean runtime of 8.85s and 13.3s respectively, no stark differences in

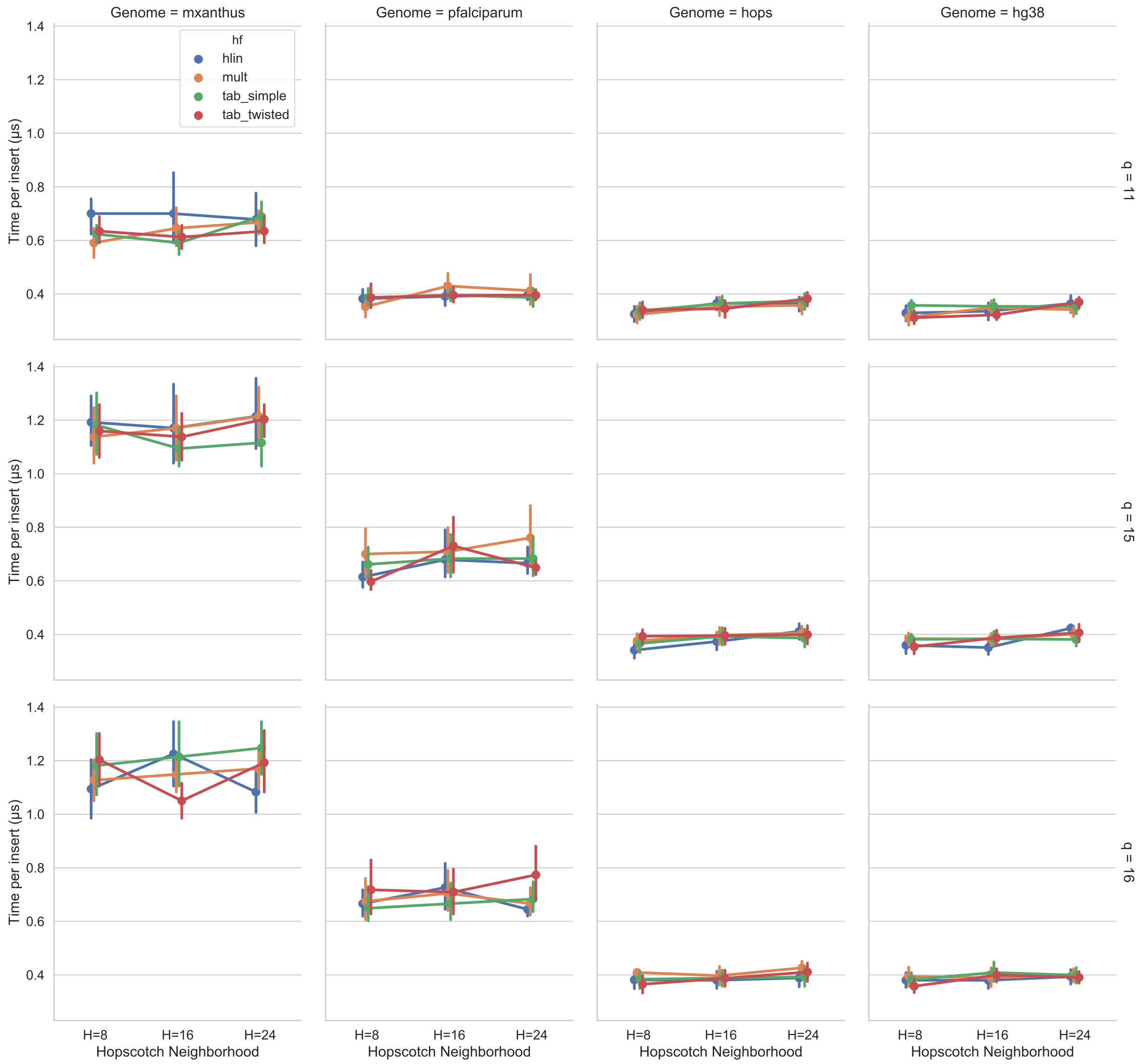


Figure 4.6: Point plot illustrating the insert time in µs per q-gram (values of q increase by row) from the given genome (column). Insert times for all items (as shown in the previous figure) here are normalized by the number of q-grams inserted.

runtime could be observed. The larger genomes—*H. lupulus* and hg38—show a mean runtime of 661.71 s (∼11min) and 1129.34 s (∼18min) respectively. Runtimes increase with increasing q-gram size, due to the fact that the existence of more q-grams increases the fill rate of the hash table. Therefore, more shift operations are required to free up slots during Insert operations.

For increasing hopscotch neighborhood size H an upward trend is noticeable, albeit with several exceptions. This can more clearly be seen in Figure 4.6, which shows the average time required per insert. In addition to showing a higher error rate, the two smaller genomes required insert times up to three times as long as larger genomes. Since for the larger genomes insert times stabilize at 0.3 µs for $q = 11$ and at 0.4 µs for $q = 15$ and $q = 16$, this can be due to external influences. With a total run time of less than 13 seconds for *M. xanthus* and less than 24 seconds for *P. falciparum* the influence of file operations, including opening the FASTA files, are more pronounced. This influence could be mitigated by pre-loading the files into memory and only measuring the insert process itself. However, our q-gram counting implementation follows an online approach to avoid the additional amount of memory required to keep whole chromosomes in memory.

For the different hash functions evaluated, no significant differences could be observed. Possible influences would have been the time required to compute each hash values, however, all evaluated hash functions require only the relatively cost effective operations multiplication, addition, bit-shifts, and bitwise XOR. Additionally, an increase in collisions due to less well distributed hash values could have been observed. Since no differences in runtime could be observed for different hash functions, other factors, like waiting for memory accesses, arguably dominated the runtime. The choice of more complex hash functions which trade stronger theoretical guarantees for an increase in runtime, could improve fill rates through better distributions of keys. We evaluated a hash function family for which we have not proven any guarantees (invertible multiplication), a 2-independent hash function family ($\mathcal{H}^{\text{lin}}$), and two 3-independent hash functions (simple and twisted tabulation). Considering that hopscotch hashing is related to linear probing, using a 5-independent hash function family could be beneficial for our approach, as suggested by Pătraşcu and Thorup for linear probing.[11]

4.5.3 *Speedup of Access Time Through Bit-packing*

To evaluate the speedup of our bit-packed approach over plain hopscotch hashing with two arrays we added a modified version of our BPHT implementation with a dedicated 32-bit hop-bit array (plain hopscotch hash table, PLHT). Apart from the hop bit computation, no parts of the code were modified. We performed two evaluations, one for hash tables in counting mode with only soft-colliding keys and one for normal mode (i.e. allowing keys to be added to the table multiple times) with hard collisions.

For the counting evaluation, we compared query times with keys sampled without replacement from $[2^{32}]$, using a number of keys identical to the size of the address space. As above, keys that could not be inserted were stashed and served as SEARCH operations with negative result during evaluation. For each set of keys, we filled a BPHT and a PLHT with the keys and verified an identical assignment by comparing both their fill rates and number of stashed keys. We then queried each hash table using the previously simulated key set, measuring the total wall clock time required to perform all queries.

To judge the influence of both the quotienting and bit-packing approaches, as well as the size of the hopscotch neighborhood, we evaluated different combinations of the hash table address space

[11] Pătraşcu and Thorup, "On the k-Independence Required by Linear Probing and Minwise Independence", 2010.

power $\mathfrak{a}$ and the neighborhood size H. All experiments were performed 10 times.

Figure 4.7 shows the query times required to retrieve the counts for all inserted keys using counting mode. We illustrated the range of $\mathfrak{a} \in \{28, \ldots, 31\}$ with their respective permissible values of H. Recall, that the address space power $\mathfrak{a}$ limits the number of hop bits that can be used, since the 32 administrative bits are required to be encoded into 32 bits in this implementation. For hash tables sizes of up $\mathfrak{a} = 31$, BPHTs require less time than the their PLHT counterparts.

As to be expected, query times rise with the size of the hash table and therefore with the number of queries performed. For larger hopscotch neighborhoods, query times also rise, as more array lookups have to be performed during each query. Across all runs, BPHTs perform the total number of queries faster than their PLHT counterparts.

For $\mathfrak{a} = 32$, a different behavior emerges, since for this case the remainder length is reduced to $\mathfrak{r} = 0$ and hence no soft collisions can occur in the input data. Consequently, using a key set as described above results in a permutation of the key space without any collisions. In combination with the hash tables in counting mode, this resulted in trivial problem instances. The total time to perform the queries dropped by approximately 75% with respect to the runtimes observed for $\mathfrak{a} = 31$, with PLHTs gaining a speedup over BPHTs. However, since this is not a representative evaluation of the hopscotch hashing approach, we also evaluated a dataset containing hard collisions and using hash tables in normal mode.

For the evaluation in normal mode, i.e. allowing multiple entries for each key in the table, we kept the same setup as with the previous evaluation. To create a dataset with similar properties to the ones used above, we sampled $2^{\mathfrak{a}}$ keys without replacement from a multiset containing all items from $[2^{32}]$ with a multiplicity of 2, i.e. $\{0, 0, 1, 1, \ldots\}$. This results in the same distribution of collisions as the case $\mathfrak{a} = 31$ for the counting evaluation, however, with hard collisions instead of soft collisions. For all values $\mathfrak{a} < 32$, soft collisions occur in addition to hard collisions.

Figure 4.8 shows the query times required to retrieve the values for all inserted keys using normal mode. In addition to the parameter combinations evaluated for counting mode, we also evaluated $\mathfrak{a} = 32$, which allows also evaluating the largest hopscotch neighborhood $H = 32$ supported by our current implementation.

As before, BPHTs outperformed PLHTs in all instances, barring single outliers for $\mathfrak{a} = 31, H = 24$ and $\mathfrak{a} = 32, H = 8$. The overall runtime of all queries increased with regards to counting mode, since queries cannot be terminated after the first successful access.

The mean speedup of BPHTs over PLHTs across all instances is illustrated in Figure 4.9. For all instances, BPHTs achieved a speedup of at least 1.082 and up to 1.405 for counting mode and from 1.077 up to 1.620 for normal mode. While for BPHTs in-

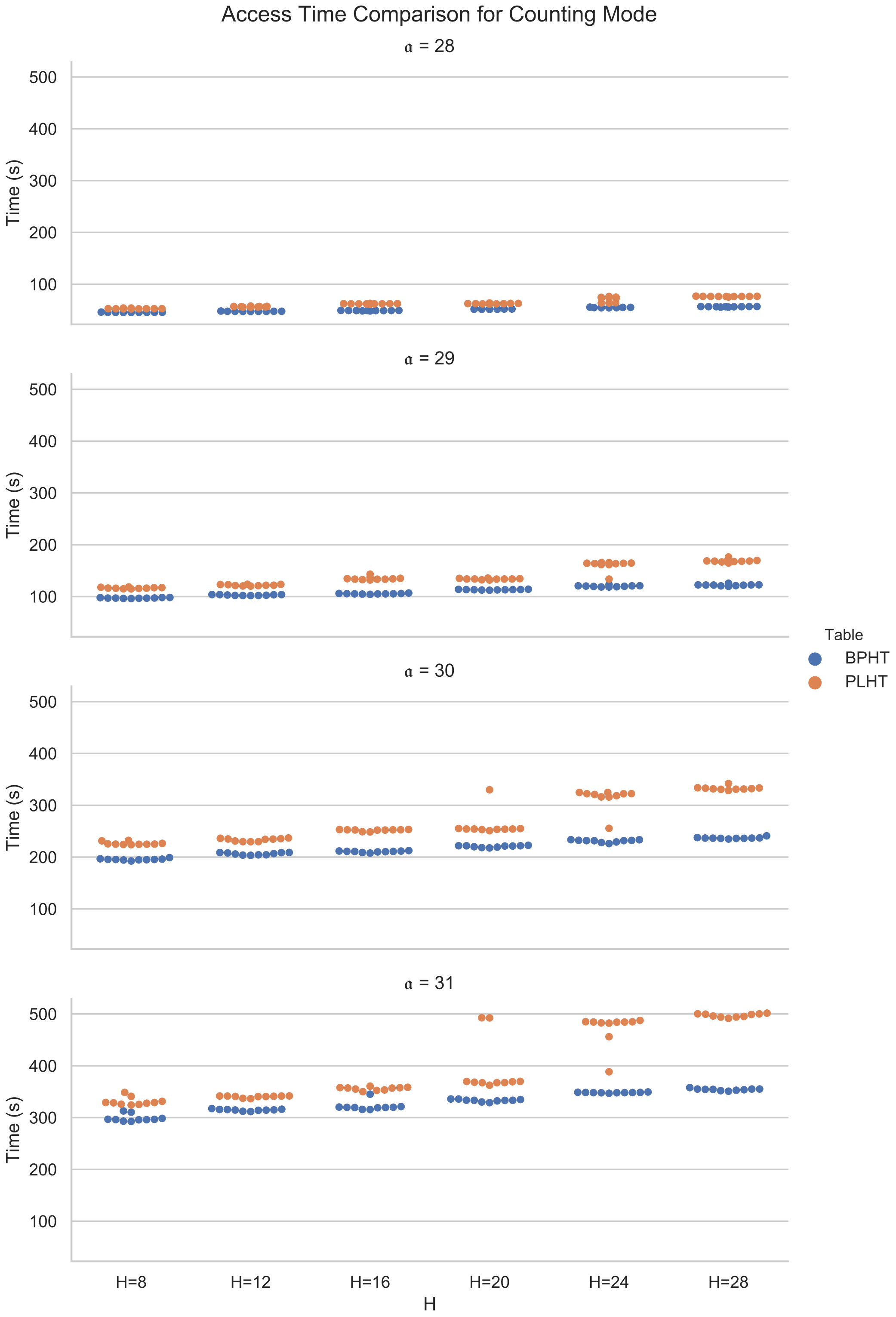


Figure 4.7: Swarm plot illustrating the total wall clock access time in seconds for BPHTs (blue) and PLHTs (orange) in **counting mode**. Different sizes of hopscotch neighborhoods H are shown on the x-axis of each facet, while hash table sizes given by α increase along the rows. This displayed query time includes SEARCH operations for all simulated keys inserted into the respective table.

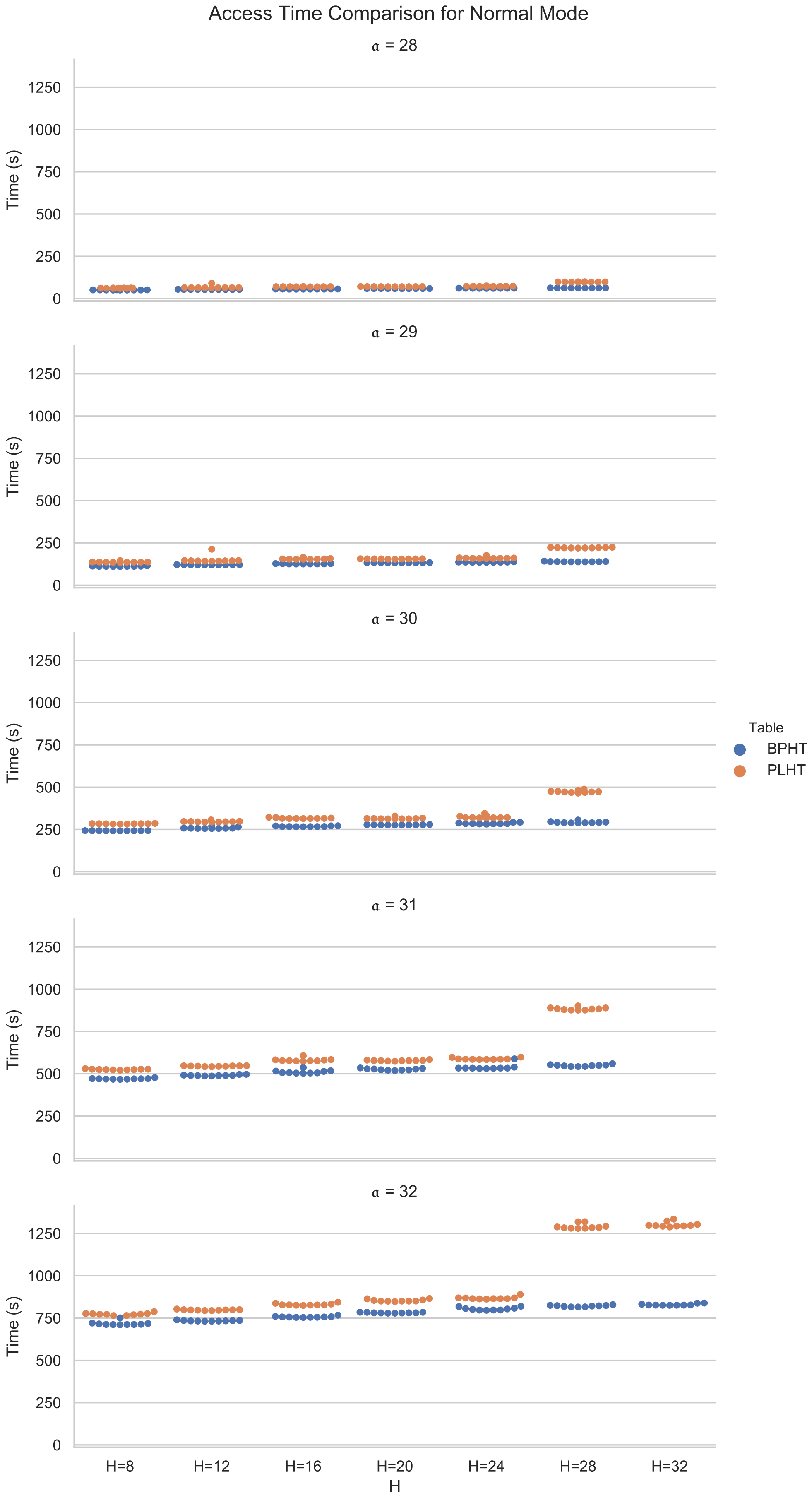
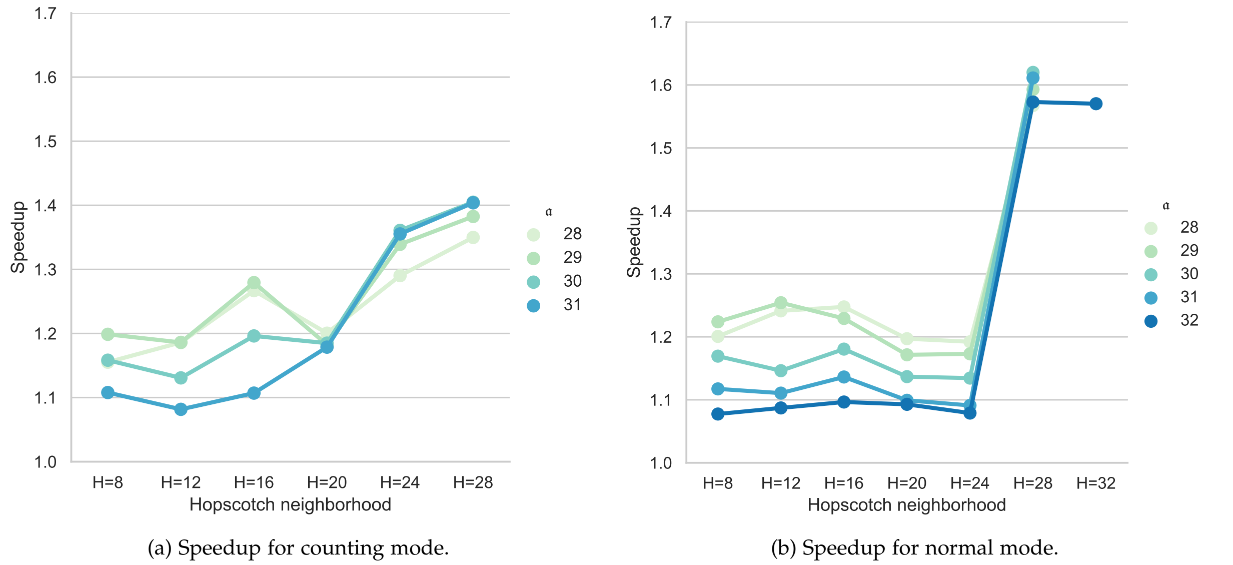


Figure 4.8: Swarm plot illustrating the total wall clock access time in seconds for BPHTs (blue) and PLHTs (orange) in **normal mode**. Different sizes of hopscotch neighborhoods H are shown on the x-axis of each facet, while hash table sizes given by $\mathfrak{a}$ increase along the rows. This displayed query time includes Search operations for all simulated keys inserted into the respective table.



(a) Speedup for counting mode.

(b) Speedup for normal mode.

Figure 4.9: Mean speedup of BPHTs over PLHTs in counting mode (a) and normal mode (b) for different hash table sizes (color) and hopscotch neighborhood sizes (x-axis). Hash tables of size $\mathfrak{a} = 32$ were not evaluated for the counting case, since these degenerated to trivial instances. Measurements for the same hash table size are connected by lines to guide the eye. The speedup was computed as $\frac{\text{Mean PLHT runtime}}{\text{Mean BPHT runtime}}$.

creasing hopscotch neighborhood sizes result in a linear runtime increase for both counting and normal mode, PLHTs show a significant increase in query time depending on the value of H. For counting mode (Figure 4.7), this first occurred at $H = 20$, while single instances using $H = 24$ still obtained a faster runtime. Normal mode instances (Figure 4.8) show a similar behavior for $H \geq 28$. The fact that these runtimes all increase by a similar fraction could indicate that these instances surpassed a break point of the hardware architecture. Potential causes for this effect could be cache line alignment, additional cache evictions resulting in additional cache misses, or different memory allocation behavior used for larger data structure. Further research and evaluations, focusing on the caching and memory allocation behavior of these instances, are required to answer this question.

4.6 *Conclusion and Discussion*

We have constructed and implemented a hash table using a bit-packed version of hopscotch hashing that halves the number of compulsory cache misses. Through bit-packing hop bits into the data array alongside the values (and remainders), we omit the access into the hop bit array which caused a second cache miss that is required by plain hopscotch hashing. Additionally, we could show that RESIZE operations of our hash table can be performed with $H \cdot 64$ bits of additional memory. In comparison with other implementations of hopscotch hashing, our approach offers the additional flexibility to use less hop-bits if required. With slight modifications, this flexibility can be extended to prevent unused bits, resulting in better memory usage.

We could show, that BPHTs can reach high fill rates, close to 100%, as long as a small fraction of keys can be stashed into a secondary data structure. The fraction of stashed keys is reduced with larger hopscotch neighborhoods. In practice, the achievable fill rate also depends on the used hash function.

The runtime required to fill a BPHT with q-grams from a selection of different genomes has shown that insert times per q-gram stabilize on a value dependent on the size the q-grams. An upward trend in runtime for larger hopscotch neighborhoods could be observed, since more array operations are required for these instances (especially more shifts of empty slots). The different evaluated hash functions all performed equally well for this case and did not show any stark deviations in runtime.

For query times, our BPHT implementation outperformed a comparable implementation of plain hopscotch hashing using a dedicated hop bit array on all instances (but for one outlier) in both normal and counting mode. Especially for larger hopscotch neighborhood sizes $H \geq 24$ we observed a significant speedup of more than 1.29. While the overall speedup of our approach was expected due to the reduced number of array accesses and the resulting reduction in compulsory cache misses a precise explanation of the runtime increase for PLHTs with large hopscotch neighborhoods remains open.

In our current implementation we limited the hash function codomain to $[2^{32}]$ and the size of entries to 64 bits to simplify the description. A downside of this approach, apart from the limited size, is that for larger address spaces, a part of the administrative bits remains unused. Both of these limitations can be overcome at the cost of additional computation by bit-packing entries of size $\neq$ 64 into the array through word packing. For the word packing technique, data types with sizes that are no multiple of a machine word are stored in a contiguous array. Consider, for example entries comprising 64 bits of payload, 20 remainder bits, and 8 hop bits, resulting in a total size of 92 bits per entry. While there is no native data type that can hold 92-bit values, they can be encoded into one 64-bit integer and the $92 - 64 = 28$ high bits of a subsequent slot in the array. To access a value, we need to compute the beginning slot of the value and read the following 92 bits from the following 64-bit slots of the array. On the other hand, entries with less than 64 bits can also be packed into 64-bit values using this technique to reduce the memory footprint of the hash table. This approach would lift the restriction on the hash function codomain, the value size, and additionally can prevent unused administrative bits. However, all operations have to perform an additional address computation and potentially access multiple slots per access, since the slots of the data array and the hash table slots no longer align.

In the implementation described above, a large part of each hash table entry is claimed by the remainder of the quotienting function. The quotienting approach is required to resize the hash table or to reclaim keys from hash values. In use cases where this is not required, e.g. for use as a q-gram index for a large protein or DNA reference database, the remainder bits can be used as value bits to store larger payloads. This would allow values of size $64 - H$. Using a hash function of the $\mathcal{H}^{\text{lin}}$ hash function family[12] would for example be well suited and combines 2-independence with the ability to restrict hash values to a power-of-two address space.

Lifting the restriction that hash tables have a size that is a power of two would allow to achieve higher fill rates for given key sets. Using the current architecture and inserting keys from an input set, if we had to resize shortly before the end of the input, we receive a table with a low fill rate. However, for hash table sizes that are no power of two, we cannot split keys into address and remainder directly. Instead we have to rely on modulo and division operations, which are considerably slower to compute. This can be achieved with an arbitrary size quotienting function

$$\begin{aligned}\delta_A(y) : [2^{32}] &\to \left([A], \left\lceil \frac{2^{32}}{A} \right\rceil\right)\\ \delta_A(y) &:= \left(y \bmod A, \left\lfloor \frac{y}{A} \right\rfloor\right)\end{aligned}$$

where $y \in \mathcal{C}$, $\mathcal{C} := [2^{32}]$ is a hash value and A is the size of the hash table's address space. The respective inverse function is defined as

$$\begin{aligned}\delta_A^{-1}(a, r) : \left([A], \left\lceil \frac{2^{32}}{A} \right\rceil\right) &\to [2^{32}]\\ \delta_A^{-1}(a, r) &:= r \cdot A + a\end{aligned}$$

where $(a, r) \in \mathcal{A} \times \mathcal{R}$ is an address remainder pair. To store remainders for this approach, we require

$$\mathfrak{r} = \left\lceil \log_2 \left(\frac{2^{32}}{A}\right) \right\rceil$$

bits in the remainder.

Through this modification, the positions at which entries are placed after the resize cannot be predicted as easily. Consequently, we could no longer resize the hash table in-place with our approach detailed above.

[12] Cf. The $\mathcal{H}^{\text{lin}}$ family, p. 45f

5
Computing and Approximating Resemblance and Containment

Consider two DNA sequences, obtained from different sources. When comparing them, there are several question that might be of interest:

- Are they identical?
- Are they similar?
- Does one occur within the other?

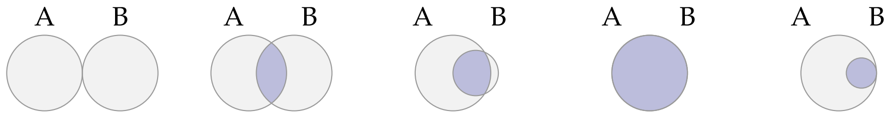
These questions correlate with biological questions. For example locating a DNA read in a reference genome or comparing the genomes of two bacterial strains. On a more abstract level these question can be answered through the use of sequence similarity measures. But how can we quantify similarity?

5.1 Resemblance and Containment

Following the nomenclature proposed by Broder,[1] we are interested in the resemblance $r(A, B)$ and the containment $c(A, B)$ of two documents A and B. In this abstract description, a document denotes anything that can be reasonably compared using its parts: For example, a DNA sequence can be described using its q-grams, or a mathematical set by its items. Resemblance quantifies the notion of *how similar two sets are*, whereas containment of A and B quantifies if *A occurs in B*.

Both resemblance and containment are values within $[0, 1]$. A resemblance of $r(A, B) = 0$ denotes two completely dissimilar sets, while $r(A, B) = 1$ denotes identical sets. Resemblance grows proportionally with the number with the number of items shared between the sets. Note that resemblance is symmetric: $r(A, B) = r(B, A)$. For the containment case, $c(A, B) = 1$ describes that A is completely contained within B. Unless $A = B$, $c(A, B) = c(B, A)$ does not hold; Containment is asymmetric. Some example of resemblance and containment values for two documents are given in Figure 5.1.

[1] Broder, "On the Resemblance and Containment of Documents", 1997.



$r(A,B)$	0	0.2	0.3	1.0	0.16
$c(A,B)$	0	0.2	0.2	1.0	0.16
$c(B,A)$	0	0.2	0.8	1.0	1.0

Figure 5.1: Consider two sets A and B in various cases of overlap. The values in columns two and three are rough approximations for illustration.

In our biological use cases, the question of resemblance arises when comparing genomes of different species. How much genetic material do these two strains of bacteria have in common? Containment, on the other hand, is relevant to judge the quality of assignments. How well does my mapped read fit to its assigned protein in the database? While the notions of resemblance and containment seem rather straight forward, there are many ways to implement them, which interact in different ways with the expected documents.

5.2 *Similarities and Distances*

We can quantify the resemblance of two documents using distances and similarity measures.[2] Abstractly, a similarity function $s(A, B)$ is a function that returns a value between 0 (not similar at all) and 1 (identical), which is what we need to describe resemblance. However, depending on the kind of documents, the given application, and many other factors, there are many possible similarity functions. For biological sequences from SGS, Hamming similarity, edit similarity, and Jaccard similarity best model the expected effects.

Complementary to a similarity measure $s(A, B)$, we define a normalized distance function $d(A, B) := 1 - s(A, B)$. Hence, identical documents $A = B$ have a distance of $d(A, B) = 0$ and the higher the distance, the further A and B are apart. Similarity measures are often derived from their complementary distance function. However, distance functions are not required to fall between 0 and 1 and can depend on a cost function assigning weights to different kinds of deviations.

Formally, a distance function or metric is defined as a function $d : \mathcal{K} \times \mathcal{K} \to [0, \infty)$ for an item space $\mathcal{K}$ with $A, B \in \mathcal{K}$, that satisfies the following four conditions:

NON-NEGATIVITY $d(A, B) \geq 0$ Distances are at least 0.

DEFINITENESS $d(A, B) = 0 \Leftrightarrow A = B$ Documents with distance 0 are identical.

SYMMETRY $d(A, B) = d(B, A)$ The order of documents is irrelevant for the distance function.

[2] Deza and Deza, *Encyclopedia of Distances*, 2009.

TRIANGLE INEQUALITY $d(A,C) \leq d(A,B) + d(B,C)$ There cannot be a shorter way between A and B than the direct one.

The following sections describe three similarity and distance measures used for biological sequences: Hamming distance, Edit distance, and Jaccard similarity.

5.2.1 *Hamming Distance and Similarity*

A common and very intuitive distance measure for documents of *identical* length is the Hamming distance.[3] We count the number of positions at which both documents differ.

The Hamming distance of two sequences $A, B \in \Sigma^n$ is defined as

$$d_H(A,B) := \sum_{i=0}^{n-1} \delta(A_{[i]}, B_{[i]}) \qquad \delta(x,y) = \begin{cases} 0, & x = y \\ 1, & \text{else} \end{cases} \tag{5.1}$$

where $\delta(a,b)$ is a cost function penalizing mismatches between the documents. In this case we assume unit costs, i.e. each error is penalized with a cost of 1. Since there is only one kind of possible error, all other (positive) cost functions are just a linear scaling of this case.

Further, the normalized Hamming distance

$$d_{HN}(A,B) := \frac{1}{n} \sum_{i=0}^{n-1} \delta(A_{[i]}, B_{[i]}) \qquad \delta(x,y) = \begin{cases} 0, & x = y \\ 1, & \text{else} \end{cases} \tag{5.2}$$

is the Hamming distance, normalized to the range $[0,1]$. In other words, the fraction of non-matching to total items in the sequences. Based on the normalized Hamming distance, we define the Hamming similarity:

$$s_H(A,B) := 1 - d_{HN}(A,B), \qquad A, B \in \Sigma^n \tag{5.3}$$

An example for the Hamming distance (plain and normalized) and their Hamming similarity is given in Figure 5.2.

$A =$ `GATTACAT`

$B =$ `AGTTACAT`

$$d_H(A,B) = 1 + 1 + 6 \cdot 0 = 2$$

$$d_{HN}(A,B) = \frac{2}{2+6} = 0.25$$

$$s_H(A,B) = 0.75$$

Figure 5.2: Hamming distance and similarity of two DNA strings that differ in the first two characters.

While the restriction to documents of the same length seems limiting, it allows the hamming distance to be computed fast, namely in $\mathcal{O}(n)$ comparisons.[4] Additionally, there are many use cases in which documents of identical length are guaranteed. For example the comparison of bit words, which are bound to a fixed size (most of the time 32- or 64-bit) by the hardware architecture. Most importantly for us, however, the Locality Sensitive Hashing techniques described in this chapter rely on computing the Hamming similarity between reduced representations of documents.

[3] Hamming, "Error Detecting and Error Correcting Codes", 1950.

[4] Note that (weakly) approximating the hamming distance is even possible in sub-linear time (Batu et al., "A Sublinear Algorithm for Weakly Approximating Edit Distance", 2003).

Note that computing the Hamming distance between documents $A \in \Sigma^n, B \in \Sigma^m, n > m$ is also possible by padding the smaller document. We extend B using the padding character `-`, with $\delta(x, \texttt{-}) = 1,\ \forall x \in \Sigma$, until A and B have equal length. The padding itself introduces $m - n$ mismatches, but, more importantly, the position of the padding can greatly influence the distance, as illustrated in Figure 5.3.

$$\begin{aligned} A &= \texttt{GATTACAT} & & \\ B &= \texttt{-ATTACAT} & s_H(A,B) &= 0.875 \\ B' &= \texttt{ATTACAT-} & s_H(A,B') &= 0.125 \end{aligned}$$

Figure 5.3: Hamming distance with padding for two strings of unequal lengths. The string B has a significantly higher similarity to A than B'.

5.2.2 Edit Distance and Similarity

Another distance that is widely used in bioinformatics (and many other contexts) is the edit distance (also named Levenshtein distance[5]). Given two sequences $A \in \Sigma^n, B \in \Sigma^m$, it is defined as the minimal number of operations needed to transform one text into the other using the following three operations:

REPLACE one item with another item.

INSERT a new item at any position in one of the documents.

DELETE an item in one of the documents.

Noticing the similarities of these operations with the different types of mutations described in the Section Genomic Mutations, making the usefulness of edit distances on DNA strings apparent. Since the edit operations listed above behave analogous to SNVs (replacements), insertions, and deletions, respectively, the edit distance can be used to describe mutation processes.

The edit distance can be computed recursively as

$$d_{\mathrm{E}}(A,B) = E_{A,B}(n,m) = E_{A,B}(|A|,|B|) \tag{5.4}$$

where the recursive function E is defined as follows:[6]

$$E_{A,B}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0 \\ \min \begin{cases} E_{A,B}(i-1,j)+1 \\ E_{A,B}(i,j-1)+1 \\ E_{A,B}(i-1,j-1)+\delta(A_{[i-1]},B_{[j-1]}) \end{cases} & \text{otherwise.} \end{cases} \tag{5.5}$$

and, as above, using a function

$$\delta(x,y) = \begin{cases} 0, & x = y \\ 1, & \text{else} \end{cases}$$

To normalize the edit distance of two documents, we need to take their lengths into account:

$$d_{\mathrm{EN}}(A,B) = \frac{d_{\mathrm{E}}(A,B)}{\max(|A|,|B|)} \tag{5.6}$$

[5] Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals", 1966.

[6] Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals", 1966.

Using the normalized edit distance we can describe edit similarity as:

$$s_{\mathrm{E}}(A,B) = d_{\mathrm{EN}}(A,B) = 1 - \frac{d_{\mathrm{E}}(A,B)}{\max(|A|,|B|)} \tag{5.7}$$

While the recursive definition can be elegantly written down, its implementation is inefficient, since many duplicate computations are performed. Hence, in practice dynamic programming (DP) approaches that compute the complete $|A| \times |B|$ matrix are employed, most notably, the Wagner-Fisher Algorithm.[7] However, several variations of DP algorithms to compute the edit distance have been developed.

Recall that DP relies on maintaining a data structure for intermediate results of recursive calls. If an intermediate result is required, we check if it has already been computed and stored, otherwise we compute it and store it.

For biological use cases, it is often helpful to weight the different operations. This is the realm of alignment algorithms, which compute an assignment that is maximal in respect to a given scoring function. An example for this is BLOSUM[8] substitution matrices for amino acids, which assign a weight to each pair $(a_1, a_2) \in \Sigma_{AA}^2$, depending on both their chemical properties (cf. Figure 2.13) and the biological question at hand. The aforementioned Wagner-Fisher Algorithm computes the edit distance by minimizing unit costs, i.e. all operations other than a match are penalized with a cost of 1.

A similar algorithm which can incorporate a different scoring system (like BLOSUM matrices), including negative weights and adaptive gap costs[9] has been proposed by Needleman and Wunsch.[10] This algorithm computes a global alignment using dynamic programming and an $|A| \times |B|$ matrix. In a global alignment, all positions of A and B are aligned. Each entry in the matrix is computed depending on its neighbors and the cost-optimal way to reach this entry. After all entries have been filled, the operations for the optimal alignment can be retrieved by backtracking through the matrix.

The Smith-Waterman algorithm[11] computes a local alignment, again using dynamic programming and an $|A| \times |B|$ matrix. A local alignment is not required to incorporate all positions, but comprises the highest scoring aligned subsequences. Similar to the Needleman-Wunsch algorithm, Smith-Waterman can use scoring systems and adaptive gap costs. The highest scoring sequence is also determined using a traceback starting from the highest scoring cell in the DP matrix.

Given special restrictions, other specialized alignment algorithms have been developed. Banded alignments restrict the parts of the DP matrix that are computed, reducing the runtime to $\mathcal{O}(w \cdot \min(|A|, |B|))$, where w is the width of the band.[12] The size of w can be set to restrict alignments to a certain range of scores.

Another possible optimization for small patterns and unit costs is Myers bit-parallel alignment algorithm for semi-global alignments, which relies on bit-encoding the DP matrix in integer values.[13] For a semi-global alignment, gaps at the end of the smaller document are not penalized, i.e. it needs to incorporate all positions of one document, but not the other. If the pattern can be encoded

[7] Wagner and Fischer, "The String-to-String Correction Problem", 1974.

[8] Henikoff and Henikoff, "Amino Acid Substitution Matrices From Protein Blocks", 1992.

[9] For example affine gap costs, meaning the penalty for opening a gap can be high, but the cost to extend an existing gap can be low.

[10] Needleman and Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins", 1970.

[11] Smith and Waterman, "Identification of Common Molecular Subsequences", 1981.

[12] Chao, Pearson, and Miller, "Aligning Two Sequences Within a Specified Diagonal Band", 1992.

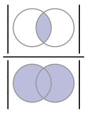
[13] Myers, "A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming", 1999.

in a processor word, a column of the DP matrix can be computed using bit-wise operations on integer values, resulting in a runtime of $\mathcal{O}(\frac{|A|\cdot|B|}{w})$, where w is the size of the processor word.

5.2.3 *Jaccard Similarity and Distance*

The similarity of two sets can be described with the Jaccard Index,[14] which is defined as the ratio of items shared by both sets divided by the total number of unique items in both sets. More formally, the Jaccard similarity (or Jaccard index) of two sets A and B is defined as:

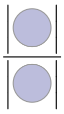
$$\mathcal{J}(A,B) = \frac{|A \cap B|}{|A \cup B|} =$$



$$(5.8)$$

Since identical sets have a Jaccard similarity of

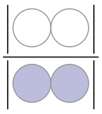
$$\mathcal{J}(A,A) = \frac{|A \cap A|}{|A \cup A|} = \frac{|A|}{|A|} =$$



$$= 1 \qquad (5.9)$$

disjoint sets have a similarity of

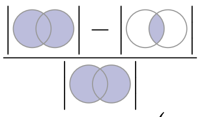
$$\mathcal{J}(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{0}{|A \cup B|} =$$



$$= 0 \qquad (5.10)$$

and all values in between are proportional to the ratio of shared items, Jaccard similarity realizes the resemblance of two documents. The complementary Jaccard distance is defined as:

$$d_{\mathcal{J}}(A,B) = 1 - \mathcal{J}(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} =$$



$$(5.11)$$

Computing the Jaccard similarity of two sets is straight forward: Count the shared elements of both documents, count the number of unique elements present in the union of both sets and divide these numbers. However, this can be time consuming to compute for large sets. More specifically, computing union and intersection of the sets requires inspecting each item in each set at least once, for each pair of sets. Consequently, the time required to compare two documents grows at least linearly with the size of the documents.

5.2.4 *Weighted Jaccard Similarity*

Limiting similarity computation to sets can be a stark restriction. Consider two sequences $A, B \in \Sigma_{\text{DNA}}^{15}$

$$\begin{aligned} A &= \texttt{AAAAAAAAAAAATTT} \\ B &= \texttt{AAATTTTTTTTTTTT} \end{aligned} \qquad (5.12)$$

which are very dissimilar on an intuitive level. However, computing the Jaccard similarity of their q-gram sets reveals the limitation of

[14] Jaccard, "Lois de distribution florale dans la zone alpine", 1902.

this measure:

$$\begin{aligned} Q(A,3) &= \{\,\texttt{AAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTT}\,\} \\ Q(B,3) &= \{\,\texttt{AAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTT}\,\} \\ \mathcal{J}(A,B) &= \frac{4}{4} = 1 \end{aligned} \tag{5.13}$$

This example illustrates that the choice of similarity measure greatly influences similarity values.[15]

Here, the multiplicity of items is lost by using item sets, which can be circumvented by using multisets. In a multiset, items can occur multiple times and we maintain a multiplicity function $\chi_A(x)$, which returns the number of times the item x occurs in A. Using this, we can define weighted Jaccard similarity[16], which incorporates multiplicity.[17] Given two multisets A and B, both containing items from a set X, the weighted Jaccard similarity is defined as:

$$\mathcal{J}^{\mathrm{W}}(A,B) = \frac{\sum_{x \in X} \min\left(\chi_A(x), \chi_B(x)\right)}{\sum_{x \in X} \max\left(\chi_A(x), \chi_B(x)\right)} \tag{5.14}$$

Moving back to the example, where A and B contain items from Σ^3_{DNA}, we get the following multiplicities:

$$\begin{array}{lllll} A: & \chi_A(\texttt{AAA}) = 9, & \chi_A(\texttt{AAT}) = 1, & \chi_A(\texttt{ATT}) = 1, & \chi_A(\texttt{TTT}) = 1 \\ B: & \chi_B(\texttt{AAA}) = 1, & \chi_B(\texttt{AAT}) = 1, & \chi_B(\texttt{ATT}) = 1, & \chi_B(\texttt{TTT}) = 9 \end{array}$$

The weighted Jaccard similarity of A and B is then computed as follows:

$$\begin{aligned} \mathcal{J}^{\mathrm{W}}(A,B) &= \frac{\sum_{x \in X} \min\left(\chi_A(x), \chi_B(x)\right)}{\sum_{x \in X} \max\left(\chi_A(x), \chi_B(x)\right)} \\ &= \frac{\min(9,1) + \min(1,1) + \min(1,1) + \min(1,9)}{\max(9,1) + \max(1,1) + \max(1,1) + \max(1,9)} \\ &= \frac{1+1+1+1}{9+1+1+9} = \frac{4}{20} = 0.2 \end{aligned}$$

in this example, we used the absolute abundance of q-grams for the multiplicity function χ. There are many other possible weighting functions, for example based on the frequency of items, their setup, or domain specific information.

5.3 *Containment*

Containment of documents can not as easily be described using similarity measures. As shown in the rightmost columns in Figure 5.1, documents with low resemblance can have high containment. Intuitively, we are looking for a subset of the larger set which has a high resemblance with the smaller one. For this section we will denote the smaller set as $A =$ • and the larger set as $B =$ ● . Additionally, in contrast to similarity measures, containment measures are asymmetric:

ASYMMETRY $c(A,B) \neq c(B,A),\ A \neq B$ The order of documents influences their containment value.

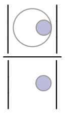
[15] This example is derived from one presented in the preprint for Marçais et al., "Locality-sensitive Hashing for the Edit Distance", 2019.

[16] Also known as Ruzicka similarity.

[17] Deza and Deza, *Encyclopedia of Distances*, 2009.

For some similarity measures, there are corresponding containment measures. For example, Jaccard containment describes the ratio of shared items to items in the contained set:

$$\mathcal{J}^{C}(A,B) = \frac{|A \cap B|}{|A|} = \tag{5.15}$$



For Hamming similarity, we can define a containment variant by not penalizing the leading and trailing run of mismatches. However, this also complicates the computation, since, as with end padding for not equally sized documents, the position of the smaller document within the larger one greatly influences containment scores.

Similarly, for edit similarity, a containment score can be derived from a semiglobal alignment. This alignment variant does not penalize gap opening and gap extensions at the ends of the smaller documents. Intuitively, this allows shifting the smaller document to the highest scoring position with the larger one. Using the Needleman-Wunsch Algorithm for two documents A and B and not penalizing end gaps for B yields a containment score for B in A.

Containment measures offer a way to deal with documents of different sizes in a way that similarity measures cannot. Jaccard similarity, for example, does not cope well with differently sized sets. Consider a typical read of length 100 bp and target chromosome of length 1 000 000 bp. If the read is contained perfectly in the chromosome the maximal possible similarity can be

$$\frac{100}{1\,000\,000} \approx 0.001$$

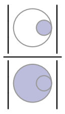
while the resemblance of the chromosome subsequence that the read aligns to is

$$\frac{100}{100} = 1.$$

Observation 5.3.1. *The Jaccard similarity of two sets A and B, with $|A| < |B|$ is at most $\frac{|A|}{|B|}$.*

Proof. The similarity $\mathcal{J}(A,B)$ is maximized if all items of A are contained in B. Consequently, the size of the intersection of A and B can be at most $|A|$, while the size of their union is $|B|$ (no new items are added to B through A).

$$\mathcal{J}(A,B) = \frac{|A \cap B|}{|A \cup B|} =$$



$$= \frac{|A|}{|B|}$$

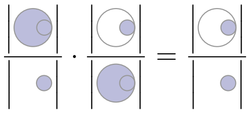
□

If the sizes of B and the intersection of A and B are known, we can express $\mathcal{J}$ using $\mathcal{J}^{C}$ and vice versa.

Lemma 5.3.2. *For the Jaccard similarity of two sets A and B, with $|A| < |B|$, there exists a scaling factor $\gamma = |A \cup B| / |A|$, so that $\mathcal{J}^{C}(A,B) = \gamma \cdot \mathcal{J}(A,B)$.*

Proof.

$$\gamma \cdot \mathcal{J}(A,B) = \frac{|A \cup B|}{|A|} \cdot \frac{|A \cap B|}{|A \cup B|} =$$



$$= \frac{|A \cap B|}{|A|}$$

which is the definition of Jaccard Containment (see Equation 5.15).

$$\mathcal{J}^C(A,B) = \frac{|A \cap B|}{|A|}.$$

□

Note that while the size of the union $|A \cup B|$ might also be unknown, it can be estimated, for example using HyperLogLog sketching.[18]

Unfortunately containment measures are harder to compute efficiently. In many applications, we would rather quickly estimate similarities or containment of documents and compute the exact values only for likely candidates. While there are many efficient ways to estimate similarity of documents, estimating containment is more complicated, as we will describe in the following section.

5.4 *Estimation of Similarity and Containment*

Referring back to the beginning of this chapter, we can see that all similarity measures described implement the resemblance of two documents. Given a specific (biological) task, we can select a similarity measure informed by the characteristics of the available data. However, the time to compute similarities grows quickly with the sizes of the compared documents. In practice, especially when working with large datasets like reference genomes or reads from high coverage sequencing experiments, explicitly computing similarity measures is very costly.

Computing containment measures suffers from the same problem. The containment of documents can be described using explicit containment measures, like Jaccard containment, and can be computed by alignment algorithms, using free end costs. Computing the containment of one document within another requires reading both documents in full at least once. This motivates using estimates of resemblance and containment measures, which can be computed more quickly, and restrict explicit computations to likely candidates.

Consider the case where a newly crafted reference genome for a microorganism is compared to a reference database. Instead of explicitly comparing our new sequence with all existing reference sequences, we can rule out all comparisons that are not even remotely similar. Using hash tables, we can maintain a set of reduced representations (sketches) of known reference sequences which are much smaller than the actual genomes. By querying these tables with the sketch of our new sequence, we can quickly estimate its similarity to existing sequences in the database. Starting from these

[18] Flajolet et al., "Hyperloglog: The Analysis of a Near-optimal Cardinality Estimation Algorithm", 2007.

similar sequences we can then compute alignments to quantify the actual sequence similarities.

Typical use cases for resemblance estimation include one-versus-all and all-versus-all distance computation of (reference) genome sets, for example to identify the strain of a sampled Ebola virus from TGS reads.[19] Containment estimation can be employed for read mapping and to query protein databases. The estimation of resemblance and containment measures can be performed using hash functions that assign similar keys to the same hash value with a probability depending on their similarity, so called locality sensitive hash functions. Their implementation and use will be described in the remainder of this chapter, starting with estimation of resemblance. Estimating containment remains a harder problem, for which general practical solutions have been proposed only recently. Finally, we will provide some examples of bioinformatics software which harness the Locality Sensitive Hashing paradigm.

5.5 *Locality Sensitive Hashes as Estimators for Resemblance*

Locality Sensitive Hashing (LSH), as introduced by Indyk, Motwani, and Gionis,[20] describes hash functions that hash similar keys to similar hash values. The polar opposite of cryptographic hash functions, LSH purposefully incites collisions to find similar items. Since then, different variants of LSH have been developed and applied to a variety of tasks ranging from DNA sequence analysis[21] over spam detection[22] to querying music databases.[23] Most notably, these include SimHash, MinHash, and Winnowing.

SimHash[24] estimates the cosine similarity[25] of documents to estimate their Jaccard similarity. Documents are projected onto a unit sphere, which is partitioned using random hyperplanes. A bin is defined as an area surrounded by a set of hyperplanes. Feature vectors (i.e. documents) falling into the same bin are considered similar.

MinHash[26] exploits the properties of min-wise independent hash functions, to uniformly sample items from the documents to estimate their Jaccard similarity. For each document a reduced set of items with minimal hash values, called *sketch*, is computed. The Jaccard similarity of the sketches is an estimator for the Jaccard similarity of the documents.

Winnowing[27] computes minimal hash values for a sliding window, so that the range of minimal hash values is restricted to a local neighborhood. Comparing the values and order of the selected items allows judging the similarity and containment of two documents.

Note that both SimHash and MinHash do not allow to estimate containment, nor do they incorporate (long range) position information within the documents. However, variants of the MinHashing approach can be applied to solve these problems.

[19] Ondov et al., "Mash: Fast Genome and Metagenome Distance Estimation using MinHash", 2016.

[20] Indyk and Motwani, "Approximate Nearest Neighbors: Towards Removing The Curse of Dimensionality", 1998; Gionis, Indyk, and Motwani, "Similarity Search in High Dimensions via Hashing", 1999.

[21] Buhler, "Efficient Large-Scale Sequence Comparison by Locality-Sensitive Hashing", 2001.

[22] Damiani et al., "An Open Digest-based Technique for Spam Detection", 2004.

[23] Ryynanen and Klapuri, "Query by Humming of Midi and Audio Using Locality Sensitive Hashing", 2008.

[24] Sadowski and Levin, *Simhash: Hash-Based Similarity Detection*, 2007.

[25] $\mathcal{CS}(A,B) = \frac{|A \cap B|}{\sqrt{|A||B|}}$

[26] Broder, "On the Resemblance and Containment of Documents", 1997.

[27] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003; Roberts et al., "Reducing Storage Requirements for Biological Sequence Comparison", 2004.

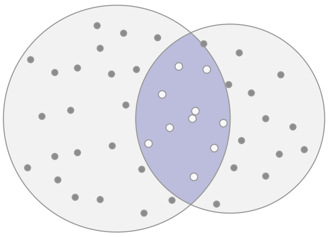


Figure 5.4: Visualization of a MinHash sketch (small dots) for two sets. The fraction of sketch entries that are shared by both sets (light purple) allows estimating the number of the shared items.

5.6 *MinHash*

The core concept of the MinHash strategy as described by Broder[28] relies on comparing randomly uniformly selected subsets of documents, called sketches[29]. As visualized in Figure 5.4, if we place a number of probes randomly within two sets, the fraction of probes within the intersection of the sets to those in their union is proportional to their Jaccard similarity.

It can be seen that this estimation is only accurate if the probes are distributed evenly throughout the set. Additionally, when the number of probe positions is increased, the estimated value approaches the real Jaccard similarity, as the size of the sketch approaches the size of the document.

More formally, consider $A \subseteq [n]$ and $B \subseteq [n]$ and a random permutation $\pi : [n] \to [n]$. The value $\min \pi(A)$ is the smallest value after the permutation has been applied to A. Note that if π is a perfectly random permutation, all items from A have the same chance to be $\min \pi(A)$. Using this, the probability that two documents have the same minimal value is equal to their Jaccard similarity:[30]

$$\mathbb{P}[\min \pi(A) = \min \pi(B)] = \frac{|A \cap B|}{|A \cup B|} = \mathcal{J}(A, B) \tag{5.16}$$

A minimal item can be present only in A or only in B, but the probability that both documents share a minimal item is equal to the fraction of items they share (cf. Figure 5.4). Hence, by comparing the minimal values of two documents under several permutations $\pi_1, \ldots, \pi_k$, we can estimate the Jaccard similarity of these documents.

The error of this process can be described using Chernoff bounds.[31] Define random variables $X_i \in \{0, 1\}$, which are 1 if $\min \pi_i(A) = \min \pi_i(B)$, i.e. if the documents share the same minimum. If the permutations π_i are independent, X_i can be interpreted as a sequence of independent Bernoulli trials with $p_i = \mathcal{J}(A, B)$ (via Equation 5.16), with $\mu = \mathbb{E}[X] = \sum_{i=1}^{k} p_i = k \cdot \mathcal{J}(A, B)$ so that $X = \sum_{i=1}^{k} X_i$ is binomially distributed.

[28] Broder, "On the Resemblance and Containment of Documents", 1997.

[29] Note that sketching strategies are also applied for several other applications other than MinHashing, including cardinality estimation of sets, estimating the number of unique items in data streams, and feature extraction for machine learning.

[30] Broder et al., "Min-Wise Independent Permutations", 1998.

[31] Motwani and Raghavan, *Randomized Algorithms*, 1995, Chapter 4.1.

The probability to over- and underestimate $\mathcal{J}(A,B)$ by a factor of δ for a given number of permutations k can then be described as

$$\begin{aligned} & \mathbb{P}[X > (1+\delta)\mu] && \leq \frac{e^{\delta}}{(1+\delta)^{1+\delta}} \\ \equiv \quad & \mathbb{P}[X > (1+\delta)k \cdot \mathcal{J}(A,B)] && \leq \frac{e^{\delta}}{(1+\delta)^{1+\delta}} \end{aligned} \tag{5.17}$$

and

$$\begin{aligned} & \mathbb{P}[X < (1-\delta)\mu] && \leq e^{-\frac{\mu\delta^2}{2}} \\ \equiv \quad & \mathbb{P}[X < (1-\delta)k \cdot \mathcal{J}(A,B)] && \leq e^{-\frac{k\cdot\mathcal{J}(A,B)\delta^2}{2}} \end{aligned}$$

respectively.

In literature, the error of this approach is often stated in a simplified way to be $\mathcal{O}(1/\sqrt{k})$ using the central limit theorem (CLT). The CLT provides information about the sum and mean of independent random variables leveraging the law of large numbers (LoLN)[32]. Given k identical independent Bernoulli trials, as described above, the expected mean of these trials is approximately normally distributed around the mean of a single Bernoulli trial. The accuracy of this approximation increases with the number of trials k. As shown above, the mean of a single Bernoulli trial X used to model a MinHash probe is $\mathbb{E}[X] = p = \mathcal{J}(A,B)$ and its standard deviation is $\sigma = \sqrt{p(1-p)}$. According to the CLT, the standard deviation $\sigma_{\overline{X}}$ of the mean of k such Bernoulli trials is

$$\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{k}} = \frac{\sqrt{p(1-p)}}{\sqrt{k}}. \tag{5.18}$$

Consequently, the mean of k Bernoulli trials behaves as if chosen from a normal distribution with mean p and a standard deviation of $\sigma/\sqrt{k}$. Intuitively, with increasing k the standard deviation of this distribution narrows. By observing this distribution, we can describe the expected deviation of the estimated similarities, for example ~95% of values sampled from a normal distribution fall into the range $\mu \pm 2\sigma$. Finally, since the similarity of A and B is constant and independent of the number of trials, the standard deviation behaves asymptotically like $\mathcal{O}(1/\sqrt{k})$. Additionally, a tighter bound

$$\mathcal{O}\left(\frac{1}{\sqrt{\mathcal{J}(A,B)k}}\right)$$

has been proven.[33]

Computing a series of random permutations π that is sufficient for this approach has been shown to be computationally infeasible. However, random permutations can be realized using (ϵ-approximately) min-wise independent hash functions, as described in the section Universality, Independence, and Min-Wise Independence. We denote the minimal hash value and its generating item as follows:

[32] For details, cf. Montgomery and Runger, *Applied Statistics and Probability for Engineers*, 2014

[33] Thorup, "Bottom-k and Priority Sampling, Set Similarity and Subset Sums with Minimal Independence", 2013.

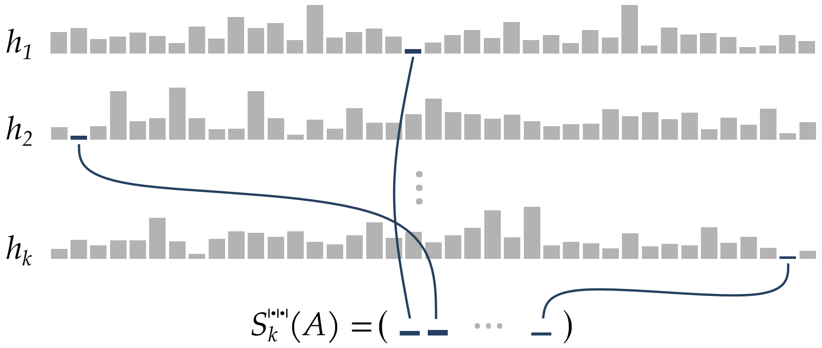


Figure 5.5: Illustration of a k-mins sketch. Each line of gray blocks denotes the sequence of hash values for one hash function, where the height of a block represents the "size" of the hash value. The MinHash value for each hash function is denoted by a blue block. Each entry in the sketch is the minimum of one of k different hash functions.

Definition 5.6.1 (MinHash values and minimizers). *Let h be an (ϵ-approximately) min-wise independent hash function, and $Q(A, q)$ the q-gram set for a document A. We define the* ***MinHash value*** *as the smallest hash value for all q-grams of A under hash function h:*

$$m(A) := \min \{ h(g_i) \mid g_i \in Q(A, q) \}$$

and the ***minimizer*** *as the q-gram that hashed to this this value:*

$$g^m(A) := \arg\min \{ h(g_i) \mid g_i \in Q(A, q) \}$$

As a shorthand notation we use m and g^m to refer to MinHash values and minimizers respectively.

A set or sequence of MinHash values can serve as a sketch of a document, which allow estimating their Jaccard similarity. With MinHashing, there are two kinds of sketches, k-mins sketches and bottom-k sketches, where k denotes the size of the sketch.

5.6.1 k-Mins Sketches

The technique described above—selecting items for a MinHash sketch using the minima of k different hash functions—is called k-mins sketching.[34]

Definition 5.6.2 (k-mins sketch). *Let $\mathcal{H}$ be a family of (ϵ-approximately) min-wise independent hash functions, and $Q(A, q)$ the q-gram set for a document A. For a sequence of hash functions $(h_i)_{i=1}^{k}$ $k \in \mathbb{N}_+$, we define the* ***k-mins sketch*** *of A as:*

$$S_k^{|\cdot|\cdot|}(A) := (m_i(A))_{i=1}^{k} \qquad m_i(A) := \min \{ h_i(g) \mid g \in Q(A, q) \}$$

Furthermore, if we describe properties that hold for all possible values of k, we use the short hand notation $S^{|\cdot|\cdot|}(A)$.

Figure 5.5 provides an example of how a k-mins sketch is assembled from k sequences of hash values.

Given the two sketches $S^{|\cdot|\cdot|}(A)$ and $S^{|\cdot|\cdot|}(B)$, the Jaccard similarity of the original documents can be estimated.[35]

[34] Broder et al., "Min-Wise Independent Permutations", 1998.

[35] Broder, "On the Resemblance and Containment of Documents", 1997.

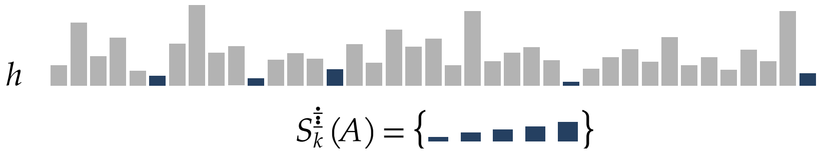


Figure 5.6: Illustration of a bottom-5 sketch. Each entry is one of the k smallest values under *one* given hash function h.

Theorem 5.6.3. *Given k-mins sketches for two documents A and B, the Jaccard similarity of their sketches*

$$\mathcal{J}^*(A,B) := \frac{1}{k}\sum_{i=1}^{k} \left[\!\left[S^{\text{|·|·|}}(A)_{[i]} = S^{\text{|·|·|}}(B)_{[i]} \right]\!\right] \approx \mathcal{J}(A,B)$$

is an unbiased estimator for the Jaccard similarity of A and B. Here we use $[\![\ ]\!]$ *as the Iverson bracket.*[36]

Proof. See (Broder, "On the Resemblance and Containment of Documents", 1997). □

Considering the runtime performance of MinHash strategies, two metrics are of interest for k-mins sketches: Computing a sketch and comparing two given sketches. The computation of a k-mins sketch for a document A can be performed in $\mathcal{O}(k|A|)$ time. By iterating through all q-grams of the document, computing their hash values using all k hash functions, and keeping track of the smallest hash value for each h_i. Comparison of two sketches is performed in k steps by comparing $m_i(A) = m_i(B)$ for all $i = 1, \ldots, k$ hash functions. Note that $\mathcal{J}^*(A,B) = d_{\mathrm{HN}}(S^{\text{|·|·|}}(A), S^{\text{|·|·|}}(B))$ (cf. Equation 5.2, p. 91).

Computing a k-mins sketch comes with the downside of evaluating k hash functions $|A|$ times. A similar approach for the generation of MinHash sketches that is feasible for many applications offers a solution with only one hash function.

5.6.2 *Bottom-k Sketch*

Where a k-mins sketch comprises the smallest hash value of k hash function, a bottom-k sketch contains the k smallest hash values using only one hash function.[37] Figure 5.6 provides an illustration for a bottom-5 sketch.

Definition 5.6.4 (Bottom-k sketch)**.** *Let h be a (ϵ-approximately) min-wise independent hash function, and $Q(A,q)$ the q-gram set for a document A. We define the* ***bottom-k sketch*** *of A as:*

$$S^{\vdots}_k(A) := min^k \{\, h(g) \mid g \in Q(A,q) \,\}$$

where $min^k X$ denotes the k smallest items in a set or sequence X (see p. 29).

Corollary 5.6.5. *A bottom-k sketch $S^{\vdots}_k(A)$ can be constructed by repeatedly sampling MinHash values without replacement. Let $Q' = Q(A,q)$ be the q-gram set of A and g^m_i $i = 1, \ldots, k$ be the minimizer selected for*

[36] The Iverson bracket $[\![P]\!]$ denotes an indicator function that is 1 if a condition P holds, and 0 otherwise.

[37] Cohen and Kaplan, "Summarizing Data Using Bottom-k Sketches", 2007; Broder, "On the Resemblance and Containment of Documents", 1997.

the i-th smallest entry of $S^{\ddagger}_k(A)$, and $|\,\{\, h(g^m_1), \dots, h(g^m_k) \,\}\,| = k$ (i.e. no collisions occur). The bottom-k sketch can be recursively defined as:

$$S^{\ddagger}_k(A) := \{\, m(Q'), m(Q' \setminus \{\, g^m_1 \,\}), \dots, m(Q' \setminus \{\, g^m_1, \dots, g^m_{k-1} \,\}) \,\}$$

Proof. Removing a minimizer from the set does not influence other hash values. Hence, removing a minimizer g^{m}_i from a set leaves a new minimizer g for the next iteration. □

Estimating the Jaccard similarity is only slightly more complicated than with k-mins sketches. Given the two sketches $S^{\ddagger}(A)$ and $S^{\ddagger}(B)$, the Jaccard similarity of the original documents can be estimated.[38]

Theorem 5.6.6. *Given bottom-k sketches for two documents A and B, the Jaccard similarity of their sketches*

$$\mathcal{J}^*(A, B) := \frac{|S^{\ddagger}(A \cup B) \cap S^{\ddagger}(A) \cap S^{\ddagger}(B)|}{|S^{\ddagger}(A) \cup S^{\ddagger}(B)|} \approx \mathcal{J}(A, B)$$

is an unbiased estimator for the Jaccard similarity of A and B.

Proof. See (Broder, "On the Resemblance and Containment of Documents", 1997). □

Note that the bottom-k sketch of the union $A \cup B$ can be computed from the sketches of A and B as

$$S^{\ddagger}(A \cup B) = S^{\ddagger}(S^{\ddagger}(A) \cup S^{\ddagger}(B)) \tag{5.19}$$

since the k smallest items of the union can only be selected from the smallest k items of each set.[39]

Computing a bottom-k sketch is less run time intensive than a k-mins sketch since it requires fewer hash value evaluations. By maintaining a maximum heap of size k, we can quickly ($\mathcal{O}(1)$) check if the item is eligible to be inserted into the sketch. If that is the case, we remove the old largest value in the sketch ($\mathcal{O}(1)$) and insert the new value ($\mathcal{O}(\log k)$). This results in a worst case run time of $\mathcal{O}(|A| \log k)$, however the expected runtime is more benevolent.

Since the hash function is assumed to be ϵ-approximately minwise independent, the k minimizers can be assumed to be uniformly distributed throughout A. Additionally, assuming an independent hash function (see Definition 3.1.4) we can also assume that all (other) hash values are distributed approximately randomly throughout the codomain $\mathcal{C}$ of the hash function. Consequently, the expected time is lower since most hashed q-grams can be skipped after comparing with the max-value of the heap. The further we progress through the document, the higher the chance that a randomly chosen hash value is smaller than the k smallest of all values we already saw up until that point. We discuss the distribution of minimizers, albeit in a slightly different context in chapter Distribution of Minimizer Segment Lengths.

[38] Cohen and Kaplan, "Summarizing Data Using Bottom-k Sketches", 2007.

[39] Cohen and Kaplan, "Summarizing Data Using Bottom-k Sketches", 2007.

Comparing two bottom-k sketches is possible in $\mathcal{O}(k)$ time by advancing linearly through both sketches and counting which items are present in one of the sketches or in both. To ensure that only the k smallest items of the union are evaluated (cf. Equation 5.19), we stop the iteration after k comparisons (matches or no matches) were made.

A beneficial property of bottom-k sketches is that their estimate $\mathcal{J}^*$ converges towards $\mathcal{J}$ as k approaches $|A|$.

Corollary 5.6.7. *Let $S^{\underline{\vdots}}_k(A)$ and $S^{\underline{\vdots}}_k(B)$ be the bottom-k sketches of two documents A and B. If $k = |A| = |B|$, then $\mathcal{J}(A,B) = \mathcal{J}^*(A,B)$.*

Proof. $S^{\underline{\vdots}}_k(A)$ contains the k smallest hash values of A, hence, if $k = |A|$, all items of A are contained in $S^{\underline{\vdots}}_k(A)$. Consequently:

$$\mathcal{J}^*(A,B) = \frac{|S^{\underline{\vdots}}_{|A|}(A) \cap S^{\underline{\vdots}}_{|B|}(B)|}{|S^{\underline{\vdots}}_{|A|}(A) \cup S^{\underline{\vdots}}_{|B|}(B)|} = \frac{|A \cap B|}{|A \cup B|} = \mathcal{J}(A,B)$$

□

This does not hold for k-mins sketches, since there is no guarantee that each item of A is selected at least once as minimizer by a collection of $k = |A| = |B|$ hash functions.

While the reasoning for the Chernoff bounds given in the previous section is slightly different, the error bounds remain the same. Each Poisson experiment is described by random variables $X_i \in \{0, 1\}$, which are 1 if a minimizer $x \in S^{\underline{\vdots}}(A)$ is also present in $S^{\underline{\vdots}}(B)$.

A third technique we will only briefly mention is k-partition sketching, which combines aspects of both bottom-k and k-mins sketches. This approach was initially used for cardinality estimation under the name stochastic averaging.[40] When assembling a k-partition sketch, each hash value is assigned to a set using on a subset of its bits, e.g. its k most significant bits. Within each of these 2^k sets, a bottom-1 sketch is computed using the remaining $n - k$ bits of the initial n-bit hash values. This emulates the behavior of a 2^k-mins sketch using only one hash function, albeit using a reduced range of possible hash values.

All three MinHash techniques provide an efficient way to estimate the Jaccard similarity and thereby the resemblance of documents. However, since all MinHash approaches inherit the properties of Jaccard similarity, different sizes of A and B remain a problem. Especially for read mapping of SGS short reads, this limitation becomes apparent, as illustrated in the section Containment. Estimating document containment, unfortunately, poses a new set of challenges. Another limitation of Jaccard-based approaches is that translations cannot be detected. This would require using another distance measure, like edit similarity.

[40] Flajolet and Martin, "Probabilistic Counting Algorithms for Data Base Applications", 1985.

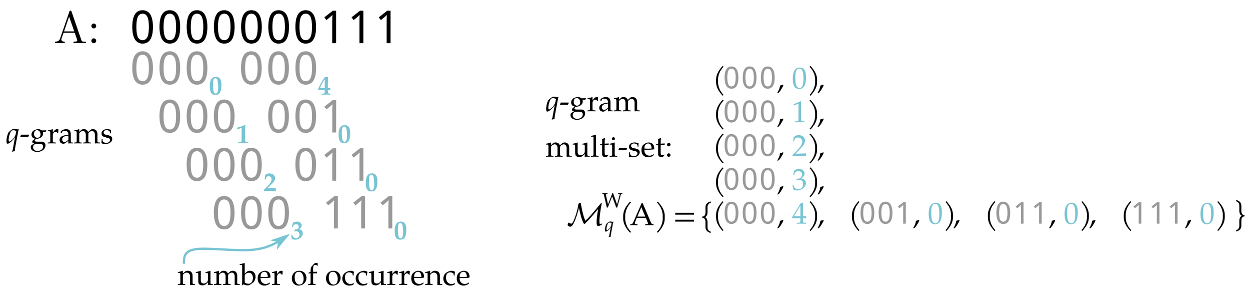

A: 0000000111
q-grams
000 0
000 4
000 1
001 0
000 2
011 0
000 3
111 0
number of occurrence
q-gram multi-set:
$\mathcal{M}_q^W(A) = \{(000,0), (000,1), (000,2), (000,3), (000,4), (001,0), (011,0), (111,0)\}$

Figure 5.7: Example of uniquified q-grams. The q-grams of a document A (gray monospace text) are annotated with their occurrence number (teal). This allows assembling them in a "multiset" containing 8 uniquified 3-grams, as shown on the right.

5.7 *Locality Sensitive Hashing for Edit Distance*

An LSH approach for the edit similarity based on MinHashing was recently presented by Marçais et al.[41] The Order Min Hash (OMH) approach combines bottom-k sketching with weighted Jaccard similarity. To tackle the specific weaknesses of Jaccard based similarity measures, OMH addresses two problems:

Multiplicity: How often does a q-gram occur in the document?

Order: Where does a q-gram occur in the document?

OMH sketches comprise k entries, each of which is a bottom-ℓ sketch of uniquified q-grams ordered by their position in the input sequence.

To add multiplicity information to a q-gram set, OMH augments each q-gram with an occurrence number in a process called uniquification. Each generated q-gram is reported with the number of times it has already occurred to assemble the set:

$$\mathcal{M}_q^W(A) = \{\, (g,o) \mid g : q\text{-gram},\ o : \text{occurrence number} \,\}$$

An example for this approach using a bit string is illustrated in Figure 5.7. Since q-grams can occur multiple times in M_q^W, it behaves similarly to a multiset.

Recall that the Jaccard similarity of multisets can be computed using weighted Jaccard similarity. Since uniquified q-grams encode multiplicity within the boundaries of sets, the weighted Jaccard similarity can be computed as

$$\mathcal{J}^W(A,B) = \mathcal{J}(M_q^W(A), M_q^W(B))$$

The relative order of q-grams is encoded by keeping the position of a minimizer within the input sequence to compile a text-ordered sketch. As mentioned above, each sketch entry of an OMH sketch comprises a bottom-ℓ sketch. This sketch is generated from the uniquified q-grams $\mathcal{M}_q^W(A)$ using a function $h_{\ell,\pi}^W(A)$, where π is a random permutation. The function $h_{\ell,\pi}^W(A)$ returns a text-ordered bottom sketch of A, comprising the ℓ items of $S_\ell^{\ddagger}(\mathcal{M}_q^W(A))$ in the order in which they appear in the input sequence.

An example of a bottom-ℓ sketch and its text-ordered counterpart is illustrated in Figure 5.8.

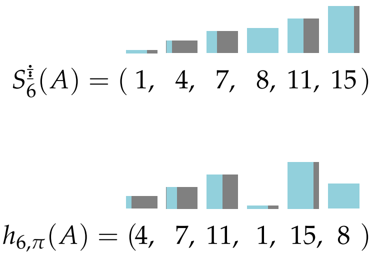


Figure 5.8: Comparison of a classic bottom sketch (above, represented as a sequence ordered by hash value size) and a text-ordered bottom sketch (below). Each bar respresents a sketch entry, where the height of a bar denotes the size of the hash value and the fraction shaded in teal denotes the position within the sequence.

[41] Marçais et al., "Locality-sensitive Hashing for the Edit Distance", 2019.

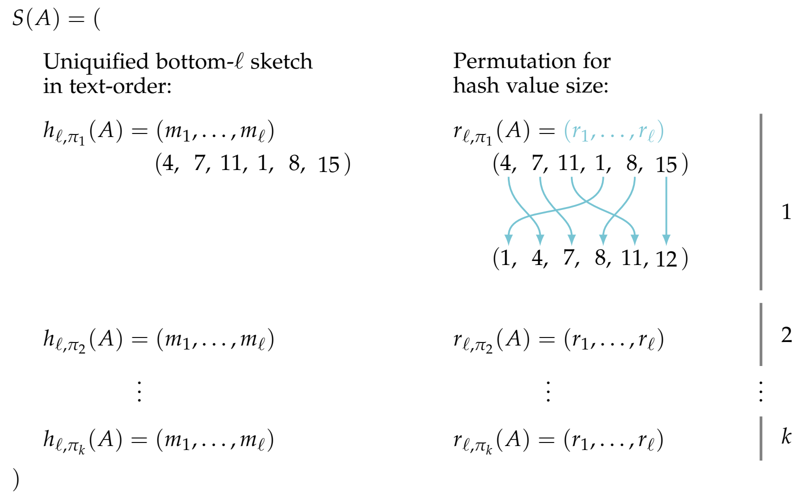

$S(A) = ($
Uniquified bottom-ℓ sketch in text-order:
Permutation for hash value size:
$h_{\ell,\pi_1}(A) = (m_1, \dots, m_\ell)$
$(4,\ 7,\ 11,\ 1,\ 8,\ 15)$
$r_{\ell,\pi_1}(A) = (r_1, \dots, r_\ell)$
$(4,\ 7,\ 11,\ 1,\ 8,\ 15)$
$(1,\ 4,\ 7,\ 8,\ 11,\ 12)$
1
$h_{\ell,\pi_2}(A) = (m_1, \dots, m_\ell)$
$r_{\ell,\pi_2}(A) = (r_1, \dots, r_\ell)$
2
$h_{\ell,\pi_k}(A) = (m_1, \dots, m_\ell)$
$r_{\ell,\pi_k}(A) = (r_1, \dots, r_\ell)$
k
$)$

Figure 5.9: Illustration of an OMH sketch with k entries. Each row contains the sketch for one random permutation π. In the first column of each row, the uniquified bottom-ℓ sketch values are shown. The second column contains the permutation required to sort the sketch entries by size (denoted by teal arrows).

Using these building blocks, a sketch for Edit similarity can be defined. Such an OMH sketch comprises k text-ordered bottom-ℓ sketches. For a practical implementation, the sketch $h^{\mathrm{W}}_{\ell,\pi}(A)$ (containing (q-gram, occurrence number)-tuples) is replaced by $h_{\ell,\pi}(A)$, which contains only q-grams in text order. Additionally, a function $r_{\ell,\pi}(A)$ encoding a permutation ordering the elements of $h_{\ell,\pi}(A)$ by size is used to synthesize $S^{\ddagger}_{\ell}(\mathcal{M}^{\mathrm{W}}_{q}(A))$. An example of an OMH sketch is shown in Figure 5.9.

Text-order sketches with $h_{1,\pi}$ allow constructing an LSH for weighted Jaccard similarity:[42]

$$\mathbb{P}[h_{1,\pi}(A) = h_{1,\pi}(B)] = \mathcal{J}^{\mathrm{W}}(A, B).$$

For all values $1 < \ell < n - q + 1$, where n is the length of the input, OMH is a gapped LSH for the edit distance.

A limitation of this approach is that it does not work with canonical q-grams, since the position information becomes inconsistent. Additionally, the size of the sketch and consequently the time required to compare sketches is higher than with other MinHashing approaches.

5.8 *Locality Sensitive Hashes as Estimators for Containment*

The main problem with estimating containment is that the MinHash sketches described above, by design, always represent their whole respective set. In the case of radically different sizes, say A is much smaller than B, as illustrated in Figure 5.10, items that are no minimizer for B can easily become minimizers for A.

While Broder already proposed a way to estimate Jaccard containment using MinHash with modulo sketches, the required sketch size is large. Other approaches to estimate containment via MinHash have only been proposed recently by Koslicki et al.[43] and Ondov et al.[44]. This problem can also be tackled by decomposing

[42] Marçais et al., "Locality-sensitive Hashing for the Edit Distance", 2019, Theorem 1.

[43] Koslicki and Zabeti, "Improving MinHash via the Containment Index with Applications to Metagenomic Analysis", 2019.

[44] Ondov et al., "Mash Screen: High-throughput Sequence Containment Estimation for Genome Discovery", 2019.

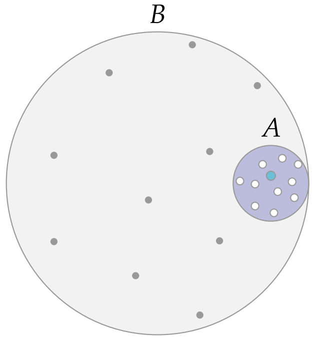


Figure 5.10: Two sketches with $k = 11$ entries for the sets A (light gray and teal dots) and B (dark gray and teal dots), where $A \subset B$. The 11 sketch members of A hit only one of the sketch members of B (shown as a larger, teal circle). MinHash values within $S(A)$ are larger than the largest MinHash value in $S(B)$, otherwise they would also have been selected for $S(B)$.

the larger of two sets into subsets, which brings forth the question of how to find a good decomposition. We will describe static window decompositions in this section while dynamically sized decompositions are subject of the following section.

5.8.1 *Modulo Sketches*

The first approach for containment estimation with MinHash was proposed by Broder in the original paper.[45] He proposed to compute modulo sketches S_{2^i}, containing all items with hash value $h(g) \equiv 0 \mod 2^i$ instead of minimal hash values.

The modulo size 2^i, used to regulate the number of items in the modulo sketch, needs to be chosen with respect to the document size. Given two sketches with modulo values 2^i and 2^{i+1} respectively, a sketch for 2^{i+1} can be computed from the one for 2^i by dropping all elements that are no multiple of 2^{i+1}. When computing containment for two documents A and B with $A < B$, the modulo value used for the larger document is computed as above. The resulting sketches are compared to compute the estimated Jaccard containment

Example: starting from the sketch for $S_{64} = \{128, 320, 512, 576\}$ the sketch $S_{128} = \{128, 512\}$ is computed by removing $320 \equiv 64 \mod 128$ and $567 \equiv 64 \mod 128$.

$$\mathcal{J}^{C*}(A, B) = \frac{|S_i(A) \cap S_i(B)|}{|S_i(A)|}$$

A downside of this approach is that the size of modulo sketches grows linearly with respect to document size, which sacrifices the memory efficiency of bottom-k and k-mins sketches. Additionally, it is not robust against size differences between documents,[46] which is one of the main reasons to choose containment estimation above resemblance estimation. As a result, modulo sketches have not seen widespread application.[47]

5.8.2 *Containment MinHash*

Just as containment measures are asymmetric, so are approaches to estimate containment values. Koslicki et al. proposed computing a containment index by combining k-mins sketches with a bloom filter[48], to estimate both Jaccard similarity and containment.

[45] Broder, "On the Resemblance and Containment of Documents", 1997.

[46] Broder, "On the Resemblance and Containment of Documents", 1997.

[47] A notable example being (Yang, Zola, and Aluru, "Parallel Metagenomic Sequence Clustering via Sketching and Maximal Quasi-Clique Enumeration on Map-Reduce Clouds", 2011)

[48] Koslicki and Zabeti, "Improving MinHash via the Containment Index with Applications to Metagenomic Analysis", 2019.

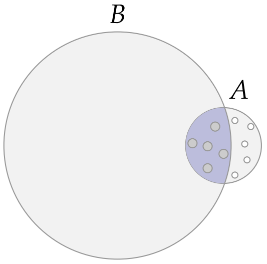


Figure 5.11: Illustration of the containment index. The MinHash values from the k-mins sketch $S^{|\bullet|\bullet|}(A)$ are shown as small circles within the sets A and B. The fraction of MinHash values contained within B is an estimate for the Jaccard containment of A in B.

Given two sets A and B, where $|A| < |B|$, we want to compute the Jaccard containment of A in B:

$$\mathcal{J}^C(A,B) = \frac{|A \cap B|}{|A|}$$

When using classic resemblance MinHash, the k minimizers of A and B are distributed uniformly within each document (cf. Figure 5.10). This leads to low Jaccard similarity, since many[49] minimizers of B cannot be matched by A.

Note that in Figure 5.11, overlapping circles denote shared values, i.e. all values symbolized by the purple shaded area are contained in both A and B. Consequently, the minimizers of A that fall into the purple area are contained in B, albeit not as minimizers, and the remaining minimizers of A are not present in B. This property can be modeled using a bloom filter, containing all items of B. Recall that a bloom filter $\mathcal{B}$ is a data structure that can answer membership queries with a false positive error rate (see Section Bloom Filter).

Koslicki et al. populate a bloom filter $\mathcal{B}(B)$ with all elements of the bigger set B and query it with items from $S_k^{|\bullet|\bullet|}(A)$. The number of minimizers of A that are contained in B allow estimating the containment $\mathcal{J}^C(A,B)$ as

$$\mathcal{J}^{C*}(A,B) = \frac{1}{k}\sum_{i=1}^{k} m_i(A,B) \qquad m_i(A,B) = \begin{cases} 1, & \text{if } m_i \in \mathcal{B}(B) \\ 0, & \text{else} \end{cases}$$

where $m_i \in S_k^{|\bullet|\bullet|}(A)$ is the minimizer of the i-th hash function and $\mathcal{B}(B)$ is the bloom filter containing all items of B. Intuitively, this is equivalent to counting the minimizers within the overlapping area of A and B in Figure 5.11. In addition to containment estimation, the containment estimate allows computing a Jaccard similarity estimate

$$\mathcal{J}^*(A,B) = \frac{|A|\mathcal{J}^{C*}(A,B)}{|A| + |B| - |A|\mathcal{J}^{C*}(A,B)}$$

Furthermore, Koslicki et al. have shown that computing the containment index requires less hash functions than classic MinHash.[50]

A downside of this approach is that it requires a potentially large bloom filter for each reference set. While the authors argue that the size of the bloom filter is amortized by the reduced number of hash functions required by the Containment MinHash technique,

[49] Depending on the size difference of A and B.

[50] Koslicki and Zabeti, "Improving MinHash via the Containment Index with Applications to Metagenomic Analysis", 2019.

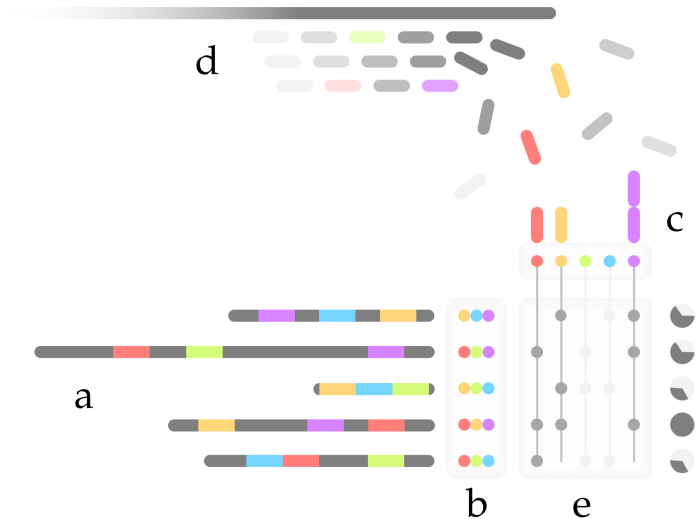


Figure 5.12: Illustration of the mash screen workflow with $k = 3$. For each reference sequence (a), a bottom-3 sketch (b) is computed. Iterating through the read data (d), hashing each value yields key to query (c) a count hash table (e). The number of minimizers (out of $k = 3$ possible one) detected in the input is an estimate for the containment (f). This figure has been derived from Figure 2 of (Ondov et al., "Mash Screen: High-throughput Sequence Containment Estimation for Genome Discovery", 2019), released under CC BY 4.0 license.

this gain depends on the properties of the application. The use case described by Koslicki et al. is a large reference set B and a collection of several smaller query sets A_i, for which the space required is reduced significantly.

5.8.3 *Mash Screen*

Similarly to the Containment MinHash approach, Mash Screen[51] only sketches one of the two input sets to estimate containment. This extension of the Mash software, which uses bottom-k sketches to estimate resemblance[52] is used for the analysis of metagenomic sequencing data. The Mash software[53] as published in 2016 is able to estimate the resemblance ($\frac{|A \cap B|}{|A \cup B|}$) of (meta-) genomes using bottom-k sketches.

Given a sequencing dataset B with reads from an unknown combination of species, and a set of reference genomes $A_i, |A_i| \ll |B| \, \forall i$, which reference genomes are contained within B? In contrast to Containment MinHash, Mash Screen uses the complementary asymmetric solution, sketching the references instead of the query.

First, a bottom-k sketch $S_k^{\underline{\vdots}}(A_i)$ for each reference sequence is computed and stored. Recall that we assume that minimizers of the MinHash values $m_j \in S_k^{\underline{\vdots}}(A_i)$ are uniformly distributed throughout each set A_i. To compare these minimizers of the A_i against all items in the dataset B, all items from B are compared with all sketches $S_k^{\underline{\vdots}}(A_i)$ directly. In other words, instead of entering the items into a data structure, Mash Screen performs a linear scan of the input set to successively compute containment estimates

$$\mathcal{J}^{C*}(A_i, B) = \frac{\#\text{ minimizers of } A_i \text{ found in } B}{k} = \frac{|S_k^{\underline{\vdots}}(A_i) \cap B|}{k}$$

for each A_i.

This approach, which is illustrated in Figure 5.12, can be used as an online (streaming) algorithm, for example during the anal-

[51] Ondov et al., "Mash Screen: High-throughput Sequence Containment Estimation for Genome Discovery", 2019.

[52] Cf. Overview of LSH in Bioinformatics.

[53] Ondov et al., "Mash: Fast Genome and Metagenome Distance Estimation using MinHash", 2016.

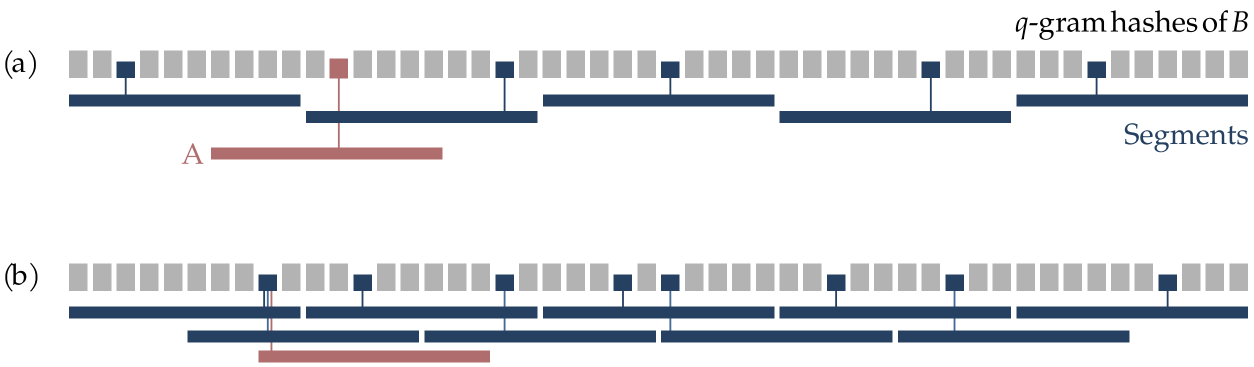


Figure 5.13: Two decompositions of the sequence B using static window sizes. Minimizers of a sequence are denoted by colored blocks and a line to the sequence they are a minimizer of. Simple, non-overlapping decomposition (a), the sequence A can *fall through* the minimizers. When using overlapping windows, in the case of (b) with an overlap of $0.5w$, this can be averted at the cost of increased memory consumption.

ysis of a MinION dataset. For implementation, sketches are represented using a count hash table, containing the sketch entries of all genomes. Each q-gram g in the input is hashed and used to query the table. If $h(g)$ is in the table, its counter is incremented. The number of sketch entries $m \in S_k^{\ddagger}(A_i)$ with non-zero entries in the count hash table are collected and counted, which yields $|S_k^{\ddagger}(A_i) \cap B|$. Dividing this count by k yields the containment estimate $\mathcal{J}^{C*}(A_i, B)$.

Through the use of atomic data types,[54] this computation can also be parallelized.

5.8.4 *Static Window Decomposition*

Another approach to avoid the size difference problem is to limit its impact by decomposing the larger set into smaller segments. Given the use cases presented above, i.e. finding occurrences of small sets within one or more large ones, an intuitive way to do this is to split up the large set into windows of static size w. This approach, combining k-mins sketching with overlapping static windows, is employed by the variant tolerant read mapper VATRAM.[55]

If a read fits perfectly into the window, a high Jaccard similarity between reference subsequence $B_{[iw:(i+1)w]}$ and query sequence A_j is akin to a high containment of A_j within B (see Figure 5.13 (a)). In the less benevolent case that A_j spans two reference windows $B_{[iw:(i+1)w]}$ and $B_{[(i+1)w:(i+2)w]}$, the Jaccard similarities

$$\mathcal{J}(A_j, B_{[iw:(i+1)w]}) \qquad \mathcal{J}(A_j, B_{[(i+1)w:(i+2)w]})$$

are bounded by the overlap A_j has with each of them. This can be mitigated by using overlapping windows at the cost of additional memory (see Figure 5.13 (b)).

The main problem with static window approaches is that window boundaries and positions of minimizers are independent. It is not possible to shift window boundaries to react to the sequence composition. These can be circumvented by directly incorporating minimizer information into the window decomposition through dynamically sized windows.

[54] In this context atomic means that operations are guaranteed to complete in multi-thread environments and cannot be interrupted.

[55] Quedenfeld and Rahmann, "Variant Tolerant Read Mapping using Min-Hashing", 2017.

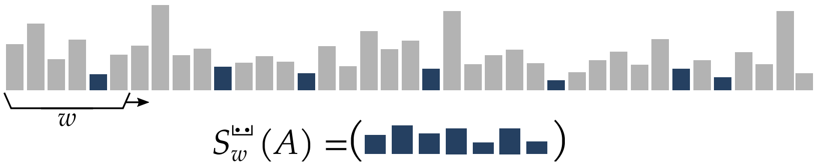


Figure 5.14: Illustration of a winnowed sketch with size w. The window (black bracket on the left) is moved along the sequence of hash values (gray blocks). At each position, the window's MinHash value is computed and compared with the active minimum. If a new minimum is found, it is added to the sketch (blue blocks) and a new segment starts.

5.9 *Winnowing and Minimizers*

A class of LSH approaches which relies on sketches generated from minimal hash values, but is not considered a MinHash variant, are the Winnowing and Minimizer strategies independently described by Schleimer et al.[56] and Roberts et al.[57] Roberts' minimizer definition focuses more on encoding biological sequences using their minimizers only,[58] while Schleimer's approach aims at document fingerprinting. To avoid confusion with the term minimizers introduced above to denote a q-gram hashing to a minimal hash value, we will refer to both techniques as winnowing from here on.

Where MinHash approaches with k-mins and bottom-k sketches always consider whole sets, winnowing introduces locality information. The core idea is to compute a MinHash value not for the whole document but for a moving window, thereby reducing the range a single minimizer can influence.

A MinHash value is computed for each window of w consecutive hash values[59] as illustrated in Figure 5.14.[60] However, only changed minima are reported: if the windows starting at q-gram i and $i+1$ have the same MinHash value m and the window at $i+2$ has a different MinHash value $m' \neq m$, the sequence of MinHash values is reported as $(m, m', \dots)$.

A simplified version of the winnowing algorithm is illustrated in Algorithm 2. Note that in a more efficient implementation, explicitly computing the minimum for each window can be replaced by maintaining a list of minimizer positions and only comparing the new value with the acting MinHash value.

Winnowed MinHash values can be compiled into sketches which behave similar to the document sketches described above.

Definition 5.9.1 (Winnowed Sketch)**.** *Let h be a (ϵ-approximately) min-wise independent hash function and $G := \mathfrak{Q}(A, q)$ the q-gram sequence for a document A. We define the **winnowed sketch** or the **winnowed MinHash values** of A as:*

$$S^{\sqcup}_w(G) := ((p_i, m_i))_{i=0}^{j-1} \quad 0 \leq j \leq |G|$$

*where $m_i := h(g_i^m)$ is the i-th different MinHash value of the hash values of G, obtained by computing a MinHash value for each window of size w. We refer to the corresponding minimizers as the **winnowed minimizers** of the document. Additionally, the following property holds for all winnowed minimizer positions:*

$$p_{i+1} - p_i \leq w \quad \forall i \in [|S^{\sqcup}_w(G)| - 1]$$

[56] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

[57] Roberts et al., "Reducing Storage Requirements for Biological Sequence Comparison", 2004.

[58] This is possible if the window size is $w \leq q$, since all characters in the input sequence are covered by a minimizer.

[59] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

[60] Note that windows of size w are effectively w-grams of the q-gram sequence. For a sequence A there are $|A| - q + 1$ q-grams and $(|A| - q + 1) - w + 1$ windows.

Algorithm 2: Implementations of simple and robust winnowing. Both differ only in the way a new minimizer with the same (minimal) hash value is handled. While simple winnowing updates the minimizer for each new occurrence of the minimal value, robust winnowing keeps a minimizer as long as possible.

```
Input:
• Sequence of $q$-grams $\mathcal{Q}(A, q)$
• Window size $w$

▷ Initialization (fill the first window)
Create an empty list $M$ for computed minima;
Create a window $W$ of size $w$;
Fill $W$ with the first $w$ $q$-gram hashes;
Set current minimum $m := \min(W)$;

▷ Winnowing iterations for all remaining q-grams
foreach $g \in \mathcal{Q}(A, q)_{[w:]}$ do
    Compute hash value $v = h(g)$;
    Update window: Pop $W[0]$, insert $W[w] := v$;
    Compute new window minimum $m' = \min(W)$;
    if $m' \neq m$ then
        Append $m$ to $M$;
        Set new current minimum $m := m'$;
    else
        ▷ Identical minima are resolved differently by simple or robust winnowing:
        Simple Winnowing:
            Select the rightmost occurrance as minimizer;
            Append $m$ to $M$;
            Set new current minimum $m := m'$;

        Robust Winnowing:
            if $m$ was shifted out of the window then
                Select the rightmost occurrance as minimizer;
                Append $m$ to $M$;
                Set new current minimum $m := m'$;
            else
                Keep $m$ as minimum and continue;
Append last minimum $m$ to $M$;
return $M$;
```

In contrast to bottom-k and k-mins sketches[61], the size of a winnowed sketch is not constant. How many minimizers are selected into the sketch depends on both the window size w and the distribution of hash values in the data.

Corollary 5.9.2. *Let A be a document of length $|A|$, with its q-gram sequence $G := \mathfrak{Q}(A, q)$ containing $\ell_q = |A| - q + 1$ q-grams, and a window size $w < \ell_q$ for winnowing. This results in $\ell_w = \ell_q - w + 1$ window starting positions. Then the winnowed sketch $S_w^{\boxdot}(G)$ of A contains between $\left\lceil \frac{\ell_w}{w} \right\rceil$ and ℓ_w MinHash values.*

Proof.
- Consider a sequence of ℓ_q q-grams with the first MinHash value m_0 at position w: Slide the window across the q-gram sequence and place the next minimum m_{i+1} so that it enters the window, as m_i leaves the window. A minimum stays in the window for w consecutive starting positions, in other words it covers w input q-grams. Since all window positions must report a MinHash value, it is not possible to encounter fewer minima than this.

- Consider a sequence of ℓ_q q-grams sorted in (w.l.o.g.) decreasing order: The first window of w q-grams receives one MinHash value (initialization). For each of the remaining $\ell_q - w$ start positions of the window, the minimum changes, since a smaller value is observed.

□

The density of minimizers, i.e. the amount of q-grams that are selected to be minimizers in relation to the size of the document, is a measure how suited the input sequence is for a winnowing approach.

Definition 5.9.3 (Minimizer Density)**.** *The **minimizer density** of a q-gram sequence $G := \mathfrak{Q}(A, q)$ is defined as the fraction*

$$\frac{\#\ \textit{minimizers}}{\ell_w} = \frac{|S_w^{\boxdot}(G)|}{(|A| - q + 1) - w + 1}$$

i.e. the fraction of windows for which a new MinHash value is reported.

Applying Corollary 5.9.2, the minimizers density of a document can range between $\left\lceil \frac{\ell_w}{w} \right\rceil / \ell_w \approx \frac{1}{w}$ (Widely spaced minimizers:) and 1 (all q-grams are minimizers:).

A winnowed sketch with $|S_w^{\boxdot}(G)| = |G| - w + 1$ MinHash values does not offer any edge over just using the q-gram sequence G. Since a change of MinHash values is expected once every half window width,[62] the expected density assuming independently uniformly distributed keys is $d = \frac{2}{w+1}$.[63]

Assuming the minimizers were computed using an (ϵ-approximately) min-wise independent hash function, the density is also an estimator for the entropy of the input sequence. As seen in the proof of Corollary 5.9.2, the density of minimizers does not only depend

[61] But similar to modulo sketches.

[62] Roberts et al., "Reducing Storage Requirements for Biological Sequence Comparison", 2004.

[63] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

on the qualities of the hash function, but also on setup of the input sequence.

Recall that in genomic sequences repetitive regions[64] can be present. These have the potential to create repeating runs of equiminimal minimizers leading to long sketches that contain mostly redundant information. Since sketches need to be saved, this can become a performance concern with respect to memory.

A solution for this problem is a slight variation of the winnowing algorithm. Instead of updating the sketch with the rightmost occurrence of an equiminimal hash value, we keep its first occurrence as long as possible. This approach is called robust winnowing[65] and intuitively condenses runs of equiminimal sequences. If not explicitly mentioned otherwise, we consider all winnowed sketches to be obtained through robust winnowing.

Computing a minimizer using the winnowing strategy allows the following interpretation:

Lemma 5.9.4. *Let $G := \mathfrak{Q}(A,q)$ be a q-gram sequence and $G' := G_{[i:i+j]}$, $j \geq w$ a subsequence of G with length at least w. Then the smallest MinHash value of G' is contained in the winnowed sketch of G:*

$$S_1^{\vdots}(G') \in S_w^{\sqcup}(G)$$

Proof. Following Corollary 5.9.2 minimizers can be at most w positions apart. Since $|G'| \geq w$, G' covers at least one minimizer $m \in S_w^{\sqcup}(G)$. Consequently, m dominates all other hash values in at least one window of length w of G and must therefore be contained in $S_w^{\sqcup}(G)$.

Consider a MinHash value $m \in S_1^{\vdots}(G') \notin S_w^{\sqcup}(G)$. Since $m \in S_1^{\vdots}(G')$, there is no smaller hash value in the hashes of G' and consequently in the subsequence $G_{[i:i+j]}$, $j \geq w$. The value m is not selected as MinHash value of the windows fully contained within $G_{[i:i+j]}$, from which follows that there is a smaller hash value $m' < m, m' \in S_w^{\sqcup}(G)$. This leads to a contradiction, since m is the smallest value of the segment $G_{[i:i+j]}$ ↯. □

In other words, based on the window size w, we can guarantee that two sequences which are contained within each other share a winnowed minimizer, if the smaller contains at least one full window.

Winnowing approaches are commonly used in plagiarism detection[66] through detecting local matching subsequences, e.g. copied paragraphs. However, using them to estimate similarity measures is not as well established. While it is possible to treat winnowed MinHash values like a bottom-k sketch, or derive a bottom-k sketch to estimate Jaccard similarity,[67] this approach removes locality information. An approach that preserves the order of minimizers would be to compute an alignment between winnowed sketches of two documents. However, this does not allow the detection of structural

[64] Low entropy regions like homopolymers or short tandem repeats.

[65] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

[66] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

[67] This approach is used by the MASHMAP software described below (p. 118).

variations like exchanged subsequences. Due to this, winnowing approaches are most commonly used to identify single matching windows for further analysis.

Winnowed sketches can be used to estimate containment of linear sequences. Given two sequences $A < B$, where A is a subsequence of B, then $S_w^{\sqcup}(A)$ is a subsequence of $S_w^{\sqcup}(B)$. The containment of A in B can be estimated as:

$$c^*(A, B) = \frac{|\text{matching subsequences of } S_w^{\sqcup}(A) \text{ and } S_w^{\sqcup}(B)|}{|S_w^{\sqcup}(A)|} \tag{5.20}$$

To avoid performing all-vs-all comparison, in practice, winnowed MinHash values of the reference sequences $B_i, i \in [n]$ can be stored in an index like a hash table mapping winnowed MinHash values to sequence positions. For a query sequence A, we perform a hash table lookup with all MinHash values $m_j \in S_w^{\sqcup}(A)$ and estimate the containment $c^*(A, B_i)$ for each observed target sequence as the number of matches found for that sequence divided by $|S_w^{\sqcup}(A)|$.

Storing MinHash values from k-mins, bottom-k, or winnowed sketches in a hash table offers a new challenge. Through selecting for small values, the key set used to index the hash table is skewed. The size distribution of MinHash values in a sketch is influenced by the hash function, document size, and input data. Sketches computed using the k-mins or bottom-k strategies are expected to contain smaller MinHash values for large documents, since there are more input items that can be selected as minimizer. Through the locality of winnowing, larger MinHash values are more likely to occur, since one subsequence of larger hash values is enough to introduce a large MinHash Value. This influences the decision for data structures used to save MinHash values. We analyze the behavior of winnowed MinHash values obtained using different hash functions, especially focusing on the distribution of segment lengths in the following chapter. The remainder of this chapter provides examples for the flexible and widespread use of locality sensitive hashing strategies applied in bioinformatics.

5.10 Overview of LSH in Bioinformatics

Bottom-k sketches are widely used in current MinHashing implementations,[68] since they offer superior runtime performance, as illustrated in Table 5.1. The properties of k-mins sketching, on the other hand, are beneficial for parallelization, i.e. allowing to maintain sketches distributed across processor cores. Especially the tolerance for small errors implemented by the Jaccard similarity makes MinHash strategies a popular tool in bioinformatics. This section will briefly introduce several uses of MinHash variants and in bioinformatics software (in alphabetical order of software names).

[68] Thorup, "Bottom-k and Priority Sampling, Set Similarity and Subset Sums with Minimal Independence", 2013.

Strategy	Sketch Document	Compare Sketches
k-mins sketching	$\mathcal{O}(k \cdot n)$	$\mathcal{O}(k)$
bottom-k sketching	$\mathcal{O}(\lg k \cdot n)$	$\mathcal{O}(k)$
winnowed sketching	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 5.1: Runtimes for k-mins and bottom-k sketching of documents with n items. For a k-mins sketch, k hash functions are evaluated per item. For a bottom-k sketch only one function is evaluated, but maintaining a sorted list of hashes is logarithmic in k. Computing winnowed sketches requires one hash function evaluation per item, as does comparing them since in the worst case a sketch can contain all items. Note that computing multiple winnowings with k hash functions is also possible, resulting in a runtime of $\mathcal{O}(k \cdot n)$ for sketching and comparison.

Balaur The BALAUR[69] framework is a read mapping software designed to run in commercial cloud systems, like Amazon Elastic Compute Cloud, Azure or Google Cloud. When working with data from human patients, this poses additional restrictions to protect the privacy of sensitive genomic data. BALAUR locally computes candidate alignments using MinHashing which are then verified in the cloud environment.

After discarding overrepresented q-grams, a k-mins sketch is computed using a two stage hash function:

$$h_i(x) = h_{a,b}(h'(g)) \quad h_{a,b} \in \mathcal{H}^{\text{lin}}, \quad h' \text{ is a GPHF, } g \text{ is a } q\text{-gram}$$

For a set read length l, a sketch $S_k^{|\cdot|\cdot|}(r)$ is computed for each window r of length l of the reference sequence. The computation of these sketches uses a rolling hash approach which only recomputes MinHash values if necessary. Using these sketches, a reference index, comprising T tables with $B = 2^m$ slots each, is constructed by projecting k-dimensional sketches down to $[B]$ for each table using a multiply-shift hash function. The projection for each table is performed by selecting b items from $S_k^{|\cdot|\cdot|}(r)$, similar to a gapped (b,k)-gram. When analyzing a read, all segments that share more than a certain number of hits in the tables are passed to the alignment step. Verification of alignment position in the cloud is performed using q-gram voting.

The parameters used to index a human genome (GRCh37) with this approach as described in the evaluation were: sketches with $k = 128$, $T = 78$ hash tables with $M = 18, B = 2^{18}$ entries and projections of size $b = 2$.

Canu, MHAP Both MHAP[70] and Canu[71] find overlaps of TGS long reads for assembly using a two step process. Since both employ a similar approach, with Canu being an updated version of MHAP,[72] we will focus on describing Canu here. Furthermore, we omit the actual long read assembly and focus this description on step one, the identification of overlapping fragments.

Overlap identification itself is a two step process. First, likely candidates are identified using a weighted k-mins sketch, followed by computing the Mash distance (described in the following paragraph, see Definition 5.10.1) between the overlapping subfragments using bottom-k sketches. The weighted MinHash approach applied by Canu is a variation of tf-idf weighting[73] with discrete weights. In this context tf stands for term frequency, i.e. how often a q-gram

[69] Popic and Batzoglou, "A Hybrid Cloud Read Aligner Based on MinHash and Kmer Voting That Preserves Privacy", 2017.

[70] Berlin et al., "Assembling Large Genomes with Single-Molecule Sequencing and Locality-Sensitive Hashing", 2015.

[71] Koren et al., "Canu: Scalable and Accurate Long-Read Assembly via Adaptive k-Mer Weighting and Repeat Separation", 2017.

[72] Koren et al., "Canu: Scalable and Accurate Long-Read Assembly via Adaptive k-Mer Weighting and Repeat Separation", 2017.

[73] Chum, Philbin, and Zisserman, "Near Duplicate Image Detection: min-Hash and tf-idf Weighting", 2008.

g occurs in a specific read, and idf denotes inverse document frequency, i.e. the frequency of g across all reads. Each q-gram in a read is hashed using a number of hash functions based on the weight

$$\begin{aligned} w(g) &= \mathrm{tf}(g) \cdot \mathrm{idf}(g) \\ \mathrm{tf}(g) &:= \#\text{ occurrences of } g \text{ in the active read} \\ \mathrm{idf}(g) &:= T\left(\log\left(\frac{f_{\max}}{f(g)} - \alpha\right)\right) \end{aligned} \tag{5.21}$$

where T linearly transforms the idf value to the range $[1, \mathrm{idf}_{\max}]$ and $\alpha \in [0, 1]$ parametrizes the influence of less common q-grams.[74] Consequently, overrepresented q-grams are only hashed with 1 hash function, while uncommon and therefore specific q-grams receive up to $\mathrm{idf}_{\max}$ hash values. Highly weighted q-grams receive more than one hash function to contribute to the k-mins sketch, which increases their probability to become a minimizer. The i-th entry of a tf-idf weighted k-mins sketch is computed from the following values:

$$S_k^{|\cdot|\cdot|}[i] = \min \bigcup_{g \in Q(A,q)} \{\, h_{i,j}(g) \mid j := w(g) \,\}$$

Hash values are computed using MMH3[75] for the first hash value. This is used to compute $w(g)$ and to seed an XORShift random number generator (RNG) to derive the hash functions $h_{i,j}$.

For reads that pass this first stage filter, bottom-k sketches with $k = 1\,500$ entries are used to compute the Mash distances between active and similar reads as a second stage filter. The overlapping region is estimated using a combination of these bottom sketches as well as q-gram counting.

Mash[76] Mash computes (meta-) genome distances using bottom-k sketches to estimate genome similarity and a bloom filter to exclude erroneous q-grams. To take differences in genome size into account as well as incorporating a Poisson model for mutations, Ondov et al. define a distance measure used for clustering of sequences.

Definition 5.10.1. *Mash Distance Let $A, B \in \Sigma^*_{DNA}$ be two sequences. Assuming unique q-grams and random, independent mutations, the **Mash Distance** of A and B is defined as:*

$$D(A, B) = -\frac{1}{q} \ln \frac{2j}{1 + j}$$

where $j := \mathcal{J}^*(A, B)$ is the estimated Jaccard similarity from bottom-k sketching.

When working on raw TGS long reads with high error rates,[77] Mash uses a bloom filter to exclude likely erroneous q-grams. Assuming a coverage of at least 5×, q-grams that appear in the input only one time are likely caused by sequencing errors. This is

[74] Koren et al., "Canu: Scalable and Accurate Long-Read Assembly via Adaptive k-Mer Weighting and Repeat Separation", 2017.

[75] Appleby, *Murmurhash3*, 2016.

[76] Ondov et al., "Mash: Fast Genome and Metagenome Distance Estimation using MinHash", 2016.

[77] Cf. Section PacBio SMRT and ONT MinION – Single Molecule Sequencing

compensated by maintaining a MinHash candidate set S' which contains pairs of minimizer g_i^{m} alongside their abundance c_i. A MinHash value only enters the actual bottom sketch $S_k^{\vdots}$ if it is encountered at least c times. For each encountered minimizer g_i^{m}, the following two cases can arise, based on abundance:

$g_i^{\mathrm{m}} \notin S'$: The minimizer g_i^{m} has not been encountered in this dataset and might be a sequencing artifact. It is added to S' with an abundance of $c_i = 1$ to be maintained as a MinHash candidate.

$g_i^{\mathrm{m}} \in S'$: If g_i^{m} is already present in the candidate set, then it has been encountered before. Increase the abundance of g_i^{m} and compare to the abundance threshold:

$c_i \geq c$ Remove g_i^{m} from the candidate list S' and add g_i^{m} to $S_k^{\vdots}$.

$c_i < c$ Continue with the next q-gram.

Through this approach, a g-gram can only become part of the sketch, if it is present more than c times in the input data.

The applications shown in the paper include a clustering of all ∼54 118 genomes in NCBI RefSeq, using 32-bit encoded 16-grams and $k = 400$. This resulted in a total sketch size of 94 MB and required 33 hours of CPU time. Another application was the identification of an Ebola strain from MinION sequencing data within 10 minutes of the start of the sequencing run.

Mashmap, Mashmap2 MASHMAP[78] performs mapping of long, noisy TGS reads, using a combination of bottom-k sketching and winnowing. MASHMAP2[79] uses a similar approach to compute whole genome alignments.[80]

For MASHMAP, Jain et al. reformulate the read mapping problem as finding all pairs of read A and reference subsequences B_i so that the Jaccard similarity $\mathcal{J}(A, B_i)$ is higher than the expected similarity

$$\mathcal{J}(A, B_i) \geq \mathcal{G}(\epsilon, q) - \delta$$

where δ is the margin of error for the Jaccard estimation given a 90% confidence interval and $\mathcal{G}(\epsilon, q)$ is the expected Jaccard similarity based on the per-base error rate ϵ.

Through winnowing the sampling frame is reduced before selecting MinHash values for the bottom-k sketch from the winnowed sketch. Compute the set $S_w^{\sqcup}(A)$ of minimizers of read A using winnowing[81] and find the k smallest hashed q-grams

$$S_k^{\vdots}(S_w^{\sqcup}(A)) = \min{}^k \{ x : (x, \mathrm{pos}) \in S_w^{\sqcup}(A) \},$$

where x is a hash value. The estimated Jaccard similarity $\mathcal{J}'(A, B_i)$ of two bottom sketches is computed as

$$\mathcal{J}'(A, B_i) := \frac{|S_k^{\vdots}(S_w^{\sqcup}(A) \cup S_w^{\sqcup}(B)) \cap S_k^{\vdots}(S_w^{\sqcup}(A)) \cap S_k^{\vdots}(S_w^{\sqcup}(B))|}{|S_k^{\vdots}(S_w^{\sqcup}(A)) \cup S_k^{\vdots}(S_w^{\sqcup}(B))|}.$$

[78] Jain et al., "A Fast Approximate Algorithm for Mapping Long Reads to Large Reference Databases", 2017.

[79] Jain et al., "A Fast Adaptive Algorithm for Computing Whole-Genome Homology Maps", 2018.

[80] More information on this approach can be found in the dissertation of Chirag Jain (Jain, "Long Read Mapping at Scale: Algorithms and Applications", 2019).

[81] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

Algorithm 3: The MASHMAP algorithm for mapping long reads against a reference database.

```
▷ Input:
• Read sequence $A$
• Reference sequences $B_i$
• Window size $w$
• Sketch size $k$
• Mapping threshold $\tau = \mathcal{G}(\epsilon, q) - \delta$

▷ Index Reference Sequences
foreach Reference sequence $B_i$ do
    • Compute winnowed sketch $S_w^{\sqcup}(B_i)$
    • Compute bottom-$k$ sketch $S_k^{\underline{\vdots}}(S_w^{\sqcup}(B_i))$ of winnowed MinHash values
    • Save hash table $\mathcal{H}$, mapping MinHash values to a dictionary of positions.

▷ Computing Candidate Intervals
foreach Query read $A$ do
    • Compute winnowed bottom-$k$ sketch $S_k^{\underline{\vdots}}(S_w^{\sqcup}(A))$
    • Assemble candidate position set $T$ using $|S_w^{\sqcup}(A) \cap S_w^{\sqcup}(B_i)| \geq \lceil k \cdot \tau \rceil$ as filter criterion.

    ▷ Interval Validation
    foreach Candidate position in $T$ do
        • Compute similarity $\mathcal{J}'(A, B_i)$, using $\mathcal{H}$.
        • If similar enough, return $(i, \mathcal{J}(A, B_i))$
```

To avoid this computation, if possible, a second level of similarity estimation is employed: By comparing the number of shared winnowed minimizers in relation to bottom sketch size and expected error rate

$$|S_w^{\sqcup}(A) \cap S_k^{\sqcup}(B_i)| \geq k \cdot \tau \quad \tau = \mathcal{G}(\epsilon, q) - \delta$$

suitable candidates are identified.

To support variable length reads, MASHMAP uses multi-level winnowing, i.e. computing multiple references for window sizes $\{w, 2w, 4w, \dots\}$. The optimal window size w is computed with respect to the read length l. If a read is longer than a minimum read length l_0, the window size is rounded to the closest smaller reference window size $l \in \{w, 2w, 4w, \dots\}, l \geq l_0$. These sketches can be computed recursively, since $S_{2w}^{\sqcup}(A) \subseteq S_w^{\sqcup}(A)$,[82] meaning that winnowed MinHash values of a bigger window size are a subset of the MinHash values of a smaller window size.

The MASHMAP algorithm is shown in Algorithm 3. MASHMAP2 uses a similar sampling and MinHashing approach as MASHMAP.

[82] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

Based on identified fragments, it uses a variant of the Shamos-Hoey algorithm to identify overlapping segments in $\mathcal{O}(n \log n)$ time and filter out low scoring segments before alignment.

MC-MinH[83] The Metagenome Clustering using Minwise based Hashing (MC-MinH) approach uses k-mins MinHashing and greedy agglomerative clustering to solve metagenomic binning.

Given a set $\mathcal{S} = \{ s_0, \ldots, s_{n-1} \}$ of input sequences and a similarity threshold θ, sequence similarity for the clustering step is computed using k-mins sketches computed from the q-grams of each sequence using the hash function

$$g_i(x) = (a_i x + b_i \mod p) \mod m$$

where p is prime and m is the feature set size.

During the clustering step, a sequence s_i that is not yet assigned to a cluster is selected to form a new cluster. Using the similarities computed from the k-mins sketches for all still unassigned sequences, sequences that surpass the similarity threshold θ are added to the cluster.

Opal[84] The Opal software uses a guided approach to the generation of LSH functions to solve metagenomic binning, i.e. assigning reads to a reference database.

Opal computes hashes from gapped (q,t)-grams by selecting columns from the integer encoding of a q-gram. Drawing inspiration from Gallager codes,[85] a $k \times q$ binary matrix that covers all positions of the input q-gram approximately equally is used to ensure that all positions of the q-gram are covered. Each line in the matrix contains a set of t 1-positions, which describe the positions selected for one LSH function. For a given q-gram $g \in \Sigma^q_{\text{DNA}}$, and a set of 1-positions $i_1, \ldots, i_t$ obtained from a line in the Gallager matrix, the hash value

$$h(g) = (g_{[i_1]}, g_{[i_2]}, \ldots, g_{[i_t]})$$

is the concatenation of the characters selected by the 1-positions. This is equivalent to a 2-bit encoding of the (q,t)-gram defined by the line in the Gallager matrix.

minimap, minimap2, miniasm[86] Minimap2 is a mapper and pairwise aligner for TGS long reads which use winnowed sketches to identify seeds for seed-and-extend alignments. Miniasm uses overlaps identified by Minimap for de-novo assembly. Since the minimizer steps of Minimap and Minimap2 are similar, we describe the Minimap workflow here.

The core idea of Minimap computing double-strand (w,q)-minimizers (m,i,r) comprising the minimizer value m, a position i within the sequence, and the strand r. The minimizer value m is computed as

$$m = h(s_{[i:i+q-1]}, r) = \min \{ h(\pi(s^k_{j+p}, r')) : 0 \leq p \leq w, r' \in \{ 0,1 \} \}$$

[83] Rasheed and Rangwala, "MC-MinH: Metagenome Clustering using Minwise Based Hashing", 2013; Rasheed and Rangwala, "A Map-Reduce Framework for Clustering Metagenomes", 2013.

[84] Luo et al., "Metagenomic Binning Through Low Density Hashing", 2018.

[85] Gallager, "Low-density Parity-check Codes", 1962.

[86] Li, "Minimap and Miniasm: Fast Mapping and de novo Assembly for Noisy Long Sequences", 2016; Li, "Minimap2: Pairwise Alignment for Nucleotide Sequences", 2018.

where the strand function $\pi : \Sigma^*_{\text{DNA}} \times \{0,1\} \rightarrow \Sigma^*_{\text{DNA}}$ yields the reverse complement of s for $\pi(s,1) = \bar{s}$ and the sequence s for $\pi(s,0)$. The hash function h is a compound hash function $h = h'(b(g))$, applying an invertible integer hash function h' to a 2-bit encoding of q-grams. This approach prevents q-grams starting with an `A` homopolymer from being overrepresented as minimizers, since $\texttt{A} = \texttt{00}_2$ in 2-bit encoding.

The computed minimizers are stored in a two stage hash table to provide fast access for queries. To provide cache-friendly access, all minimizers are entered into an array which is then sorted, so that colliding values maintain memory locality. A hash table maintains a mapping of minimizer hash values m to triplets (t, i, r), comprising a target sequence index t, position i within the sequence, and strand orientation r. For memory efficiency, these triplets are bit-packed into 64-bit integers.

In the mapping step, this hash table is queried to find ϵ-away minimizer hits. Similar to identifying a banded alignment, these are all minimizers shared by both sequences that are max ϵ bases apart in both strand combinations (in practice, $\epsilon = 500\,\text{bp}$). Identified minimizer hits are clustered using single-linkage clustering and a longest increasing sequence problem is solved to find the longest colinear match. MINIMAP2 improves upon this step with more elaborate identification of matches.

VATRAM[87] The variant tolerant read mapper prototype VATRAM uses static window decomposition and a know reference database to reduce the impact of know genetic variants on read mapping. Certain SNPs and short indels are prevalent in human populations at a known rate. By adding the q-grams affected by these variants to the reference and using an LSH approach, VATRAM is able to mitigate the influence on mapping quality caused by known variants. This makes unknown variants more easily identifiable.

First, an index data structure is created for a given reference genome and an associated VCF file containing known variants. The reference is split into windows of constant size $w < n$, where n is the length of SGS reads to be mapped against the reference. To avoid the problem of minimizers falling between windows described above, the start position of windows is set to overlap. A new window starts every o positions, where $n < o < w$. Per default, these parameters are chosen as $w = 1.4n$ and $o = 1.25n$, so that windows overlap by $0.15n$ items.

MinHashes are computed for the q-gram of each window using k-mins sketching and the hash function family

$$\mathcal{H} := (h_i)_{i=1}^{k}\ h_i = h_{\text{enc}}(x) \oplus \pi_i$$

where π_i is a random integer value. VATRAM maintains k hash tables implemented as two layer succinct rank data structures. These map MinHash values of each window onto reference window positions.

[87] Quedenfeld and Rahmann, "Variant Tolerant Read Mapping using Min-Hashing", 2017; Quedenfeld and Rahmann, "Analysis of Min-Hashing for Variant Tolerant DNA Read Mapping", 2017.

For the read mapping step, the k-mins sketch of a read is computed to query the hash tables. Once a reference window is identified, a variant tolerant alignment is performed using a variant of Ukkonen's Algorithm for approximate pattern matching.[88] This also incorporates both SNPs and indels from the VCF file.

5.11 Conclusion

In this chapter we have given an overview of LSH approaches for the estimation of resemblance and containment values, as well as their implementation in bioinformatics. Table 5.2 provides an illustration of the LSH approaches most relevant to this work (as well as their associated notation), including one approach we describe in detail in the following section. While set-based approaches of similarity computation are sufficient for many applications, they can create false positives. One example of this would be two sequences $A = (x_1, \ldots, x_n, y_1, \ldots, y_n)$ and $B = (y_1, \ldots, y_n, x_1, \ldots, x_n)$ with $x_1 = \cdots = x_n \neq y_1 = \cdots = y_n$, which have a very high Jaccard similarity based on their q-gram sequences. This is not an uncommon occurrence in biological sequences due to structural mutations and alternative splicing. While approaches like OMH have been developed to address this issue, sequence based LSH approaches like Winnowing can offer additional solutions for this problem.

Another interesting problem is the management of MinHash and Winnowing hash values. If we want to use a MinHash value to address into a hash table, we generate an input sequence of keys which are strongly biased. This can affect the collision resolution strategy and potentially result in inefficient accesses to the data structure. In the following chapters we will describe our research covering the following topics:

- A variation of the winnowing approach which can reduce the impact of repetitive genomic regions on MinHashing strategies.
- An analysis of the distribution of winnowed segment lengths, i.e. the number of positions a MinHash value remains minimal.
- A two step hashing approach used to save MinHash values obtained from MinHash and winnowing strategies that does not rely on retaining the initial q-grams.
- Applications of MinHash and Winnowing strategies for biological problems, including protein similarity and chimera detection, including

[88] Ukkonen, "Finding Approximate Patterns in Strings", 1985.

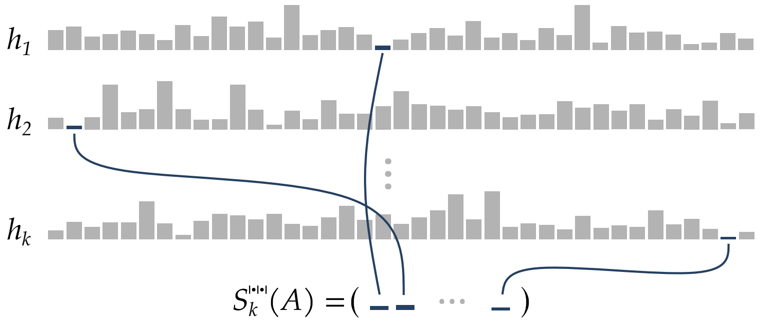
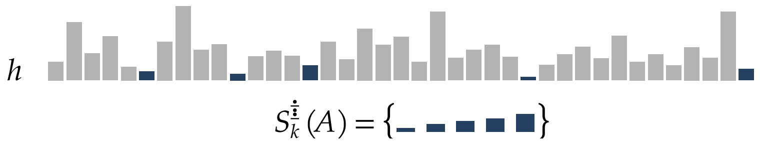
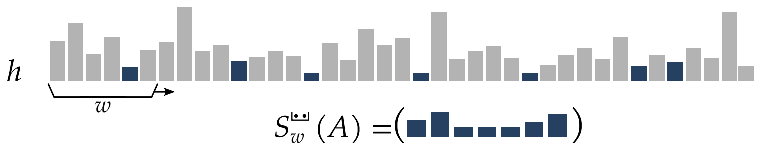
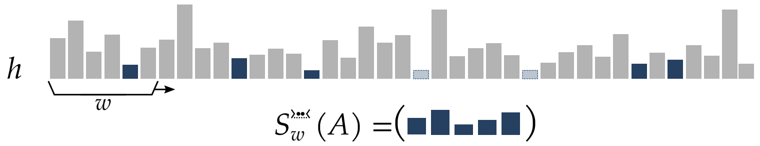
Symbol	Description	Page
$S_k(A)$	A k-mins sketch contains k entries, each of which was minimal under one of k different hash functions $h_1, \ldots, h_k$.  	101
$S_k(A)$	A bottom-k sketch contains the k smallest hash values of A using only one hash function h.  	102
$S_w(A)$	A (robust) winnowed sketch contains the minimal hash values for windows of length w using one hash function h. Equiminimal runs can persist for up to w window positions.  	111
$S_w(A)$	A compressed winnowed sketch contains the minimal hash values for windows of length w using one hash function h. Equiminimal runs have no upper limit.  	131

Table 5.2: Symbols used in this and the following chapters to denote different types of sketches for a document A. Compressed winnowed sketches are introduced in the following chapter.

6
Distribution of Minimizer Segment Lengths

In the section Winnowing and Minimizers (p. 111) we described winnowing as an LSH technique that incorporates locality information into minimizer-based strategies. The main metric used to describe how well a sequence is suited for winnowing was minimizer density, i.e. the amount of items selected as winnowed minimizers. As shown in Corollary 5.9.2, a winnowed sketch contains between $\lceil \ell / w \rceil$ and $\ell - w + 1$ MinHash values, where ℓ is the number of q-grams in the input sequence and w is the window size used for winnowing. However, the minimizer density offers no information on how minimizer positions are distributed throughout the sequence and how they interact with the composition and structure of the sequence.

In this chapter we describe the distribution of minimizers through the lens of segmentation, i.e. a decomposition of the input sequence based on its winnowed minimizers. We define a variant of the winnowing algorithm that allows compressing repetitive sequences into single minimizers and describe the length of runs with the same MinHash value. Finally, we show the expected distribution of segment lengths, assuming perfectly random hash values. Using this we show which hash functions closely approximate the behavior of perfectly random hash values with respect to winnowing.

6.1 Segmentation of a Sequence

As mentioned in the previous chapter, if two sequences share a winnowed minimizer, there is a high chance that they share an identical sequence. Using a winnowed sketch of a sequence, we can split up said sequence into subsequences that each are identified by its MinHash value, its start position in the sequence, and its length.

Definition 6.1.1 (Segment)**.** *Given a q-gram sequence $G := \mathfrak{Q}(A, q)$ for a document A and a hash function h, a* ***segment*** *$\mathfrak{s} := (m, p, l)$ is defined as a consecutive interval of winnowing window positions starting at position p, with identical MinHash value m (equiminimal values) under the hash function h for l window positions.*

A segment can be understood as a subsequence of the input text that has the same dominating minimizer. An example of seg-

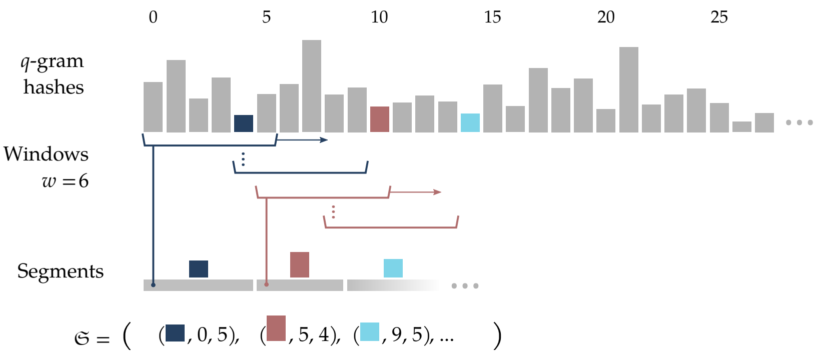


Figure 6.1: Example for a segmentation of a q-gram sequence. The blocks in the first line represent hash values, where a hash value is proportional to the height of the block. The first segment starts at window position 0 and ends at position 5 (resulting in a segment length of $5 - 0 = 5$), when the (blue) minimizer at position 4 is pushed out of the window. After that, the new (red) minimizer dominates the second segments (for four positions), until the smaller (teal) hash value enters the window.

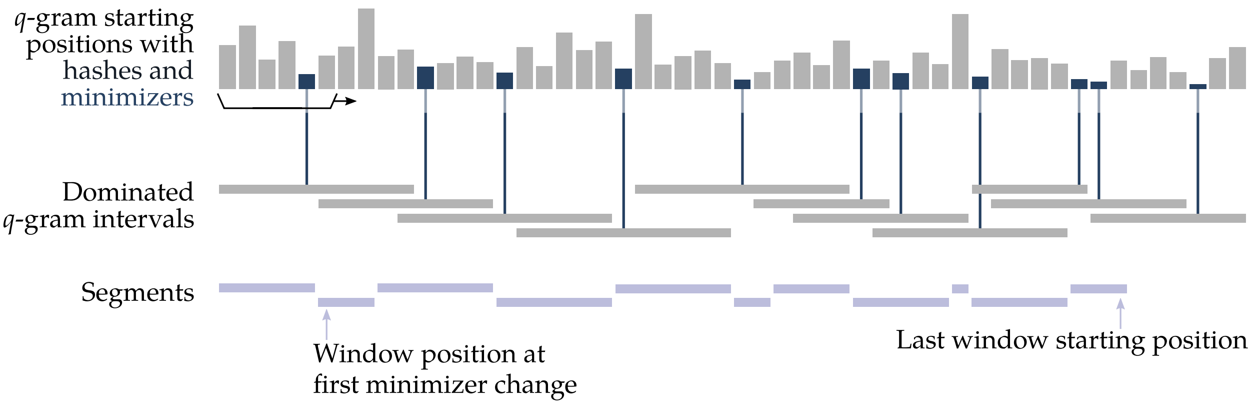

q-gram starting positions with hashes and minimizers
Dominated q-gram intervals
Segments
Window position at first minimizer change
Last window starting position

Figure 6.2: Decomposition of a sequence into segments. Hash values are symbolized by the height of the boxes in the first row; winnowed minimizers are highlighted blue. The q-grams dominated by a minimizer are shown as gray segments in the second row. Note that a q-gram can be dominated by several winnowing windows. The segmentation of the sequence is shown in purple. Each segment starts at the position its minimizer value is first encountered. This line is shorter, since for $l = |A| - q + 1$ q-grams, there are $l - w + 1$ window start positions.

ments induced by winnowed minimizers of a q-gram sequence is illustrated in Figure 6.1.

We derive the length of a segment from the number of window positions is covers, so that the sum of all segment lengths is equal to the number of window positions.

Definition 6.1.2 (Segment Length)**.** *The length of a segment $\mathfrak{s} = (m, p, l)$ is defined as the number of winnowing window positions it covers. It is denoted by $|\mathfrak{s}| := l$.*

Since during winnowing each window position returns a minimizer, the sequence of all segments of a document covers the whole sequence (cf. Algorithm 2).

Definition 6.1.3 (Segmentation)**.** *Given a q-gram sequence $G := \mathfrak{Q}(A, q)$ for a document A and a hash function h, a w-**segmentation** $\mathfrak{S}_h(G, w) := (\mathfrak{s}_i)_{i=0}^{k-1}$ is a decomposition of G into segments, so that:*

- *Each start position of a winnowing window is covered by a segment.*
- *Each q-gram $g \in G$ can be covered by up to w segments.*
- *A segmentation does not contain gaps, i.e. there is no window position that does not have a minimizer.*

For convenience, we also write $\mathfrak{S}_h(A, q, w) := (\mathfrak{s}_i)_{i=0}^{k-1}$ to denote the w-segmentation of the q-gram sequence of A.

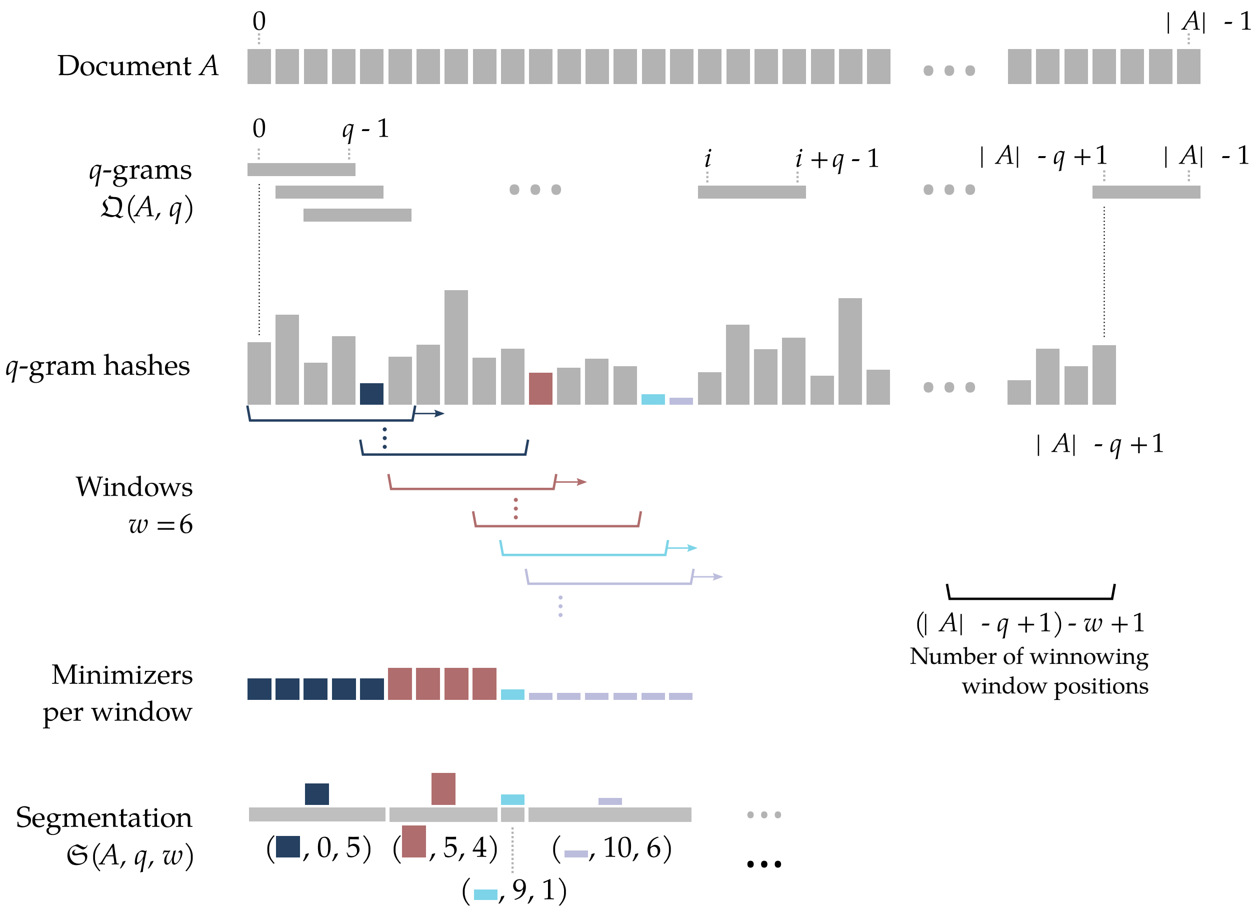


Figure 6.3: Different levels of description for a document A. The first layer shows the $|A|$ document positions, for example bases in a DNA string. Below, $|A| - q + 1$ q-grams, denoted by horizontal bars labeled with their start and end position within A are shown. For each of these q-grams, its hash value is represented by a vertical bar below its start position. As before, the numerical size of a hash is denoted by its height and minimizers are highlighted by color. Minimizers for each window are denoted the same way, with equiminimal runs shown in the same color. Finally, segments are denoted as 3-tuples, comprising a MinHash value, a window start position, and a segment length.

Note, that a segment is directly related to an entry in the winnowed sketch of a document. Therefore, the number of segments in a segmentation is described by the number of winnowed minimizers, as described in Corollary 5.9.2 (p. 113).

An example for a segmentation, illustrating hash values, associated dominated q-grams, and segments, can be found in Figure 6.2. Observing the segmentation shown in Figure 6.2, notice that it is not possible to select a subsequence of $w = 6$ consecutive q-gram hashes that does *not* cover at least one minimizer (cf. Lemma 5.9.4, p. 114). This allows us to use winnowed minimizers to estimate containment of sequences with different sizes, leveraging the locality of the computed minimizers to offset the size difference.

A segment of length k obtained by winnowing with a window size of w covers k window start positions, each of which cover w items of the winnowed sequence, which in turn are q-grams of the input sequence. This hierarchy is illustrated in Figure 6.3.

Observation 6.1.4. *A segment* $\mathfrak{s} \in \mathfrak{S}_h(A, q, w)$ *with* $|\mathfrak{s}| = k$ *covers* $k + w + q - 2$ *positions of the input sequence* A.

Proof. With length k, a segment covers $k + w - 1$ consecutive q-grams. A sequence of $k + w - 1$ q-grams cover

$$q + (k + w - 1) - 1 = k + w + q - 2$$

positions in the input sequence. For the longest possible segment covering the whole of A with $k = (|A| - q + 1) - w + 1$ we receive:

$$\begin{aligned} & k + w + q - 2 \\ = \quad & ((|A| - q + 1) - w + 1) + w + q - 2 \\ = \quad & |A| \end{aligned}$$

□

The distribution of segment lengths is a more detailed metric for the "winnowability" of a sequence. Note that in addition to sequence properties and hash function, it is also influenced by the employed winnowing strategy. After providing a use case for winnowing in the bioinformatics context, we define a variant of winnowing suitable to compress repeats. Further we empirically analyze the distribution of segment lengths using this winnowing variant and provide a recursive formula to compute the expected distribution.

6.2 *Application: A Segment Reference for Protein Similarity*

A practical application of sequence segmentation is creating a reference index using a hash table of winnowed MinHash values. For our TaxMapper software,[1] which is aimed to classify the species within a metagenomic sample, we need to solve the protein similarity problem. We have a protein database containing labeled protein sequences that support certain taxonomic classifications. Using DNA reads from a sample, we want to identify the most similar proteins in the database for each read in order to judge the capabilities and taxonomic distribution of the analyzed community. Since SGS DNA reads translated into amino acid sequences are shorter than most proteins, we are interested in the containment of reads within proteins (cf. Containment p. 95). Note that while the application we describe here uses amino acid sequences, the approach we describe here is feasible for DNA sequences as well.

In our current implementation of TaxMapper, we used the RAPsearch[2] software to compute protein similarities. However, creating and querying the index created by RAPsearch dominated the runtime of TaxMapper. While using other protein alignment tools like DIAMOND[3] or PALADIN[4] could mitigate this issue, these compute alignments for all identified candidates. Furthermore, for our application, actually computing the alignments is not even required and similarity estimation would suffice, which can be performed faster. In this section, we propose an index data structure using multiple winnowings to identify alignment candidates. Through the use of reference segmentation, we can identify likely alignment candidates and classify them before actually computing any alignment.

A workflow so solve this example application is illustrated in Figure 6.4. As Figure 6.4 (a) illustrates, reads are sourced from an

[1] Beisser et al., "TaxMapper: An Analysis Tool, Reference Database and Workflow for Metatranscriptome Analysis of Eukaryotic Microorganisms", 2017.

[2] Ye, Choi, and Tang, "RAPSearch: A Fast Protein Similarity Search Tool for Short Reads", 2011; Zhao, Tang, and Ye, "RAPSearch2: A Fast and Memory-efficient Protein Similarity Search Tool for Next-generation Sequencing Data", 2011.

[3] Buchfink, Xie, and Huson, "Fast and Sensitive Protein Alignment using DIAMOND", 2015.

[4] Westbrook et al., "PALADIN: Protein Alignment for Functional Profiling Whole Metagenome Shotgun Data", 2017.

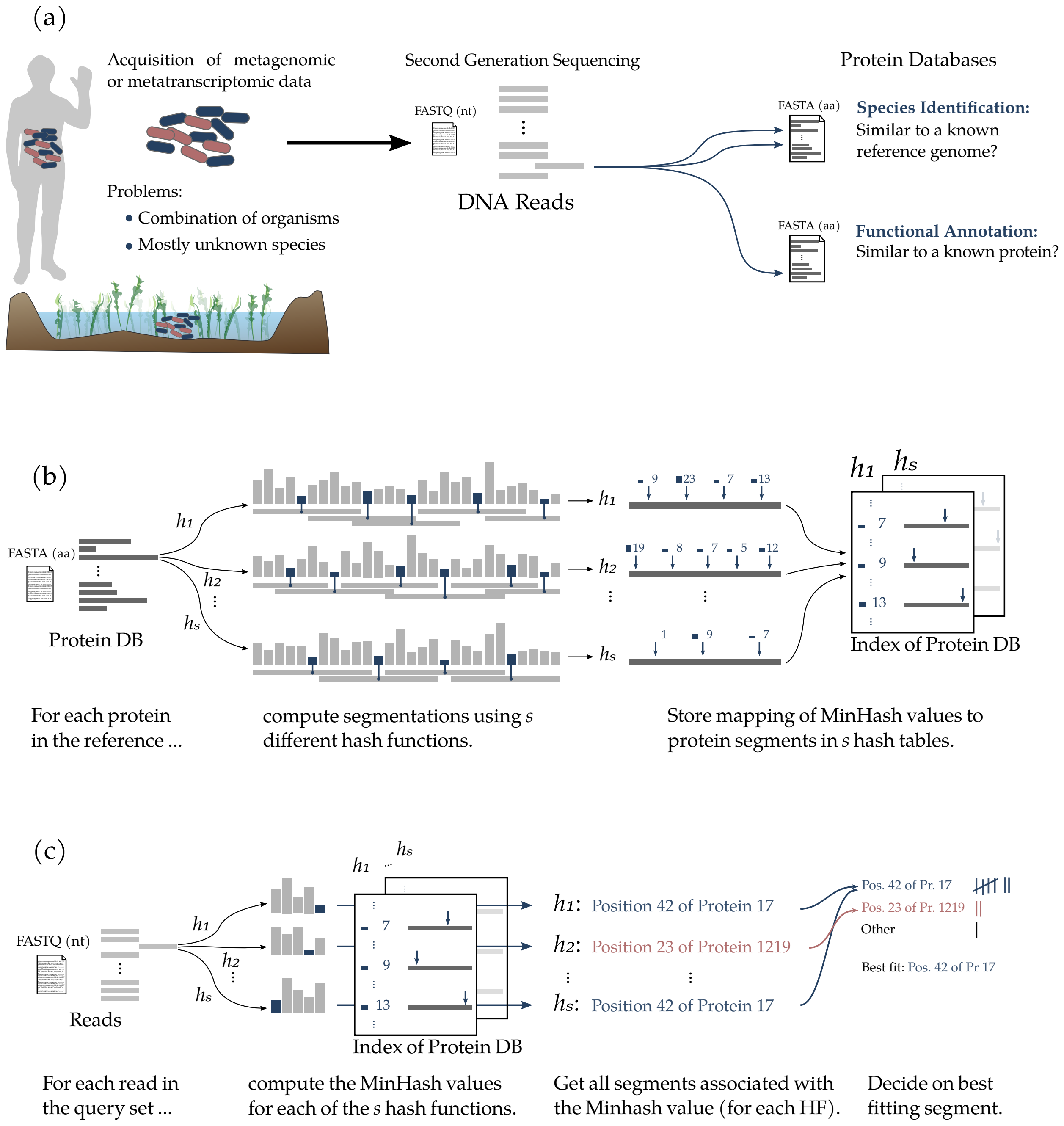

(a)
Acquisition of metagenomic or metatranscriptomic data
Problems:
• Combination of organisms
• Mostly unknown species
Second Generation Sequencing
FASTQ (nt)
DNA Reads
Protein Databases
FASTA (aa)
Species Identification:
Similar to a known reference genome?
FASTA (aa)
Functional Annotation:
Similar to a known protein?
(b)
FASTA (aa)
Protein DB
h1
h2
hs
9
23
7
13
19
8
7
5
12
1
9
7
h1
hs
7
9
13
Index of Protein DB
For each protein in the reference ...
compute segmentations using s different hash functions.
Store mapping of MinHash values to protein segments in s hash tables.
(c)
FASTQ (nt)
Reads
h1
h2
hs
h1
hs
7
9
13
Index of Protein DB
h1: Position 42 of Protein 17
h2: Position 23 of Protein 1219
hs: Position 42 of Protein 17
Pos. 42 of Pr. 17
Pos. 23 of Pr. 1219
Other
Best fit: Pos. 42 of Pr 17
For each read in the query set ...
compute the MinHash values for each of the s hash functions.
Get all segments associated with the Minhash value (for each HF).
Decide on best fitting segment.

Figure 6.4: Illustration of a metagenomic analysis. (a) DNA reads are procured from a diverse population of individuals, for example the human gut microbiome or lake water (left). For each read, similar sequences are searched for in protein databases (right). (b) Construction of a winnowed reference index with s hash functions. (c) Query process of the same index.

Pond graphic derived from "ReededPond r1" by Wikipedia user Jkwchui (CC BY-SA 3.0). Human outline derived from "Human outline by Linda Salzman Sagan (original artwork); Tompw (GIF version); Wikipedia user Holek (SVG) (CC BY-SA 3.0)."

unknown mixture of species. For the identification of alignment targets, we still need to compensate for sequencing errors and mutations. Since we compare DNA reads to a protein database, one of the input sequence sets—reads or protein reference—needs to be translated. Through performing a six frame translation of the reads (SFT; cf. Section 2.5.2) we already introduce a level of error tolerance. After translation, codons affected by silent mutations (cf. Section 2.2.1) and sequencing errors that behave the same way are already alleviated. For an additional level of tolerance, we translate input reads into a reduced amino acid alphabet like M15 (cf. Section 2.5.1, Table 2.3). Note that for SGS DNA reads, we expect a read length of approximately 100 to 200 bp, which relate to ∼30–66 amino acids.

By choosing a window size w for the winnowing that is equivalent to the read length, we can assure that the minimizer of the read is also guaranteed to be present in reference. We construct a hash table mapping MinHash values to sequence positions in the reference, comprising a sequence identifier (i.e. the number of the target protein in the reference file), a window starting position, and a segment length. When querying this table, we compute the MinHash value of the read without winnowing. Since the size w of the winnowing window and the read length align, we do not require winnowing the read using a window. Instead we can compute its MinHash value directly. All segments stored in the table for this minimal hash value are possible alignment candidates.

We propose using multiple winnowings with different hash functions to create overlapping segmentations with different minimizers to provide additional error tolerance as well as a better preselection of alignment targets. The location of the read within the target protein falls into the intersection of all s segments. Additionally, this architecture mitigates the influence of sequence deviations that are not compensated for by the translation to the Murphy alphabet. Consider, for example, reads $\mathcal{R} := \{ r_i \}, r_i \in \Sigma^{\ell}_{\text{M15}}$ with length $\ell = 30$, from which $\ell - 9 + 1 = 22$ 9-grams can be created. If a sequencing (substitution) error is present in such a read, up to 9 q-grams are affected. Using multiple winnowings generated using min-wise independent hash functions, leveraging the uniform distribution of minimizers within the read sequence, we can estimate the probability to select an affected q-gram as minimizer as:

$$\mathbb{P}(\text{unaffected } q\text{-gram selected}) = p^{\text{u}} \geq \left(1 - \frac{\#\,\text{errors} \cdot q}{\ell - q + 1}\right)$$

for each individual winnowing. Assuming a read length of 30 (amino acids), $q = 9$, and an error rate of $\epsilon = 0.01$ as a ballpark estimate, we see that we select an unaffected q-gram with a probability larger than ∼0.59. Through use of multiple winnowings, we can amplify the probability to select an unaffected q-gram at least once (given that $p^{\text{u}} > 0.5$). The expected number of times we select an unaffected q-gram using s different hash functions can be estimated as:

$$\text{Expected nr. of unaffected } q\text{-grams selected} \geq \mathbb{E}(X), \qquad X \sim BD(s, p^{\mathrm{u}})$$

By increasing the number of hash functions s, and therefore the number of trials for the binomial distribution, we can increase the probability to observe unaffected minimizers. Note that while we already implemented an early prototype of this approach, this estimate just serves illustrative purposes here and requires a more rigorous investigation for an actual implementation.

Figure 6.4 (b) and (c) illustrate the index creation and querying process respectively. The decision on the best alignment targets can be guided by several factors, like the number of segments supporting a certain protein or the total number of candidate sequences. Most notably, the segment length information obtained from the index can help to narrow down the exact location of the query sequence, by observing the overlap of segments supporting similar locations in the reference protein.

There are several desired properties of the segment length distribution that lend themselves to the index described above. A high proportion of segments with length w is beneficial for a smaller index size. Since each segment needs to be represented in the index, long segments result in less entries which in turn reduces the memory footprint and fill rate of the underlying hash tables. Short segments, while providing a more detailed fix on the position in the target sequence, can only do so at the expense of a higher (local) minimizer density and thus increased memory usage of the used data structures. Additionally, short segment sizes increase the number of reference positions expected to be associated with a single minimizer. This causes an increase in colliding entries in the hash tables used to realize the index, which can have detrimental effects on the performance.

Another effect causing collisions in the hash tables are repetitive regions within genomes or proteins. Minimizers from these regions introduce many colliding entries which can also harm the performance of downstream analysis. However, by introducing a modification to the winnowing algorithm we can alleviate this problem.

6.3 *Compressed Winnowing*

In addition to the two winnowing strategies described before—simple and robust winnowing (cf. Winnowing and Minimizers, p. 111f)—we introduce a third variant: compressed winnowing, which can extend segments beyond a length of w. Where simple winnowing would start a new segment for each hash in a run of equiminimal hashes, and robust winnowing would start a new segment every w steps, compressed winnowing condenses equiminimal runs into one segment. The main difference of compressed winnowing is that we do not start a new segment once we evict

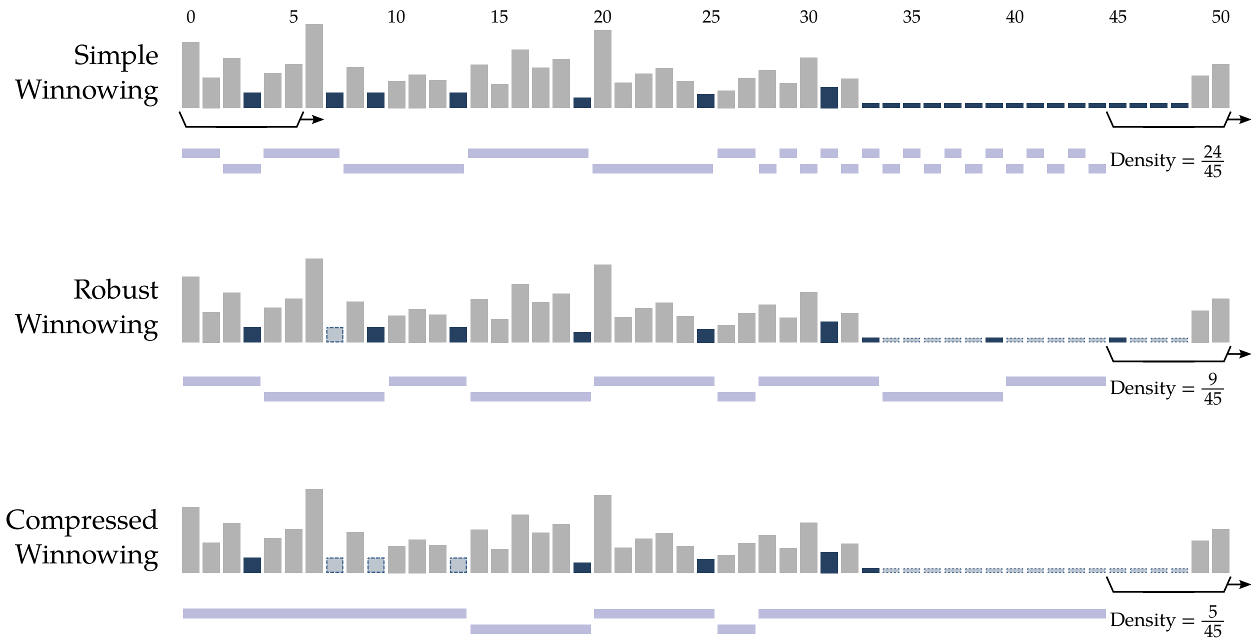


Figure 6.5: Comparison of the behavior of three winnowing variants with $w = 6$ on an equiminimal region (positions 33 - 48). Hash values are represented by vertical bars with their height representing the size of hash values and minimizers highlighted blue (■). Bars in light gray with a dashed blue outline (▨) represent skipped equiminimal hash values. Generated segments are denoted by horizontal purple bars. Simple winnowing reports each hash value in the equiminimal region as a new segment. Robust winnowing reports each w-th hash value, skipping 5. Compressed winnowing only reports one minimizer and skips all in the remaining run.

the minimizer that started it, as long as the subsequent windows possess the same MinHash value.

Definition 6.3.1 (Compressed Winnowed Sketch)**.** *Let h be a (ϵ-approximately) min-wise independent hash function and $G := \mathfrak{Q}(A, q)$ the q-gram sequence for a document A. We define the* ***compressed winnowed sketch*** *or the* ***compressed winnowed MinHash values*** *of A as:*

$$S_w^{\succ\!\prec}(G) := ((p_i, m_i))_{i=0}^{j-1} \quad 1 \leq j \leq |G|$$

where $m_i := h(g_i^m)$ is the i-th different MinHash value of the hash values of G. We refer to the corresponding minimizers as the ***compressed winnowed minimizers*** *of the document.*

An example showing the differences between simple, robust, and compressed winnowing is illustrated in Figure 6.5.

As a result of this approach, the compressed winnowing strategy does not possess a (non-trivial) lower bound on the number of segments:

Corollary 6.3.2 (Size of Compressed Winnowed Sketches)**.** *Let A be a document of length $|A|$, its q-gram sequence $\mathfrak{Q}(A, q)$ containing $\ell = |A| - q + 1$ q-grams, and a window size $w < \ell$ for winnowing. Then the compressed winnowed sketch $S_w^{\succ\!\prec}(A)$ contains between 1 and $\ell - w + 1$ MinHash values.*

Proof. Since each window receives a minimizer, a sketch always contains at least one entry.

Consider a sequence of ℓ identical hash values: Since with compressed winnowing, equiminimal q-grams do not create a new minimizer, there is no window position that changes the minimizers.

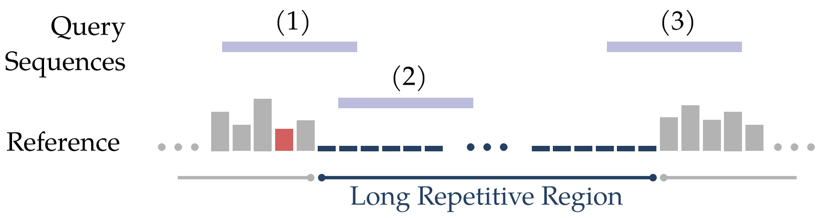


Figure 6.6: Given a reference (hash values symbolized by bars) that contains a long repetitive sequence (low blue bars) and a query sequence that overlaps the repetitive region there are two possible outcomes: Either the query sequence is completely contained within the repetitive sequence, as (2), or it covers both repetitive and normal sequence, as (1) and (3). In the first case, precise positioning information is not useful and might result in a high number of uninformative reference positions. This is also true for the second case. All sequences that touch a repetitive sequence can only be meaningfully interpreted using their neighboring windows.

A sketch contains at most one entry for each window position. Proof analogous to Corollary 5.9.2. □

Compressed winnowing sacrifices exact positioning information to achieve less segments and reduces redundant information. Depending on the use case, precise positioning information might not be required, for example within repetitive regions of a biological sequence. As an example, consider the repetitive amino acid sequence

AHHAADAHH**AHHAADAHH**AHHAADAHH**AHHAADAHH**AHHA

which is part of the proteome of *Plasmodium falciparum*.[5] The q-gram sequence derived from such a sequence, and in turn its hashes, follow a repeating pattern, which, depending on the choices of q and w, can results in one long equiminimal run. Furthermore, such repetitions in proteins are fairly common.[6]

However, for most sequence analysis tasks, the precise localization of a read within a repetitive region is neither helpful nor possible. Consider the segment reference described in Section 6.2, where the size of the query sequence (a read) is much smaller than that of its reference sequences (genomes, chromosomes, or proteins). The two cases that need to be distinguished, as illustrated in Figure 6.6, are reads that are completely contained within a repetitive region and those that reach into such a region. Reads that are completely contained within a repetitive region do not benefit from precise placement. Even worse, they might receive a large number of potential alignment targets within the repetitive region, as illustrated in Figure 6.7 which need to be resolved. Reads that only overlap a repetitive region on one side can still be placed on the correct side of the repetitive region.

Note that a compressed winnowing and a robust winnowing of a sequence only differ, if a run of windows longer than w has the same minimizer. Assuming the hash values were sampled from an ϵ-approximately min-wise independent hash function as well as a random sequence, this is unlikely. However, as shown above, biological sequences regularly contain subsequences that violate the latter assumption.

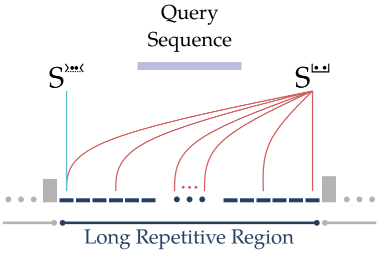


Figure 6.7: Segments obtained from a compressed winnowed sketch $S_w^{\succ\cdots\prec}$ and from a robust winnowed sketch $S_w^{\llcorner\cdot\cdot\lrcorner}$ for a repetitive region. The robust sketch contains one entry for each w positions within the repetitive region. Each segment is a potential target for alignments.

[5] Davies et al., "Repetitive Sequences in Malaria Parasite Proteins", 2017.

[6] Marcotte et al., "A Census of Protein Repeats", 1999.

6.4 Expected Segment Length Distribution

In this section we describe the expected distribution of segment lengths of compressed winnowed minimizers. We want to compute the expected distribution $\Psi = (\psi_k)_{k=1}^{\infty}$, where ψ_k is the probability to observe a segment of length k as part of a compressed winnowed sketch $S_w^{\rtimes\ltimes}$.

The expected distribution of segment lengths obtained from compressed winnowing is equivalent to the probability with which we observe segments of a certain length. For input sequences of length K, we can compute the probabilities to observe segments of length $k \in \{1, \dots, k\}$ independently. Note the we use 1-based indexing, both for the hash function codomain as well as for window indices, for a more convenient formulation. Furthermore, we will assume for our analysis that hash values are independently and identically chosen. While this does not precisely model our specific use case of biological q-grams—after a specific DNA q-gram there are only four possible successors, violating independence—it is a reasonable assumption for general hash values. For the case of hashed q-grams, however, this effect diminishes, if the hash function codomain $\mathcal{C}$ is smaller than the number of possible q-grams and the employed hash functions distribute well over $\mathcal{C}$.[7] In the evaluation of this method in the following sections we will show, that for practically employed q-gram spaces and hash functions our model is able to predict the behavior of hashed q-grams.

Consider an input sequence of hash values $X = (x_i)_{i \in \mathbb{Z}}$ with its elements x_i chosen independently and identically from $\mathcal{C} = \{1, \dots, C\}$. We denote the k-th window of length w, i.e. the subsequence of length w starting at position k, as $\mathfrak{X}_k := (x_i)_{i=k}^{k+w-1}$.

The MinHash value of the connected (inclusive) subsequence $X_a^b := X_{[a:b+1]}$, i.e. the smallest value contained in said subsequence, is denoted as

$$\underline{X}_a^b = \min \{ x_i \mid a \leq i \leq b \} . \tag{6.1}$$

A window minimum, i.e. the smallest value in a window of size w starting at position k, is denoted as $\underline{\mathfrak{X}}_k$. For a sequence of consecutive windows, ranging from position a to position b inclusively, we denote the minimum of this sequence as

$$\underline{\mathfrak{X}}_{a \dots b} . \tag{6.2}$$

Note, that for $a < b$

$$\underline{\mathfrak{X}}_{a \dots b} = \underline{X}_a^{b+w} .$$

Throughout this section, we will also use **max**ima instead of *min*ima for a more convenient formulation. With respect to the definitions for minima give above, we denote the subsequence maximum as $\overline{X}_a^b$, the window maximum as $\overline{\mathfrak{X}}_k$, and the consecutive window maximum as $\overline{\mathfrak{X}}_{a \dots b}$. All of these describe the maximal hash value within their respective domain. A complete compilation of the notation for minima and maxima can be found in Table 6.1.

[7] Consider a hash value x that can be generated by two different colliding q-grams: $h(g_0) = h(g_1) = x$. Since both the preceding and succeeding q-grams of g_0 and g_1 are different, their resulting hash values are different as well (with high probability), leaving more the four possible options. If $|\mathcal{C}| \ll 4^q$ holds there are many possible q-grams g_i with $h(g_i) = x$ and thus q-gram hashes tend to behave more like i.i.d. hash values.

Symbol	Description
C	Size of the hash function codomain $\mathcal{C} := \{1, \dots, C\}$ used to compute Ψ.
m	A specific MinHash value $m \in \mathcal{C}$.
K	Maximum segment length used to compute Ψ.
$\Psi^{C}_{w,K}$	Expected distribution of segment lengths computed with a codomain of size C, with window size w for segments up to length K.
$\mathfrak{X}_k$	The k-th length w window of X: $(x_i)_{i=k}^{k+w-1}, x_i \in X$.
$\underline{X}_a^b$	*Min*imum of the subsequence containing positions a to b (inclusive).
$\underline{\mathfrak{X}}_k$	*Min*imum of the window starting at position a. $\underline{\mathfrak{X}}_k = \underline{X}_k^{k+w-1}$.
$\underline{\mathfrak{X}}_{a\dots b}$	*Min*imum of the windows starting at the positions a through b. $\underline{\mathfrak{X}}_{a\dots b} = \underline{X}_a^{b+w-1}$.
$\overline{X}_a^b$	**Max**imum of the subsequence containing positions a to b (inclusive).
$\overline{\mathfrak{X}}_k$	**Max**imum of the window starting at position a. $\overline{\mathfrak{X}}_k = \overline{X}_k^{k+w-1}$.
$\overline{\mathfrak{X}}_{a\dots b}$	**Max**imum of the windows starting at the positions a through b. $\overline{\mathfrak{X}}_{a\dots b} = \overline{X}_a^{b+w-1}$.
$\underline{p}_{m,k}$	Probability for a segment of length at least k with a specific *min*imum $m \in \{1, \dots, C\}$.
$\underline{p}^l_{m,k}, \underline{p}^r_{m,k}, \underline{p}^{lr}_{m,k}$	Probability for a segment of length at least k with a specific *min*inimum $m \in \{1, \dots, C\}$ that is bounded on the left, on the right, or on both sides.
$\underline{p}_k$	Probability for a segment with length at least k with any *min*imum.
$\underline{p}^l_k, \underline{p}^r_k, \underline{p}^r_k$	Probability for a segment of length at least k with any *min*imum that is bounded on the left, on the right, or on both sides.
$\overline{p}_{m,k}$	Probability for a segment of length at least k with a specific **max**imum $m \in \{1, \dots, C\}$.
$\overline{p}^l_{m,k}, \overline{p}^r_{m,k}, \overline{p}^{lr}_{m,k}$	Probability for a segment of length at least k with a specific **max**imum $m \in \{1, \dots, C\}$ that is bounded on the left, on the right, or on both sides.
$\overline{p}_k$	Probability for a segment with length at least k with any **max**imum.
$\overline{p}^l_k, \overline{p}^r_k, \overline{p}^r_k$	Probability for a segment of length at least k with any **max**imum that is bounded on the left, on the right, or on both sides.

Table 6.1: Symbols used in this section.

We write $\underline{p}_k := \mathbb{P}(\underline{\mathfrak{X}}_{1\dots k})$ as the probability, that any minimizer covers (at least) the windows 1 to k (which is equivalent to a segment of length at least k). For a given hash function codomain $\mathcal{C}$, we can compute $\underline{p}_k$ from the probabilities $\underline{p}_{m,k}$ that there exists a segment of length at least k with a specific minimum $m \in \mathcal{C}$.

6.4.1 Segment Length Distribution

First we describe the basic properties and behavior of MinHash values, i.e. how a minimum can change between two consecutive windows. Since a new segment can only be started with a changed window minimum, there are two options: the new hash value is either smaller, or larger than the last one.

Lemma 6.4.1. *A minimum m_{i+1} can only be larger than its predecessor m_i if the hash value x_i is the minimizer of window $\mathfrak{X}_i$ and is pushed out of the window (and no other copies of that minimizer remain in the window).*

Proof. Consider the windows $\mathfrak{X}_i = (x_i, \dots, x_{i+w-1})$ and $\mathfrak{X}_{i+1} = (x_{i+1}, \dots, x_{i+w})$. After the move, x_i is no longer part of the active window, but x_{i+w} entered the window.

Assume there exists a new, larger minimum $\min \mathfrak{X}_{i+1} > \min \mathfrak{X}_i$ and $\min \mathfrak{X}_i \neq x_i$.

$$\begin{aligned} \Rightarrow \min \mathfrak{X}_i = \min(\mathfrak{X}_i \setminus x_i) < \min \mathfrak{X}_{i+1} = \min((\mathfrak{X}_i \setminus x_i) \cup x_{i+w}) \\ \Rightarrow \exists x_j \in \mathfrak{X}_i \setminus x_i < \min((\mathfrak{X}_i \setminus x_i) \cup x_{i+w}) \end{aligned} \tag{6.3}$$

Hence, by adding x_{i+w} the minimum has to increase. However, adding another element can only further decrease the minimum, not increase it. $\lightning$

Hence, if $\min \mathfrak{X}_{i+1} > \min \mathfrak{X}_i$, this can only be due to the minimizer of $\mathfrak{X}_i$ being x_i, which was pushed out. □

Lemma 6.4.2. *A minimum m_{i+1} can only be smaller than its predecessor m_i if a new, smaller minimizer x_{i+w} enters the window.*

Proof. Consider the windows $\mathfrak{X}_i = (x_i, \dots, x_{i+w-1})$ and $\mathfrak{X}_{i+1} = (x_{i+1}, \dots, x_{i+w})$. After the move x_i is no longer part of the active window, but x_{i+w} entered the window.

Assume there exists a new smaller minimum $\min \mathfrak{X}_{i+1} < \min \mathfrak{X}_i$ and $\min \mathfrak{X}_{i+1} \neq x_{i+w}$ (it is not the value just pushed in).

$$\begin{aligned} \Rightarrow \min(\mathfrak{X}_{i+1} \setminus x_{i+w}) < \min \mathfrak{X}_i \\ \Rightarrow \exists x_j \in \mathfrak{X}_{i+1} \setminus x_{i+w} < \min \mathfrak{X}_i \end{aligned} \tag{6.4}$$

However, since $\mathfrak{X}_{i+1} \setminus x_{i+w} \subset \mathfrak{X}_i$, any minimum of $\mathfrak{X}_{i+1} \setminus x_{i+w}$ would have already been a minimizer of $\mathfrak{X}_i$. $\lightning$

Hence, a new, smaller minimizer has to be the new value x_{i+w}. □

As noted above, we will work with maxima instead of minima throughout this description. While able to express the same probabilities, this allows for a more concise formulation of the following equations. Instead of $\underline{p}_{m,k}$ for a minimum m, we compute $\overline{p}_{C-m+1,k}$ for a maximum $C - m + 1$, which have the same probability.

Lemma 6.4.3 (Interchangeability of Minimum and Maximum Probability)**.** *For a subsequence containing positions a through b, a hash codomain of size C, and specific hash value $m \in \{1, \dots, C\}$,*

$$\mathbb{P}(\underline{X}_a^b = m) = \mathbb{P}(\overline{X}_a^b = C - m + 1)$$

holds, i.e. the probability that m is the minimum the segment is equal to the probability that $C - m + 1$ is its maximum.

Proof. To maintain a *min*imum m, we rely on the probability to select only values above or equal to m, as well as values truly larger than m. This probability depends on the number of values $m' \geq m, m' \in \mathbb{N}$. Since

$$|\{m, \dots, C\}| = C - m + 1 = |\{1, \dots, C - m + 1\}|$$

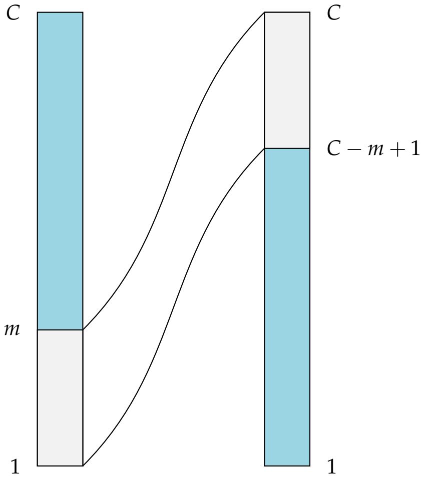


Figure 6.8: Two partitions of the same hash function codomain $\mathcal{C}$. To retain a *min*imum m, in the left codomain, only values from the teal part may be chosen. On the right side, we inverted the orientaion of values to choose values retaining the **max**imum $C - m + 1$ with the same probability as a *min*imum in the left codomain.

the probability to randomly choose a value larger than m is equal to choosing a value smaller than $C - m + 1$. Intuitively, we invert the orientation of permissible values as shown in Figure 6.8. □

Since for every segment of length k with minimum m, there is an equally probable maximum of length k with maximum $C - m + 1$, $\underline{p}_k = \overline{p}_k$ holds.

The probability that a consecutive subsequence of windows has the maximum m is:

Lemma 6.4.4 (Segment Maximum Probability)**.** *For a fixed hash value $m \in \{1, \ldots, C\}$, and window positions $a, b \in \mathbb{N}$ with $a < b$, the probability that m is the largest value in a segment covering positions a to b is*

$$\mathbb{P}(\overline{X}_a^b = m) = \frac{m^{b-a+1} - (m-1)^{b-a+1}}{C^{b-a+1}}$$

Proof. For m to be the largest value in a segment of $b - a + 1$ values it needs to contain m at least once and all other values have to be smaller than m. We count all events in which all values are at most m and subtract the number of events that contain only values smaller than m (but not m itself). The probability to select a sequence of hash values, that satisfies the first, but not the second condition is:

$$\begin{aligned}
\mathbb{P}(\overline{X}_a^b = m) &= \mathbb{P}(\text{all values are at most } m) & &-\mathbb{P}(\text{all values are at most } m-1) \\
&= \mathbb{P}\left(\{\, x_i \in \{1, \ldots, m\} \mid a \le i \le b \,\}\right) & &-\mathbb{P}\left(\{\, x_i \in \{1, \ldots, m-1\} \mid a \le i \le b \,\}\right) \\
&= \left(\frac{m}{C}\right)^{b-a+1} & &-\left(\frac{m-1}{C}\right)^{b-a+1} \\
&= \frac{m^{b-a+1} - (m-1)^{b-a+1}}{C^{b-a+1}}
\end{aligned}$$

□

Since the actual positions a and b do not affect the probability $\mathbb{P}(\overline{X}_a^b = m)$, segments can be translated without affecting the probability for the event.

Observation 6.4.5 (Translation Invariance of $\mathbb{P}(\overline{X}_a^b = m)$ and $\mathbb{P}(\overline{\mathfrak{X}}_{a\ldots b})$)**.** *For any offset $r \in \mathbb{Z}$, we can translate a segment and maintain the same probabilities for*

$$\mathbb{P}(\overline{X}_a^b = m) = \mathbb{P}(\overline{X}_{a+r}^{b+r} = m)$$

and

$$\mathbb{P}(\overline{\mathfrak{X}}_{a\ldots b}) = \mathbb{P}(\overline{\mathfrak{X}}_{a+r\ldots b+r})$$

as long as $b - a$ remains unchanged.

For a specific maximum m, we compute the probability to maintain this maximum throughout several windows.

Theorem 6.4.6. *The probability for a single segment with maximum $m \in \{1, \dots, C\}$ to cover at least k windows, with $C, w, k \in \mathbb{N}$, $k' := \min(w, k-1)$, is*

$$\overline{p}_{m,k} = \mathbb{P}(\overline{\mathfrak{X}}_{1\dots k} = m) = \frac{m^w(m-1)^{k'} - m^{k'}(m-1)^w}{C^{w+k-1}} + \sum_{j \in 1,\dots,k'} \left(\frac{(m-1)^{j-1}}{C^j} \cdot p_{m,k-j} \right) \tag{6.5}$$

Proof. For every sample in $\overline{p}_{m,k}$ we have $\overline{\mathfrak{X}}_1 = \max(x_1, \dots, x_w) = m$. Observing the first occurrence of m in the sequence $(x_1, \dots, x_w)$, we can use the law of total probability. Given that the first occurrence of m is at position j, we get

$$\overline{p}_{m,k} = \sum_{j \in \{1,\dots,w\}} \mathbb{P}(\overline{\mathfrak{X}}_{1\dots k} = m \text{ and } (x_1, \dots, x_{j-1}) < m \text{ and } x_j = m) \tag{6.6}$$

Since $j < w$, the first occurrence of m already covers the first j of k windows. Hence, we can reduce the above statement to

$$\overline{p}_{m,k} = \sum_{j \in \{1,\dots,w\}} \mathbb{P}(\overline{\mathfrak{X}}_{j+1\dots k} = m \text{ and } (x_1, \dots, x_{j-1}) < m \text{ and } x_j = m) \tag{6.7}$$

We now consider the cases $k > w$, i.e. segments that exceed the window length through repeated maximum values, and $k \leq w$, segments that can be covered by the first maximizer. For $k > w$, the events $[\overline{\mathfrak{X}}_{j+1\dots k} = m]$ and $[(x_1, \dots, x_{j-1}) < m, x_j = m]$ are independent. Using translation invariance, we obtain

$$\begin{aligned} \overline{p}_{m,k} &= \sum_{j \in \{1,\dots,w\}} ((x_1, \dots, x_{j-1}) < m \text{ and } x_j = m) \cdot \mathbb{P}(\overline{\mathfrak{X}}_{j+1\dots k} = m) \\ &= \sum_{j \in \{1,\dots,w\}} \frac{(m-1)^{j-1} \cdot 1}{C^j} \cdot \overline{p}_{m,k-j} \end{aligned} \tag{6.8}$$

For $k \leq w$, k windows can be covered by the first maximum at position $j \in \{k, \dots, w\} \subseteq \{1, \dots, w\}$. Aggregating the probability for all $(w - k + 1)$ possible values of j and applying Lemma 6.4.4, we obtain

$$\begin{aligned} \overline{p}_{m,k} &= \mathbb{P}(x_1, \dots, x_{k-1} < \overline{X}_k^w = m \geq x_w, \dots, x_{w+k-1}) + \sum_{j \in \{1,\dots,k-1\}} \frac{(m-1)^{j-1} \cdot 1}{C^j} \cdot \overline{p}_{m,k-j} \\ &= \frac{(m-1)^{k-1}}{C^{k-1}} \cdot \frac{m^{w-k+1} - (m-1)^{w-k+1}}{C^{w-k+1}} \cdot \frac{m^{k-1}}{C^{k-1}} + \sum_{j \in \{1,\dots,k-1\}} \frac{(m-1)^{j-1} \cdot 1}{C^j} \cdot \overline{p}_{m,k-j} \\ &= \frac{m^w(m-1)^{k-1} - m^{k-1}(m-1)^k}{C^{k+w-1}} + \sum_{j \in \{1,\dots,k-1\}} \frac{(m-1)^{j-1} \cdot 1}{C^j} \cdot \overline{p}_{m,k-j} \end{aligned} \tag{6.9}$$

The first summand of Equation 6.5 collapses to 0 for $w = k - 1$. By using $k' = \min(w, k-1)$ instead of k, we can unify the cases $k > w$ and $k \leq w$. □

Using the probability $\overline{p}_{m,k}$, we can derive the probabilities to observe bounded segments. We denote the probability for the case that k is the right segment boundary as $\overline{p}^r_{m,k}$, the left segment boundary as $\overline{p}^{l}_{m,k}$, and the case that the segment is bounded on both sides as $\overline{p}^{lr}_{m,k}$.

Lemma 6.4.7. *For segments of length $k \in \mathbb{N}$, the following connections hold:*

1. $\overline{p}^{r}_{m,k} = \overline{p}^{l}_{m,k}$ *The probabilities that a segment can be extended in one direction are symmetric.*

2. $\overline{p}_{m,k} = \overline{p}_{m,k+1} + \overline{p}^{r}_{m,k}$ *The probability for a segment with length k is the sum of the probabilities to observe a segment that can be extended to the right and the probability that the segment ends with k, i.e. is right-bounded.*

3. $\overline{p}^{l}_{m,k} = \overline{p}^{l}_{m,k+1} + \overline{p}^{lr}_{m,k}$ *The probability to observe a left-bounded segment of length k is the sum of probabilities that a left-bounded segment can be extended and the probability that it is bounded on both sides.*

Proof. For 1, we reverse the hash value sequence by bijectively mapping $(x_i)_i$ to $(x_i)_{\pi(i)}$ where $\pi(i) = k + 1 - i$.

For 2, we partition all events leading to $\overline{p}_{m,k}$ by observing the two possible outcomes for $\overline{\mathfrak{X}}_{k+1}$ and exploit translation invariance (Observation 6.4.5):

$$\begin{aligned}\overline{p}_{m,k} &= \mathbb{P}(\overline{\mathfrak{X}}_{1\dots k} = m = \overline{\mathfrak{X}}_{k+1}) + \mathbb{P}(\overline{\mathfrak{X}}_{1\dots k} = m \neq \overline{\mathfrak{X}}_{k+1}) \\ &= \overline{p}_{m,k+1} + \overline{p}^{r}_{m,k}\end{aligned}$$

For 3, we analogously get:

$$\begin{aligned}\overline{p}^{l}_{m,k} &= \mathbb{P}(\overline{\mathfrak{X}}_0 \neq \overline{\mathfrak{X}}_{1\dots k} = m = \overline{\mathfrak{X}}_{k+1}) + \mathbb{P}(\overline{\mathfrak{X}}_0 \neq \overline{\mathfrak{X}}_{1\dots k} = m \neq \overline{\mathfrak{X}}_{k+1}) \\ &= \overline{p}_{m,k+1} + \overline{p}^{r}_{m,k}\end{aligned}$$

□

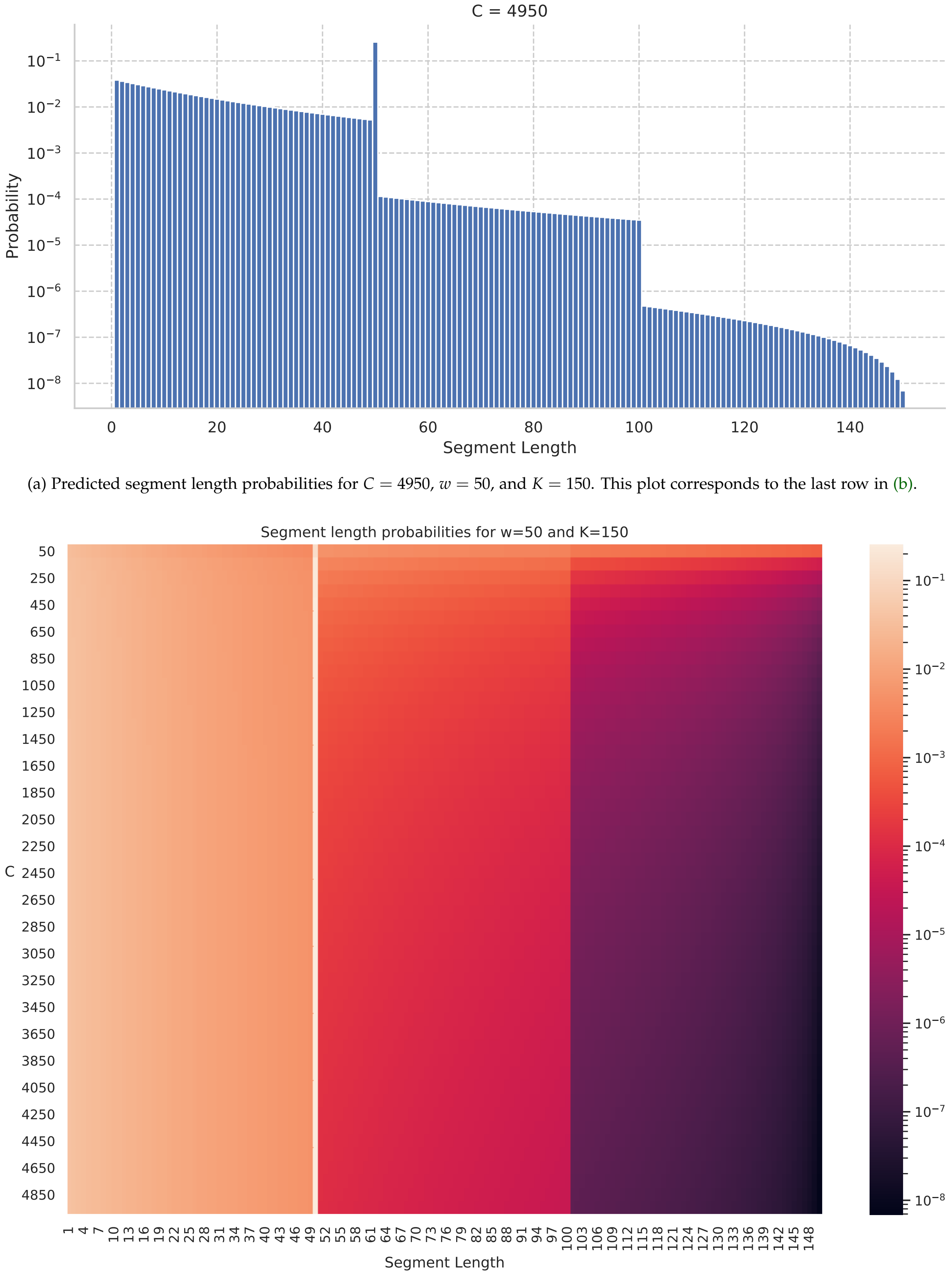
Using the relations described in Lemma 6.4.7, we can express the probability $\underline{p}^{lr}_{m,k}$ to observe a segment of length k with *min*imum m using the probability for the unbounded case $\overline{p}_{m,k}$:

$$\begin{aligned}\underline{p}^{lr}_{m,k} &= \overline{p}^{lr}_{C-m+1,k} \\ &= \overline{p}^{l}_{C-m+1,k} - \overline{p}^{l}_{C-m+1,k+1} \\ &= (\overline{p}_{C-m+1,k} - \overline{p}_{C-m+1,k+1}) - (\overline{p}_{C-m+1,k+1} - \overline{p}_{C-m+1,k+2})\end{aligned} \tag{6.10}$$

Using this, we can compute the distribution of segment lengths obtained by compressed winnowing for a given hash function codomain $\mathcal{C}$.

6.4.2 Implementation and Evaluation

We implemented a Python program to compute the probability distribution for segment lengths obtained by compressed winnowing for a given sequence length K and hash function codomain



(a) Predicted segment length probabilities for $C = 4950$, $w = 50$, and $K = 150$. This plot corresponds to the last row in (b).

(b) Distribution heatmap of multiple values for C. Each line corresponds to a distribution as illustrated in (a) for $C = 4950$.

Figure 6.9: Expected distributions $\Psi^{C}_{w,K}$ for small hash function codomains and fixed values $w = 50$ and $K = 150$. Both plots are in logarithmic scale, with the probability to observe a segment of length w at ~ 0.25.

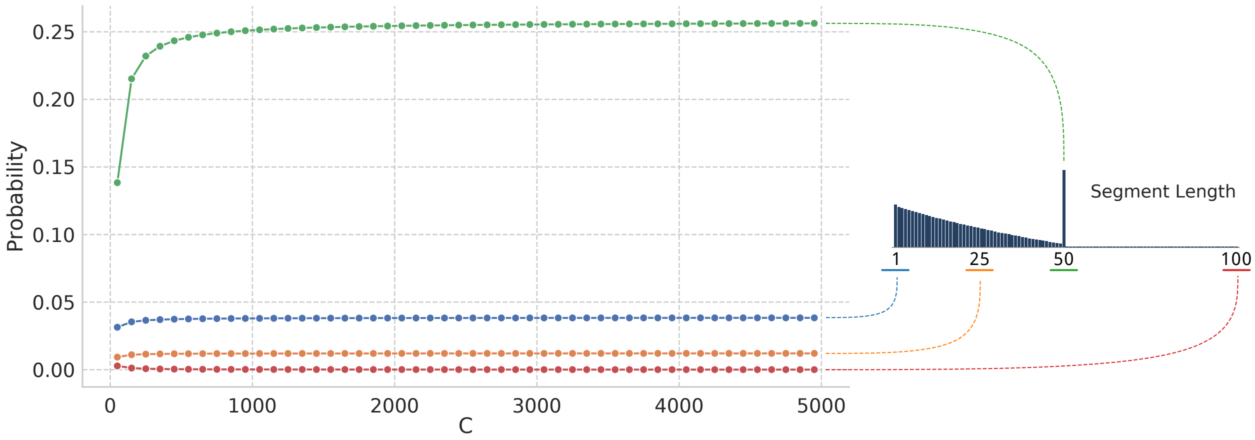


Figure 6.10: Development of four selected segment lengths for increasing values of C. Each line corresponds to a column in the heatmap in Figure 6.9(b), but shown in linear scale. For better visualization dashed lines connect the measurement curves to a visualization of a linearly scaled segment length distribution.

$\mathcal{C} = \{1, \ldots, C\}$. The code is available as part of the evaluation workflow described in Section 6.5. We are ultimately interested in the probability distribution $\Psi^C_{w,K} = (\psi^C_k)^K_{i=k} \cup \psi^C_{>K}$, where

$$\psi^C_k := \frac{\underline{p}^{l,r}_k}{\underline{p}^l_1}$$

is the probability to observe a segment of length exactly k when extending segments (that are left-bounded) to the right. Using the probability $\overline{p}_k$ to observe an unbounded segment of length at least k we can compute $\underline{p}^{lr}_k$ as follows:

$$\begin{aligned}
\underline{p}^{lr}_k &= \underline{p}^{lr}_k \\
&= \underline{p}^l_k - \underline{p}^l_{k+1} \\
&= (\underline{p}_k - \underline{p}_{k+1}) - (\underline{p}_{k+1} - \underline{p}_{k+2}) \\
&= (\overline{p}_k - \overline{p}_{k+1}) - (\overline{p}_{k+1} - \overline{p}_{k+2}) \\
&= (\sum_{m \in \{1,\ldots,C\}} \overline{p}_{C-m+1,k} - \sum_{m \in \{1,\ldots,C\}} \overline{p}_{C-m+1,k+1}) - (\sum_{m \in \{1,\ldots,C\}} \overline{p}_{C-m+1,k+1} - \sum_{m \in \{1,\ldots,C\}} \overline{p}_{C-m+1,k+2})
\end{aligned} \tag{6.11}$$

We compute an $C \times K$ matrix P containing all values $\overline{p}_{C-m+1,k}$. Then, for each value $k \in \{1, \ldots, K\}$, we compute the probability $\underline{p}^{lr}_k$ using P as a dynamic programming matrix to access already computed values.

Since P contains K entries for each possible hash value $m \in \mathcal{C}$, the size of P grows fast. While realistic sizes for C and K can currently not be computed with this approach, we observed that even for small hash function codomains the computed distribution is consistent with distributions computed for simulated and reference genomes. For the same reason we had to limit K to values that fall significantly below the sequence lengths present in reference genomes.[8]

Figure 6.9 illustrates the overall shape of the computed segment length distributions. The most probable case by far are segments of

[8] For reference, computing the expected distribution $\Psi^{2\,000\,000}_{100,300}$ required 89.4 hours, using one processor core on the system described on page 79.

length w with $\psi_w^C \approx 0.25$, which are caused by small minimizers that enter the window and remain dominant for the whole w positions before leaving the window. Segment lengths between 1 and $w-1$ monotonously fall. For segment lengths above w this trend continues until $2w$, albeit with an overall probability that is more than an order of magnitude smaller than for $w-1$. These segment lengths are still achievable with only one additional (equiminimal) minimizer. Beyond segment length $2w$, we require at least three equiminimal minimizers, which coincides with further reduced probabilities for such segments. Finally, we can observe in Figure 6.9(b) that while the overall shape of the distribution remains identical for increasing values of C, the probability for segments of length $> w$ decreases. This is expected, since these segments rely on the fact that one or more minimizer with the same hash value occur less than w positions apart from each other. For increasing hash function codomains C, this probability drastically reduces due to the increased number of possible hash values.

This can be observed in Figure 6.10, which depicts four columns from the heatmap shown in Figure 6.9(b). With increasing values of C, the probability for segments with length 100 decreases, while the probability for the lengths 1, 25, and 50 increase. Note that the lower total probability for small values of C is caused by the fact, that we restrict $\Psi_{w,K}^C = (\psi_{w,k}^C)_{k=1}^{\infty}$ to a constant text size $\Psi_{w,K}^C = (\psi_{w,k}^C)_{k=1}^{K}$ as mentioned above. To better show the behavior of the predicted distributions, we omitted the probability

$$\psi_{>K}^C := \frac{\underline{p}_{K+1}^l}{\underline{p}_1^l}$$

in these plots, which accumulates the probability mass for all segment lengths larger than K. This effect is further amplified by the fact that for the purpose of computation we restrict K to be smaller than the length of most biological sequences of interest. For our empirical evaluations in Section 6.5, we aggregate the probability for all segment lengths beyond w (i.e. all $w < k \leq K$ and $\psi_{>K}^C$), to incorporate this information into our prediction.

For increasing values of C, differences in subsequent distributions reduce, due to the decreasing probabilities to observe segments longer than w. This is illustrated in Figure 6.11 for hash function codomains from the range $\{50, 150, \ldots, 29950\}$ using a step size of 100. Each point symbolizes the sum of squared differences

$$\sum_{k=1}^{K} (\psi_{w,k}^C - \psi_{w,k}^{C'})^2$$

for $C \in \{50, 150, \ldots, 29850\}$ and $C' := C + 100$. Differences between successive experiments with a linear increase in hash function codomain size rapidly decrease. This is due to the fact that with increasing C the total probability mass beyond $w = 50$ is reduced and has increasingly little influence on the total distribution.

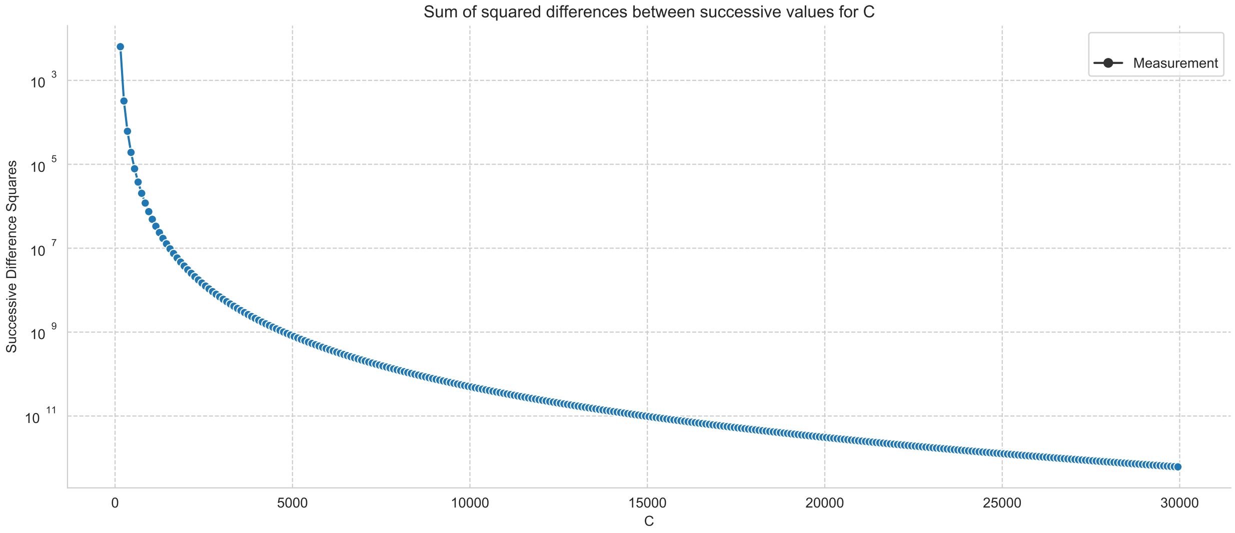


Figure 6.11: Development of sums of squared differences between successive values of C in logarithmic scale. The measurement point at position C denotes the sum of squared differences between C and $C - 100$.

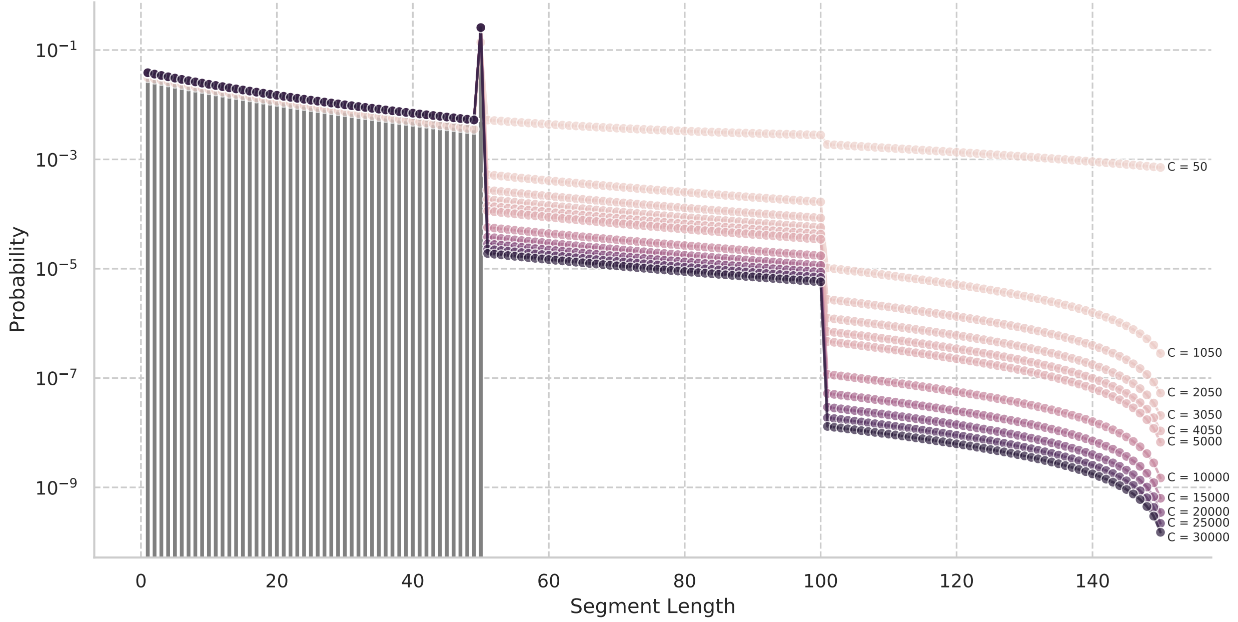
Finally, Figure 6.12 shows the differences between $\Psi_{w,K}^{C}$ for selected values of C and two distributions computed from 10 random DNA sequences of length 100 000 000 bp using 31-grams (without canonization) hashed with twisted tabulation hashing (see p. 46). For the second distribution (shown in Figure 6.12(b)), we used hash functions

$$h(g) = h'(g) \bmod 30\,011$$

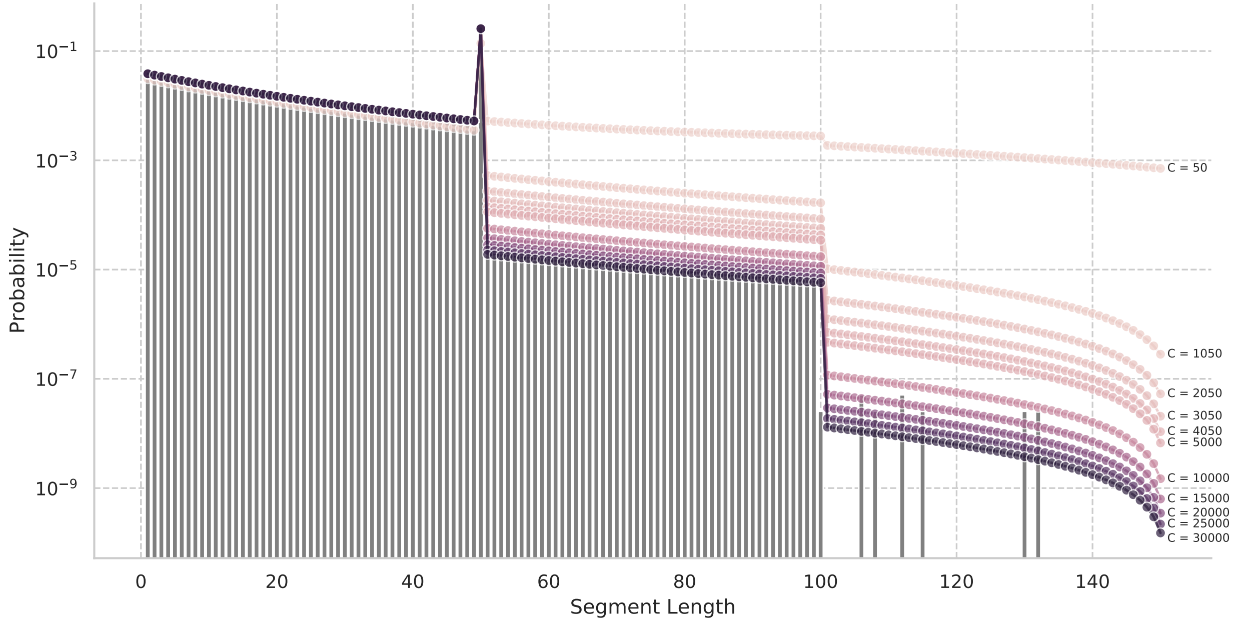
where g is a 31-gram, h' a hash function from the 64-bit twisted tabulation family, and 30 011 is a prime number close to the largest codomain size $C = 30\,000$ used for simulation.

While Figure 6.12(a) lacks segments with length $> w$, in Figure 6.12(b) the simulated and empiric distribution closely resemble each other. This is due to the fact that these segment lengths are exceedingly rare for random DNA sequences and large values of C. The limited sample that is our 10^9 simulated bases does not contain enough cases to encounter these rare events. Similarly, in Figure 6.12(b), only very few lengths ≥ 100 were observed. The empirically computed values for segment lengths ≥ 115 with a probability of $\sim 10^{-8}$ correspond to a single segment of this length in the dataset. Therefore, their probabilities exceeds the predicted probabilities.

For hash function codomains as small as $C = 10\,000$, the distribution $\Psi_{w,K}^{C}$ closely resembles the distribution computed from the simulated DNA sequences. However, as we mentioned before, neither uniformly independently distributed keys, nor completely random DNA sequences with a GC-content of 0.5 are realistic scenarios for biological sequences. In the following section we analyze different sequence types as well as different combinations of hash functions and canonization functions.



(a) Predicted segment length probabilities compared to an empiric distribution using $\mathcal{C} = [2^{64}]$.



(b) Predicted segment length probabilities compared to an empiric distribution using $\mathcal{C} = [30\,011]$.

Figure 6.12: Comparison of segment length distributions $\Psi^{C}_{50,150}$ with empirically computed distributions. The empiric distributions (gray bars), were computed for 10 DNA sequences with 100 000 000 bases chosen uniformly at random from Σ_{DNA} and 31-grams. Distributions for different hash function codomain size C are denoted as colored lines, omitting $\psi^{C}_{>K}$ to increase the readability of the plots. For (a), q-grams were hashed with 64-bit twisted tabulation hashing, while for (b) hash values were limited to $[30\,011]$ using modulo.

Genome	Length (in bp)	GC-Content
Myxococcus xanthus	9 139 763	≈ 70%
Homo sapiens HG38	3 209 286 105	≈ 40%
Humulus lupulus (Hops)	1 812 501 705	≈ 40%
Plasmodium falciparum (Malaria)	23 268 702	≈ 20%

Table 6.2: Analyzed reference and draft genomes, sorted by GC-content. For sources please refer to p. 79.

6.5 Empirical Analysis of Segment Length Distribution

To evaluate our computation of the expected segment length distribution $\Psi^{C}_{w,K}$, we compared it to empirically computed segment length distributions. While uniformly randomly chosen hash values as used above ease analysis, biological sequences seldomly behave like this. Two of the main differing aspects are sequence entropy, i.e. how (un-)repetitive the sequence is, and GC-content.[9]

To assess these effects, we analyzed simulated genomes with different sequence properties. These random DNA sequences comprised 100 000 000 bases and were simulated with different levels of GC-content. If not otherwise noted, the GC-content of simulated sequences is 0.5. We generated FASTA files comprising a single sequence with all bases chosen at random, either uniformly or with the specified GC-content.

Additionally, we analyzed a collection of reference genomes (see Table 6.2) selected to cover a great range of parameters, including different length, GC-content, and repetitiveness.

Another influencing factor are the employed hash functions and canonization strategies. We analyzed the following hash functions to cover a broad spectrum of function types: To provide a base level using trivial hash functions, we evaluated plain 2-bit encoding of DNA sequences and swap mixing, i.e. swapping the higher and lower half of a 2-bit encoded sequence, as illustrated in Figure 6.13. As an example for a general purpose hash function, we evaluated Murmur Hash 3 (mmh3),[10] specifically its Rust implementation.[11] For integer hash functions, we evaluated functions from the invertible multiplicative hash function family described in Section 4.4, the $\mathcal{H}^{\text{lin}}$ hash function family, and simple as well as twisted tabulation hashing.

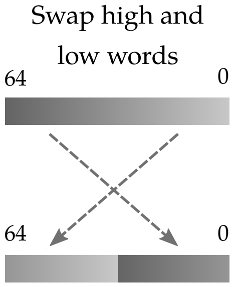


Figure 6.13: During swap mixing, the low and high 32-bit of a 64-bit integer are exchanged. A Rust implementation of swap mixing can be found in Listing 5 (p. 238).

Since using canonical q-grams as described in Section 2.5.3 approximately halves the number of possible hash values, this poses the question: Does this influence the efficiency of MinHashing strategies and of winnowing in particular? Usually, canonical q-grams are computed as the minimum of g and $\bar{g}$, for $g \in \mathfrak{Q}$, using the κ function (see Equation 2.12, p. 32). In this section, we refer to these as min-canonical q-grams. To analyze how this minimization influences winnowing techniques, we also analyzed max-canonical q-grams, using the function:

$$\kappa^{\max}(g) := \begin{cases} g & \text{if } g > \bar{g} \\ \bar{g} & \text{else} \end{cases} \tag{6.12}$$

[9] Cf. Structure of DNA and RNA Sequences p. 10

[10] Appleby, *Murmurhash3*, 2016.

[11] `https://crates.io/crates/murmur3`

We discern non-canonical q-grams, which are not canonized, min-canonical, and max-canonical, which use the smaller and larger q-gram of g and $\bar{g}$ respectively.

For q and w, we chose ranges of values that commonly occur in bioinformatics applications: We analyzed q-grams with lengths between 11 and 31 bases. Shorter q-grams are of limited value for MinHashing approaches, since they are likely to repeat themselves frequently. On the other end of the spectrum, 31-grams can still be encoded in 64-bit integer numbers using 2-bit encoding. Selecting uneven q-gram lengths prevents the existence of self-complementary sequences with $g = \bar{g}$. Notice, that using $q < 32$ also reduces the number of possible hash values and that this effect is amplified by additionally using min- or max-canonical q-grams.

For the winnowing window size w, we evaluated the range 30—which corresponds to the length of a protein sequence encoded in a 100 bp Illumina SGS read—to 100, corresponding to the length of an average Illumina SGS DNA short read.

We used $K = 3w$ as sequence length for the computation of Ψ, since these values cover the largest part of the probability mass of Ψ. For the size of the hash function codomain we used $C = 2\,000\,000$, which was the largest value still possible to compute with reasonable effort on our hardware for the sizes of w and K we analyzed.

We implemented an analysis workflow using SNAKEMAKE[12] to evaluate our approach for the parameters detailed above. This workflow can be found under the open source MIT license on Zenodo[13] and GitHub.[14].

The evaluation workflow comprises three main steps:

- Genome simulation
- Segment length evaluation
- Plotting

During the genome simulation phase, we generate FASTA files as described above, using a Rust program. To limit the influence of each single simulated sequence, we simulated a set of 10 sequences.

For segment length evaluation we computed a compressed winnowing for each simulated sequence and reference genome, using the different parameter combinations detailed above. Using this winnowing, we derived a sequence segmentation and count the occurrences of sequence lengths $|\mathfrak{s}|$ for $\mathfrak{s} \in \mathfrak{S}_h(A, q, w)$. Results for simulated genomes with normal GC-content (as evaluated in Section 6.5.1) are aggregated over all generated runs by adding the observed numbers for each segment lengths.

Finally, plots are generated using matplotlib[15] and seaborn.[16]

6.5.1 Distribution of Segment Lengths on Random DNA Sequences

In this section we compare the expected segment length distribution to empiric results obtained with different combinations of

[12] Köster and Rahmann, "Snakemake — A Scalable Bioinformatics Workflow Engine", 2012.

[13] Timm, *Segment Length Analysis Workflow*, 2021.

[14] `github.com/HenningTimm/segment_length_analysis`

[15] Hunter, "Matplotlib: A 2D graphics environment", 2007.

[16] Waskom and the seaborn development team, *mwaskom/seaborn*, 2020.

parameters. For each plot, we show the segment length on the x-axis and the amount of segments with this length on the y-axis. In the case that segment lengths $> w$ occur we plot all of these into one accumulated bar to better visualize their abundance. We will refer to these segments as compressed segments.

Figure 6.14 shows the distribution of segment lengths for simulated genomes using 31-grams. Segments were computed using compressed winnowing with a window size of $w = 50$. Each row of plots shares a hash function, denoted on the right side of the row, while each column shares a canonization strategy. The expected segment length distribution $\Psi_{50,150}^{2\,000\,000}$ is shown in each plot as black points, connected by lines to guide the eye. For better visualization, all segment lengths above w were aggregated into one bar placed on the right side, alongside with the expected probability $\sum_{k=w+1}^{K}(\psi_{50,k}) + \psi_{50,>K}$.

All distributions have in common, that—as predicted—most segments have a length of w. These are caused by relatively small minimizers, which are neither pushed out of the window by a smaller minimizer, nor become minimizer through a smaller minimizer leaving the window (cf. Lemma 6.4.1 and Lemma 6.4.2). This kind of minimizers have the highest rate of compression for robust winnowing and for compressed winnowing outside of repetitive regions, meaning that a winnowing with few minimizers covers the sequence.

On the other end of the spectrum, some combinations of hash functions and canonization strategies also show a high number of length 1 segments. Namely, for plain 2-bit encoding (denoted as *hf = 2bit*; the first row of Figure 6.14) and swap mixing (*hf = swap*) without canonical q-grams (non-canonical) and min-canonical q-grams, the peak for length 1 segments is in the same order of magnitude as that for length w. Segments with a length of 1 occur when the minimum changes after each step.

This case occurs frequently for both 2-bit encoding and swap mixing since the hash values of consecutive q-grams are not independent. Consider the DNA sequence $A =$ `TAACGTT` and its 2-bit encoding

$$\begin{array}{rcccccccccl} A = & \mathtt{T} & \mathtt{A} & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{G} & \mathtt{T} & \mathtt{T} & & \\ b(A) = & 11_2 & 00_2 & 00_2 & 00_2 & 01_2 & 10_2 & 11_2 & 11_2 & & = 11\,00\,00\,00\,01\,10\,11\,11_2 \end{array}$$

with the 3-gram sequence

$$\mathfrak{Q}(A,3) = (11\,00\,00_2,\quad 00\,00\,00_2,\quad 00\,00\,01_2,\quad 00\,01\,10_2,\quad 01\,10\,11_2,\quad 10\,11\,11_2)$$

and the windows for $w = 2$:

$$\begin{aligned} (11\,00\,00_2,\quad 00\,00\,00_2) &= (48,0) \\ (00\,00\,00_2,\quad 00\,00\,01_2) &= (0,1) \\ (00\,00\,01_2,\quad 00\,01\,10_2) &= (1,6) \\ (00\,01\,10_2,\quad 01\,10\,11_2) &= (6,27) \\ (01\,10\,11_2,\quad 10\,11\,11_2) &= (27,47) \end{aligned}$$

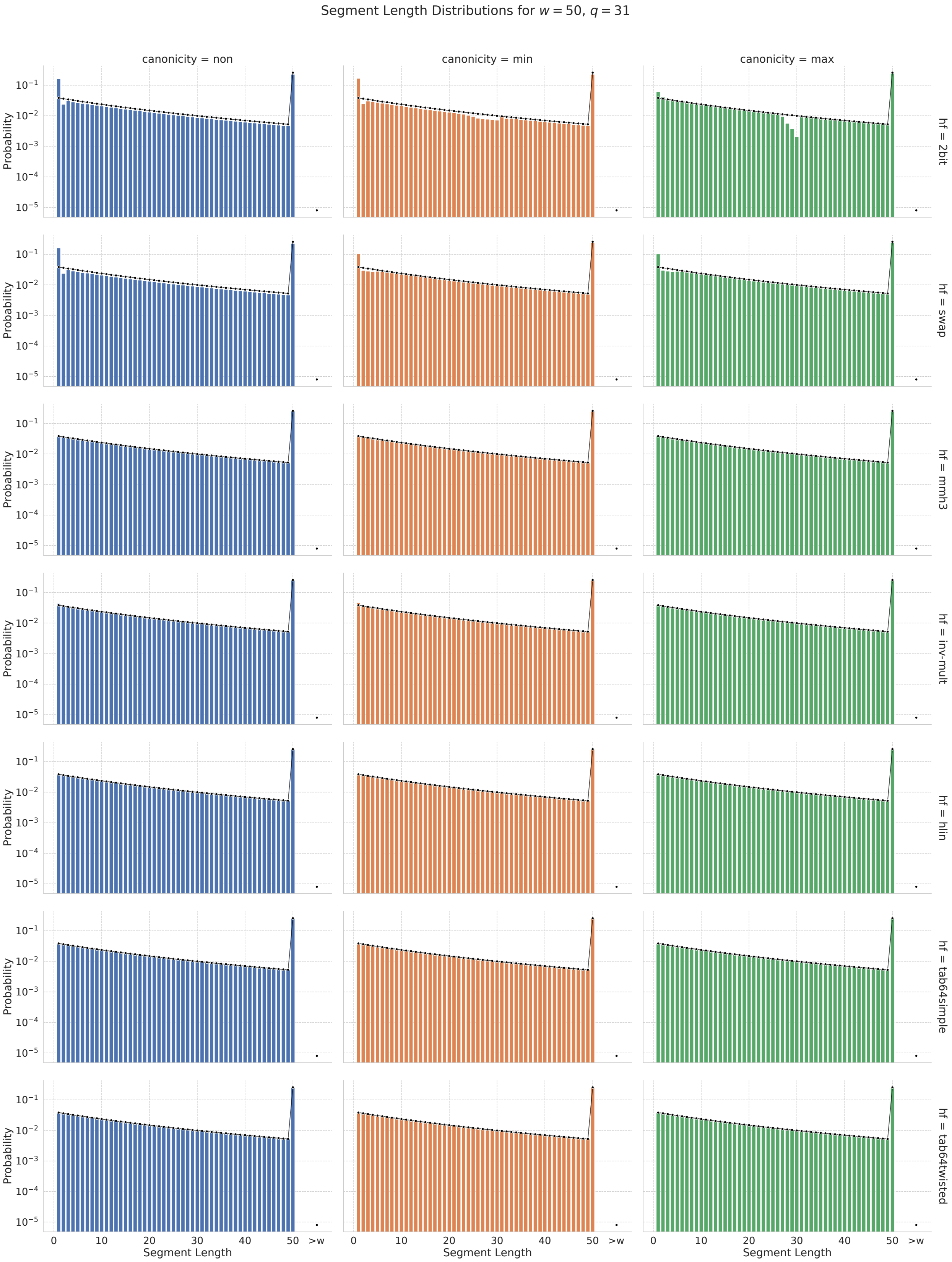


Figure 6.14: Length distribution of segments (colored bars) on 31-grams with a window of size $w = 50$, generated from 10 simulated genomes with a GC-content of 0.5 and a length of 100 000 000. The y-axis is logarithmically scaled. All plots in one column share the same canonicity (non, min, max), while all plots in one row share the same hash function. The expected distribution values from $\Psi^{2\,000\,000}_{50,150}$ are shown as black points. Both for the empirical and the expected distribution, all values $> w$ are collected into a single darker colored bar (or point for the expected value) on the right side of the plot. In this case, no such values are present.

The sequence A contains a run of As, which are 2-bit encoded to a block of consecutive zeros. When the window used for q-gram computation slides over such a segment of zeros, it first produces a sequence of zero values ($00\,00\,00_2$), which are handled according to the winnowing strategy. At the moment the window moves off the zero segment (or earlier, if its length is less than w bases i.e. $2w$ bits), the following w hash values are of the form:

$$b(g) = (b_0, \ldots, b_{w-1}) = \underbrace{00 \ldots 00}_{\text{leading 0-bits}}\ b_x \ldots b_{w-1}$$

Each additional base pushed into the q-gram, pushes out two zero bits, and thereby increases the hash value. This results in an increasing subsequence of hash values.

When winnowing such a sequence, as the window passes over the q-grams, we reach the point where the window minimizer is the leftmost entry in the winnowing window and the following values are sorted in increasing order: . With each step, a new minimizer is chosen by pushing out the new minimum. Moving forward, each step pushes out the acting minimizer and starting a new segment.

Using swap mixing as the hash function, the same effect occurs. The only difference is, that it occurs split up, one half, when the zero segment populates the most significant bits of the hash value, and when they are pushed out again. These dependencies between hash values violate the assumption we made for modeling, i.e. that hash values are i.i.d. While we noted above that hash values from functions with high independence can behave similar to i.i.d. hash values, these trivial hash functions do not distribute hash values well enough. Consequently, segment lengths deviate from the distribution expected for i.i.d. hash values. However, while 2-bit encoding shows less segments of length 1 when combined with max-canonical q-grams than swap mixing with the same canonization, swap mixing performs better using min-canonical q-grams.

For both of the trivial hash functions described above, segment lengths between 2 and w are underrepresented with regard to the expected distribution. A visualization of this effect can be found in Figure B.1 in the appendix. This effect is most pronounced when using non-canonical q-grams as well as 2-bit encoding with min-canonical q-grams. Both min- and max-canonical q-grams show a reduced presence of segments with lengths 25 to 30—even compared to the already lower distribution—followed by a higher probability for segments with length 31, which coincides with the q-gram length. An explanation for this phenomenon remains open.

Out of these two hash functions, 2-bit encoding in combination with max-canonical q-grams adhered closest to the expected distribution. Especially for the arguably most important segment lengths 1 and w, the empiric distribution most closely resembles the prediction using random hash values.

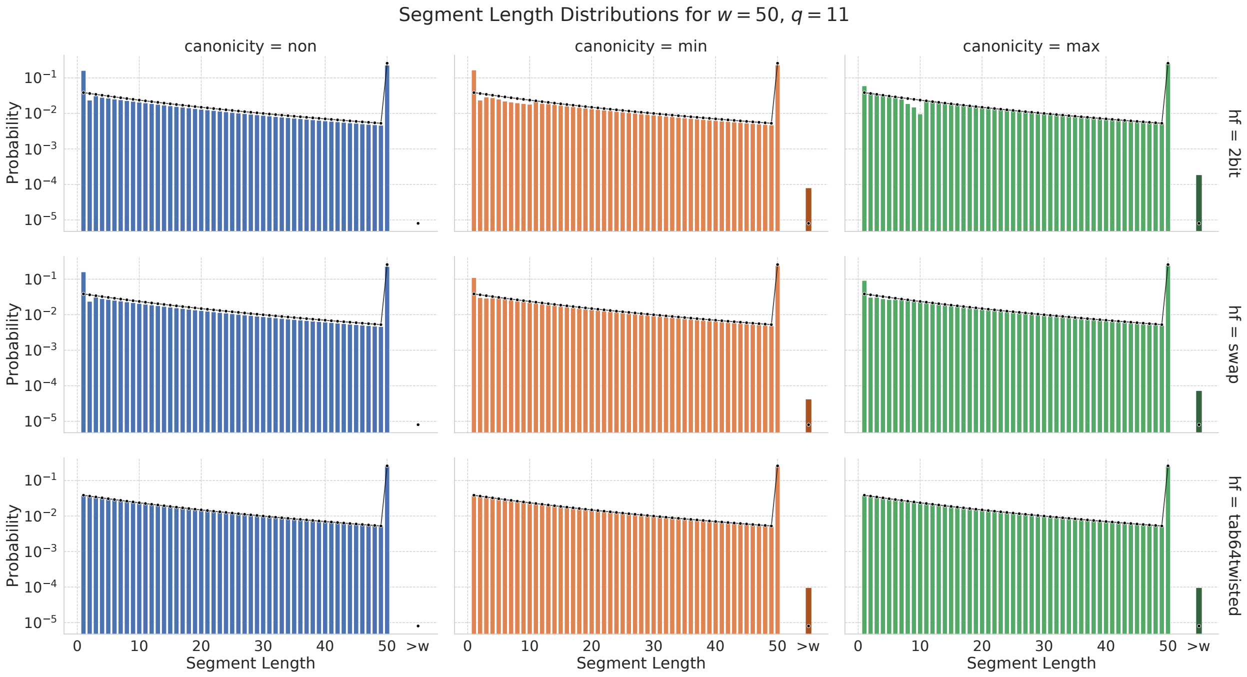


Figure 6.15: Empirical distribution of segment lengths of random DNA sequences (colored bars), compared with expected distribution (black points) using $q = 11$ and $w = 50$. Expected segment lengths were computed using a hash function codomain size of $C = 2\,000\,000$. The sum for all segments of length $> w$ is shown as a separate darker bar on the right of the plot.

All other hash functions adhere more closely to the expected segment length distribution. The only exception is the invertible multiplication hash function, which shows a small over-representation of length 1 segments, when used without canonization or with min-canonical q-grams (barely rising above the point denoting the expected value).

For none of the simulated DNA sequences, segments with length $> w$ were observed. This is to be expected, since this evaluation used 64-bit hash values, which realize a hash function codomain several orders of magnitude larger than we are currently able to evaluate for Ψ. The expected values shown for this evaluation (illustrated in black) used a hash function codomain of size $C = 2\,000\,000$, which we were still able to analyze. Thus, the probability of segment lengths $> w$ is overestimated due to technical limitations.

For different values of w—namely $w = 30$ and $w = 100$—we observed identical effects, with the drop in probability These plots can be found in Appendix B.2 and Appendix B.3 respectively. We removed the `mmh3`, invertible multiplication, $\mathcal{H}^{\text{lin}}$, and simple tabulation hash functions from these plots, keeping only twisted tabulation hashing as a reference.

For smaller values of q, we observe an increased amount of segments longer than w. In Figure 6.15 we show the distributions for $q = 11$, $w = 50$, and a reduced set of hash functions. While the overall shape of the segment length distributions remained as with $q = 31$ for all experiments, especially for $q = 11$, we observed more compressed segments. This is to be expected, since smaller q-grams reduce the space of possible hash values and therefore increase the probability to observe the same hash value twice within w positions. With 11-grams, we can observe at most 4^{11} different hash values and only approximately half that amount when using a canonization strategy. This corresponds to a universe size of $C \approx 4\,000\,000$ and $C \approx 2\,000\,000$ respectively, which is several orders of magnitude closer to the value used to compute the expected distribution. Nonetheless, in the evaluation we observed more compressed segments than expected. These deviations could also be caused by rounding errors in our computation of Ψ and require further investigation.

Considering the random DNA sequences shown here and their compressed winnowing segment length distribution all non-trivial hash functions performed well. As to be expected, using 2-bit encoded values directly or swap-mixed values resulted in sub-par results. However, using max-canonical q-grams partially mitigated this. We have stated above that DNA sequences with uniformly randomly distributed bases are a stark simplification with respect to real genomes. In the next section, we analyze random sequences that deviate from this assumption by varying the GC-content.

6.5.2 *Influence of GC-Content*

The GC-content describes the abundance of the bases G and C in a DNA sequence (cf. Structure of DNA and RNA Sequences, p. 10). It can be interpreted as the probability that a base from a given genome is either G or C. We simulated genomes with different GC-contents

$$\mathrm{GC}(\text{simulated genome}) = \gamma$$

ranging from $\gamma = 0$ to $\gamma = 1$, with $\mathbb{P}(\mathtt{G}) = \mathbb{P}(\mathtt{C}) = \frac{\gamma}{2}$ and $\mathbb{P}(\mathtt{A}) = \mathbb{P}(\mathtt{T}) = \frac{1-\gamma}{2}$. By reducing the GC-content, we reduce the probability to encounter certain q-grams, resulting in sequences with reduced complexity. Especially for $\gamma = 0$ and $\gamma = 1$, the used alphabet collapses to a size of $\sigma = 2$. We simulated sequences of length $100\,000\,000$ bases and analyzed them as before to show the influence of varying GC-content on segment lengths. Note, that as with reduced length of q, this effectively reduces the (expected) abundance of certain q-grams which results in less distinct q-grams and ultimately a reduction of the hash value space. To show a broader range of γ values within one plot, we use $w = 30$ throughout this section. Note that the center column of the following plots with a GC-content of $\gamma = 0.5$ are equivalent to the respective plots in Figure B.2 (p. 240).

Figure 6.16 illustrates segment length distributions for different GC-content values (columns), obtained by different combinations of hash functions and canonization strategies (rows). We selected the combinations of hash functions and canonization methods shown in Table 6.3. For min-canonical q-grams we also show results for `mmh3`, which performed comparably to tabulation hashing in our previous analyses.

Hash Function	Canonization
2bit	non
tab64twisted	non
2bit	min
mmh3	min
tab64twisted	min
2bit	max
tab64twisted	max

Table 6.3: Combinations of hash functions and canonization strategies used.

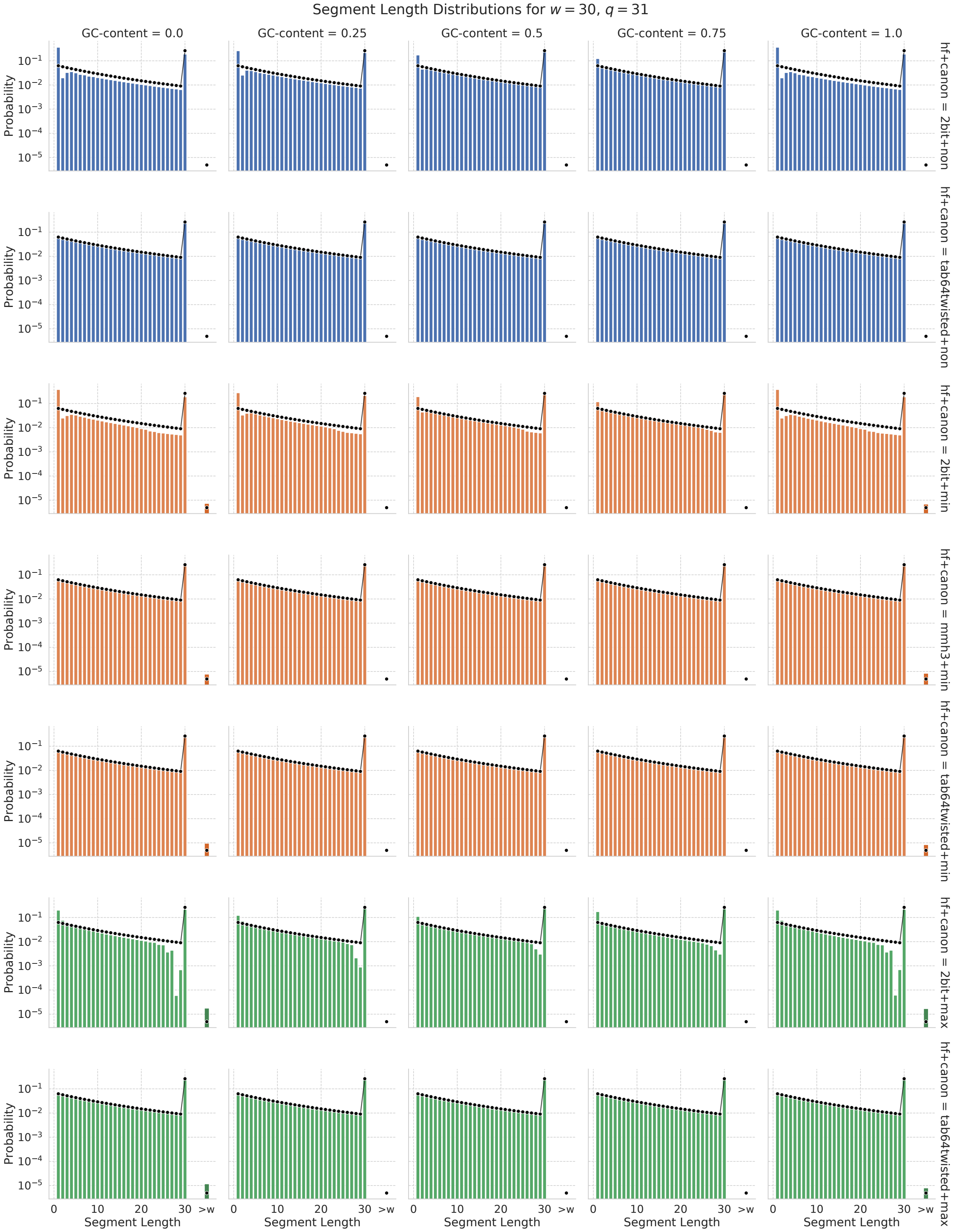


Figure 6.16: Distribution of segment lengths for random sequences of length 100 000 000 for multiple GC-contents (columns) with $w = 30$ and $q = 31$. Each row contains plots for one combination of hash function and canonization strategy, with rows using the same canonization strategy grouped by color. As before, all segments above length w are aggregated into one bar.

Both high and low GC-content values result in a higher amount of segments with length 1 with hash functions that have previously shown an increase in length 1 segments in Figure 6.14. Additionally, for the extreme cases $\gamma = 0$ and $\gamma = 1$, segments of length $> w$ are observed even for $q = 31$. This is caused by the drastic reduction in the input space from 4^{31} to $2^{31} = 4^{15.5}$, when only two of the four bases are used. Using $q = 31$, these segments only occur when using either min- or max-canonical q-grams, which amplify the reduction of the hash function codomain size. However, the number of observed segments with length $> w$ is still close to the expected value. For smaller q-gram sizes, like $q = 17$ and $q = 11$, this is no longer the case and the observed segments exceed the expected number. This is illustrated in Figure B.4 and Figure B.5 in the appendix.

As expected, the number of segments with a length larger than w increased for extreme GC-content values, as these reduce the number of possible hash values. However, for q-gram sizes of $q = 17$ and above, this effect was only measurable for $\gamma = 0$ and $\gamma = 1$. Only for smaller q-gram sizes did this effect also occurred for $\gamma = 0.25$ and $\gamma = 0.75$. This effect needs to be taken into account when analyzing genomes with more extreme GC-content. With regards to the GC-content of reference genomes (cf. Table 6.2), *M. xanthus* and *P. falciparum* have comparable levels of GC-content. While this experiment was restricted to random sequences, this effect can be expected to be present even more pronounced in these reference genomes, due to their less random sequence structure.

6.5.3 *Segment Length Distribution of Reference Genomes*

To judge the properties of real genomes with our winnowing approach, we analyzed the genomes described in Table 6.2. At this point, we restrict the set of hash functions to 2-bit encoding, Murmur hash 3, and twisted tabulation hashing. Both Murmur and twisted tabulation have performed equally well in previous tests, with tabulation hashing offering theoretical guarantees that murmur cannot. The 2-bit encoding serves as a reference to compare these hash functions to.

The plots shown in this section depict segment lengths computed with 17-grams and a window size of $w = 50$. We have selected these since they best show the performance difference between different hash functions. The versions of these plots with $w = 31$ can be found in the appendix as Figures B.6, B.7, and B.8. Figures in this section are split by canonization strategy, i.e. each figure depicts segment length distribution for different genomes (rows) and hash functions (columns) that all use the same canonization strategy (colors). The same color code for canonization introduced in Table 6.3 applies. Notice, that for shorter genomes—especially for *M. xanthus*—single segments have greater influence on the distribution through the normalization process.

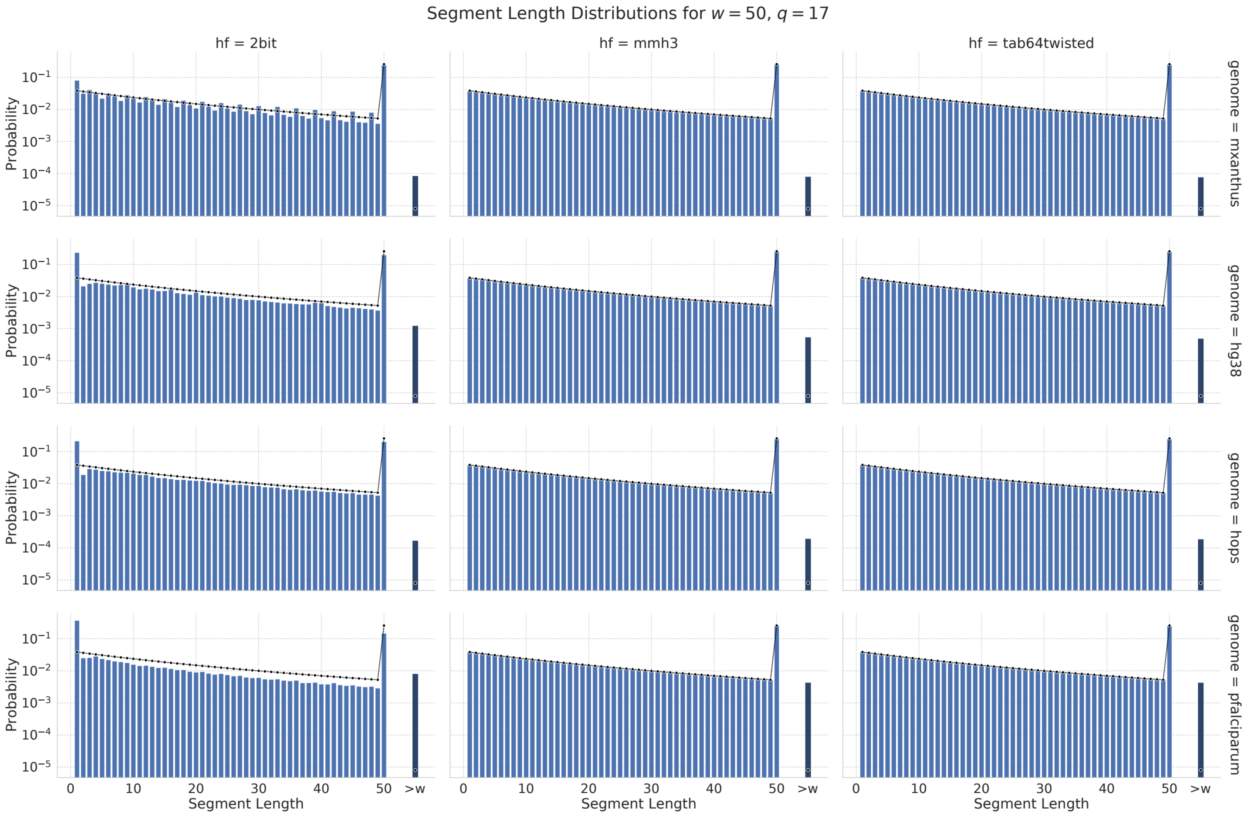


Figure 6.17: Segment length distributions for reference genomes (rows) using different hash functions (columns) and non-canonical 17-grams. The empirical segment length distribution is shown as bars with the expected distribution shown by black points. All segment lengths larger than w are aggregated into a single bar with darker shade on the right of each facet.

Figure 6.17, Figure 6.18, and Figure 6.19 illustrate the segment length distributions for non-, min-, and max-canonical q-grams respectively. Across all canonization strategies using $q = 17$, the *P. falciparum* and Hg38 genomes contained the highest fraction of compressed segments, followed by *H. lupulus* and *M. xanthus*. Notice that the number of possible hash values with $q = 17$ is larger than the value $K = 2\,000\,000$ used for the expected distribution, resulting in an overestimation of these segments' probabilities. Nonetheless, for all genomes and all canonization strategies the number segments with length $> w$ surpasses even the overestimated amount. Using 2-bit encoding, this effect is smaller for non- and max-canonical q-grams than for min-canonical q-grams. For the Murmur and tabulation hash functions, differences across canonization strategies were less pronounced. In this case, using non-canonical q-grams reduced the amount of observed segments with length $> w$. The fact that less reduction of these segments could be observed suggests that the segments are caused by repetitive subsequences within the genomes, and not by chance through reduction of the hash value space.

For the *M. xanthus* genome, we observed a repetitive pattern of segment length distributions that deviate from the expected distribution when using 2-bit encoding. Across all evaluated values for q and w and all canonization strategies, segments with lengths that are multiples of three occur with a higher than expected probability

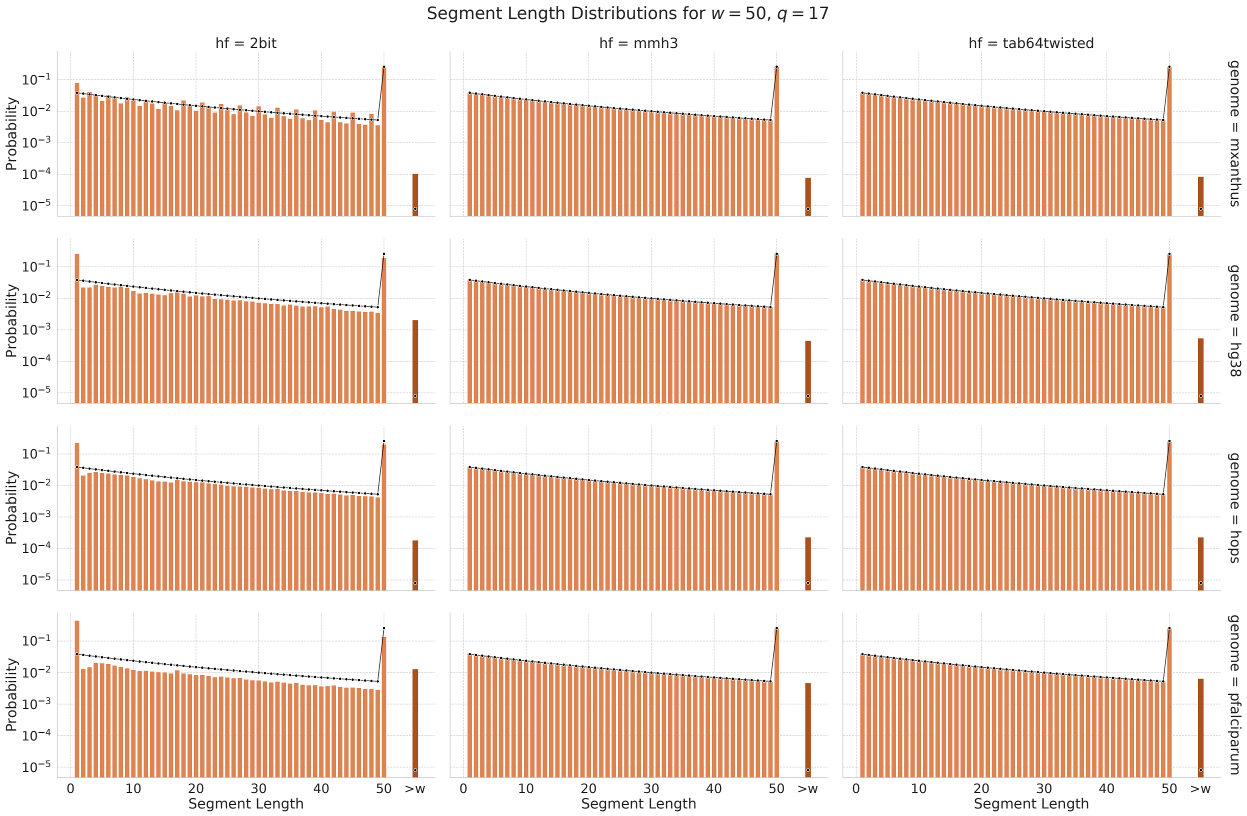


Figure 6.18: Segment length distributions for reference genomes (rows) using different hash functions (columns) and min-canonical 17-grams. The empirical segment length distribution is shown as bars with the expected distribution shown by black points. All segment lengths larger than w are aggregated into a single bar with darker shade on the right of each facet.

while segments of other lengths are less prevalent. Since this effect is only present for 2-bit encoding, a possible explanation for this behavior are (potentially regular) sequence patterns which do not cause equiminimal segments. Through the application of any kind of non-trivial hash functions, these patterns would be broken up, while they remain present under all canonization strategies using 2-bit encoding. The fact that the over-represented sequence lengths are all multiples of three could be point to the codon distribution of the *M. xanthus* genome. An in-depth analysis of this behavior, for example through evaluating the sequence neighborhood of over-represented segments, remains open for future research.

For the *P. falciparum* genome, the number of segments of length $> w$ exceed both the expected value (by three orders of magnitude) and the values computed for other genomes (by approximately one order of magnitude). This increase in segments with length $> w$, even in comparison with *M. xanthus* with a similarly extreme GC-content, can be attributed to the prevalence of low complexity regions. The *P. falciparum* genome contains such regions, which have a mean length ranging from 111 bp to 270 bp and within them GC-content can fall to $\gamma = 0$.[17] As described in Section 6.3, these subsequences are likely to generate equiminimal runs, which result in long segments when using compressed winnowing.

[17] Zilversmit et al., "Low-complexity Regions in Plasmodium falciparum: Missing Links in the Evolution of an Extreme Genome", 2010.

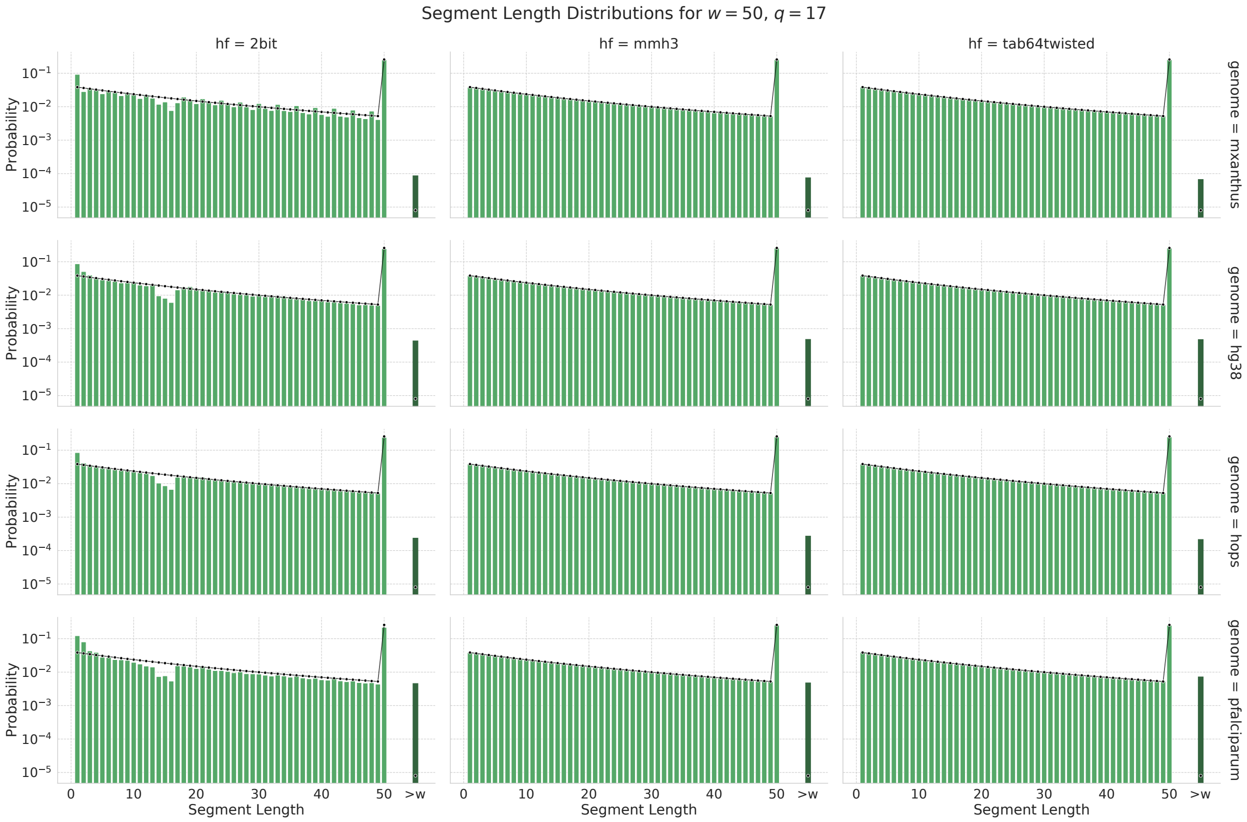


Figure 6.19: Segment length distributions for reference genomes (rows) using different hash functions (columns) and max-canonical 17-grams. The empirical segment length distribution is shown as bars with the expected distribution shown by black points. All segment lengths larger than w are aggregated into a single bar with darker shade on the right of each facet.

6.6 Distribution of MinHash Values

As mentioned above, we assumed hash values to be uniformly, independently, randomly chosen for the segment distribution analysis. In our evaluation, we used q-grams from DNA sequences, which only approximate the behavior of i.i.d. hash values for $|\mathcal{C}| \ll 4^q$. While nominally, we used $\mathcal{C} = [2^{64}]$ for the empirical analyses, multiple influencing factors reduced the effective size of $\mathcal{C}$, i.e. the number of possible distinct hash values. Our analysis in Section 6.4 has shown that the characteristic shape of segment length distributions observed in the empirical analysis occurs for very small hash function codomains. Already for values around $C = 5\,000$, the expected distribution closely approximates that for larger C (cf. Figure 6.10). The main difference for smaller universes is the probability of segments with a length larger than w, as can be seen in Figure 6.9.

Another important factor to be observed with minimizer-based strategies is the distribution of MinHash values themselves, i.e. the size distribution of hash values selected as minimum. Consider the example application described in Section 6.2. When all MinHash values entered into the hash table are recruited from a small space, a large number of collisions is to be expected. This can hamper the effectiveness of such data structures, as the very pattern they aim to prevent occurs again: A large number of alignment candidates for each read. This necessitates the usage of a large (effective) hash function codomain to prevent creating artificial collisions.

Another more technical challenge of storing minima in a hash table is the uneven distribution of input keys across the hash table itself. When directly using the MinHash values as keys for a hash table, many collision are expected to occur in the lower fractions of the address space due. This is due to the preselection of small hash values through MinHashing strategies and can result in unevenly filled hash tables. There are two possible approaches to mitigate this effect:

- Rehash the minimizer (i.e. the q-gram) with a different hash function.
- Rehash the MinHash value with a fast integer hash function.

Since the q-gram has already been selected as minimizer, we can hash it using a different hash function to store it in the hash table. While this approach is the most flexible, it introduces another hash function computation and depending on the hash function used this can be a runtime concern. The second approach, i.e. rehashing the MinHash value itself using a fast integer hash function, can be preferable in some situations.[18]

6.7 *Segment Number Estimation*

Using the expected segment length distribution presented in Section 6.4, we can estimate the number of segments in a winnowed index for a sequence A. For a given w-segmentation $\mathfrak{S}(G,w)$ of a q-gram sequence $G := \mathfrak{Q}(A,q)$, we have an unknown number of segments $\omega = |\mathfrak{S}_{G,w}|$ which we want to estimate. We know, that the sum of all segment lengths sums up to the number of window starting positions (cf. Definition 6.1.2 and Figure 6.3):

$$\sum_{\mathfrak{s}_i \in \mathfrak{S}(A,q,w)} |\mathfrak{s}_i| = (|A| - q + 1) - w + 1$$

Choosing the sequence length for the expected distribution $K = (|A| - q + 1) - w + 1$ equal to the number of window starting positions in A, we can compute the mean segment length as:

$$\mathbb{E}(|\mathfrak{s}|) = \sum_{k=1}^{K} \psi_k^C \cdot k \qquad \psi_k^C \in \Psi_{w,K}^C$$

Notice that through this choice of K, the term $\psi_{>K}^C$ of $\Psi_{w,K}^C = (\psi_k^C)_{i=k}^K \cup \psi_{>K}^C$ collapses to 0 since there are no possible segments with length > 0. For a precise estimate of expected number of segments, we would need to consider all combinations of segments lengths resulting in a length of K weighted by their respective probability.[19] Our goal—using the estimate to pick the size of an index data structure—can also be achieved by a more simple estimate that is faster to compute. Given the sequence length K and the mean

[18] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

[19] For a solution to a simplified version of this problem using six-sided dice, refer to Conroy, *A Collection of Dice Problems*, 2018, Problem 31.

segment length for the distribution $\Psi^C_{w,K}$, we can estimated number of segments required to cover said sequence as:

$$\omega^* := \left\lceil \frac{K}{\mathbb{E}(|\mathfrak{s}|)} \right\rceil$$

However, as mentioned above, currently we cannot compute Ψ for lengths K that are applicable to realistic genome sequences. In this case, our estimate ω^* can serve as an upper bound for the actual number. Through choosing $K < |A|$, we prevent long segments from influencing the expected number of segments, resulting in an overestimate of the total number of segments. Since resizing a winnowed index (i.e. its underlying hash table) is costly, overestimating the number of segments is preferable.

Notice, that for growing values of C and therefore decreasing probabilities for segment lengths $> w$ the expected segment number for compressed winnowing approaches the expected segment number for robust winnowing.

Our analysis of reference genomes has shown, that the amount of compressed segments is higher than expected using randomly chosen hash values. An approach to improve our prediction of the number of compressed segments would be to estimate the expected number of hash values for subsequences of a genome based on their complexity. By computing ω^* for each sequence, using different values for C for high and low complexity regions, a better estimate for the expected number of segments could be achieved.

6.8 Discussion and Conclusion

We presented a novel winnowing approach—compressed winnowing—which collapses repetitive regions into single segments. For the use with biological sequences this approach offers the improvement of reducing common repetitive patterns into a single characteristic segment. Thus reducing the overall number of segments in the winnowed sketch with respect to robust winnowing while only losing information that actively hinder many sequence analysis tasks. We additionally outlined a multiple winnowed sequence index leveraging our technique to identify alignment candidates for an application in protein similarity analysis for metagenomic screening.

Using a recursive function we were able to compute the expected distribution of compressed winnowing segment lengths for hash values uniformly and independently chosen from a given hash function codomain. We could show that segments obtained from reference and draft genomes, as well as from simulated genomes, adhered closely to our prediction, if certain standards to the used hash function were satisfied.

Working with simulated genomes, we could show that segments of length larger than w, which are specific to our compressed winnowing approach, do not occur at random for reasonable hash function codomain sizes, like $C = 2^{64}$.[20] This shows that through compressed winnowing we do not accidentally merge important information. Moreover, we could show that trivial hash functions like plain 2-bit encoding and swap mixing do indeed perform poorly for winnowing approaches since they do not break up the interdependence of neighboring q-grams. All other hash functions performed significantly better, with tabulation hashing (twisted more than simple) and mmh3 most closely resembling the expected distributions Ψ for randomly selected hash values. This is in line with the properties described for these hash functions in Chapter 3. Twisted tabulation hashing offers ϵ-approximately min-wise independence, and while not offering any theoretical guarantees, mmh3 has already been proven to perform well in the context of MinHash approaches (cf. Hash Functions for Use in Bioinformatics, p. 44).[21]

Using max-canonical q-grams instead of min-canonical q-grams did not offer a universal improvement. Only when used in combination with 2-bit encoding did max-canonical q-grams improve the distribution of segment lengths. However, this coincided with a reduced probability for segments with length $q - 1$ and less, for which an explanation remains open. In combination with nontrivial hash functions, no consistent changes could be observed. For these cases a detailed analysis of the MinHash value distribution could be of interest for future research.

We could show that effects that reduce the number of distinct q-grams, and therefore the number of different possible hash values, increase the number of segments that exceed the window size w. Most notably, for a lowered GC-content as well as min- and max-canonical q-grams, we could observe an increased amount of these segments. This was most visible when reducing the number of different possible hash values using simulated genomes that only comprise two bases through a GC-content of $\gamma = 0$ or $\gamma = 1$.

During the analysis of reference genomes we could show that genomes which are known to contain a high amount of repetitive sequences—namely the *P. falciparum* genome—generate a large amount of compressed segments when analyzed with compressed winnowing. A direct comparison between robust and compressed winnowing remains open. While we can observe the number and length of compressed segments and deduce the number of saved segments, this remains a topic for future research. Most notably, this could be performed with regards to the entropy (for example the Shannon entropy) of the analyzed sequences.[22]

Interestingly, we obtained the expected segment length distribution even for hash function codomains as small as $|C| = 5\,000$. However, the practical use of hash functions with such small codomains is limited, since the computed MinHash values collapse into a set too small to meaningfully distinguish entries. Most notably, for the future implementation of our segment index outlined in this chapter, using a large hash function codomain is required to restrict these to actually (strongly) repetitive regions.

[20] Notice that effectively a hash function codomain size of $C = 2^{64}$ is never reached by our evaluation through our use of q-grams shorter than 32 bp and canonization. While nominally we hash to the codomain $\mathcal{C} = [2^{64}]$ the effective codomain is much smaller, especially for smaller values of q. As noted above, this supports q-gram hashes in behaving similar to i.i.d. hash values.

[21] Thorup, "Fast and Powerful Hashing Using Tabulation", 2017; Richter, Alvarez, and Dittrich, "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing", 2015.

[22] Schmitt and Herzel, "Estimating the Entropy of DNA Sequences", 1997.

7
Analysis of ddRAD Data

As introduced in the section on Specialized Application Sequencing Workflows, restriction site associated DNA sequencing (RADseq) and double digest RADseq (ddRADseq) use restriction enzymes to fragment DNA sequences at reproducible positions. Due to its ability to process large sample sizes and relatively low cost per individual sequenced, data generated by RADseq is used for biodiversity and population genetic studies of (non-model) organisms.[1] Using restriction sites, which occur at the same position in all individuals of a sample, allows detecting mutations without performing de novo assembly, i.e. constructing a reference genome of the analyzed organisms. A possible application would be to judge the effect of anthropogenic stress on a population, as illustrated in Figure 7.1.

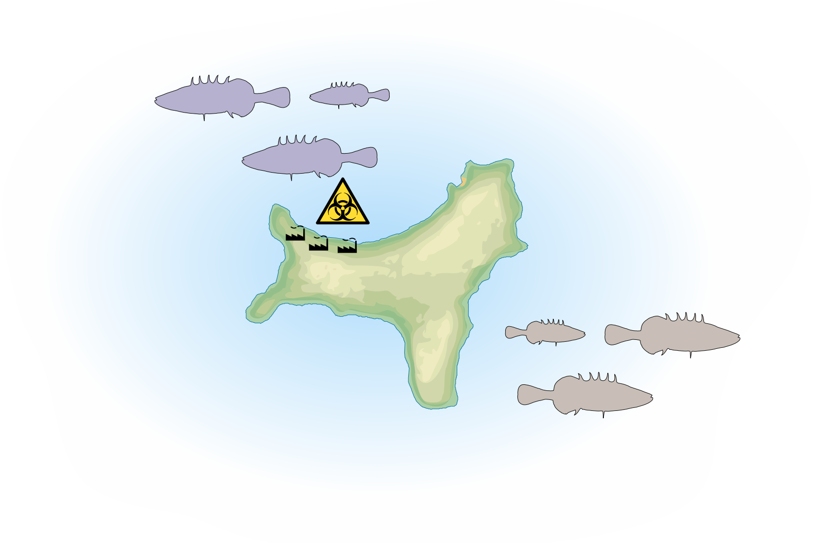


Figure 7.1: Identification of anthropogenic stress on two populations of fish. The purple population (🐟) is affected by anthropogenic stress through pollution, while the brown population (🐟) remains unaffected. It can be expected that the amount of biodiversity in the 🐟 population is lower, since individuals not optimally adapted succumb to the stressor.

Graphic derived from "Christmas Island Australia 76 FR" by Ewan ar Born CC BY 3.0 and "Biohazard" released into the public domain by Wikipedia User Knyaz-1988.

Due to the particularities of this workflow, the analysis of (dd)RAD-seq presents unique difficulties, both for the technologies to acquire this kind of data as well as for its analysis. Consequently, analyzing RADseq data is a complex task with many variables. In order to test and evaluate the performance of this pipeline, we developed ddRAGE [2] a software able to simulate ddRAD reads and generate

[1] Andrews et al., "Harnessing the Power of RADseq for Ecological and Evolutionary Genomics", 2016; Peterson et al., "Double Digest RADseq: an Inexpensive Method for de novo SNP Discovery and Genotyping in Model and Non-Model Species", 2012.

[2] Timm et al., "ddRAGE: A Data Set Generator to Evaluate ddRADseq Analysis Software", 2018; Timm and Rahmann, *ddRAGE — ddRAD Data Generator (Source Code)*, 2020.

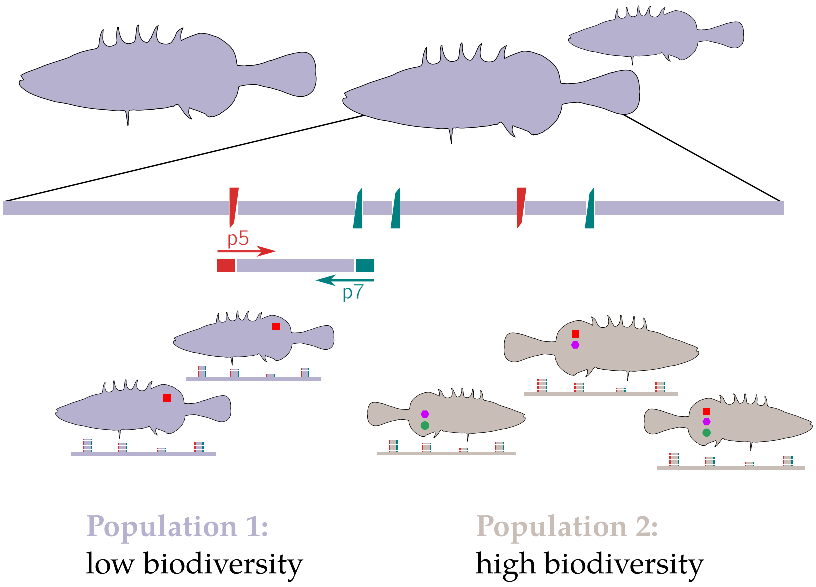


Figure 7.2: Illustration of a ddRAD analysis. The DNA sequence of an individual (purple bar), is cut using restriction enzymes (red and teal wedges). Resulting fragments are sequenced as paired end reads. For individuals of the same species, loci are likely to occur in all sequenced individuals, allowing comparison on the population level. In this example the purple and brown population can be distinguished by their level of biodiversity, denoted by icons in the fish outlines.

a diversity of challenging datasets. Additionally, we developed an analysis pipeline that automates many of the most tedious tasks.

In this chapter, we first describe the ddRADseq technique and the structure of data it generates. Following that, we describe how our simulation software DDRAGE models this structure and finally we present our analysis pipeline. Both the description of (dd)RADseq technology as well as the description and evaluation of DDRAGE are based on our publication of DDRAGE.[3] A list of probability distributions referred to in this section can be found in the section Probability Distributions (p. 34).

7.1 *Acquisition and Structure of ddRAD Data*

The core idea of all RAD sequencing approaches is to cut the DNA using restriction enzymes and sequence the resulting fragments. Given two individuals of the same species, both individuals possess almost identical genomes, apart from small genetic variations. If the genome of the first individual is cut at positions with a specific DNA motif, these restriction sites[4] have a high probability to be present and in the same position for the second individual as well. Sequences adjacent to a restriction site, which are sequenced during the RADseq process, form a so called locus.[5] Unless mutation has eroded the motif in one of the individuals (leading to one or both alleles in an individual not being sequenced), a locus sequenced in one individual will also be sequenced in the other. This property is exploited by RADseq to uniformly sample sequences from genomes, allowing to judge their genetic diversity. Comparing RADseq loci between individuals increases the ability to make population genetic inferences (see Figure 7.2).

[3] Timm et al., "ddRAGE: A Data Set Generator to Evaluate ddRADseq Analysis Software", 2018.

[4] Also know as cut sites.

[5] Depending on the source, the restriction site, the reads generated from it, or both are called locus.

From the different RAD sequencing approaches, which aim to alleviate certain weaknesses of basic RAD sequencing,[6] we focus on ddRADseq.[7] The ddRAD approach employs two different restriction enzymes to assure that the length of generated fragments does not depend on random shearing effects.

7.1.1 *Restriction Enzymes*

The cutting process used by RADseq is realized by restriction enzymes, which bind to DNA strands at a specific motif and cut the strand within or next to the motif. This process is also called digestion. Most of the restriction enzymes used in (dd)RAD sequencing leave behind part of their restriction motif after cutting, as illustrated in Figure 7.3. Table 7.1 shows a selection of restriction enzymes with their restriction site (motif) and residue (overhang or sticky ends) remaining in the reads.

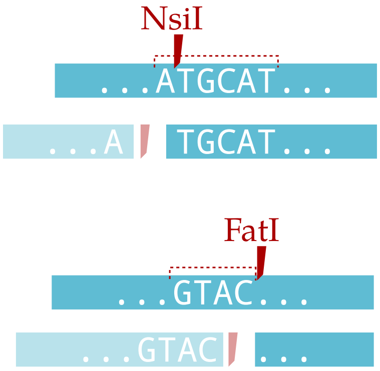


Figure 7.3: Cut sites of a restriction enzyme that leaves a residue after cutting (top) and one that does not (bottom). The first enzyme cuts the motif `ATGCAT`, leaving the sequence `TGCAT` in the fragment, while the second one cuts off the whole restriction site `GTAC`. Sequences shown in light teal are cut off, fully saturated fragments remain to be sequenced.

Name	Motif	Residue
Csp6I	`GTAC`	`TAC`
PstI	`CTGCAG`	`G`
NsiI	`ATGCAT`	`TGCAT`
BamHI	`GGATCC`	`GATCC`
FatI	`CATG`	

Table 7.1: List of selected restriction enzymes used in RADseq with their cut site motif and residue.

Since restriction enzymes bind to specific motifs, the number and location of restriction sites in a specific genome depend on several factors. These include the length of the motif (shorter motifs are more frequently present in a genome) and the `GC`-content of the target genome (if the motif contains many `G`s and `C`s, motifs are less likely to occur in low `GC` species). Consequently, the number of loci generated for a given genome can be influenced by the choice of the employed restriction enzymes. Based on the number of loci they are expected to generate, restriction enzymes are classified as rare cutters or frequent cutters.

After a DNA sequence is cut, the resulting fragments are truncated to lengths that can be analyzed by a SGS device. For basic RADseq, this is performed by trimming the reads using ultrasound or other fragmentation techniques, and performing a length selection. Due to the random nature of the trimming process, RADseq reads from one locus can have different lengths, as illustrated in Figure 7.4.

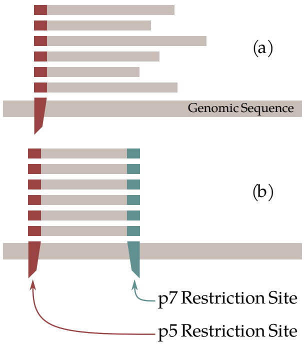


Figure 7.4: The length of basic RADseq fragments (a) varies due to the randomness of the fragmentation process, while the length of ddRADseq fragments (b) is determined by the p7 restriction enzyme.

By using both a rare and a frequent cutter and digesting the DNA twice (thus: double digest), ddRADseq can assure that fragments have fixed sizes, determined by the occurrences of rare and frequent cut sites within the target genome. A length selection is performed as well, since the cut fragments can still be too long for SGS sequencers.

As with any biological process, DNA digestion with restriction enzymes is a probabilistic process. It is possible, that a restriction site present in the genome is not cut and remains in the fragments. This incomplete digestion can influence the layout of reads generated by ddRADseq.

7.1.2 *Sequencing Process and Read Structure*

After digestion and size selection have been performed, the remaining fragments are sequenced. Parameters like the length and sequencing errors present in the reads are determined by the se-

[6] Andrews et al., "Harnessing the Power of RADseq for Ecological and Evolutionary Genomics", 2016.

[7] Peterson et al., "Double Digest RADseq: an Inexpensive Method for de novo SNP Discovery and Genotyping in Model and Non-Model Species", 2012.

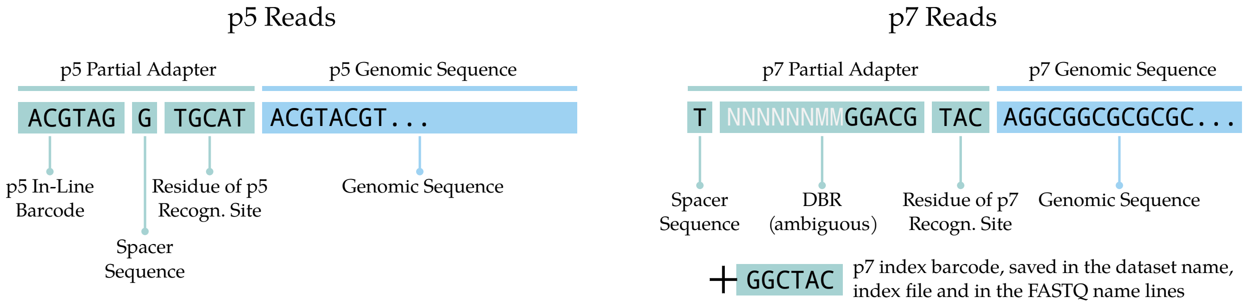


Figure 7.5: Structure of PE ddRAD reads obtained from Illumina sequencers. In the p7 read, ambiguous parts of the degenerate base region (DBR) are shown in light gray. The p7 index barcode is not part of the final read, but is removed with the rest of the adapter. This illustration is derived from Figure 1 of our publication: Timm et al., "ddRAGE: A Data Set Generator to Evaluate ddRADseq Analysis Software", 2018.

quencing technology used. While ddRADseq can be performed with several sequencing technologies, paired-end reads obtained from Illumina sequencers are widely used. Hence, we will focus our description on this technology. Due to the applied library preparation protocol, ddRAD reads contain several sequences in addition to the genomic sequence from the p5 and p7[8] end of the sequenced fragment respectively. A typical setup of ddRAD reads is illustrated in Figure 7.5. Note that depending on the specific experiment, not all of these sequences need to be be present.

To achieve a high sequencing throughput, several individuals are analyzed on the same flow cell lane using barcoding. Before sequencing, fragments from each individual are tagged with a pair of short artificial DNA sequences on the p5 and p7 end. Each pair of p5 + p7 barcode uniquely identifies one individual. Depending on library preparation and sequencing protocol, barcodes can be present as in-line barcodes, which remain in the read after sequencing, and index barcodes. The latter are part of the adapter sequence used by the sequencing device. They are trimmed off during the sequencing process and are used to dispatch the sequenced reads to different output FASTQ files. The read setup shown in Figure 7.5 contains p5 in-line barcodes and p7 index barcodes. Reads for each individual can be extracted from the sequencer output through demultiplexing, i.e. sorting reads into individual files based on their p5 + p7 barcode combination.

Spacers are small sequences added to avoid over-saturation of the sequencer's camera. Depending on context, these sequences are also called inserts, or ins for short. We use the term spacer instead, to avoid confusion with insertion mutations. Since the following sequences (restriction enzyme residues and fixed parts of the degenerate base region, both of which we will describe below) are identical for all individuals on the flow cell lane, one base per sequencing cycle yields a very high light response, while the other three remain dark. This problem is alleviated by purposefully synchronizing reads from different individuals. Each individual is assigned a pair of sequences of length 0 to 3, which are added to p5 and p7 read before the restriction enzyme residue. Individuals with a spacer of length 1 reach the enzyme residue one sequencing cycle after those with spacer of length 0 and before those with longer

[8] For Illumina reads, forward and reverse read are also called p5 and p7 read respectively, named after the flow cell binding sites.

spacers, thus distributing the high light response over several cycles. As a downside, longer spacers reduce the length of genomic sequence that can be sequenced. Since the read length is constant, each spacer base truncates the genomic sequence by one base with respect to a read without spacers.

Depending on the restriction enzymes used to digest the DNA, restriction enzyme residues are also present in the reads. As described in the section on Restriction Enzymes (p. 163), sticky ends can be left behind by some restriction enzymes (like NsiI or Csp6I) after cutting. These mark the beginning of the genomic sequence, even though they provide no usable information themselves, since they are identical in all reads.

Finally, Unique Molecular Identifiers (UMIs) or Degenerate Base Regions (DBRs)[9] are often used to identify and remove PCR duplicates. By adding a partially randomized sequence to the fragments before PCR amplification, PCR duplicates can be identified with high probability as reads with identical DBR sequence and identical genomic sequence. DBR sequences are denoted as a string from Σ_{IUPAC}, where ambiguous bases are assigned at random during library preparation. For example, in Figure 7.5, the DBR comprises 6 completely random bases (denoted by `N`), followed by two bases that can either be `A` or `C` (denoted by `M`), and finally the fixed section `GGACG`. Assuming a completely random assignment of bases, there are $4^6 \cdot 2^2 = 16\,348$ possible assignments for this DBR setup. We will describe handling of PCR duplicates in more detail in the section PCR Duplicate Removal (p. 213).

The sequences described until now were all either technical artifacts themselves, or sequences used to cope with technical limitations. However, they do not hold any information about the sequenced individual. The remaining bases in the read are the genomic sequence, sequenced from the p5 and p7 end of the fragments extracted. While the sequences described above, which we will refer to collectively as the partial adapter, are required for (some) ddRAD experiments, they limit the size of genomic sequence that can be analyzed.

Maximal read length is set by the sequencing device used. The remaining payload of genomic sequence is restricted by the length of the partial adapter, which can differ between individuals in a sample. Given the layout shown in Figure 7.5, the lengths of p5 and p7 genomic sequences for an individual I are:

$$\begin{aligned}|\text{p5 genomic sequence}(I)| = {} & \text{Total read length from sequencer} \\ & - (|\text{p5 barcode}(I)| + |\text{p5 spacer}(I)| + |\text{p5 residue}(I)|)\end{aligned}$$

$$\begin{aligned}|\text{p7 genomic sequence}(I)| = {} & \text{Total read length from sequencer} \\ & - (|\text{p7 spacer}(I)| + |\text{DBR}| + |\text{p7 residue}(I)|)\end{aligned}$$

[9] Schweyen, Rozenberg, and Leese, "Detection and Removal of PCR Duplicates in Population Genomic ddRAD Studies by Addition of a Degenerate Base Region (DBR) in Sequencing Adapters", 2014; Tin et al., "Degenerate Adaptor Sequences for Detecting PCR Duplicates in Reduced Representation Sequencing Data Improve Genotype Calling Accuracy", 2015.

Extracting the partial adapter from ddRAD reads is one of the challenges of ddRAD analysis.

7.1.3 Challenges of ddRAD Analysis

The distinctive structure of ddRAD data described above, combined with the requirements of population genetics, as well as biological and technological factors, make ddRAD analysis a uniquely challenging endeavor. It has to solve the following tasks:

Preprocessing Prepare raw reads for analysis, usually includes the following tasks:

- *Demultiplexing* Assemble one FASTQ file per individual in the dataset using barcoding information.
- *PCR Duplicate Removal* Remove PCR duplicates from the FASTQ files using DBRs in order to achieve better genotype information in downstream steps.
- *Adapter Trimming* Trim off parts of the reads that are not genomic sequences.

Locus Identification Reconstruct the loci with similar genomic sequences for each individual.

Genotype Identification Identify genotypes within and between individuals.

These tasks would be relatively simple if there were no biological or technological factors influencing the reads. Most prominently, since for most species no reference genome is available, locus identification has to be performed without a reference as well as with an unknown number of loci and an unknown number of alleles. In short, instead of read mapping, we have to perform read clustering. This means grouping reads with similar sequences while avoiding to *over-merge* loci, that were distinct in the input data. Furthermore, similar loci also have to be identified between individuals. All of this is further influenced by an unknown number of genetic variants, such as SNVs and indels, which in turn can be homozygous or heterozygous.

In the case that a genetic variant affects a restriction site, its effect can extend to whole reads or read pairs. This effect is called a null allele (NA). We distinguish two types of NAs: dropout NAs and alternative sequence NAs.

When a mutation erodes a restriction site, this site is no longer cut during digestion. The resulting fragments from the affected cut site are longer than those from individuals where the restriction site is intact. During size selection, two outcomes are possible:

1. The size of the newly formed fragment is longer than the maximum permissible size in the size selection step. Such fragments are not sequenced and drop out of the dataset, creating a *Dropout NA* for the individual at the affected locus.

2. The newly formed fragment passes size selection and is sequenced. In this case, a different sequence is observed for the mutated restriction site, e.g. if the p7 restriction site is mutated, p7 reads show an alternative genomic sequence while p5 reads show the expected sequence. This creates an *alternative sequence NA*.

Examples for both variants of NAs are illustrated in Figure 7.6. Since the p5 restriction site is a rare cutter, p5 NAs are more likely to be dropout NAs, while p7 NAs have a higher chance to be alternative sequence NAs.

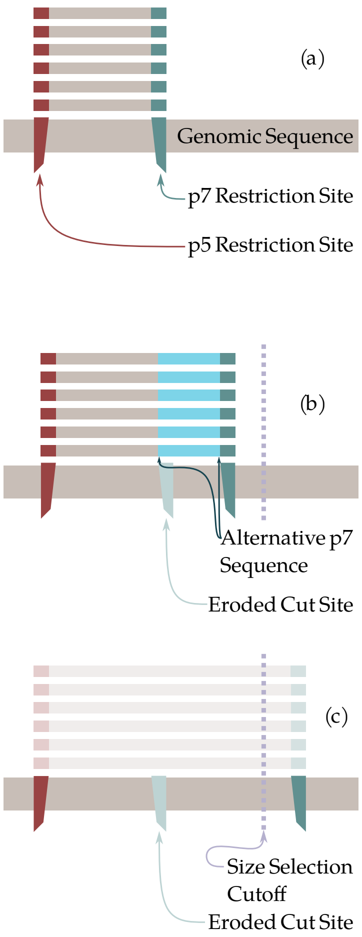


Figure 7.6: Effects of null alleles on the generated fragments, with respect to an unaffected locus (a). For alternative sequence NAs (b), the next p7 cut site ▮ is located close to the eroded one. Generated fragments show an alternate p7 genomic sequence (teal). If the subsequent p7 cut site is further away than the size selection permits, a dropout NA occurs (c).

It is also possible that a new restriction site forms in the genomic sequence. As with eroded restriction sites, their effects depend on their position relative to other restriction sites. Additionally, they can introduce new loci that are not present in other individuals.

As other genetic variants, NAs may be heterozygous or homozygous. Note that an NA caused by a heterozygous mutation of a restriction site might be mistaken for a homozygous mutation with very low (half of the expected) coverage.

Another effect that is specific to ddRAD reads are *highly repetitive loci* (HRLs)[10]. If a restriction site is contained in a highly repetitive region of the genome, reads from several restriction sites can collapse into a single highly repetitive locus (HRL). Consequently, HRLs show a very high read coverage, but their information for biological questions is limited, since they contain inseparably mixed genetic variants from different genomic sites.

While the above mentioned factors are inherent to the biological samples, there are also technological factors determined by the sequencing platforms (e.g. sequencing errors) as well as library preparation protocols (e.g. size selection, number of PCR cycles) that influence the ddRAD output.

Genotype calling is performed based on the abundance of reads that present different alleles. Hence, variations in locus coverage as a result of the sequencing process may result in the incorrect inference of genotypes. In a low coverage scenario, only one allele of a heterozygous locus might be sequenced, resulting in a dropout of the other allele.

Oppositely, PCR duplicates of the sequences may be present in the dataset which can obscure the real coverage and thereby influence the analysis significantly. PCR duplicates can lead to spurious heterozygote calls due to the duplication of PCR errors, or the failure to call heterozygotes due to uneven PCR amplification of variant alleles. Additionally, PCR errors during library preparation or sequencing errors may cause apparent changes in the genomic sequence and may be confounded with individual variation, or they may obscure the individual by changing the barcode.

Finally, incomplete enzymatic digestion (ID) of the genome may result in null alleles or dropouts. These behave similar to effects from mutated restriction sites, since a restriction site not being cut

[10] Also known as highly repetitive stacks or lumberjack stacks.

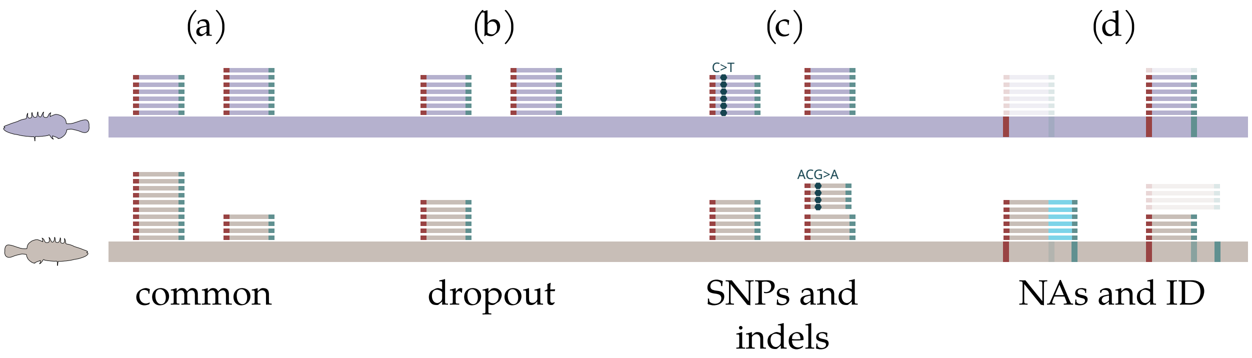


Figure 7.7: Examples for the influence of biological and technological factors on ddRAD reads. For the genomes of two individuals (upper and lower bar), two loci for each effect are shown. Bars in a locus symbolize a fragment, with its p5 adapter (red) and p7 adapter (green). Common loci (a) are present in both individuals, albeit with different levels of coverage. Loci can drop out of the analysis for single individuals or groups (b). SNPs and indels, as well as sequencing errors, can introduce slight differences between read sequences (c). Null alleles (NAs) and incomplete digestion (ID) (d) can move the effective cut sites (indicated by red and green lines on the genomes), causing dropouts (desaturated sequences) or extensions (teal sequences).

by chance, is not distinguishable from a restriction not being cut due to a mutation of said site.

Figure 7.7 visualizes how the effects discussed in this section influence fragment generation and the structure of reads sequenced from these fragments.

7.2 *Simulation of ddRAD Data*

A large variety of tools, such as Stacks [11] or pyRAD[12]/ ipyrad[13] are available to analyze ddRAD data. The two main computational tasks these tools have to solve are

- to generate clusters of reads, that likely originate from the same locus across individuals, and
- to analyze genomic variation per locus within and between individuals.

Due to the biological and technological effects described in the previous section, and in more detail by Mastretta et al.,[14] evaluating optimal parameter values[15] for an analysis is difficult. In the worst case, this has the potential to leave researchers with ill configured analysis pipelines that might obfuscate biological findings.

While the effects present in real ddRAD datasets cannot be easily verified, simulated data can be used to evaluate parameters for an analysis. In order to match the complexity of real ddRAD data, both biological and technological factors need to be simulated. Furthermore, technological factors need to be annotated as such to make them distinguishable from biological factors. A changed base, for example, could be caused by both a SNP or a sequencing error. This ground truth comprises a list of events an analysis tool has to detect, which has to be both as complete as possible and easily accessible for verification.

There are several tools available to simulate various aspects of ddRAD experiments in order to choose good parameters for the analysis. These tools range from read simulation for validation, to complete simulations of ddRADseq pipelines.

The simrrls software,[16] which is used to test pyRAD, simu-

[11] Catchen et al., "Stacks: an Analysis Tool Set for Population Genomics", 2013; Rochette, Rivera-Colón, and Catchen, "Stacks 2: Analytical methods for paired-end sequencing improve RADseq-based population genomics", 2019.

[12] Eaton, "PyRAD: Assembly of de novo RADseq Loci for Phylogenetic Analyses", 2014.

[13] Eaton and Overcast, "ipyrad: Interactive Assembly and Analysis of RADseq Datasets", 2020.

[14] Mastretta-Yanes et al., "Restriction Site-associated DNA Sequencing, Genotyping Error Estimation and de novo Assembly Optimization for Population Genetic Inference", 2015.

[15] A fact well illustrated by Paris, Stevens, and Catchen ("Lost in Parameter Space: A Road Map for STACKS").

[16] github.com/dereneaton/simrrls

lates (dd)RAD reads, but does not provide a detailed account of the simulated effects. The FRAGMATIC[17] software performs *in-silico* digestion of reference genomes closely related to the target genome to predict number and characteristics of target loci. The R library SIMRAD[18] also allows users to simulate the digestion process on both real and simulated reference genomes, enabling the prediction of the number of target loci with different protocols to optimize a ddRAD experiment. DDRADSEQTOOLS[19] simulates a ddRAD pipeline to support the optimal design of ddRAD experiments. Finally, the RADINITIO[20] software also simulates a ddRADseq pipeline for a given reference genome, while focusing on biologically correct simulation than on easy and detailed verification. However, it currently offers no options to simulate effects like HRLs and NAs, other than providing a reference genome tailored to contain such effects, which is unwieldy for simulation purposes.

We have developed DDRAGE, the ddRAD Data Generator, which simulates ddRAD reads based on a model, either from a reference genome or from a simulated genome. For each locus, influencing biological and technological effects, including mutations, coverage variations, dropouts, and sequencing errors, are chosen using mathematical models. Given a set of simulated reads and a detailed description of the effects that are detectable in this dataset, analysis software can be tested for specificity and sensitivity, to further improve their performance concerning questions such as:

- Are PCR duplicates removed from the analysis reliably during preprocessing?
- Are all loci identified and reconstructed correctly during read clustering?
- Are loci with indel mutations assembled correctly or split up?
- Are highly repetitive loci identified and removed?
- Which minimal coverage is required to reliably detect heterozygous mutations?
- How many SNPs make a locus split up using the applied read clustering method?
- Is the applied SNP calling approach robust? Are sequencing errors mistaken for SNPs or are SNPs uncalled?

In contrast to other software like RADINITIO,[21] DDRAGE does not aspire to provide a detailed phylogenetic simulation or to simulate different populations, since such simulations would not significantly enhance the ability to validate analysis tools. To achieve this, DDRAGE creates a set of output files, each covering a different aspect of the generated data. A detailed specification of all output file types can be found in the documentation.[22]

[17] Chafin et al., "FRAGMATIC: in silico Locus Prediction and Its Utility in Optimizing ddRADseq Projects", 2017.

[18] Lepais and Weir, "SimRAD: an R Package for Simulation-Based Prediction of the Number of Loci Expected in RADseq and Similar Genotyping by Sequencing Approaches", 2014.

[19] Mora-Márquez et al., "ddRADseq-Tools: a Software Package for in silico Simulation and Testing of Double Digest RADseq Experiments", 2017.

[20] Rivera-Colón, Rochette, and Catchen, "Simulation with RADinitio Improves RADseq Experimental Design and Sheds Light on Sources of Missing Data", 2020.

[21] Rivera-Colón, Rochette, and Catchen, "Simulation with RADinitio Improves RADseq Experimental Design and Sheds Light on Sources of Missing Data", 2020.

[22] Available at `ddrage.readthedocs.io/en/latest/documentation/output_format/` and as part of the source code (Timm and Rahmann, *ddRAGE — ddRAD Data Generator (Source Code)*, 2020, documentation.zip).

7.2.1 *Using* DDRAGE *to Simulate Reads*

DDRAGE is implemented as a command line tool and can be installed using the conda package manager[23] and the Bioconda[24] channel:

```
$ conda install -c bioconda ddrage
```

After installation, calling the program ddrage from the command line without parameters will simulate a small dataset containing reads from three individuals at three loci.

As input, DDRAGE needs the number of loci and individuals, a barcodes file, and the remaining partial adapter sequences (degenerate base sequence, enzyme residues and restriction sites). Instead of the number of loci, a FASTA file containing sequence fragments can be passed to DDRAGE to simulate reads from a reference genome. The barcode file is a text file containing barcode and spacer sequence pairs for each individual. As default, a barcode set utilized in the Illumina TrueSeq LT kit is used, which offers a distance of at least two bases between all barcodes to minimize the chance of collision through sequencing errors. However, other barcode sets allowing more individuals and variable length barcodes are also supported. For all input parameters reasonable default values are provided.

The generated reads are saved as annotated FASTQ files. In addition to basic FASTQ entries (header in CASAVA format, sequence, plus line, quality values), the header of each read contains a manifest of the effects added to the read. Using this information, for example, the difference between a SNP and a sequencing error can be inferred directly from the read name.

The FASTQ files contain the generated reads in the order of their creation: first all reads from valid loci, followed by PCR duplicates for each individual, and last all singletons and HRL reads, also including their PCR duplicates. While this is useful for basic validation of an analysis workflow, it is advised to shuffle the reads before analysis using the script randomize_fastq to more closely mimic a real ddRAD dataset:

```
 $ randomize_fastq in_1.fq in_2.fq out_1.fq.gz out_2.fq.gz
```

In this example, in_1.fq and in_2.fq are the files generated by DDRAGE. The randomized reads will be written to the gzipped output files out_1.fq.gz and out_2.fq.gz.

In addition to the read data, DDRAGE generates two log files: an annotation file (text) and a statistics file (PDF). The annotation file provides a first glance view of the created dataset. It contains the parameters used to create the dataset and a concise collection of measures to assess the generated dataset, including the total number of generated reads, PCR copy rate and number of valid reads. The statistics file is a PDF document with figures illustrating the structure of the created data. These plots include a breakdown of

[23] conda.io/docs/

[24] Grüning et al., "Bioconda: Sustainable and Comprehensive Software Distribution for the Life Sciences", 2018.

the read origins, events types per individual, the mean number of mutations per individual, and PCR duplicate rates. A barcode file containing a list of individuals in the dataset and their associated barcodes and spacer sequences is written as well.

Finally, a ground truth file is written with which the results of analysis tools can be compared. This ground truth file, which is written in YAML format to be easily accessible and convertible into other formats, contains three sections: individual information, loci, and HRLs. The individual information section contains entries describing the partial adapter sequences of all individuals in the sample. In the second section, locus entries each describe one simulated locus. They contain the consensus sequences of the locus, assigned alleles alongside their coverage and frequency at the locus for each individual, the total coverage of the locus, and the total number of ID reads. The last section contains an entry for each HRL locus, which contains a coverage entry for each individual.

If we use a ddRAD analysis software to analyze the FASTQ files, we receive results which we can validate using the ground truth YAML file. The profile of detected and undetected effects can give us insight on how to improve the parameters chosen for the analysis software.

7.2.2 *Simulation Workflow*

We will now describe the simulation workflow ddRAGE uses to simulate reads under the following three assumptions:

1. The simulated genome is diploid. Hence, an individual can only have either one allele per locus (homozygous) or two alleles per locus (heterozygous). Since the majority of animals analyzed in population genetics is diploid, this is not a severe limitation.

2. The target sequencing depth d_s is constant across the genome. Due to this, the average coverage of loci, meaning the number of reads simulated for one individual at one locus (without PCR duplicates), varies only statistically because of sampling effects. The total coverage of an individual i at a locus ℓ is the sum of the two allele coverages. An expected coverage value of $cov_{i,\ell} = d_s$ is desired for the simulation.

3. Simulated genomic sequences follow an i.i.d. (independently and identically distributed) model. Each nucleotide is drawn independently from the others from the same distribution. By default, genomic sequences are simulated with a `GC`-content of 0.5, but this can be changed using the `--gc-content` parameter.

Since the main challenges of ddRAD analysis are the reconstruction of loci and the analysis of said loci, the simulation approach of ddRAGE is focused on loci. To accurately mimic real ddRAD data, ddRAGE simulates individual genetic variation using a locus-based phylogenetic approach and (partial) adapter sequences induced by the technological constraints. In contrast to simulating

Algorithm 4: ddRAGE's simulation process

```
▷ Initialization
• Create $n$ individuals $\mathcal{I} := \{i_1, \dots, i_n\}$, choosing barcodes
  and spacers for each one
• Create $m$ loci $\mathcal{L} := \{\ell_1, \dots, \ell_m\}$:
foreach locus $\ell \in \mathcal{L}$ do
    • Simulate p5 and p7 genomic sequences or read them
      from FASTA file
    • Compute mutation tree for $\ell$
    foreach individual $i \in \mathcal{I}$ do
        • Create template read for $i$ using partial adapter and
          genomic sequences
▷ Read Simulation
foreach locus $\ell \in \mathcal{L}$ do
    foreach individual $i \in \mathcal{I}$ do
        • Pick individual event type from
          $\{common, mutation, dropout\}$
            and its mutation zygosity, alleles, and coverage
        • Create reads from individual template according to
          picked event type
        • Finalize reads by fixating the DBR, adding PCR
          duplicates, incomplete digestion and sequencing
          errors
        • Write reads to FASTQ files and events to YAML file
▷ Post-processing
• Simulate and write singletons and reads from highly
  repetitive loci
• Write statistics to log file and figures to PDF file
```

a whole genome and then creating loci by *in-silico* digestion, our approach allows the user to precisely specify the number of desired loci and avoids the simulation of unused sequences. The simulation process (see Algorithm 4) is structured into three phases:

- Initialization Phase
- Read Simulation Phase
- Post-Processing Phase

In the *initialization phase*, the structure of the dataset is established. As prerequisites, it requires a barcode file which contains a mapping of individuals to barcode pairs and spacer sequences. A set of individuals, each represented by a pair of barcodes and a pair of spacer sequences, is chosen from the barcode file. For each locus to be simulated, a genomic sequence is generated (either from a simulated fragment or using a sequence read from a FASTA file). This sequence is used to generate a template read for each individual, which is used to create reads in the subsequent read simulation

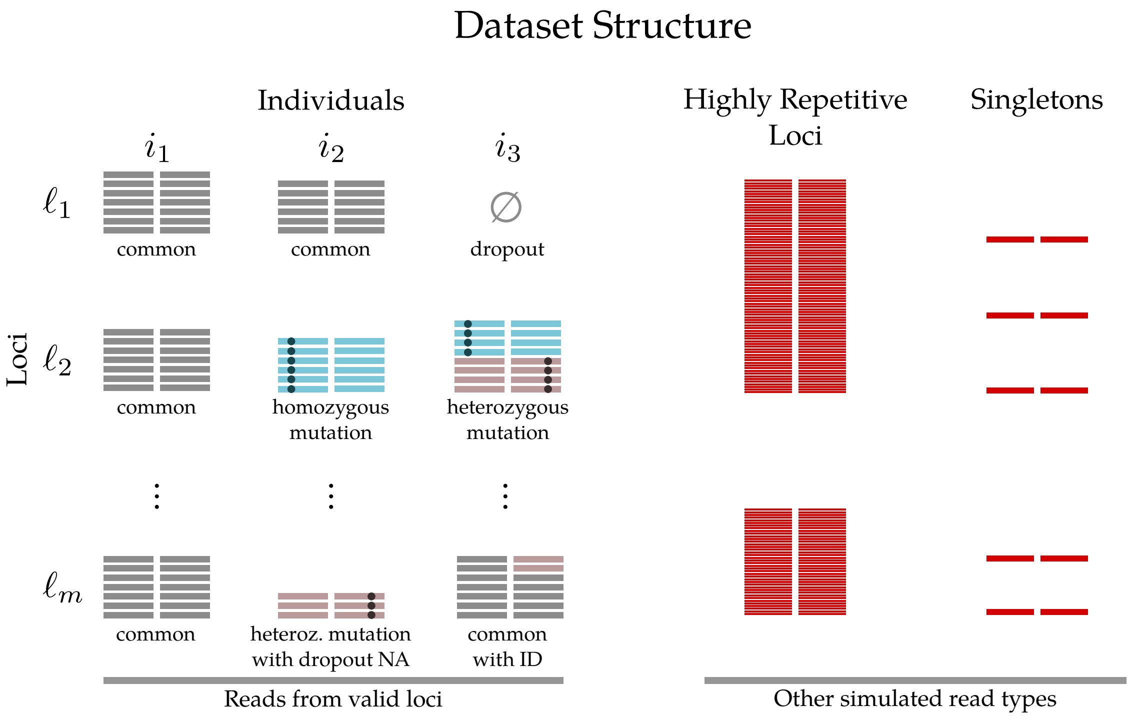


Figure 7.8: Structure of datasets generated by DDRAGE. A dataset contains valid reads, simulated for each individual at each locus (left) and reads that are not associated with valid loci (right). Each pair of lines signifies one paired-end read; the text below describes the event type of the individual at the locus. For each read pair, PCR copies and sequencing errors are added subsequently.

step. Additionally, a mutation tree, comprising relations between possible alleles that can occur at the locus, is simulated. Each allele in the tree can contain one or more mutations of the common sequence. Based on this foundation, read simulation for each locus can begin.

In the *simulation phase*, the effects to be simulated for each individual at each locus are chosen. For each individual, the sequence at the underlying locus is modified according to the randomly drawn individual event for that locus (common –– no mutation/modification, mutation, dropout), before generating reads for that individual at that locus (see next section for additional details on locus and read simulation). After the valid reads have been created, reads affected by incomplete digestion are added. Additionally, simulated PCR copies are added to the locus. As a final step, sequencing errors are simulated for all generated reads and the reads are written to the FASTQ files, while the simulated events are recorded in the YAML file. An example for a dataset with three individuals is illustrated in Figure 7.8.

Finally, in the *post-processing phase*, reads from other sources are added. These are singletons, single reads that do not share the sequence of any locus, and reads from highly repetitive loci (HRL). Reads added in this step cannot be interpreted in a meaningful biological way.

In the following sections we describe the effects simulated as part of these phases.

7.2.3 Locus and Read Simulation

ddRAGE simulates three different types of events that can occur for an individual at a locus: *common*, *dropout* and *mutation*. This event type determines what kind of reads will be simulated and how the coverage is sampled. For each individual at each locus, events are chosen independently. Hence, different individuals can show different event types and the type information for one individual conveys no information about other individuals at the locus. The probabilities for the different event types can be changed using the `--event-probabilities` parameter. We now discuss each event type in detail.

Common A *common* individual only has reads that do not deviate from the expected genomic sequence. No mutations are present and all reads are coverage copies. Apart from sequencing errors, which are added in the post-processing phase, and reads resulting from incomplete digestion (ID), the simulated reads from a *common* event are identical to the consensus sequence.

Dropout A locus may not be present at all (dropout) for some individuals e.g. because of library preparation artifacts.[25] While mutations of the p5 and p7 restriction sites can also increase or decrease the fragment length, preventing valid fragments from being formed, this effect is modeled as part of the mutation event type. An individual with a *dropout* event does not have any reads from the locus in question.

Mutation(s) This event type denotes the presence of mutated alleles at the current locus for the current individual. Alleles are chosen from the mutation tree that was created for each locus in the preprocessing phase, so that all individuals at the same locus draw from the same pool of alleles. An allele comprises a set of mutations, namely SNPs, indels, and NAs, which are applied to the given template reads. For example an allele could contain a `C>T` polymorphism at position 21 of the p5 read, a `G>C` polymorphism at position 25 of the p5 read, and a deletion of three bases after position 17 in the p7 read. The structure of the simulated mutations and the employed tree process that generates the mutations are described in the section Mutation Tree (p. 177).

Two subtypes of this event are distinguished: homozygous and heterozygous; their relative probabilities are user-defined. This allows, for example, the (unrealistic) generation of a dataset with purely homozygous genotypes, which can be useful in a simulated benchmark dataset to test particular features of the analysis tools.

For the homozygous event, one allele is chosen from the mutation tree of the locus. The locus' template sequence is transformed for this individual according to the mutations. Hence, all reads gen-

[25] Mastretta-Yanes et al., "Restriction Site-associated DNA Sequencing, Genotyping Error Estimation and de novo Assembly Optimization for Population Genetic Inference", 2015.

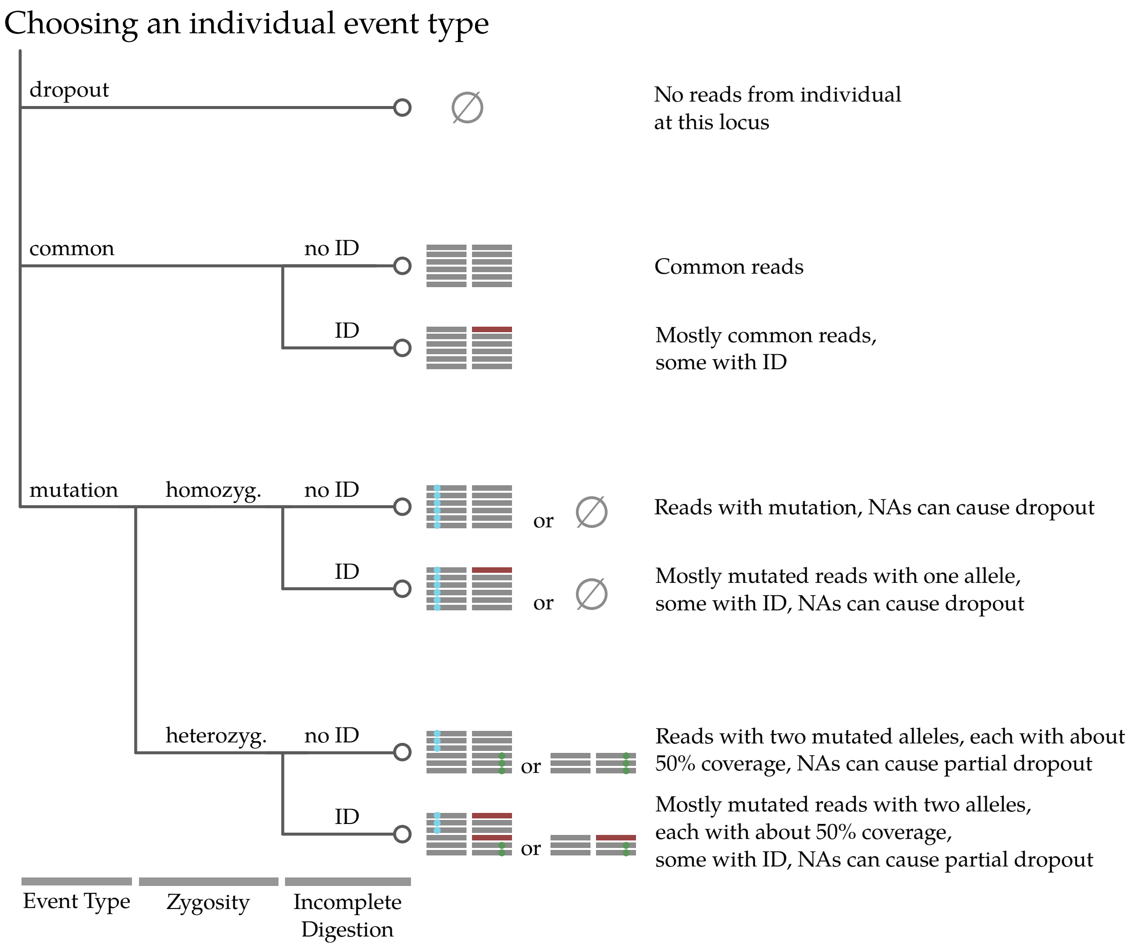

Choosing an individual event type
dropout
No reads from individual at this locus
common
no ID
Common reads
ID
Mostly common reads, some with ID
mutation
homozyg.
no ID
or
Reads with mutation, NAs can cause dropout
ID
or
Mostly mutated reads with one allele, some with ID, NAs can cause dropout
heterozyg.
no ID
or
Reads with two mutated alleles, each with about 50% coverage, NAs can cause partial dropout
ID
or
Mostly mutated reads with two alleles, each with about 50% coverage, some with ID, NAs can cause partial dropout
Event Type
Zygosity
Incomplete Digestion

Figure 7.9: Depending on the event type, a different profile of reads will be created. For all types that create reads, reads with incomplete digestion (ID, shown in red) can occur. If a mutation event occurs, the zygosity of the mutation is also determined.

erated by this event share the same mutations and differ from the consensus sequence.

For the heterozygous event, two different alleles are chosen; here, the common allele, i.e. the unchanged consensus sequence, is also considered. Reads are then generated according to the two selected alleles. The coverage is divided between the two alleles using a binomial distribution. If coverage is assigned to a dropout NA, no reads will be generated for the allele and the coverage is lost.

Once the event type has been chosen, additional locus effects are applied with probabilities defined by the user. These include PCR duplicates, sequencing errors, etc. and are described in the section Additional Locus and Dataset Effects (p. 180). Most notably at this point, for all events that generate reads, there is a chance for incomplete digestion, which can introduce new p7 sequences in a manner similar to NAs. The complete decision tree illustrating all possible combinations of events, including incomplete digestion, is illustrated in Figure 7.9. Figure 7.10 illustrates the expected distribution of reads using default parameters.

In the following sections we describe how mutations are simulated and how they affect simulated reads, which additional effects

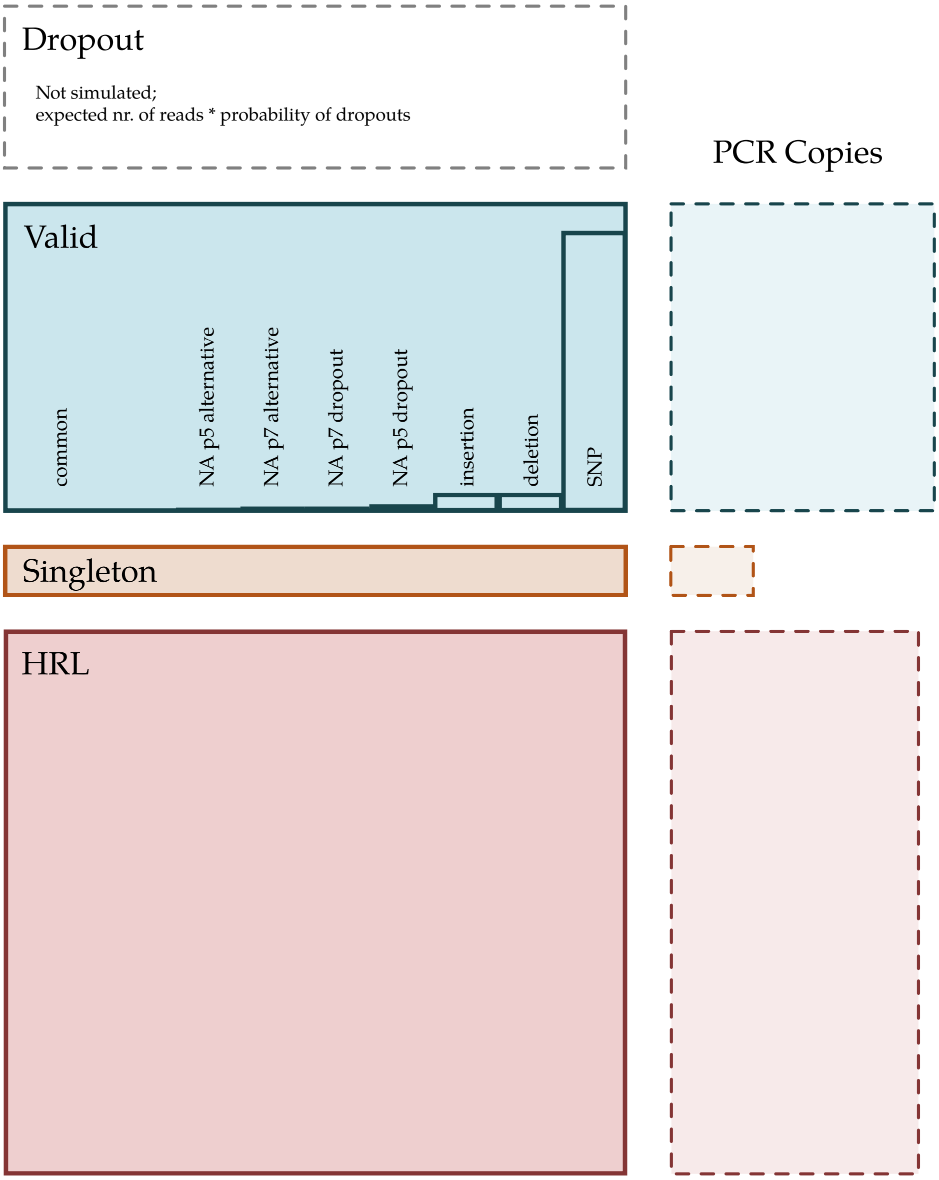


Figure 7.10: Structure of the simulated datasets using standard parameters visualizing the expected distribution of simulated reads for all loci and all individuals as area of rectangles. Each rectangle symbolizes one specific kind of reads. Valid reads comprise reads from *common* events (most reads) and from *mutation* events, with the expected amount of reads from each mutation type symbolized as bars on the right of the valid read area. PCR duplicates are shown in a lighter color in the right column. The probabilities of NA mutations are not to scale to make them visible in this graphic.

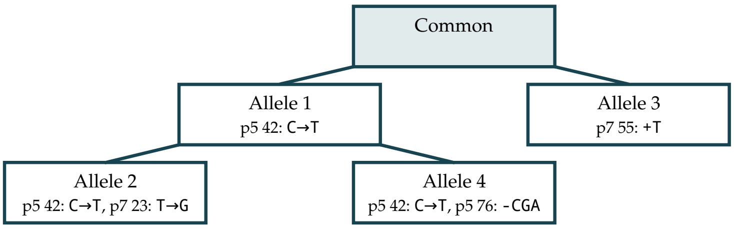


Figure 7.11: Mutation tree with five alleles, including the common allele. Allele 4 is a child of Allele 1 and hence contains all mutations of Allele 1 plus a deletion. Hence, the total mutations of Allele 4 are one substitution of `C` for `T` at position 42 of the p5 read and the deletion of the sequence `CGA` after position 76 in the p5 read.

are simulated, and how the coverage of simulated loci is determined.

7.2.4 Mutation Tree

Mutation events simulated by DDRAGE follow a tree based model, from which alleles are chosen to form a genotype. This process has several parameters that need to be taken into account. These include the total number of different genotypes that can be observed at a locus, the number and type of mutations per allele, and the structure of the different alleles. DDRAGE computes a mutation tree for each locus, which contains a selection of possible alleles each of which comprises one or more mutations. Alleles are stored as a set of modifications to the original locus sequence, for example a `C>T` polymorphism at position 17 of the p7 read is stored as `p7 17:C>T`. From this model, alleles are randomly chosen to create a genotype for the *mutation* event.

During the initialization phase, a mutation tree is constructed for each locus. Its base is a common allele, representing the unmodified locus sequence. Building on this sequence, different alleles are added in an iterative process: An already existing allele is randomly selected from the tree, to serve as parent for the new allele. To this, a new child with an additional mutation[26] is added, until the desired number of alleles is reached. Due to this construction method, the alleles in the model can share some similarity with each other. Alleles from the same subtree share common mutations, while alleles from two different subtrees might be disjoint. The structure of the resulting model is illustrated in Figure 7.11.

To make sure that the added mutations can be detected in the analysis, the last bases, which might be cut off due to variability in the p5 spacer sequences, are excluded from the model. By this, mutations of the last bases of a sequence are prevented from being truncated. Additionally, a position cannot be affected by several mutations at once. For example, if a SNP has been added to an allele, no deletion that removes the mutated position can be added. The one exception from this are null allele mutations, which change the whole p5 or p7 sequence. NA mutations overwrite existing mutations or make the read drop out completely. Additionally, no new mutations will be generated on null alleles.

The number of different mutations that are added to the model can be changed using the `--diversity` parameter, while the chance that a *mutation* event is chosen during simulation can be altered

[26] SNPs, indels, NAs; For a full list see the following section.

with the `--event-probabilities` parameter. The relative abundance of the different mutation types is controlled by the `--mutation-type-probabilities` parameter.

To create genotypes for homozygous and heterozygous *mutation* events, alleles are uniformly chosen from the mutation tree. The tree structure is not considered at this point, so alleles from different subtrees are chosen with equal probability.

For a homozygous genotype, one allele other than the common allele is selected to assure that a detectable mutation is present. To simulate a heterozygous mutation, two different alleles are chosen from the model. One of these alleles can be the common allele, since it is possible that only one allele deviates from the expected sequence. The genotype for the individual with the *mutation* event is saved as the combination of the selected alleles.

The diversity of mutations at the same locus is influenced by the total number of alleles present in the model. Assuming two homozygous mutation events (for two different individuals), identical alleles are chosen with probability $1/|\mathcal{A}|$, where $\mathcal{A}$ is the set of all non-common alleles. Hence, by increasing the number of alleles in the mutation tree through the `--diversity` parameter, the expected number of different mutated positions per locus is increased.

7.2.5 *Number of Alleles per Locus*

The number of alleles that are added to the mutation tree is chosen from a zero-truncated Poisson distribution (ZTPD) (see Probability Distributions, p. 34f). This ensures that if a mutation occurs at least one allele, in addition to the common allele, is present. The `--diversity` command line parameter is used as the parameter λ for the ZTPD. Hence, the expected number of different alleles can be influenced by the user and is a measure for the simulated biological diversity, as it increases the pool of alleles to sample from. With the default value of `--diversity 1.0`, the expected number of available mutated alleles per locus is 1. In this case, for heterozygous mutations there is only one possible genotype (the common allele and the mutated allele). If the diversity value is increased, more combinations are possible. For example with a diversity value of 10, the expected number of mutated alleles is 10. Together with the common allele, there are $\binom{10+1}{2} = 55$ possible genotypes, from which one is chosen uniformly at random.

By increasing the diversity value, the expected number of mutations per allele is implicitly increased as well. Hence, datasets simulated with a high diversity value are expected to show both a higher number of different alleles and a higher number of mutations per allele.

7.2.6 *Mutation Types and Their Effects on Simulated Reads*

Each mutation added to a mutation tree can be one of the following mutation types. For a description of the biological effects of SNPs and indels, see Section Genomic Mutations.

SNPs Point mutations, substitutions of a single base in the read, are the most common mutation type simulated by ddRAGE. Using default parameters, ~90% of the added mutations are SNPs. A randomly selected base in the read is changed to a different base.

Insertions and Deletions An insertion is the addition of one or more bases that are not present in the common sequence, while a deletion describes the absence of bases that are present in the common sequence. These two behave similarly and are collectively described as indels. Using default parameters, 5% of the added mutations are insertions and 5% are deletions.

To use reasonable lengths for insertions and deletions (e.g. in coding regions, the length is frequently a multiple of three, but coding regions are rare), we use a published table of the empirical length distribution of insertion and deletion lengths in the Icelandic human population.[27] The inserted sequence for an insertion mutation is randomly chosen. It matches the GC-content of the read and does not contain any restriction sites.

Null Alleles NAs arise when a restriction site in an individual mutates, preventing the restriction enzyme from cutting the DNA at that position. Based on the kind of restriction site (p5 or p7) and the surrounding sequence, this has different effects, as described in the section Challenges of ddRAD Analysis (p. 166).

Dropout NAs on either the p5 and p7 side make the whole allele drop out of the simulation. If a dropout allele is chosen for an individual, no reads are generated and the assigned coverage is lost. For a homozygous mutation, this has a similar effect as a dropout event, while for a heterozygous mutation only half of the assigned coverage is lost.

If the fragment generated by an alternative restriction site fits the size restriction, reads are generated, but one of both read sequences does not match the old one. Such alternate sequence NAs results in p5 and p7 (depending on the location of the NA) genomic sequences that do not match the consensus sequence of the locus.

For an alternative sequence NA mutation, the whole p5 or p7 sequence of the read is replaced with a different sequence. These alternate sequences are the same for all individuals at the locus. After an alternative sequence NA mutation has been added, no other mutations can be added to the affected side of the read (e.g. after a p5 alternative sequence NA has been added, no additional mutations are added to the p5 read but can be added to the p7 read).

[27] Gudbjartsson et al., "Large-scale Whole-genome Sequencing of the Icelandic Population", 2015.

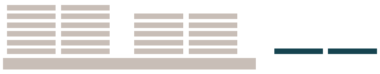
Using default parameters, 10^{-4} of the added mutations are NAs (i.e. the sum of all NA mutation types makes up $\frac{1}{10000}$ of all mutation events). This fraction is a rough estimate of the probability that a mutation in the simulated sequences hits the restriction sites. As default 89.9% of NAs simulated are p5 dropout NAs, 0.1% are p5 alternative sequence NAs, 5% are p7 dropout NAs, and 5% are p7 alternative sequence NAs. These fractions can be changed using the last four values of the `--mutation-type-probabilities` parameters.

7.2.7 Additional Locus and Dataset Effects

A number of additional effects increases the complexity of the dataset. These all have in common that they obscure the observable events, for example by altering the coverage of a valid locus, and should therefore be accounted for. Some of these are specific to ddRAD, like HRLS or ID, while others are present in most sequencing data, including sequencing errors and PCR duplicates. The following effects are simulated by ddRAGE:

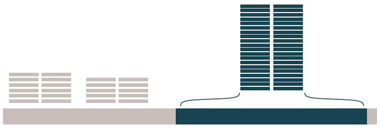
Incomplete Digestion (ID) A simulated locus can be subject to incomplete digestion, the event that a restriction site stays undigested (uncut). As with NAs, if the resulting fragment is too long, the locus drops out due to the size selection in the ddRAD pipeline. If the resulting fragment is short enough, different p5 or p7 sequences are presented by the reads. Since the p5 restriction enzyme is a rare cutter, only 1% of simulated ID events affect the p5 side of the reads.

In contrast to NAs (and dropout events), which have this effect on all reads from one allele of an individual at this locus, ID occurs randomly and not systematically and may occur when valid reads are present. For example, a *common* locus may suffer from incompletely digested reads (with a user defined probability). After the true coverage for the event has been determined, about 20% (default) of the coverage is removed due to ID reads (determined by drawing from a Poisson distribution).



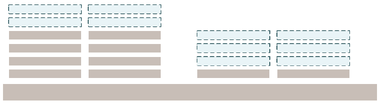
Singleton Reads Most ddRAD datasets will contain singleton reads, which cannot be associated with any locus of the source DNA. They can be introduced by different factors, like contamination of the samples with non-target DNA, incomplete size selection, or errors during the sequencing pipeline. Hence, singleton reads are noise that hinder the analysis.

ddRAGE simulates singletons by adding a set of reads that do not share a genomic sequence with any of the generated loci. They can also be subject to PCR duplicates, but the PCR duplicate rate for singletons is by default lower than for valid reads. This rate can be changed using the `--singleton-pcr-copies` parameter. Note that a singleton, in combination with its PCR duplicates, might pass as reads from a valid locus.



Highly Repetitive Loci (HRLs) When a restriction site falls into a highly repetitive region of the source genome, many fragments with a similar genomic sequence can be generated from different loci. The different origin loci of these reads cannot be distinguished in the analysis and collapse into one big HRL. Reads from HRLs are typically not analyzed further, as detected mutations cannot be assigned to a specific locus. However, HRL reads can make up a significant part of real ddRAD datasets.

HRLs are simulated by DDRAGE as additional loci with a higher coverage, which are added after all reads from valid loci have been created. The number of HRL loci simulated can be specified as a fraction of the number of valid loci using the `--hrl-number` parameter. The default value is 0.05, hence for 20 valid loci one HRL locus is added. All other steps that are applied to valid reads are applied to HRL reads as well. They also receive PCR duplicates, albeit by default with a lower PCR duplicate rate than normal reads. This rate can be changed using the `--hrl-pcr-copies` parameter. HRLs are, however, not added to the list of valid loci in the output files.



PCR Duplicates PCR duplicates are copies of reads that result from necessary PCR amplification steps during library preparation (see Section Polymerase Chain Reaction, p. 20). They distort the analysis of allele frequencies and should be removed prior to the analysis. Methods for the removal of PCR duplicates from ddRAD data have been proposed by Schweyen et al.[28] and Tin et al.[29] Both rely on adding a degenerate base region (DBR) to the fragments, which can be used as an identifier for unique fragments. A DBR is given as a pattern of possible base types represented as a sequence of IUPAC ambiguity codes. During library preparation, a sequence of fitting structure is generated and added to the reads prior to PCR amplification, thus PCR duplicates contain an exact copy of the fragments original DBR. If two reads with similar sequence also have the same DBR, they can be assumed to be PCR duplicates of each other (with a small false positive rate due to randomly identical sequences).

DDRAGE simulates PCR duplicates both for valid locus reads and other reads like singletons and HRL reads. After simulating all independent reads (by replacing the DBR with a random matching concrete nucleotide sequence from Σ_{DNA}, PCR duplicates are created by inserting a random number of copies of randomly selected reads. The probability of a read having a PCR duplicate can be adjusted using the `--prob-pcr-copy` parameter.

Sequencing Errors Sequencing errors are simulated and added to all generated reads as a final step. The applied sequencing technology determines which sequencing errors can occur in the reads. As the ddRADseq pipeline is tailored towards Illumina sequencing platforms, the characteristic error model of Illumina sequencers is

[28] Schweyen, Rozenberg, and Leese, "Detection and Removal of PCR Duplicates in Population Genomic ddRAD Studies by Addition of a Degenerate Base Region (DBR) in Sequencing Adapters", 2014.

[29] Tin et al., "Degenerate Adaptor Sequences for Detecting PCR Duplicates in Reduced Representation Sequencing Data Improve Genotype Calling Accuracy", 2015.

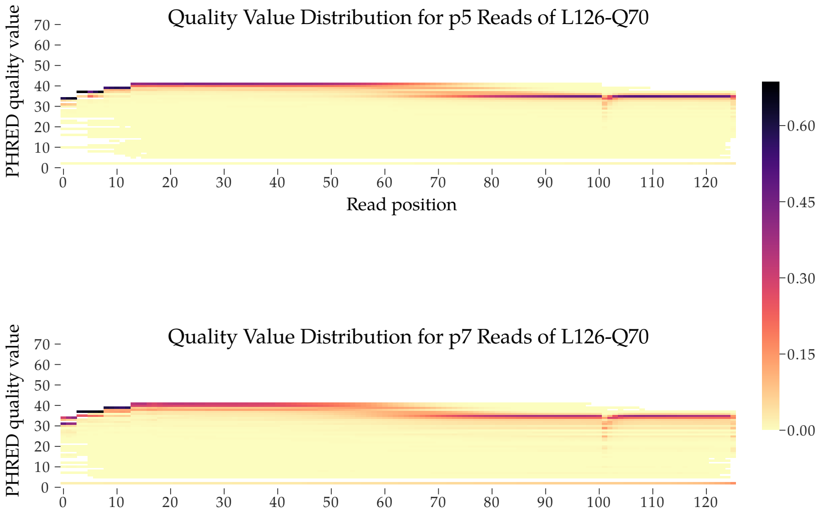


Figure 7.12: Distribution of quality values learned from the in-house ddRAD dataset L126-Q70, sequenced on an Illumina HiSeq 2500. The color of each bin denotes the relative abundance of a quality value per position (i.e. per column). The darker a bin is, the more likely it is to be chosen when sampling quality values.

used. Specifically an error rate of $p_e \approx 0.01$ for substitution errors is assumed for all bases in the read. This can be changed using the `--prob-seq-error` parameter.

Each error is guaranteed to change the base in the sequence; for example, a sequencing error changing `A` to `A` is not possible. The p7 index barcode, which is saved in the FASTQ name line of the read and is not part of the read sequence, is also affected by this step. Simulated sequencing errors are logged in the FASTQ name line of the read using read positions.

7.2.8 Quality Values

Quality values for the generated reads are chosen from a position-specific distribution learned from several real ddRAD datasets. The distribution of quality has been extracted by computing the relative abundance of all Phred values per read position. We used three in-house ddRAD datasets (L126-Q70, L100-Q70-A, L100-Q70-B) and one publicly available dataset (L150-Q70; extracted from NCBI SRR5424823) all of which were sequenced on Illumina sequencers. The resulting distribution for L126-Q70 is illustrated in Figure 7.12.

When assigning quality values to a simulated read, for each position the quality value is chosen using the position specific distribution (i.e. a column in Figure 7.12).

The four models listed above can be used with the `-q <qmodel-name>` parameter. For example:

```
$ ddrage -q L150-Q70
```

To use the quality value distribution of custom datasets, distribution can be learned using the script `learn_qmodel`. This script is installed along with DDRAGE. It takes one or more FASTQ files as input and creates a qmodel file as output:

```
$ learn_qmodel -1 my_dataset_1.fq -2 my_dataset_2.fq -o my_dataset.qmodel
$ ddrage -q my_dataset.qmodel.npz
```

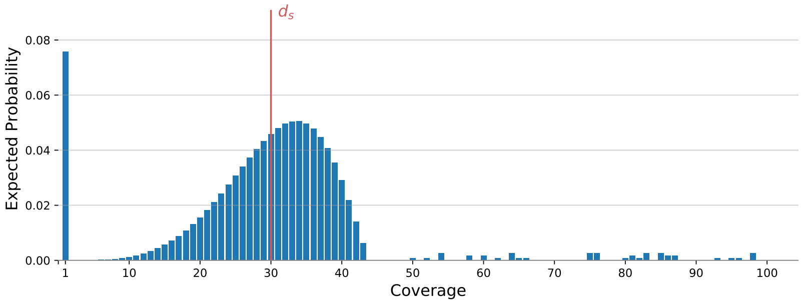


Figure 7.13: Simplified coverage distribution of ddRAD data for one individual. The red line denotes the target sequencing depth d_s. The x-axis has been truncated at coverage 100 to increase readability, since coverage values higher than 1000 may be observed.

Alternatively, a model can also be provided as a matrix (in numpy format), as described in the documentation.[30]

7.2.9 *Coverage Simulation*

The coverage, i.e. the number of reads simulated for a specific individual at a locus before any PCR duplicates are added, is simulated as a function of the target sequencing depth $d_s \in \mathbb{N}_+$. DDRAGE uses several discrete probability distributions to create a realistic distribution of reads.

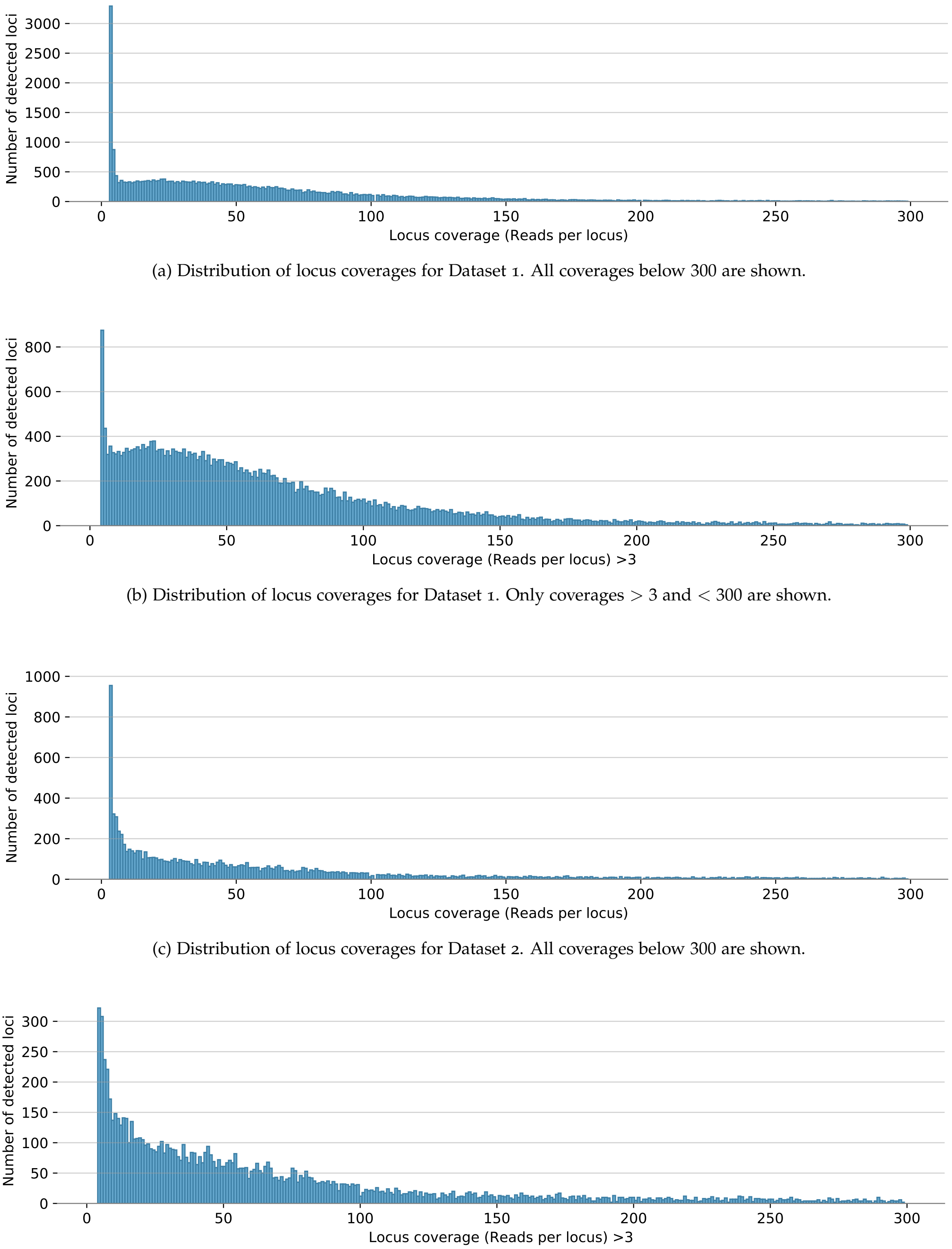
A simplified coverage profile of ddRAD data derived from in house datasets, as illustrated in Figure 7.13, shows three distinct influencing factors: singletons, HRLs and valid loci. By definition, coverage for singletons is one, so no special coverage distribution is needed for them. When considering the coverage of all loci however, singletons create a substantial peak due to their abundance. Reads from HRLs vary widely in coverage and can reach coverage values beyond 1000. In Figure 7.13, they are visible as the scattered low probability values beyond coverage 50. The peak generated by valid loci, which falls between the coverage values of 2 and 40 in Figure 7.13, is left-skewed and mostly falls below d_s with only a few values above d_s.

However, the analysis of ddRAD data has shown also that this pattern is often heavily distorted in real datasets. As illustrated in Figure 7.14, coverage might follow a right-skewed distribution without distinguishable second peak. Due to this pattern of coverages, a special coverage model is applied for each of the three locus types (singleton, valid, HRL). The coverage model for valid loci needs to be adaptable to different coverage profiles.

We denote the total coverage for an individual $i \in I$ at a locus $\ell \in L$ as $\text{cov}_{i,\ell} = \text{cov}_{i,\ell}(a_1) + \text{cov}_{i,\ell}(a_2)$, where a_1 and a_2 are the alleles assigned to individual i at locus ℓ.

Poisson Coverage Model The Poisson coverage model uses a Poisson distribution (PD) with a λ parameter that is dependent on d_s. To

[30] Available at `ddrage.readthedocs.io/en/latest/documentation/input_format/#quality-model` and as part of the source code (Timm and Rahmann, *ddRAGE — ddRAD Data Generator (Source Code)*, 2020, documentation.zip).



(a) Distribution of locus coverages for Dataset 1. All coverages below 300 are shown.

(b) Distribution of locus coverages for Dataset 1. Only coverages > 3 and < 300 are shown.

(c) Distribution of locus coverages for Dataset 2. All coverages below 300 are shown.

(d) Distribution of locus coverages for Dataset 2. Only coverages > 3 and < 300 are shown.

Figure 7.14: Coverage profiles per locus (sum of all individuals) derived from two ddRAD datasets, analyzed with STACKS using default parameters. Loci with a coverage above 300 have been truncated in all plots to increase readability. The two upper plots show Dataset 1, and contain all sizes (a) and sizes >3 (b), while the two lower plots show Dataset 2, and contain all sizes (c) and sizes >3 (d). Note that through the default parameters of STACKS, specifically a minimum stack depth (`-m`) of 3, there are no loci with 1 or 2 reads. PCR duplicates in the data have the ability to smoothen peaks, like those shown in Figure 7.13, making them indistinguishable.

generate coverage values, we use a PD with the following values for λ:

$$\lambda_c = \begin{cases} \left\lfloor 2 \cdot (1 - \frac{d_\mathrm{s}}{10}) + d_\mathrm{s} \right\rfloor & \text{for } d_\mathrm{s} < 10, \\ d_\mathrm{s} & \text{otherwise.} \end{cases} \tag{7.1}$$

Since the variance of the PD is directly dependent on the parameter λ, low coverage values are not expected to receive a lot of variance from this model. To increase the variance for low values of d_s the values of λ are calculated relative to d_s.

Since the PD is right-skewed, to yield the desired left-skewed distribution, it is reflected around its mean:

$$\mathrm{cov}_{i,\ell} = d_\mathrm{s} - (X - \mathbb{E}(X)) = 2 \cdot d_\mathrm{s} - X, \qquad X \sim \mathrm{PD}(\lambda_c) \tag{7.2}$$

A downside of this model is that the variance of the PD is smaller than what is observed in reality. Additionally, for larger values of d_s, the skew of the PD reduces and it approaches a normal distribution; hence the desired property of a left-skewed distribution of coverages is not achieved.

Nevertheless, the Poisson model provides valid coverage values and can be selected as a coverage model when using ddRAGE. One application of this model is the creation of easy datasets, where the simulated coverage is concentrated around the target coverage.

Beta-Binomial Coverage Model A model that better fits the observed coverage distributions is the Beta-binomial distribution (BBD) with three parameters $\alpha > 0$, $\beta > 0$, $n \in \mathbb{N}$ (see Probability Distributions p. 34f for details). Using the number of trials n and the shape parameters α and β, the distribution can be tailored to the desired shape. To obtain the expected coverage d_s, the parameter n must be chosen as $n = d_\mathrm{s}(\alpha + \beta)/\alpha$.

The two shape parameters control the left and right tailing of the distribution respectively. Hence, to generate a right-skewed distribution, parameters $\alpha < \beta$ have to be chosen. For $\alpha = \beta$ and $\alpha, \beta > 1$ the BBD approximates a binomial distribution with equal tailing on both sides. Hence, to simulate a dataset that can contain very low coverage values, but is unlikely to have coverage values far above d_s choose $\alpha >> \beta$.

The Beta-binomial coverage model with parameters $\alpha = 6$ and $\beta = 2$ is the default coverage model used by ddRAGE. This generates the expected coverage profiles as described above. To create a distribution similar to the distorted cases shown in Figure 7.14, $\alpha = 0.55$ and $\beta = 1.35$ can be used:

```
$ ddrage --BBD-alpha 0.55 --BBD-beta 1.35
```

Coverage of HRLs Coverage for highly repetitive loci (HRLs) significantly surpasses the coverage of valid loci. In principle, we would add the simulated coverages of valid loci (according to the beta-binomial model) for each individual and each repeat. However, we have no information about the repeat number distribution.

DDRAGE samples coverage values for HRLs from a discrete uniform distribution (DUD) ranging from a minimal to a maximal coverage value. The minimal value is determined as $|\mu + 2\sigma|$, where μ and σ are mean and standard deviation of the coverage model used for valid reads. This separates HRL coverage from valid coverage and maintains a high variance. In real datasets, coverage values up to the thousands can be observed, so as maximum value, a constant coverage of 1000 is used by default. As for valid loci, coverage values are sampled for each individual at the locus independently.

Distribution of Coverage for Heterozygous Mutations When facing a heterozygous mutation, the coverage, generated by a coverage model, needs to be distributed between two alleles of an individual. The coverage ratio is expected to be 1:1, assuming that reads are generated equally from both chromosomes. To allow for statistical fluctuations, a binomial distribution (BD) with success parameter $p = 0.5$ is used.

Redistribution of Coverage Through Incomplete Digestion The probability for an individual to possess incompletely digested (ID) reads at a locus is handled by the `--prob-incomplete-digestion` parameter, which is $p_{\mathrm{ID}} = 0.1$ by default. Hence, each individual at each locus has a 10% chance to lose coverage to ID. In this case, part of the coverage assigned to the individual is transferred from valid reads to ID reads to simulate this loss of coverage. The lost amount X is sampled from a Binomial distribution using the individuals total locus coverage and the ID rate parameter, which is $r_{\mathrm{ID}} = 0.2$ by default; this can be changed with the `--rate-incomplete-digestion` parameter,

$$X \sim \mathrm{BD}(\mathrm{cov}_{i,\ell}, r_{\mathrm{ID}}) \,. \tag{7.3}$$

Hence, the expected number of ID reads for an individual is

$$\mathbb{E}(\mathrm{cov}_{i,\ell}(\mathrm{ID})) = p_{\mathrm{ID}} \cdot \mathbb{E}(\mathrm{BD}(\mathbb{E}(\mathrm{cov}_{i,\ell}), r_{\mathrm{ID}})) = p_{\mathrm{ID}} \cdot r_{\mathrm{ID}} \cdot \mathbb{E}(\mathrm{cov}_{i,\ell}) \tag{7.4}$$

The number of individuals with ID is controlled by the ID probability parameter (p_{ID}), while the amount of ID reads per individual with ID is controlled by the ID rate (r_{ID}).

7.2.10 Evaluation of Resource Requirements

We implemented DDRAGE in Python3, using the just-in-time compiler numba[31] for computationally expensive tasks. DDRAGE simulates one locus after the other (for all individuals jointly) to keep a low memory footprint and to allow execution on a consumer grade PC. After all reads for a locus have been generated, they are written to disk and removed from memory. Only statistics about the generated data and annotations are kept.

[31] Lam, Pitrou, and Seibert, "Numba: A llvm-based Python Jit Compiler", 2015.

		number of individuals			
#loci		12	48	96	
1 000	time	1.59	7.30	15.46	mins
	memory	180.44	228.60	264.70	MB
10 000	time	19.14	76.39	176.08	mins
	memory	196.41	256.48	306.63	MB
100 000	time	191.17	770.36	1555.49	mins
	memory	252.02	436.41	590.26	MB

Table 7.2: Running times and peak memory usage of several runs of ddRAGE using different parameter combinations. The values shown are wall-clock running times in minutes and peak memory usage in megabytes. The running time depends linearly on both the number of loci and on the number of simulated individuals, allowing the extrapolation of running times for larger datasets.

Both running time and memory requirements of the simulation are linear in the number of loci, number of individuals, coverage, and read length. Note that the read length can be considered constant, as it is fixed to 100–300 bp for Illumina sequencers. While the simulation of longer reads with ddRAGE is possible, those do not accurately reflect real ddRAD experiments. ddRAGE officially supports read lengths between 50 and 500 bp. Increasing the amount of HRLs (`--hrl-number`) which is computed as a fraction of the number of total valid loci, also linearly increases the running time. Other parameters, such as different mutation rates, sequencing error probability, etc. do not affect the running time. Table 7.2 shows a benchmark of ddRAGE when called with different parameter combinations for the number of individuals and number of loci simulated (using a coverage of 30, read length of 100). For all other parameters, the default values were used. All experiments were conducted on workstation with 16 GB of RAM and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor.

The results show that even large datasets, containing more than 100 000 loci from 96 individuals with a coverage of 30, can be created using a consumer grade computer. The main limiting constraint is the available disk space.

We compared the running time of ddRAGE with that of other read simulation tools. Since neither ddRADSeqTools nor simrrls simulates HRLs or singletons, we disabled these aspects of the simulation in ddRAGE for this analysis. We did not evaluate the performance of RADInitio, since it does not offer the option to select a number of loci to simulate. The simulation of PCR duplicates was disabled for both ddRAGE and ddRADSeqTools. To simulate a dataset with 100 000 loci and 96 individuals, ddRAGE required 514 minutes and 188.60 MB of RAM. For a similarly sized dataset ddRADSeqTools required 327 minutes and 121.20 MB of RAM, while simrrls required 216 minutes and 468.04 MB of RAM. This shows that ddRAGE works in the same order of magnitude for both running time and memory usage as similar tools. However, each tool offers different features and options, resulting in different time and memory requirements that are not easily comparable.

7.2.11 *Application Examples*

To show how ddRAGE can be used to evaluate a ddRAD analysis tool and help choosing parameter combinations for such a tool, we analyzed a generated dataset with both pyRAD version 3.0.64[32] and Stacks version 1.46.[33] Note that the goal of this analysis is not to compare the performance of the two tools, but to show possible applications of ddRAGE. We analyzed only one dataset and performed no parameter optimization for the analysis tools.

Both tools required preprocessing the data, specifically removing the DBR. Stacks additionally required removing the annotation from the FASTQ name lines. For the analysis with Stacks the p5 and p7 reads were concatenated to allow for better comparison of loci. We tested a dataset simulated with ddRAGE with 1 000 loci from 6 individuals using a coverage of 30 and read length of 100. It contains 285 unique SNPs, 14 insertions and 25 deletions.

We analyzed the results using bottom-s sketching to estimate Jaccard similarity of the q-gram sets of the locus sequences simulated by ddRAGE to the locus sequences computed by Stacks and pyRAD. The parameters used for the sketching were a q-gram size of $q = 6$ and a sketch size of $s = 30$. A similarity threshold of $\mathcal{J} = 0.2$ was used, which allows for a sensitive clustering while keeping a low false positive rate. The results are aggregated in Table 7.3.

	pyRAD	Stacks
Total number of loci identified	309	727
Valid loci (correctly assembled)	300	640
% of total loci simulated	*30%*	*64%*
Valid loci (split up)	0	26
% of total loci simulated	*0%*	*2.6%*
False positive loci	9	35
SNPs identified	62	92
% of total SNPs	*21.78%*	*32.80%*
% of SNPs in valid loci	*68.13%*	*53.77%*
False positive SNPs	0	3

Table 7.3: Analysis results of Stacks and pyRAD on a simulated dataset with 1 000 valid loci and 285 SNPs.

Example of Stacks *evaluation* On the simulated dataset, Stacks identified 727 loci using default parameters. Stacks was run with a minimum stack depth of `-m 3`, `-M 2` allowed mismatches between loci, `--gapped` alignment, and the `-t` parameter to remove HRLs.

Of the 1 000 loci simulated, 640 loci were correctly assembled, but some loci were split up during analysis. 26 loci simulated by ddRAGE were associated with two Stacks loci (52 of the total 727 identified loci). Of these 26 loci, 23 were loci for which ddRAGE simulated ID reads, which are likely responsible for the splitting. One might argue that this is intended behavior, since it cannot be determined with certainty which p7 sequence is the correct se-

[32] Eaton, "PyRAD: Assembly of de novo RADseq Loci for Phylogenetic Analyses", 2014.

[33] Catchen et al., "Stacks: an Analysis Tool Set for Population Genomics", 2013.

quence. However, to not skew the analysis, loci with identified ID reads should either be removed completely or resolved (keeping only one of the two, preferably the one present in more individuals). By joining the p5 and p7 reads for the analysis, Stacks cannot easily identify and resolve this kind of error.

A total of 334 loci of the 1 000 loci simulated by ddRAGE were not identified by Stacks at all. These might have been sorted out because of low coverage, for example introduced by a large fraction of *dropout* events or low values sampled from the coverage model. On the other end, Stacks identified 35 loci that were not part of the valid loci simulated by ddRAGE. These loci have been assembled from HRL reads and singletons.

Stacks correctly identified 32.8% of the simulated SNPs with at least one locus that could be linked to the respective ddRAGE locus. The remaining 192 SNPs were not identified. For the 640 assembled loci, Stacks found 46.23% (92 out of 199) of the SNPs simulated for these loci. However, Stacks identified three SNPs that were not simulated by ddRAGE and are likely either the result of a simulated sequencing error or an indel mutation. Stacks also did not report any indel mutations, since the identification of indel variants is not supported by Stacks.

For the coverage per individual, Stacks found values between 28.6 and 30.4, which is close to the simulated coverage of 30.

Example of pyRAD *evaluation* We analyzed the same dataset following a tutorial[34] provided by the author of pyRAD. pyRAD was run with a minimum cluster coverage of `Mindepth 6`, a clustering threshold of 0.85, allowing for `NQual 4` low quality sites per read, and a maximum of 10 heterozygous sites per consensus sequence.

Using the default parameters provided by the tutorial, pyRAD identified 309 loci. Of these 309 loci, 300 could be assigned to a valid ddRAGE locus. The remaining 9 loci did not show similarity with any valid locus simulated by ddRAGE and are likely to originate from HRLs or singletons, as described above. pyRAD did not identify any split up loci. However, many loci were not identified at all. Loci with many *dropout* events or low simulated coverage might be responsible for these missing loci. We speculate that the default pyRAD parameters are not well suited for the simulated dataset; so the situation may improve with parameter optimization.

pyRAD excluded 21 loci from the analysis because they did not pass its final filtering step. All of these were valid loci simulated by ddRAGE. However, 33.33% of these 21 rejected loci had ID reads, which may have been a factor for the exclusion.

pyRAD found 21.78% of the unique SNPs in the dataset and did not report any indels. For the 300 assembled valid loci, pyRAD found 68.13% of the SNPs simulated for these loci. Additionally it identified four SNPs for the 9 invalid loci. Since SNPs were neither simulated for singletons nor for HRLs, these SNPs likely stem from sequencing errors. For the valid loci, pyRAD did not call any SNPs

[34] nbviewer.jupyter.org/gist/dereneaton/dc6241083c912519064e/tutorial_pairddRAD_3.0.4.ipynb

that were not simulated by DDRAGE. These observations suggest that the SNP calling approach used by PYRAD works well, even with default parameters.

Some locus alignments discovered by PYRAD show gaps that cannot be explained by sequencing errors or mutations. Neither indel mutations nor SNPs were simulated for these loci. However, the individuals for which gaps were added in the alignment were affected by ID, while the ones correctly assembled were not. This suggests that the alignment step of PYRAD is not coping well with ID reads.

7.2.12 Discussion and Conclusion

We developed DDRAGE, a simulation software for ddRAD reads with the goal to evaluate ddRAD analysis pipelines. Instead of optimizing experiment design, its generated datasets are aimed at evaluating the analysis process of ddRAD software. The ability to validate early analysis steps holds great potential, as errors arising early in the analysis process of ddRAD data, for example during sequence clustering, can propagate into later stages of the analysis and either skew results or increase the computational complexity of the analysis. If a read is removed during the analysis process, knowing the effects that might have led to this decision can help with parameter optimization in order to craft better analysis pipelines. While other simulation software exists, none of them provide a detailed ground truth alongside the simulated reads which is necessary for such an evaluation.

The closest comparable software — DDRADSEQTOOLS — is also able to simulate reads from scratch or from a given reference genome. However the software does not simulate sequencing errors, quality values, and technological artifacts like singletons and HRLs. SNPs, indels, and dropout events are simulated, but null alleles are not modeled. Due to that, NAs with alternate sequence are not simulated, and dropout events by technological effects and by NAs cannot be simulated independently. For each locus, DDRADSEQTOOLS simulates a number of mutated sequences that can occur as alleles. However, each allele is simulated independently, making closely related alleles highly uncommon. Most importantly though, DDRADSEQTOOLS does not provide a record of the simulated effects. While a verbose output mode is available that prints all simulated mutations for all loci and the simulated reads are marked either as mutated or unmutated, there is no readily available account for which effect is present in which read. But since the main focus of DDRADSEQTOOLS is the optimization of ddRAD experiments, this is not a required feature.

The SIMRRLS software is able to simulate ddRAD reads from scratch. It simulates SNPs, indels, NA dropout, and sequencing errors, but does not include the simulation of PCR duplicates, singletons and HRLs. Additionally NAs with alternate sequence are not simulated. SIMRRLS models genotypes using a phylogenetic tree and a coalescent model, however, these simulated genotypes are not provided as output. While read data simulated by SIMRRLS might be sufficient to test the functionality of PYRAD, their use for a comprehensive evaluation of ddRAD analysis software is limited.

RADINITIO is able to simulate population level structures and dependencies for a given reference genome, but is not able to simulate reads from scratch. This comes with the limitation that the number of loci, mutations, etc. cannot be fine tuned for evaluation purposes.

The SIMRAD and FRAGMATIC tools both perform *in-silico* digestion of a reference genome and do not generate simulated reads. While this is valuable information for the design of ddRAD experiments, these tools cannot be used to evaluate ddRAD analysis software.

DDRAGE provides both the simulated reads and a detailed record of all simulated effects, including genotypes assigned to each individual, position and kind of mutations, and sequencing errors. This enables the users and designers of ddRAD analysis software to evaluate the performance of their software.

The implementation of DDRAGE is memory-efficient and allows users to simulate datasets of sizes that are realistic for a real ddRAD experiment. Due to its modular structure, DDRAGE facilitates inclusion of future features, such as phylogenetic simulations, the simulation of individuals from different populations, or additional coverage models. The code is available under the open source MIT license on Bitbucket, and users are encouraged to contribute to the project.

As shown in the application examples, different analysis tools show different error profiles in their results. This can most prominently be seen in the performance of the applied genotyping, error correction and sequence assembly solutions. The genotyping process of PYRAD, for example, did not introduce any false positives and found a higher amount SNPs in the assembled loci, while STACKS assembled more loci in the first place and found a higher number of SNPs. Using a ground truth simulated by DDRAGE, these profiles can be identified and used as a starting point for further development of ddRAD analysis software. This may include the development and testing of new software, improvements of established tools, and allows testing of analysis pipelines for sensitivity and specificity. The development of such an analysis pipeline is described in the following section.

7.3 *A Workflow for ddRAD Data Analysis*

In the previous section, we described which effects influence the analysis of ddRAD data. Coping with these effects requires a combination of different measures:

- Preprocessing steps to mitigate the influence of technical effects.
- Intermediate processing steps to facilitate and optimize the interaction between analysis steps.
- Parameter choices for the analysis software, tailored to the expected data.

The different, sequential steps required for the analysis are usually performed by a collection of specialized tools. Such a combination of tasks, performed in a reproducible manner, is called pipeline or workflow and can be realized in different ways, ranging from shell scripts to workflow management systems like SNAKEMAKE.[35] The main advantage of managed workflows is producing reproducible results through a controlled environment paired with the reuse of intermediate results.

We used STACKS[36] to implement a ddRAD analysis workflow, which leverages these capabilities. Since STACKS follows a workflow architecture itself, the single steps of this software can be easily integrated into another workflow system. The main components of the STACKS workflow that we focus on are locus reconstruction per sample followed by merging into a catalog of loci across all samples. Locus reconstruction is performed by the USTACKS program, followed by the merging step performed by the CSTACKS program. One of the main influencing factors on the analysis of ddRAD data with the STACKS software is the choice of the following three parameters:

- The minimum depth of coverage required to form a locus. This is controlled by the `-m` parameter of USTACKS with a default value of 3.
- STACKS merges loci with similar sequences, after they were assembled. The maximal number of different nucleotides allowed between loci is controlled by USTACKS parameter `-M` (default: 2).
- When building the catalog, STACKS performs another locus merging step, this time between samples. The default number of different nucleotides allowed between loci for a merge is controlled by the `-n` parameter of CSTACKS (default: 1).

These parameters control the reconstruction of loci and their postprocessing. Depending on the expected number of mutations and sequencing errors, read length, and several other factors, radically different numbers and sizes of loci may be detected by STACKS within the same dataset. However, since in the absence of a reference genome, many of these influencing factors cannot be deter-

[35] Köster and Rahmann, "Snakemake — A Scalable Bioinformatics Workflow Engine", 2012.

[36] Catchen et al., "Stacks: an Analysis Tool Set for Population Genomics", 2013; Rochette, Rivera-Colón, and Catchen, "Stacks 2: Analytical methods for paired-end sequencing improve RADseq-based population genomics", 2019.

mined, it is useful to explore several different parameter combinations for each dataset in a structured manner.

We developed a workflow that automates the preprocessing of ddRAD datasets and allows the exploration of STACK's parameter space while reusing intermediate results. To achieve our goal of reproducible results, we implemented our workflow using SNAKEMAKE.[37] SNAKEMAKE allows to retain control over software versions used for the analysis, via the integration of CONDA environments, which allows us to use the BIOCONDA[38] repository for bioinformatics software. We implemented additional Python scripts using our own open source input/output library DINOPY,[39] which is also available via BIOCONDA. The workflow itself is available under the MIT License as part of the SNAKEMAKE-WORKFLOWS project.[40]

In the following sections, we describe our workflow for ddRAD analysis as well as its evaluation with DDRAGE.

7.3.1 Workflow Structure

A SNAKEMAKE workflow is defined through a set of rules, each of which can generate a set of output files from a set of input files using a given shell command or script. The root of a workflow is a rule named `all`, which has no output files, but contains a set of all input files that need to be generated. Starting from this rule, SNAKEMAKE constructs a directed acyclic graph of rules (rule graph) that can be used to generate these files. During this process, SNAKEMAKE automatically determines the values of wildcards in the input and output file names, which can also encode parameters used to generate the files.

We developed a more detailed visualization for SNAKEMAKE workflows called a filegraph, showing the input and output files for each rule. The option to generate a filegraph was added to SNAKEMAKE in version 5.7.0.[41] While filegraphs are best viewed on digital devices, a filegraph version of our workflow can be found in the appendix on p. 246f.

Our workflow can be structured into three stages, as illustrated by the rule graph shown in Figure 7.15: In the preprocessing phase, we prepare paired end reads for analysis, then we evoke the STACKS analysis workflow using different parameter sets. Finally, we assemble quality control plots for each parameter to compare the performance of different parameter sets.

The input data for our workflow is a set of paired end (gzipped) FASTQ files, split up by p7 index barcode. These are called the sample files. Each sample file contains reads from a set of sequenced individuals with different p5 in-line barcodes. The files `individuals.tsv` and `samples.tsv` contain a mapping of individuals to a p5 + p7 barcode pair and a p5 + p7 spacer pair, as well as paths to all sample files.

[37] Köster and Rahmann, "Snakemake — A Scalable Bioinformatics Workflow Engine", 2012.

[38] Grüning et al., "Bioconda: Sustainable and Comprehensive Software Distribution for the Life Sciences", 2018.

[39] Timm and Hartmann, *Dinopy — DNA input and output for Python and Cython*, 2020.

[40] Köster and Timm, *snakemake-workflows/rad-seq-stacks*, 2021.

[41] `snakemake.readthedocs.io/en/stable/project_info/history.html`

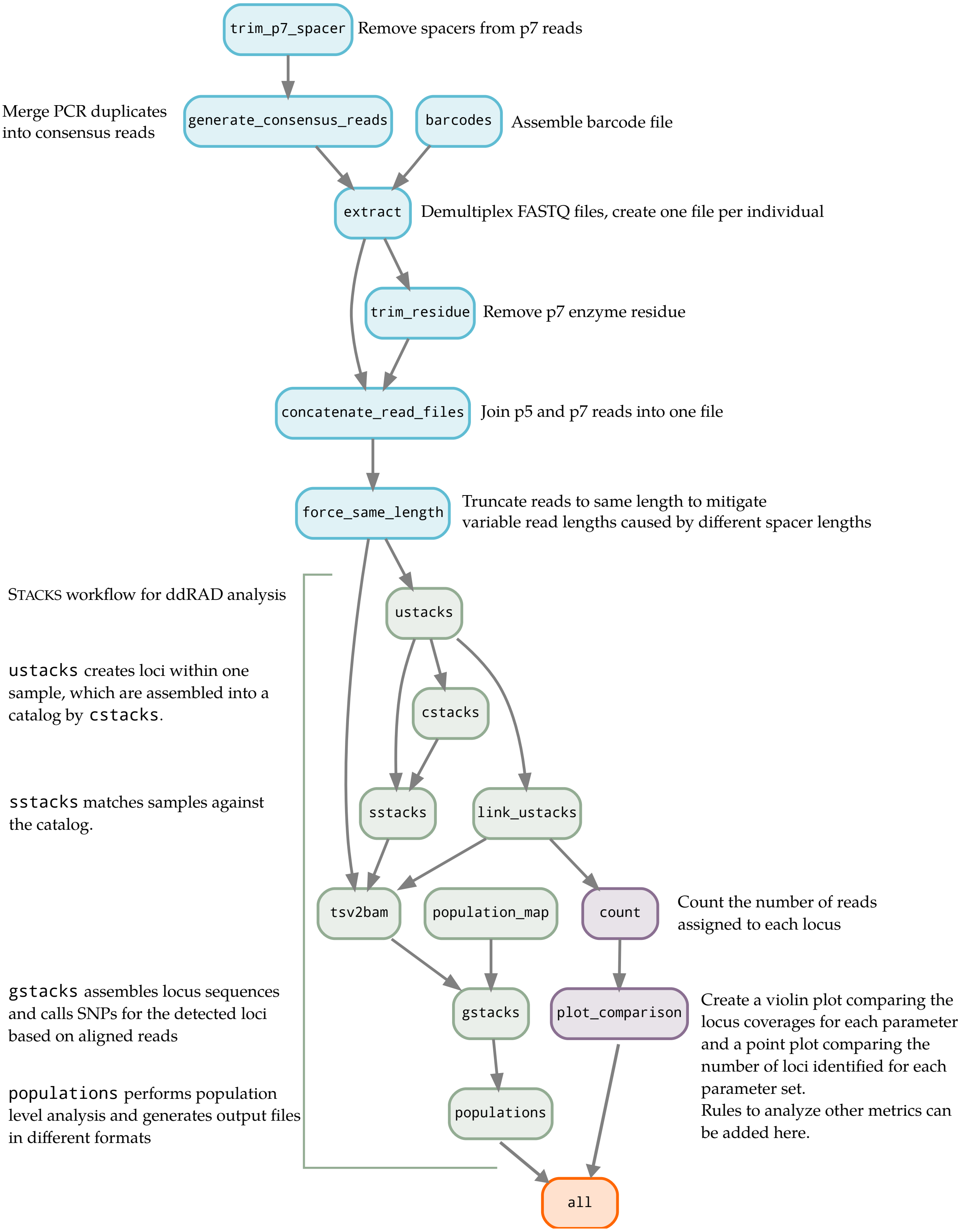

trim_p7_spacer
Remove spacers from p7 reads
Merge PCR duplicates into consensus reads
generate_consensus_reads
barcodes
Assemble barcode file
extract
Demultiplex FASTQ files, create one file per individual
trim_residue
Remove p7 enzyme residue
concatenate_read_files
Join p5 and p7 reads into one file
force_same_length
Truncate reads to same length to mitigate variable read lengths caused by different spacer lengths
STACKS workflow for ddRAD analysis
ustacks creates loci within one sample, which are assembled into a catalog by cstacks.
sstacks matches samples against the catalog.
gstacks assembles locus sequences and calls SNPs for the detected loci based on aligned reads
populations performs population level analysis and generates output files in different formats
ustacks
cstacks
sstacks
link_ustacks
tsv2bam
population_map
count
Count the number of reads assigned to each locus
gstacks
plot_comparison
Create a violin plot comparing the locus coverages for each parameter and a point plot comparing the number of loci identified for each parameter set.
Rules to analyze other metrics can be added here.
populations
all

Figure 7.15: This SNAKEMAKE rule graph for our ddRAD analysis workflow illustrates the different steps performed during analysis. Note that rules can be evoked multiple times, e.g. `generate_consensus_reads` is called once per pair of FASTQ files, while `populations` is only called once per run. Our workflow begins with preprocessing steps (shown in teal). In the second stage, we start the STACKS analysis pipeline with different parameters supplied by the user (green). To judge the influence of the parameter choices, a series of quality control plots is assembled (purple). For a more detailed description of the rules including the generated files, please refer to the Appendix File Graph for our ddRAD Analysis Workflow (p. 245).

Preprocessing During preprocessing, we first trim off p7 spacers for each sample file using SEQTK.[42] Since p7 spacers are linked to p7 barcodes, all reads within a sample file also have the same p7 spacer.

In the following rule, PCR duplicates are removed and merged into consensus reads using the CALL-CONSENSUS-READS software we contributed to the RUST-BIO-TOOLS[43] project. We describe the details of this PCR deduplication process in the subsequent section PCR Duplicate Removal (p. 213). This step also removes the DBR sequence from the reads.

Using the barcodes file, we demultiplex the sample files into individual FASTQ files utilizing the PROCESS_RADTAGS tool of STACKS. After this step, p7 restriction enzyme residues are removed.

Our analysis has shown, that the paired-end analysis mode of STACKS has trouble identifying a high number of loci. By setting the user defined `mode` parameter of our pipeline, we can pass reads to STACKS in three different configurations to address this problem:

- `p5_only` is the default behavior, where only the p5 reads are used for clustering and paired-end information can be incorporated in a later step.
- The `merged` mode horizontally joins the reads by merging p5 and p7 sequence into one read.
- In `concatenated` mode, the p7 read file is vertically appended to the p5 read file. The p5 and p7 reads are added to the same file.

As a final preprocessing step, we assure, that all reads have the same length. After trimming variable length sequences like spacers from the reads, total read lengths can differ (cf. Sequencing Process and Read Structure, p. 165). Since the last bases of the reads cannot support a strong biological signal, we trim all reads to the shortest length in the dataset.

STACKS *Workflow* Following this, we call the remaining steps of the STACKS analysis workflow. These perform the tasks mentioned at the beginning of this chapter, i.e. reconstruct loci and identify genotypes within the analyzed population. We call this process several times with different combinations of the parameters `m` (minimal number of reads per locus), `M` (maximum number of differing nucleotides between loci within samples), and `n` (maximum number of mismatches for loci between samples). Note, that all steps until this point only need to be executed once. Different instances of the STACKS workflow with their respective parameter sets can process the same input files.

Evaluation and Quality Control To evaluate the results of a parameter set, we generate a histogram of locus sizes, i.e. the number of loci with a certain coverage. These can be compared between

[42] `github.com/lh3/seqtk`

[43] `github.com/rust-bio/rust-bio-tools`

parameter sets and act as a guide when optimizing the analysis parameters. In order to make this comparison easier, we also generate violin plots comparing the distributions of locus sizes across all parameter sets, as well as a scatterplot illustrating the number of detected loci per individual. Due to the structure of our workflow, parameters for the analysis can be changed in a central configuration file.

7.3.2 *Workflow Evaluation*

To evaluate the performance of our workflow, we simulated datasets with ddRAGE and compared several metrics of its results with the ground truth. We analyzed the number and the size distribution of loci identified by Stacks, depending on the presence of our PCR duplicate removal step (PCR deduplication) and other scenarios. For this evaluation, we restricted the analysis to the locus level. Workflows to reproduce the evaluations described in this section can be found on Zenodo[44] and GitHub.[45]

Influence of PCR deduplication To show the influence of PCR deduplication, we simulated a dataset using ddRAGE with the parameters illustrated in Table 7.4. For each individual, approximately 9 500 valid loci with an expected coverage of 30 are present in the dataset (10 000 loci, 5% of which are expected to be dropout events). Additionally, 500 highly repetitive loci (HRLs) with reads from all individuals were simulated, in addition to ∼1 200 000 singleton reads. We then analyzed this dataset using two identical versions of our workflow, one with and the other without PCR deduplication. The PCR deduplication was performed with a DBR distance of 1 and a sequence distance of 6. For the workflow variant without deduplication, we only trimmed out DBR sequences, so that subsequent rules in the workflow could be executed as before.

Parameter	Value
`--loci`	10 000
`--individuals`	24
`--prob-seq-error`	0.01

Table 7.4: Parameters for the dataset simulated with ddRAGE. For all parameters that are not explicitly mentioned here, we used default parameters.

For calls to the Stacks workflow, we evaluated the parameter combinations illustrated in Table 7.5 to span a variety of signatures. We maintained a minimum locus size of 3 at this point, since this dataset has a high coverage where variations in the minimum locus size are unlikely to influence valid loci. We performed the analysis using the `p5_only` read handling mode. This means, that after extraction, we passed only the p5 reads on to Stacks for analysis.

Max. dist. (sample) `n`	Max. dist. (indiv.) `M`	Min. locus size `m`
2	2	3
2	3	3
3	4	3
5	5	3
15	16	3

Table 7.5: Parameter sets evaluated for our pipeline. `n` is the number of mismatches for merging loci between samples, `M` is the permissible number of mismatches for merging loci, and `m` denotes the minimum coverage required to form a locus.

Figure 7.16 illustrates the distributions of locus sizes with and without deduplication enabled. It can be seen that workflow runs without deduplication step possess a large peak for coverage values between 2 and 10. These are caused by singleton reads which, in combination with their respective PCR duplicates, are identified by Stacks as loci. For runs with our deduplication enabled, these peaks vanish. Through PCR deduplication, these false loci are not identified by Stacks, since PCR duplicates collapse back into singular reads. These do not satisfy a locus size of $\geq m = 3$, and are

[44] Timm, *Rad-seq-stacks Evaluation Workflow*, 2021.

[45] `github.com/HenningTimm/rad-seq-stacks_evaluation`

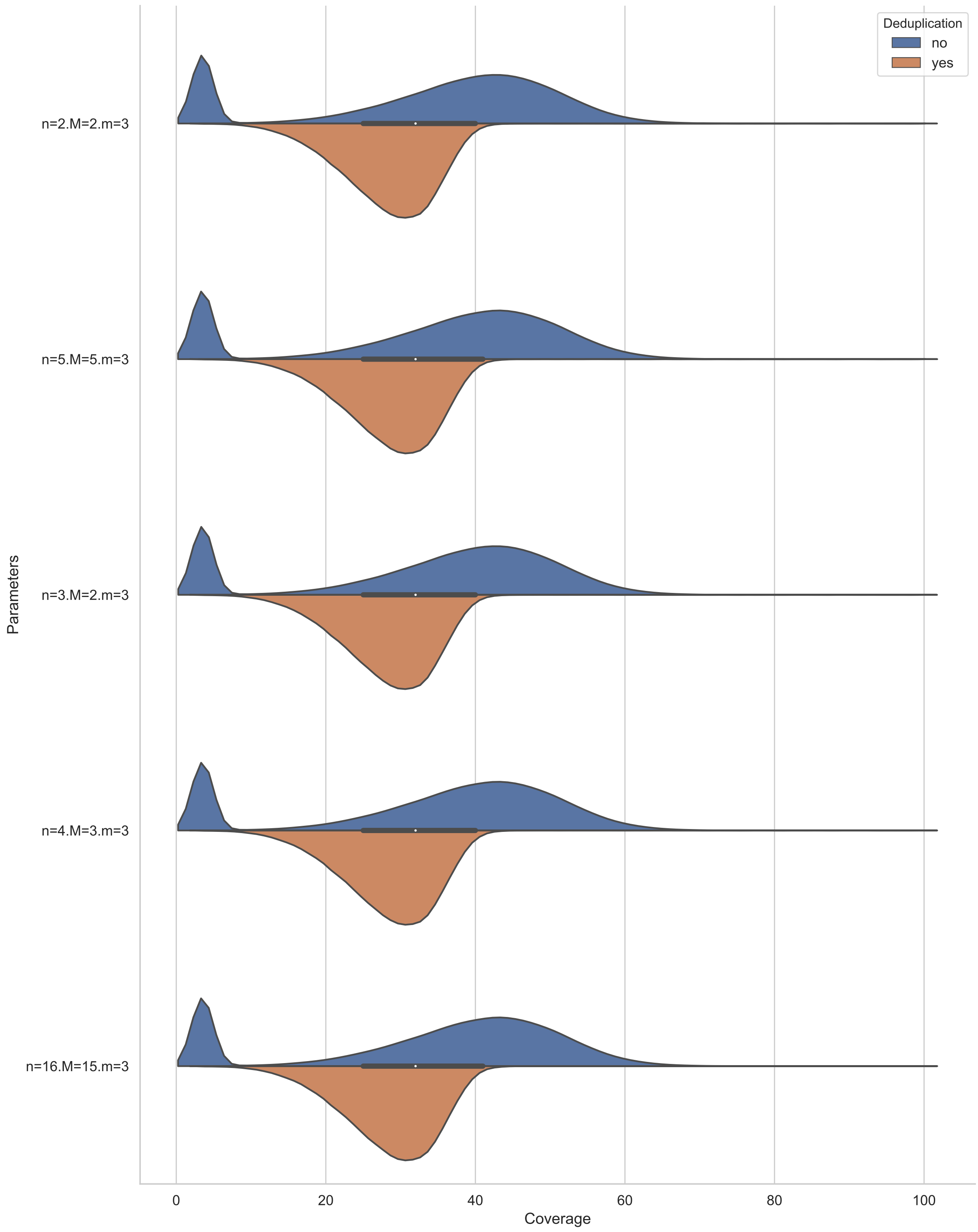


Figure 7.16: Violin plots comparing the distributions of locus sizes (x-axis in reads per individual and locus) for different parameter sets for Stacks (rows). (*n*: maximum distance between sample loci for merging, *M*: maximum distance of loci for merging within a sample, *m*: minimum stack stack size.) Each parameter set was run both with (orange) and without (blue) our PCR deduplication step. Coverage values above 100 were truncated to increase readability. Locus sizes of up to 1 400 reads per individual, introduced by HRLs were contained in the data. An untruncated version of this plot can be found in Appendix D (Figure D.1).

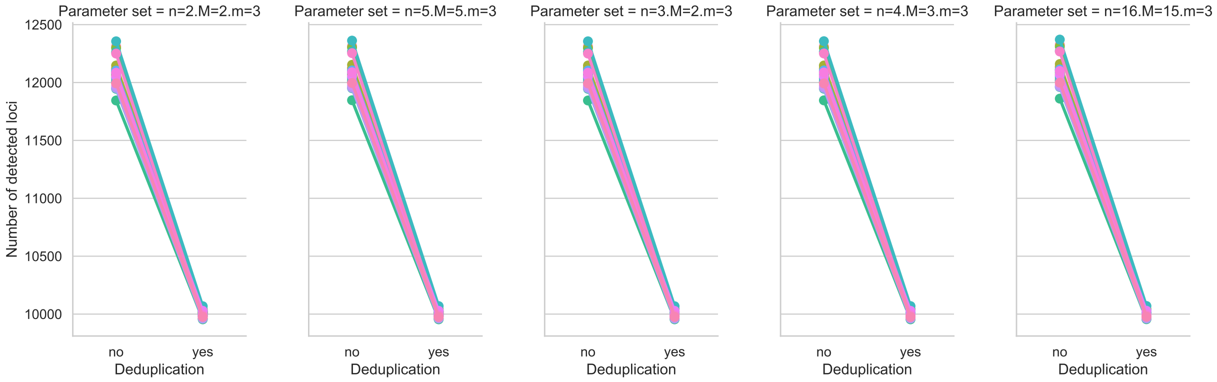


Figure 7.17: Point plots comparing the number of loci identified for each individual (y-axis) for runs without (left) and with (right) PCR deduplication. Individuals share a color and are connected. The expected number of loci for an individual in this dataset was ∼9 500.

consequently excluded. Note, that for the deduplicated workflow, the simulated left-tailed coverage distribution can be seen.

Figure 7.17 shows the number of loci identified per individual by ustacks, again without (left) and with (right) PCR deduplication. For all parameter sets and individuals, the number of loci identified dropped significantly when using deduplication. The additional loci identified in non-deduplicated runs are mostly singleton loci, which were removed by PCR deduplication. This is supported by the total number of loci contained in the cstacks catalog after loci from all samples are merged, as illustrated in Table 7.6.

	Deduplication	
Parameter set	no	yes
n=2.M=2.m=3	57 759	10 659
n=3.M=2.m=3	57 756	10 656
n=4.M=3.m=3	57 753	10 654
n=5.M=5.m=3	57 756	10 654
n=16.M=15.m=3	57 761	10 656

Table 7.6: Number of loci contained in the cstacks catalog for different parameter sets, with and without deduplication.

The additional loci identified for the single individuals accumulate during catalog creation and result in a large number of invalid loci, which need to be processed in downstream analysis steps.

Figure 7.18 displays the number of *stacks* blocklisted (excluded from further analysis) as HRLs by Stacks as well as the mean size of loci during the first clustering step of ustacks. Note, that at this point we refer to a *stack* as a cluster of reads identified by ustacks. In later steps, stacks of reads can be merged with other stacks. The remaining postprocessed stacks returned by ustacks are treated as loci.

For runs without deduplication, a much higher number of stacks are blocklisted than for runs with deduplication enabled. The cause for this is twofold:

1. HRLs are split into many different loci during the first clustering step.

2. The mean size of stacks identified in the first clustering step is lower.

HRLs possess a very high coverage and every read is expected to contain 1 sequencing error. Since the first clustering step of ustacks builds clusters from perfect matches, HRLs are split into many different stacks. The blocklisting of HRL loci is performed based on the mean size of all initial stacks. Stacks with a coverage higher than $\mu + 3\sigma$, where μ and σ are the mean and standard

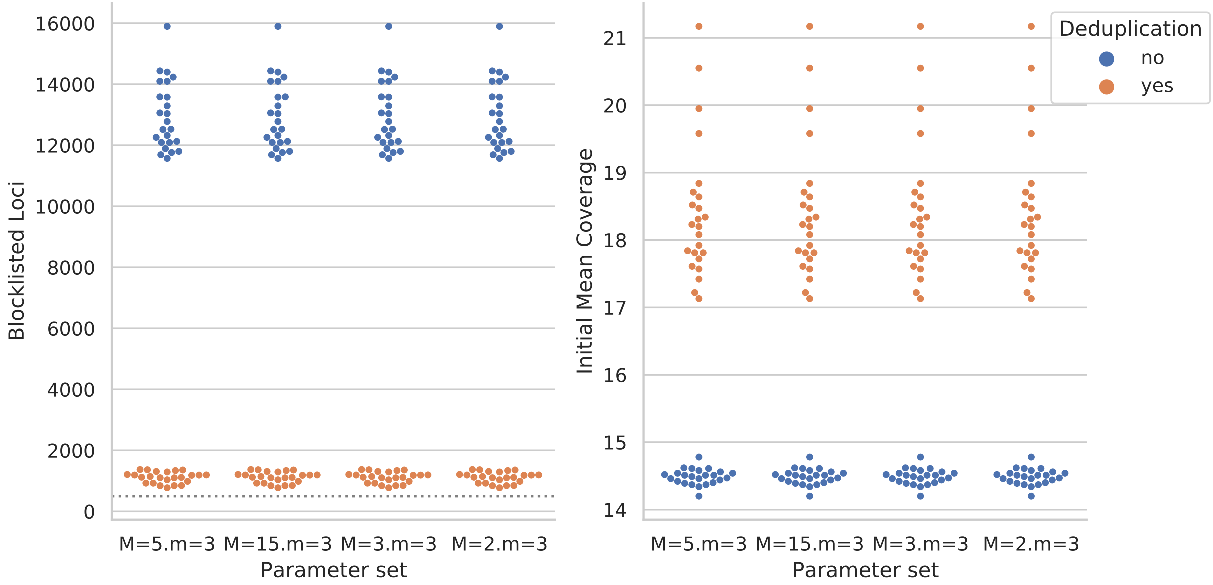


Figure 7.18: Swarm plots comparing the number of loci blocklisted as HRLs (left) and the mean size of initial loci identified by USTACKS before merging (right). In both plots, each point of the same color symbolizes one individual from the dataset. Runs with deduplication enabled are shown in orange, runs without deduplication in blue. The simulated dataset contained 500 HRLs (shown as a gray, dotted line in the left plot). Note that the n parameter for CSTACKS is not shown here since this data is produced before CSTACKS is run.

deviation of the distribution of stack coverage, are blocklisted as HRLs.

The deduplication step mitigates both of these effects, since consensus reads have a much lower error rate than unprocessed reads. Since sequencing errors occur for each read and read position independently, the chance for n reads (one original and $n-1$ PCR duplicates) to show the same error twice is low. Consequently, during PCR deduplication, many sequencing errors are removed. This results in both less initial stacks, since less differing reads are present, and a higher mean size, which in turn raises the coverage ceiling for valid stacks. Especially in the left part of Figure 7.18, it can be seen that with deduplication, the amount of HRLs identified is very close to the 500 HRLs simulated for the dataset.

For all evaluations, the parameters chosen for STACKS did not strongly influence the results. This is caused mainly by the high coverage and low diversity of the simulated dataset. We did not change the minimum number of reads per locus, however, with a simulated coverage of 30, this parameter is unlikely to have any effect. For the maximum number of mismatches between loci, the low number of simulated mutation events paired with the low number of affected positions does not introduce enough diversity to break apart loci. In the following evaluations we will show that our pipeline can detect such events through the parallel exploration of different parameter sets.

Influence of Minimum Reads per Locus (m) First, we simulated a dataset with low expected coverage to explore the influence of different STACKS parameters. Most notably, a low coverage interacts with the minimum number of reads required to form a locus (m).

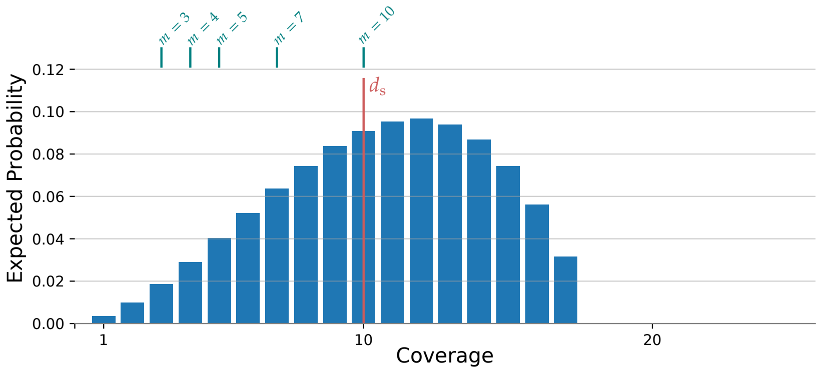


Figure 7.19: Expected coverage distribution of valid loci for one individual in the low coverage dataset. The red line denotes the target sequencing depth d_s. Teal lines denote the explored values for m, the minimal number of reads required to form a locus.

The parameters for ddRAGE used to simulate the dataset to explore the influence of this parameter are detailed in Table 7.7. The left-tailed coverage distribution described by the BBD parameters $\alpha = 3$ and $\beta = 2$ paired with a low expected coverage of 10 results in a high chance of loci containing less than m reads. We analyzed this dataset both with and without PCR deduplication using the parameter sets shown in Table 7.8.

Parameter	Value
`--loci`	10 000
`--individuals`	24
`--prob-seq-error`	0.01
`--coverage`	10
`--BBD-alpha`	3
`--BBD-beta`	2

Table 7.7: Parameters for the low coverage dataset simulated with ddRAGE. For all parameters that are not explicitly mentioned here, we used default parameters.

Max. dist. (sample) `n`	Max. dist. (indiv.) `M`	Min. locus size `m`
2	2	3
2	2	4
2	2	5
2	2	7
2	2	10

Table 7.8: Parameter sets evaluated for the low coverage dataset. `n` is the number of mismatches for merging loci between samples, `M` is the permissible number of mismatches for merging loci, and `m` denotes the minimum coverage required to form a locus.

The expected coverage distribution, alongside the values for m we used for the analysis, is illustrated in Figure 7.19. We used a DBR distance of 1 and a sequence distance of 6 for the deduplication step.

Figure 7.21 (p. 202) shows the number of loci detected by Stacks using the different parameter sets. With a minimum number of $m = 3$ reads to form a locus, approximately 8 000 – 9 000 loci are detected for all individuals using PCR deduplication. As above, without deduplication the number of detected loci per individual is higher. For an increasing number of reads required, the number of loci detected continually falls, until for $m = 10$ less than 500 loci are identified.

Figure 7.20 shows the distribution of locus sizes with and without deduplication enabled. It can be seen that for a minimal locus size of $m = 5$, the peak for singleton loci disappears even without PCR deduplication. This is due to the fact, that singletons cannot reach this limit even with their PCR duplicates. However, not only singletons are removed, but also most valid loci (cf. Figure 7.21, p. 202). Therefore, the coverage values shown for the lower parameters are not a robust quality measure in this case.

This pattern of decreasing locus numbers with increasing values of m is specific for low coverage data and can guide the analysis towards the parameter sets best suited for this scenario. In this case, the results computed with low m yield the arguably best results, since they contain more correctly identified loci. On the other hand, using both a low and a high value for m can serve as a filter: Loci identified using the higher value can be attributed a high weight, since they are supported by many reads.

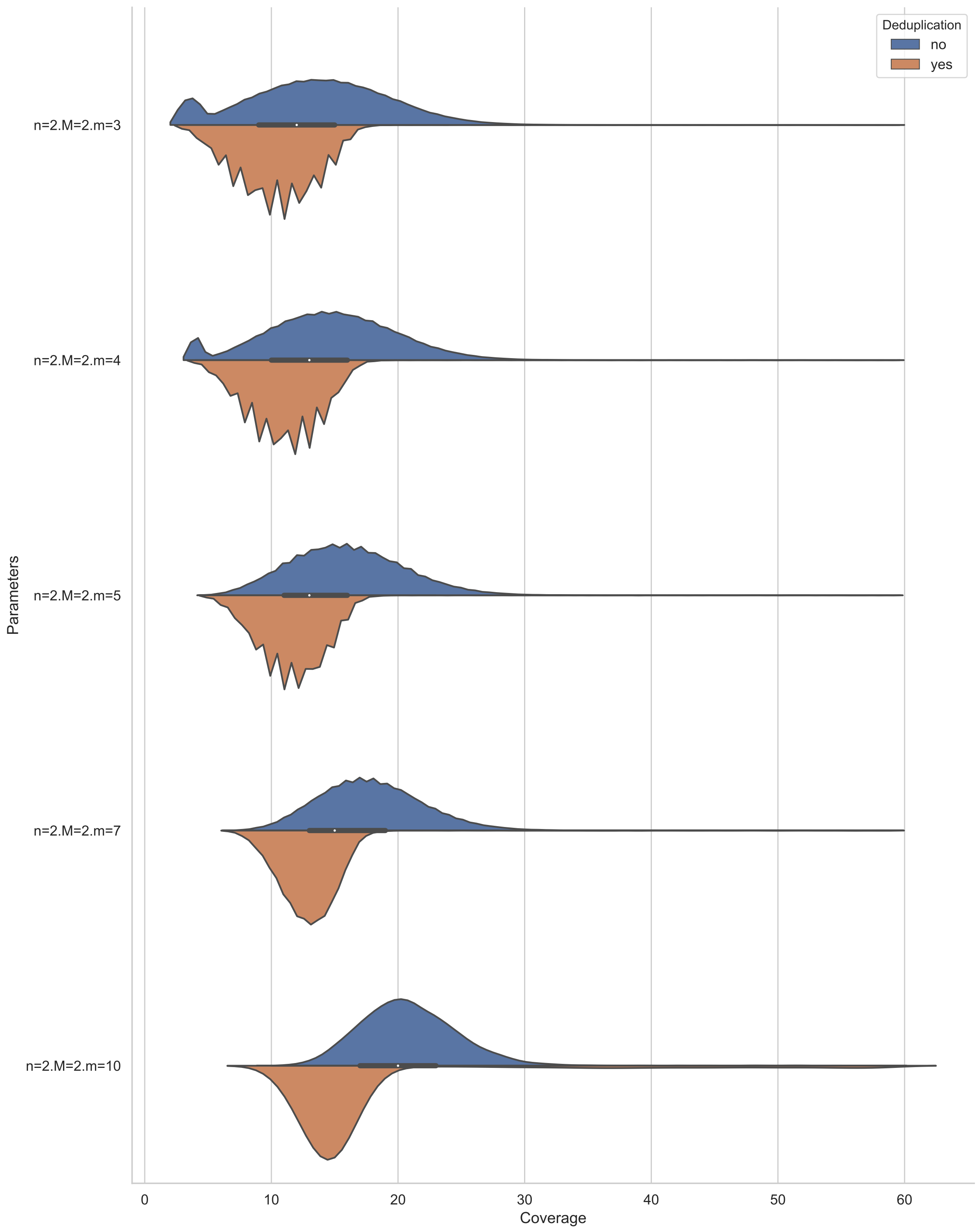


Figure 7.20: Violin plots for the low coverage case comparing the distributions of locus sizes (x-axis in reads per individual and locus) for different parameter sets for Stacks (rows). (*n*: maximum distance between sample loci for merging, *M*: maximum distance of loci for merging within a sample, *m*: minimum number of reads to form a locus.) Each parameter set was run both with (orange) and without (blue) our PCR deduplication step. Due to the lower coverage, coverage values above 60 were truncated to increase readability. Jagged shapes for low parameter values in the upper three plots are an artifact of Kernel Density Estimation used by the violin plots.

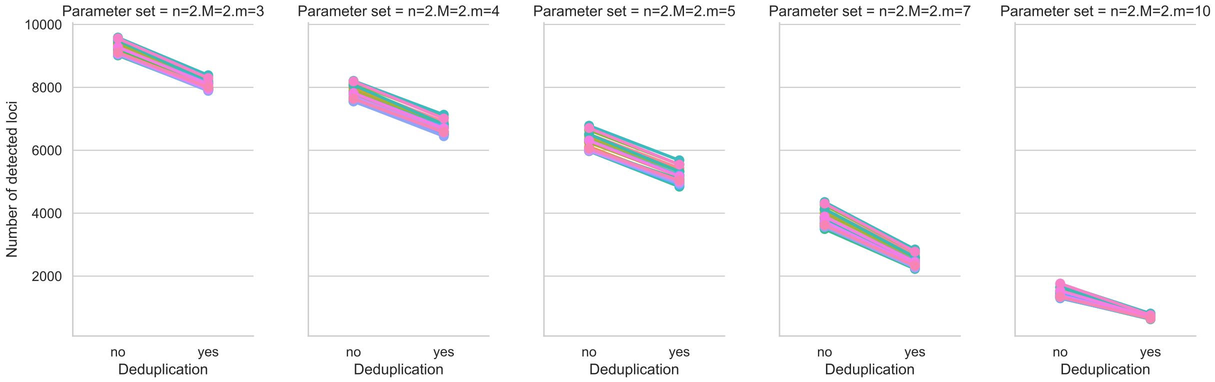


Figure 7.21: Point plots comparing the number of loci identified for each individual (y-axis) for runs without (left) and with (right) PCR deduplication. Individuals share a color and are connected. The expected number of loci for an individual in this dataset was ∼9 500 (10 000 simulated, with 500 expected dropouts).

Influence of Locus Distance (M, n) The M and n parameters both govern the merging of loci within and between samples. To judge the influence of these parameters, we simulated a dataset with a high level of diversity using the parameters shown in Table 7.9.

The high diversity parameter for DDRAGE increases the number of alleles in the mutation tree. Through this, alleles chosen for a mutation event have a higher chance to contain multiple mutations, and thus generate reads that differ significantly from the root allele. These changes are expected to cause loci to break up during the USTACKS clustering. Additionally, we chose a longer read length of 150 bp to allow for more sequence deviations. Increasing the values for the parameters M and n should mitigate the influence of this and allow the loci to be merged again. Higher values for M and n are expected to merge more loci and in turn result in a higher locus coverage.

Parameter	Value
`--loci`	10 000
`--individuals`	24
`--prob-seq-error`	0.01
`--read-length`	150
`--diversity`	12.0
`--event-probabilities`	0, 0, 1

Table 7.9: Parameters for the dataset simulated with DDRAGE. For all parameters that are not explicitly mentioned here, we used default parameters. The vector 0, 0, 1 passed to `--event-probabilities` denotes that no common and dropout events are simulated, only mutation events.

We analyzed the simulated dataset using two pipelines with and without PCR deduplication. The deduplication was performed with a DBR distance of 1 and a sequence distance of 8 (the maximum distance value possible, cf. Section 7.4.2), to compensate for the longer read length. For each pipeline, we evaluated the parameters described in Table 7.10.

Max. dist. (sample) `n`	Max. dist. (indiv.) `M`	Min. locus size `m`
1	2	3
2	2	3
3	3	3
5	5	3
10	10	3

Table 7.10: Parameter sets evaluated for the high diversity dataset. `n` is the number of mismatches for merging loci between samples, `M` is the permissible number of mismatches for merging loci, and `m` denotes the minimum coverage required to form a locus.

Figure 7.22 shows that the number of loci detected for each individual by USTACKS does not change with different parameter sets. However, when observing all individuals (i.e. the catalog generated by CSTACKS), the number of detected loci approaches the simulated number of 10 000 loci (no dropout events were simulated), as illustrated in Table 7.11. This indicates, that USTACKS misses different subsets of loci for each individual based on their specific alleles and allele coverages, but most simulated loci are contained in the catalog. For higher values of n, more loci are merged between samples when building the catalog, resulting in a number of loci that closely resembles the simulated data. While an increased number of merged loci can be seen both with and without deduplication, only

Another effect, which can be observed in Figure 7.23, is an increase in locus coverage with increasing parameter values for M

	Number of loci	
Parameter set	No deduplication	With deduplication
n=1.M=2.m=3	62 883	16 510
n=2.M=2.m=3	58 541	12 222
n=3.M=3.m=3	57 241	10 931
n=5.M=5.m=3	56 877	10 608
n=10.M=10.m=3	56 864	10 601

Table 7.11: Number of loci from the high diversity dataset contained in the CSTACKS catalog for different parameter sets, with and without deduplication.

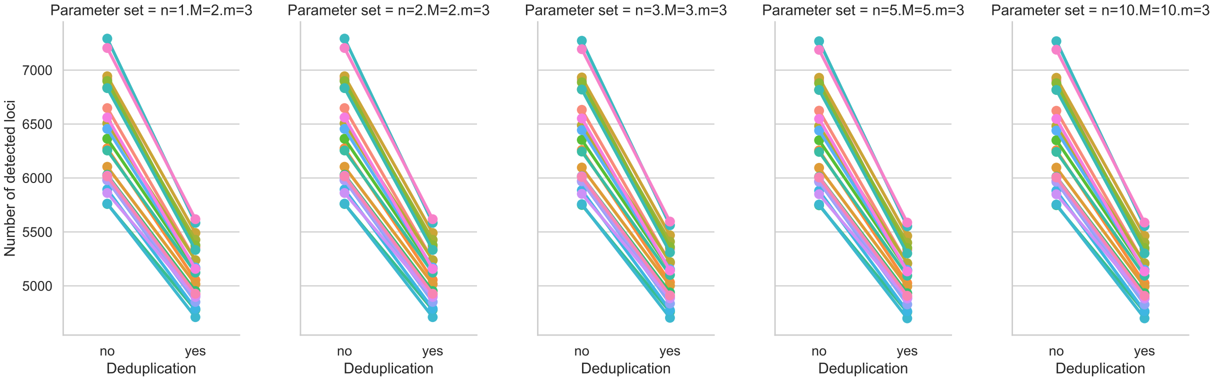


Figure 7.22: Point plots comparing the number of loci identified in the high diversity dataset for each individual (y-axis) for runs without (left) and with (right) PCR deduplication. Individuals share a color and are connected. The expected number of loci for an individual in this dataset was 10 000.

and n. This is to be expected, since merged loci result in higher locus coverage. However, it can also be seen that for higher parameter values, i.e. the lower subplots in Figure 7.23, the coverage distribution exceeds the simulated coverage of 30 both with and without deduplication. While overmerging loci is a possible explanation for this, collisions between sequences of 150 bp are unlikely. A more probable explanation is that this is caused by PCR duplicates, since for these longer read lengths the effect of PCR deduplication is reduced. The modes for deduplicated runs indicate a lower mean coverage than for non-deduplicated runs, which supports this hypothesis. However, it is not possible to choose a higher sequence similarity value, due to limitations of the clustering software used in our PCR deduplication workflow.

Note, that for low parameter values for M and n, i.e. the upper subplots in Figure 7.23, the coverage distributions are closer to the simulated coverage value of 30. When considering that not all PCR duplicates could be removed from the dataset, this indicates a lower locus coverage than simulated. The increase in locus coverage for higher values, relative to those for lower values, indicates that more loci are merged.

While high values for locus merging allow a more precise reconstruction of loci, they also increase the runtime of the USTACKS and CSTACKS programs. Values for M and n need to be chosen in relation to the available computational resources.

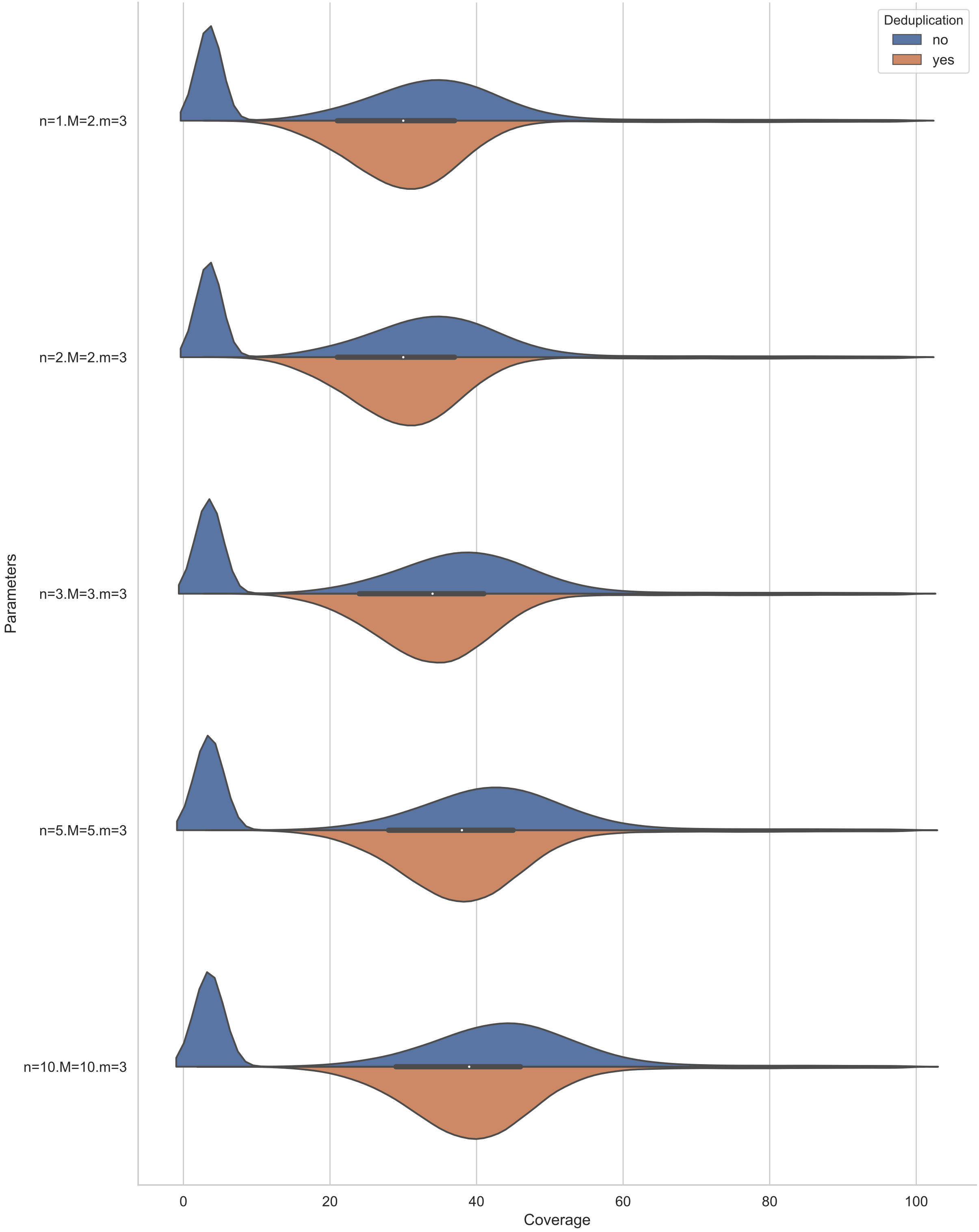


Figure 7.23: Violin plots for the high diversity dataset comparing the distributions of locus sizes (x-axis in reads per individual and locus) for different parameter sets for STACKS (rows). (*n*: maximum distance between sample loci for merging, *M*: maximum distance of loci for merging within a sample, *m*: minimum number of reads to form a locus.) Each parameter set was run both with (orange) and without (blue) our PCR deduplication step. Coverage values above 100 were truncated to increase readability.

Detection of SNPs To evaluate the mutations detected by our workflow, we compared the mutations identified by STACKS to the mutations contained in a simulated dataset. As before, we simulated a dataset using DDRAGE, using the parameters shown in Table 7.12. Since STACKS is not capable of detecting indel mutations and null allele mutations, we restricted the simulated mutations to SNPs. To judge the influence of PCR deduplication on the workflow, we executed two versions of the analysis workflow, one with and the other without PCR deduplication enabled. In order to rule out influence of incomplete digestion, we analyzed the dataset using only p5 reads through the `p5_only` read handling mode of our analysis pipeline.

Parameter	Value
`--loci`	10 000
`--individuals`	24
`--prob-seq-error`	0.01
`--coverage`	30
`--event-probs`	0.85, 0.05, 0.10
`--mutation-probs`	1, 0, 0, 0, 0, 0, 0

Table 7.12: Parameters for the SNP detection dataset simulated with DDRAGE. For all parameters that are not explicitly mentioned here, we used default parameters. The vector $0.85, 0.05, 0.10$ passed to `--event-probabilities` denotes that 10% of simulated events are *mutation* events. All of these mutations are SNPs, since the vector passed to the mutation probabilities parameter denotes a probability of 1 to select SNPs when adding an allele to the mutation tree.

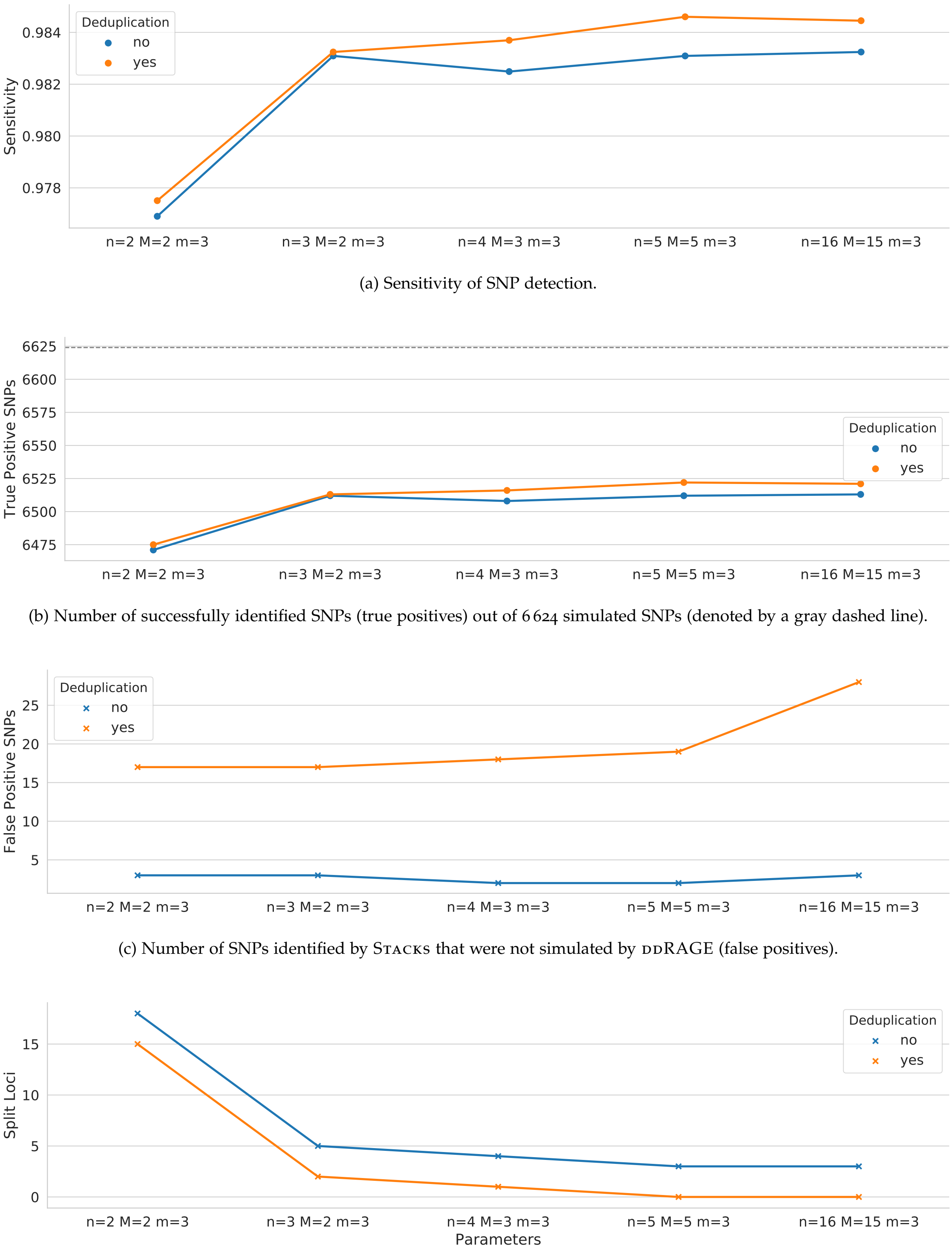
However, identifying that the correct mutations are detected by STACKS remains a challenging problem. During the USTACKS clustering, a gapped alignment is performed to compensate for (but not detect and classify) indel mutations and errors. This can introduce a difference between simulated mutation position and detected mutation position. Additionally, when the allele frequency $F(A, P)$ (cf. Allele Frequency, p. 17) for a simulated allele A in the simulated population P is higher than $F(A, P) = 0.5$, it is possible that the mutated allele is detected as reference. Introducing both of these effects into the analysis requires a high amount of error tolerance, which can lead to an increased false positive rate in the evaluation. We simplified this evaluation to comparing SNPs simulated by DDRAGE and detected by STACKS across all genotypes.

For each parameter set, we computed sequence similarities between locus sequences from the STACKS catalog and ground truth sequences using bottom-k sketching. Through this, we assigned STACKS loci to ground truth loci to compare their respective genotypes. We deemed locus sequences similar that received an estimated similarity of $\mathcal{J}^*(L_1, L_2) \geq 0.2$ and validated their similarity by computing their global alignment. The bottom sketch $S^{\underline{\vdots}}_{30}(Q(L, 7))$ of a locus sequence L was computed as described in section Bottom-k Sketch (p. 102). From this assignment, we computed the number of split-up loci, i.e. the number of ground truth loci, for which more than one STACKS locus was identified.

We subsequently compared all simulated p5 mutations to all detected p5 mutations on the locus level. To compensate for deviations during locus assembly, we deem the simulated mutation as detected if its base change (e.g. `C>G`) is detected with an offset of up to 2 positions from its simulated position. We additionally consider the respective inverted base changes (e.g. `C>G` and `G>C`) for these offset position to be a successfully detected SNP.

Figure 7.24 (a) and (b) show that the amount of correctly identified SNPs increases with higher parameter values for n and M.[46] The highest amount of correctly detected SNPs is reached by the parameter set n=5, M=5, m=3, with a sensitivity of ∼0.985. The parameter set n=2, M=2, m=3 shows a lower amount of correctly classified SNPs than the other parameter: About 35 SNPs less out

[46] To provide better context for the sensitivity values shown in (a), (b) depicts the absolute numbers of correctly detected SNPs.



(a) Sensitivity of SNP detection.

(b) Number of successfully identified SNPs (true positives) out of 6 624 simulated SNPs (denoted by a gray dashed line).

(c) Number of SNPs identified by Stacks that were not simulated by ddRAGE (false positives).

(d) Number of ground truth loci split up by Stacks.

Figure 7.24: Metrics for SNP analysis of a dataset simulated with ddRAGE both with (orange) and without (blue) PCR deduplication. (a) shows the sensitivity of SNP detection (b) show the total amount of correctly detected SNPs, (c) illustrates the amount of falsely identified SNPs, and (d) shows the number of split-up loci. Lines have been added to guide the eye for easier comparison of values.

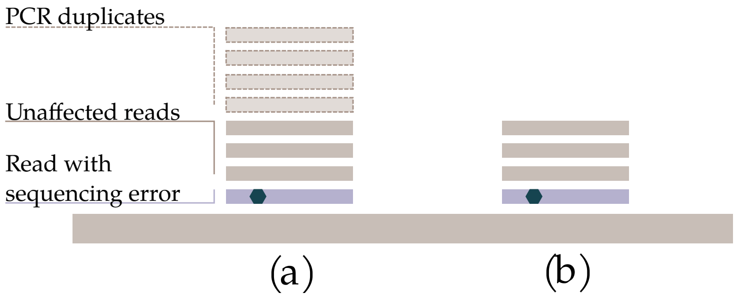


Figure 7.25: (a) Shows a locus without PCR deduplication. Valid reads (brown) in combination with their PCR duplicates (light brown, dashed) decrease the influence of the read with a sequencing error (purple). When removing PCR duplicates, as shown in (b), the ratio of unaffected reads to reads with sequencing errors is reduced, which could lead to errors being misclassified as mutations.

of 6 624 simulated SNPs were detected, compared with the following parameter set (n=3, M=2, m=3). For all parameter sets, the amount of true positive SNPs increased when using PCR deduplication. This increase in sensitivity can be attributed to better locus reconstruction due to deduplicated reads. For the lowest value of n=2, which denotes the maximum distance between sample loci, loci containing SNPs cannot be reliably merged. Thus, loci shared between individuals cannot be connected and identified as the same locus, resulting in two (or more) loci without mutations rather than one locus with a mutation.

Figure 7.24 (c) shows the number of erroneously identified SNPs, i.e. SNPs detected by Stacks that are not contained in the ground truth data. The amount of false positives for all parameter sets is higher when using PCR deduplication. Especially for n=16, M=15, m=3 with deduplication, an increase in false positives can be observed. A possible explanation for this effect is the reduced amount of PCR duplicates in combination with singular reads containing sequencing errors, as illustrated in Figure 7.25. During the simulation, not all reads receive PCR duplicates, hence there are reads that are analyzed as they were simulated. Sequencing errors in reads without PCR duplicates cannot be mitigated by PCR deduplication through consensus read computation. Removing PCR duplicates in this case reduces the (apparent) ratio of reads without this error to reads with the error which can result in it being misinterpreted as a SNP by Stacks. Another aspect that could potentially affect this is the PCR deduplication step itself, which could consolidate sequencing errors. However, identifying the exact cause of this increase is a topic for future research.

The amount of split loci decreases with increasing values of n and M. For all parameter sets, the amount of split loci is reduced by PCR deduplication, with no split-up loci remaining for n=5, M=5, m=3 and n=16, M=5, m=3 with PCR deduplication. The highest amount can be observed with 18 split loci for n=2, M=2, m=3 without deduplication. This is expected, as loci are split due to mismatching positions. PCR deduplication reduces the amount of mismatches through consensus read generation and the parameters n and M govern the merging process to restore such loci.

While this evaluation has shown, that the parameter set n=5, M=5, m=3 has arguably produced the best results, showing the

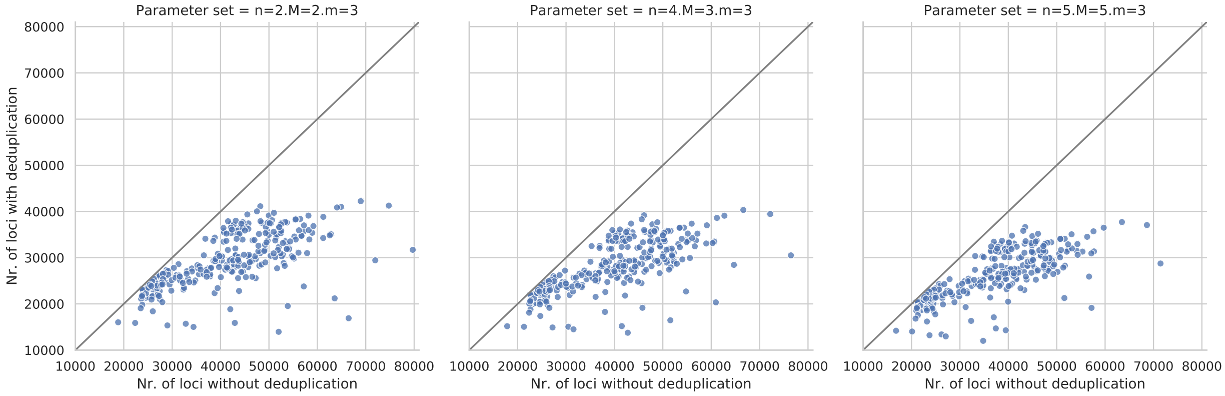


Figure 7.26: Scatter plot comparing the number of loci identified for each individual without deduplication (x-axis) and with deduplication (y-axis). Each point denotes one individual, columns show different parameter values. The gray diagonal denotes an identical number of loci before and after deduplication.

highest sensitivity without the increase in false positive rate and no remaining split up loci, these results cannot be generalized from our simulated dataset. However, for realistic datasets an analysis of split loci could be introduced to help identify the optimal parameter set. This evaluation of our pipeline could further be improved, by taking genotypes per individual, including their allele frequencies, into account.

Analyzing a Large In-house Dataset To judge the performance of our pipeline on real data, we analyzed an in-house ddRADseq dataset of *Gammarus fossarum*.[47] The dataset contained paired-end reads from 315 individuals, sequenced in 4 different lanes using an Illumina HiSeq 2500 device. Three lanes were sequenced with a read length of 125 bp, the remaining lane was sequenced with a length of 150 bp. Note, that before reads are passed to Stacks for analysis, all reads are trimmed to the same length. In total, the input data comprise ∼103 GB of gzipped FASTQ files.

We analyzed this dataset using the parameter sets described in Table 7.13. As above we applied our workflow twice, once with deduplication enabled and once without deduplication, using a DBR distance of 1 and a sequence distance of 7.

Max. dist. (sample) `n`	Max. dist. (indiv.) `M`	Min. locus size `m`
2	2	3
4	3	3
5	5	3

Table 7.13: Parameter sets evaluated for the *G. fossarum* dataset. `n` is the number of mismatches for merging loci between samples, `M` is the permissible number of mismatches for merging loci, and `m` denotes the minimum coverage required to form a locus.

Figure 7.26 shows the number of detected loci both with and without PCR deduplication. A point plot as shown in the previous evaluations can be found as Figure D.2 in Appendix D. As before, applying PCR deduplication reduces the number of detected loci per individual. It can be seen that the number of detected loci is reduced for all individuals, since all data points fall below the diagonal. While without deduplication some individuals show a number of up to 800 000 loci, the maximum number of loci observed after deduplication is reduced to ∼41 000. This overall drop in locus detection through PCR deduplication can be attributed to the fact that singleton reads can no longer form loci after their PCR duplicates have been removed.

Another effect that can be observed is that the amount of loci per individual decreases with increasing parameter values. This can be

[47] Clade 11 (Type B)

	Number of identified SNPs	
Parameter set	No deduplication	With deduplication
n=2, M=2, m=3	545 524	184 355
n=4, M=3, m=3	752 467	216 449
n=5, M=5, m=3	802 459	230 391

Table 7.14: Number of SNPs identified by our workflow in the *G. fossarum* dataset for different parameter sets, without (left) and with (right) PCR deduplication.

explained by a higher number of loci merged during clustering and catalog assembly in the Stacks pipeline. Higher values of n and M allow loci with more differing bases to be merged, regardless if these deviations stem from sequencing errors or from SNPs.

Figure 7.27 show the distribution of stack sizes for the *G. fossarum* dataset. Compared with simulated datasets shown in the previous sections, the effect of deduplication is less pronounced. This is the case due to the lower coverage of the real dataset, while our simulated ddRAD loci were simulated with very high coverage. In this scenario, low coverage loci comprise both actual loci with only a few associated reads as well as singletons with PCR duplicates.

It can be seen that the number of loci with coverage $3-5$ is reduced through PCR deduplication. This is caused by the removal of singleton reads. In contrast to the simulated datasets analyzed in the previous sections, the coverage distribution of this dataset is strongly biased towards lower coverages. The fact that a large number of low coverage loci remain after PCR deduplication can be attributed to two effects: These loci are either valid low coverage loci that should not be removed or that they comprise PCR duplicates that exceed the sequence distance that can be compensated by our PCR deduplication workflow. As we describe in Section 7.4.2, the number of sequence differences that can be compensated is currently limited by the employed clustering software. The longer read lengths present in this dataset result in a higher amount of possible sequencing errors, which can prevent the successful deduplication of reads.

Concerning the number of identified SNPs in the dataset, we can also see a strong influence of both deduplication and different parameter choices. Table 7.14 shows the number of SNPs identified in the *G. fossarum* dataset.

Regardless of deduplication, for the parameter sets with higher values of M and n, which allow more loci to be merged, the number of SNPs increases. This is expected, since loci containing SNPs are split up during the first step of the ustacks clustering. If they cannot be merged again, either in the later steps of the ustacks clustering or during catalog construction by cstacks, such SNPs cannot be identified. The multiple locus fragments that emerge through this behavior show a low amount of genetic variation, since different alleles of mutations are distributed to the different fragments. This increases the number of loci detected, while also reducing the number of identified SNPs. Through higher tolerances for

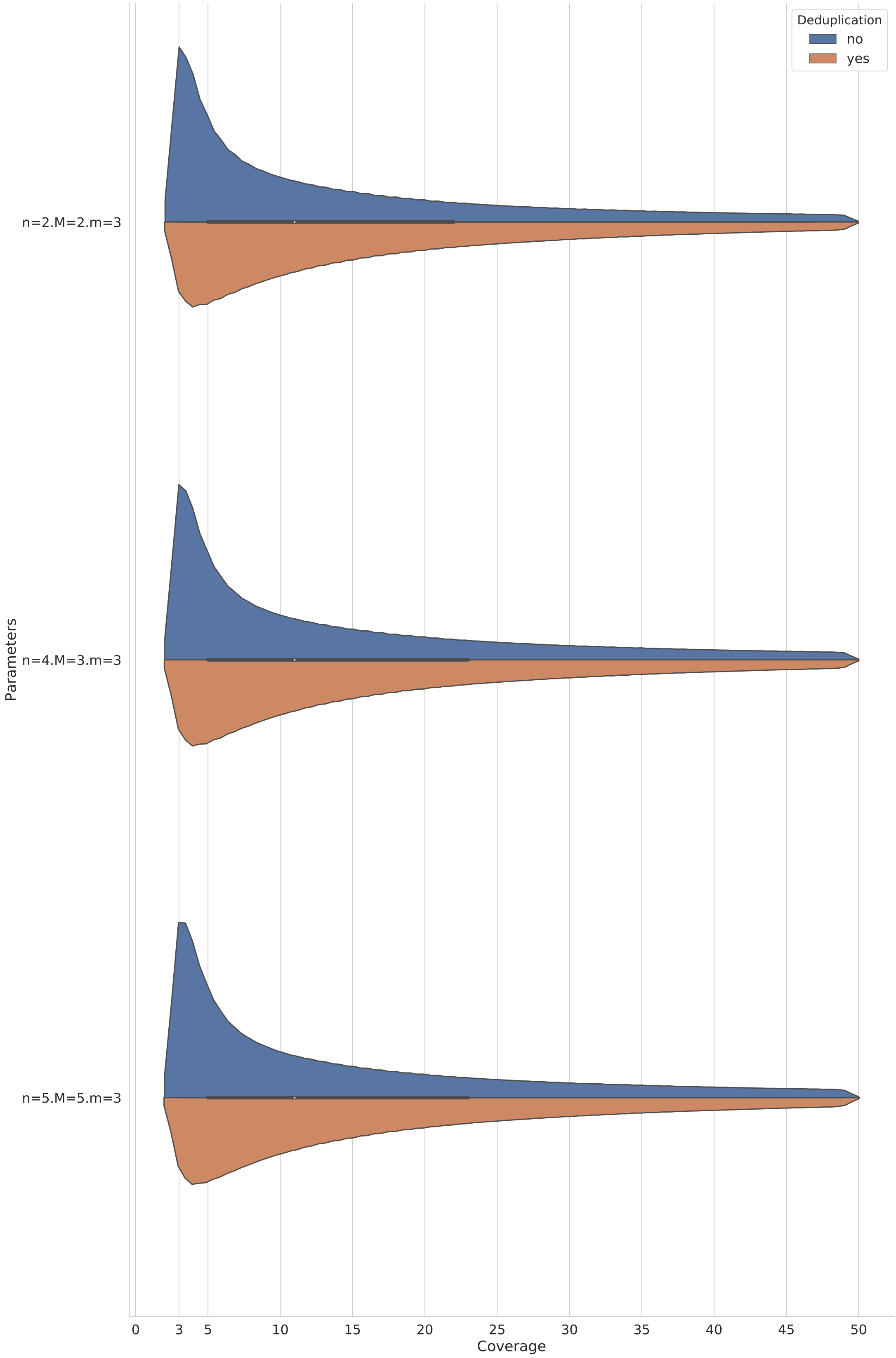


Figure 7.27: Violin plots for the *G. fossarum* dataset comparing the distributions of locus sizes (x-axis in reads per individual and locus) for different parameter sets for Stacks (rows). (*n*: maximum distance between sample loci for merging, *M*: maximum distance of loci for merging within a sample, *m*: minimum number of reads to form a locus.) Each parameter set was run both with (orange) and without (blue) our PCR deduplication step. Coverage values above 50 were truncated to increase readability. An additional tick on the *x*-axis at a coverage of 3 denotes the smallest used value of *m*. A version of this plot showing coverages up to 1 000 can be found in Appendix D (Figure D.3).

merging, the probability to merge these locus fragments increases, resulting in the observed increasing number of SNPs.

The number of identified SNPs is about ∼3 times higher without deduplication. This can be attributed to the removal of small loci as well as reducing the influence of sequencing errors. Small loci have a higher chance to misidentify sequencing errors as SNPs, since the available evidence that could distinguish these cases is limited. Especially for singletons and their PCR duplicates, single sequencing errors can be mistaken as SNPs. Removing these loci reduces the number of misidentified SNPs. Additionally, our PCR deduplication computed consensus reads, which reduces the impact of sequencing errors. If a read and several of its PCR duplicates are present, single sequencing errors do not persist in the consensus read. Especially in a low coverage scenario as with the dataset analyzed here, PCR deduplication has the potential to reduce the number of misidentified SNPs.

7.3.3 Discussion

We implemented a workflow for ddRAD analysis, which allows to explore several parameter choices for the analysis in parallel, based on the Stacks software. Using several preprocessing steps, we prepared read data once, and were able to reuse them for all further analysis steps. Through the use of Snakemake and conda, our workflow generates reproducible results and is easily portable to new platforms, including cluster or cloud computing environments. Finally, we provide a condensed overview of the number of loci and the size distribution of loci, both per individual and per sample, as crucial metrics to judge the analyzed data.

Using simulated data, we could show that our PCR deduplication step is able to remove singleton reads from the dataset, thereby significantly reducing the number of loci that require downstream analysis. Through exploring multiple parameter sets, we could identify loss of loci through high minimal locus sizes in a low coverage scenario and offered means to detect this effect. Additionally, we have shown that for high diversity datasets, increasing values for locus distances receive a higher locus coverage. For datasets with less diversity, low choices for the evaluated parameters in combination with our PCR deduplication approach yielded high quality results. Since values that allow higher differences between loci increase the overall runtime, especially of the cstacks program, these should not be used extensively. However, our workflow structure allows the exploration of these parameters limited to one parameter set and compare the results of both executions. We suggest combining increasing values for the minimal locus size (m) with several low and at most one high parameter set for M and n. This allows the identification of both unexpectedly low coverage data as well as high diversity data.

For a real dataset, we could show that the amount of SNPs detected is linked to the choice of the M and n parameters. We observed an increased number of detected SNPs for higher values of M and n, likely due to the re-merging of loci split up by their SNPs during clustering as well as correctly merging loci with variants between samples. Applying PCR deduplication reduced the number of identified SNPs, which can be attributed to the removal of both singletons and PCR duplicates forming loci as well as to compensation of sequencing errors through consensus read computation.

For all analyses, the result quality was increased by the application of PCR deduplication. Coverage distributions were closer to the simulated coverage values, which can increase the quality of genotype detection. Through the removal of PCR duplicates, allele frequencies can be reconstructed more precisely and erroneously called genotypes in low coverage scenarios can be mitigated. To further optimize this process, especially for longer reads, a version of our PCR deduplication process that allows larger sequence differences is required. Since this is a restriction of the STARCODE[48] software[49] used for the read clustering in this process, a different clustering approach is required.

During the SNP analysis, PCR deduplication increased the sensitivity of SNP detection and reduced the number of split-up loci at the cost of an increased false positive rate. Our simplified SNP evaluation could be improved by using a more sophisticated approach for variant comparison. While the comparison of locus sequences is straight forward, validating whether the correct mutation was detected is impeded by small deviations in alignment in combination with the uncertainty which allele will be identified as normal and which as mutated by STACKS. A solution for this could be to convert the ground truth file generated by DDRAGE to a VCF file and compare the VCF file generated by STACKS with this ground truth VCF. For this comparison, specialized tools like `hap.py`[50] could be employed.

[48] Zorita, Cusco, and Filion, "Starcode: Sequence Clustering Based on All-pairs Search", 2015.

[49] Cf. Deduplication Workflow p. 214.

[50] `github.com/Illumina/hap.py`

7.4 PCR Duplicate Removal

Removing PCR duplicates from SGS reads is a variation of the read clustering problem. We have to find reads with identical DBR[51] as well as identical genomic sequences and from these keep only one read, which is representative for the cluster. To solve this problem, we implemented the CALL-CONSENSUS-READS software in cooperation with Johannes Köster and Felix Mölder, which is able to merge PCR duplicates of PE reads into consensus reads. CALL-CONSENSUS-READS is implemented in the Rust programming language and is available as part of the RUST-BIO-TOOLS[52] software package.

7.4.1 Problem Definition and Related Work

Consider a read and its PCR duplicates, for example a ddRAD read as described in the previous section. Differences between the original and its copies can be classified into two categories:

- Sequencing errors, added during sequencing. As before, we focus on SGS reads and the Illumina error model (cf. Illumina – Cyclic Reversible Termination p. 21).
- PCR errors, introduced during the PCR duplication process (cf. Polymerase Chain Reaction p. 20).

Due to their respective origins, sequencing errors are expected to be present in one read, while PCR errors can affect multiple reads, depending on the PCR cycles in which they were introduced.

To merge all PCR duplicates of a read, we need to account for these deviations, using an error tolerant clustering approach. Since error rates can vary, errors rates of the clustering need to be chosen depending on the number of sequencing errors and PCR errors expected in the data.

Finally, for a computed cluster, we need to decide on a final read to return. Picking one read from the input, either at random or using some heuristic, has a high probability to contain at least one error. There are several approaches to solve this problem.

UMI-TOOLS[53] can deduplicate reads using aligned SAM or BAM files. For each cluster it selects the read with highest mapping quality and lowest number of mapping coordinates as representative. If these measures cannot be used, a read is chosen at random. The FGBIO toolkit[54] also offers PCR deduplication for aligned SAM or BAM files. Consensus bases are computed using a likelihood-based approach, incorporating the information about base qualities. STARCODE[55] can also perform PCR deduplication through the STARCODE-UMI script. The consensus read returned by STARCODE-UMI is the centroid of the cluster, i.e. a read selected from the input. For PE reads, STARCODE-UMI requires a DBR sequence on each read. The CALIB software[56] for UMI clustering and deduplication performs a base-wise majority voting to determine consensus bases

[51] We continue to use the term DBR here, to be consistent with the previous sections. For other fields, the term UMI is more widely used.

[52] `github.com/rust-bio/rust-bio-tools`. CALL-CONSENSUS-READS was stabilized in version 0.5.0. Note that it has since been renamed COLLAPSE-READS-TO-FRAGMENTS in newer versions.

[53] Smith, Heger, and Sudbery, "UMI-tools: Modeling Sequencing Errors in Unique Molecular Identifiers to Improve Quantification Accuracy", 2017.

[54] `https://github.com/fulcrumgenomics/fgbio`

[55] Zorita, Cusco, and Filion, "Starcode: Sequence Clustering Based on All-pairs Search", 2015.

[56] Orabi et al., "Alignment-free Clustering of UMI Tagged DNA Molecules", 2018.

with four possible quality values. This does not preserve the known and quantifiable uncertainty about these bases in the consensus read.

Approaches that rely on the presence of aligned reads in SAM or BAM files are not feasible for ddRADseq data for which usually no reference is available. For example the consensus read computation of FGBIO requires BAM files sorted by DBR sequences and mapping positions. While the mapping positions could be derived from an external clustering step and we could construct a BAM file to pass to the consensus calling step of FGBIO, this approach requires several additional files to be written. This significantly increases the disk space requirements and runtime overhead of this approach. The STARCODE-UMI approach requires a specific read setup and DBR position, which is not satisfied by ddRADseq reads. While CALIB is able to process FASTQ files, its quality value computation is too coarse too accurately reflect the certainty of the selected consensus base. Consequently, we developed a PCR deduplication workflow that works on FASTQ files directly and offers a configurable DBR position. We compute a consensus sequence assembled from all reads within the cluster, which also quantifies and preserves the certainty of each consensus base.

7.4.2 *Deduplication Workflow*

Our workflow to solve the problem stated above comprises three steps:

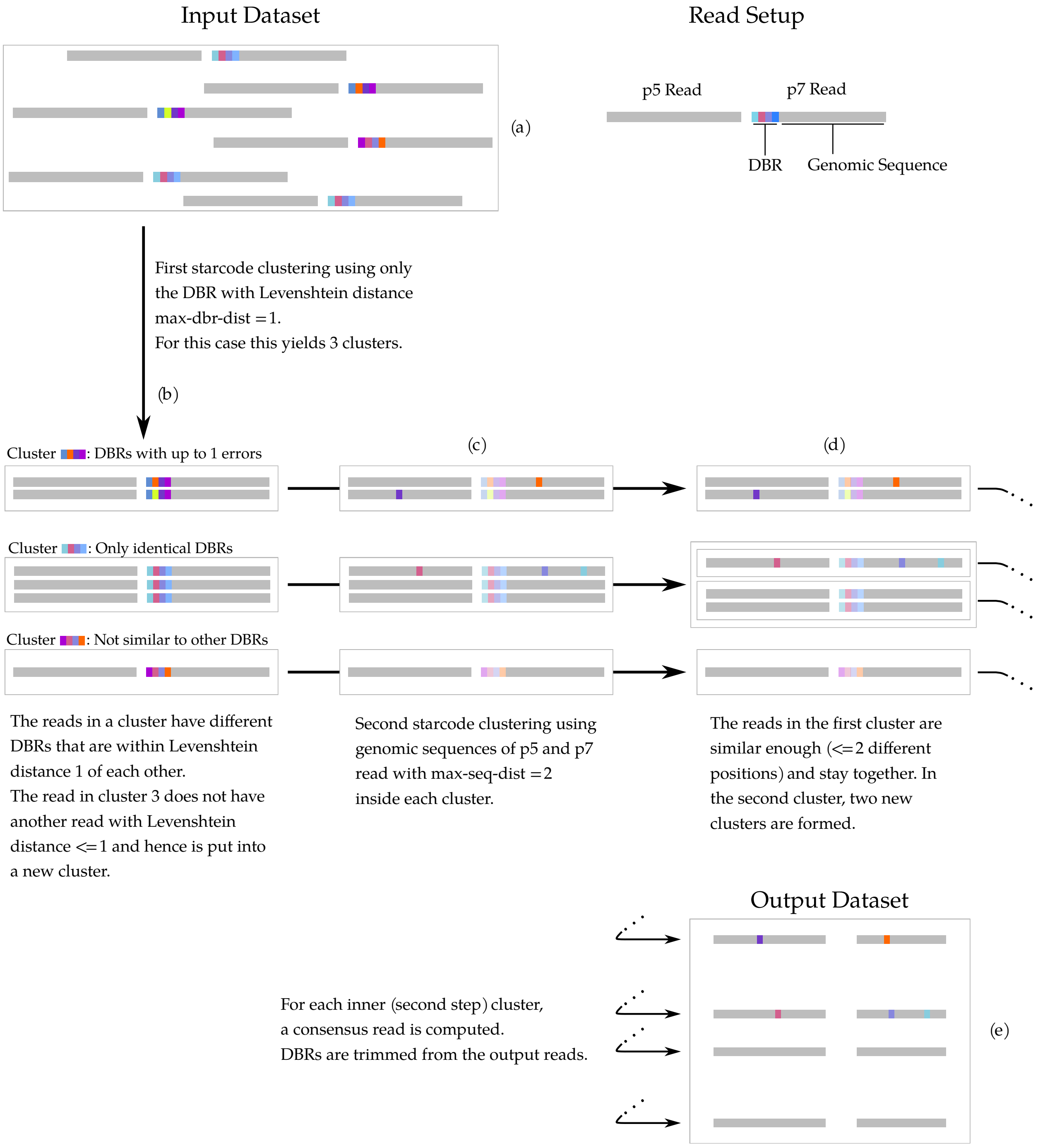
1. Cluster reads by DBR (first order clustering)
2. Cluster each read within a first order cluster by sequence (second order clustering)
3. Compute consensus reads for each second order cluster

We perform both clustering steps using STARCODE,[57] which clusters sequences using all-pairs search and a variation of the Needleman-Wunsch algorithm. This recursive clustering algorithm is designed to correct sequencing errors while preventing overmerging. The STARCODE clustering can be parameterized to allow matches within a certain Levenshtein distance (at most 8). A sequence clusters is merged with its closest neighbor as long as this neighboring cluster is within the specified distance and contains at least five times the number of reads.

An illustration of our deduplication workflow can be found in Figure 7.28.

First Order Clustering During the first step of our pipeline, we extract the DBRs from the reads and cluster them with STARCODE. Since the DBR usually is short with respect to the read length, we allow only a few errors. We discuss the parameter choices in detail in the section Evaluation and Parameter Choices. To avoid

[57] Zorita, Cusco, and Filion, "Starcode: Sequence Clustering Based on All-pairs Search", 2015.


Input Dataset
(a)
Read Setup
p5 Read
p7 Read
DBR
Genomic Sequence
(b)
(c)
(d)
Cluster ▨: DBRs with up to 1 errors
Cluster ▨: Only identical DBRs
Cluster ▨: Not similar to other DBRs
Output Dataset
(e)

First starcode clustering using only the DBR with Levenshtein distance max-dbr-dist = 1.
For this case this yields 3 clusters.

The reads in a cluster have different DBRs that are within Levenshtein distance 1 of each other.
The read in cluster 3 does not have another read with Levenshtein distance <= 1 and hence is put into a new cluster.

Second starcode clustering using genomic sequences of p5 and p7 read with max-seq-dist = 2 inside each cluster.

The reads in the first cluster are similar enough (<= 2 different positions) and stay together. In the second cluster, two new clusters are formed.

For each inner (second step) cluster, a consensus read is computed.
DBRs are trimmed from the output reads.

Figure 7.28: Example for the PCR deduplication workflow of CALL-CONSENSUS-READS. Paired-end SGS reads are denoted by a pairs of gray bars, where colored segments on the p7 read denote a specific DBR configuration. Light gray boxes that enclose groups reads denote clusters and datasets. In the first step, reads from the input set (a) are clustered with STARCODE, using only their DBR sequences, allowing for a Levenshtein distance of 1. The resulting clustering (b), groups all reads with similar DBRs. Since $|\text{DBRs}| << |\text{reads}|$, each cluster contains reads with potentially differing genomic sequences, denoted by colored squares outside the DBR in (c). For each cluster in (c), another clustering with STARCODE is performed, to split these reads apart. The resulting inner clusters (d) contain reads with both identical DBR as well as genomic sequences and are merged into consensus reads, which are written into the output dataset (e).

reading input files multiple times, we keep read sequences within a RocksDB[58] key value store. Since we cluster only on DBR sequences, this step partitions the problem into approximately as many sub-problems, as there are different possible combinations of DBR sequences.

Second Order Clustering For each of the clusters generated in the first step, we retrieve the read sequences without the DBR from the RocksDB. We concatenate p5 and p7 genomic sequence into one sequence and perform another clustering on these sequences. Since all reads within one first order cluster share (very) similar DBRs, reads that also share genomic sequences are likely to be PCR duplicates. For this second clustering we allow more errors, due to the longer sequences.

Consensus Read Computation For each cluster of reads identified by the second order clustering, both DBR and genomic sequence are similar. To merge these reads into a consensus read, we use a maximum a-posteriori estimator over each position of the read. At this point, we require all reads within one second order cluster to be of same length. Singleton clusters, i.e. second order cluster that contain only one read, are written directly to the output file.

We compute the bases and quality values for the consensus read based on the work presented by DePristo et al.[59] and Li,[60] similar to the approach used by FGBIO.[61] For each position, the maximum a-posteriori estimate for an allele $\theta \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ at position i are computed as

$$\hat{\theta}_i = \underset{\theta \in \{\texttt{A,C,G,T}\}}{\arg\max}\ L_i(\theta) \tag{7.5}$$

with the likelihood function:

$$L_i(\theta) = \prod_{j=1}^{n} f_i(\theta, j) \qquad f_i(\theta, j) := \begin{cases} 1 - \omega_{i,j} & \theta = r_{j,i} \\ c \cdot \omega_{i,j} & \text{else} \end{cases} \tag{7.6}$$

Here, n is the number of reads in the cluster, r_j is the j-th read assigned to the cluster with $r_{j,i}$ being the i-th base in said read. The function $f_i(\theta, j)$ computes the posterior probability to observe the allele $r_{j,i}$ in the read, given that the real allele was θ. The confusion constant $c = \frac{1}{3}$ models the probability of observing each of the remaining three alleles from $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\} \setminus \theta$ in case of a miscalled base. As weight for each base, we use the error probability $\omega_{i,j}$ of read j at position i, computed from the Phred score of the corresponding base (as reported by the sequencer). In other words, for one column, we examine each of the four alleles $\theta \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ and compute the likelihood to observe the bases present in the column, given that the real base was θ. The base written to the consensus read is the allele with the highest likelihood, i.e. the allele with the most supporting reads weighted by Phred score. An example for this is illustrated in Figure 7.29.

[58] github.com/facebook/rocksdb

[59] DePristo et al., "A Framework for Variation Discovery and Genotyping Using Next-generation DNA Sequencing Data", 2011.

[60] Li, *Mathematical Notes on SAMtools Algorithms*, 2010.

[61] Fennel and Homer, *fgbio*, 2017.

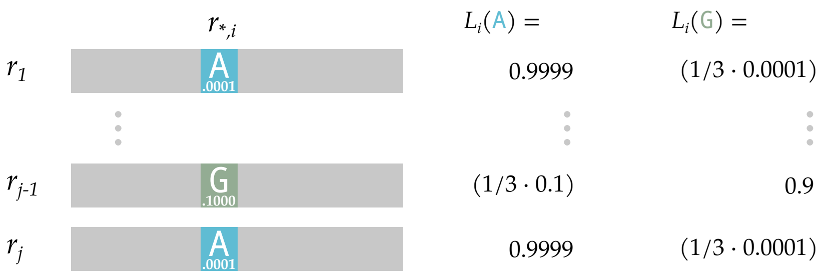


Figure 7.29: Example of likelihood computation for the alleles A and G for one second order cluster. Gray bars denote reads with colored blocks denoting bases (top) and error probabilities (bottom) of the active column i. The likelihood for the allele A is high, since two of the shown bases support this with low error probability. Likelihood computations for the alleles A and G are shown on the right.

We compute an error probability $\hat{\omega}_i$ for each position of the consensus read as

$$\hat{\omega}_i = 1 - \frac{L_i(\hat{\theta})}{\sum_{\theta \in \{\texttt{A,C,G,T}\}} L_i(\theta)} \tag{7.7}$$

where $\hat{\theta}$ is the maximum posterior from Equation 7.5.

For a singular read we retain both bases and quality values of the input. Consider a read that contains the allele θ^* at position i. Assuming θ^* was called with a Phred score of at least 2 ($\omega < 0.75$), the likelihood $L_i(\theta^*) = 1 - \omega_{i,1}$ is larger than for the other alleles with $L_i(\theta) = c \cdot \omega_{i,1} \quad \theta \neq \theta^*$.

$$\begin{aligned}
& & \frac{1}{3} \cdot \omega_{i,1} &< 1 - \omega_{i,1} \\
&\equiv & \omega_{i,1} &< 3(1 - \omega_{i,1}) \\
&\equiv & \omega_{i,1} &< 3 - 3\omega_{i,1} \\
&\equiv & 4\omega_{i,1} &< 3 \\
&\equiv & \omega_{i,1} &< \frac{3}{4}
\end{aligned}$$

Consequently, θ^* is selected as base for the consensus read. The quality value of θ^* is computed

$$\begin{aligned}
\hat{\omega}_i &= 1 - \frac{1 - \omega_{i,1}}{3 \cdot \frac{1}{3} \cdot \omega_{i,1} + 1 - \omega_{i,1}} \\
&= 1 - \frac{1 - \omega_{i,1}}{\omega_{i,1} + 1 - \omega_{i,1}} \\
&= 1 - (1 - \omega_{i,1}) \\
&= \omega_{i,1}
\end{aligned}$$

and collapses back into $\omega_{i,1}$. In our implementation we omit this computation for singleton clusters by directly returning the input read.

As an example for more than one read, consider a cluster with two reads. At position i, the first read contains A with $\omega_{i,1} = 0.001$, the second read contains G with $\omega_{i,2} = 0.1$. The likelihood values

$$\begin{aligned}
L_i(\mathtt{A}) &= (1-0.001)\cdot\frac{0.1}{3} &&= \frac{0.0999}{3}\\
L_i(\mathtt{C}) &= \frac{0.001}{3}\cdot\frac{0.1}{3} &&= \frac{0.0001}{9}\\
L_i(\mathtt{G}) &= \frac{0.001}{3}\cdot(1-0.1) &&= \frac{0.0009}{3}\\
L_i(\mathtt{T}) &= \frac{0.001}{3}\cdot\frac{0.1}{3} &&= \frac{0.0001}{9}
\end{aligned}$$

are maximized by $L_i(\mathtt{A}) = 0.0333$, for which we compute the error probability $\hat{\omega}_i$ as:

$$\hat{\omega}_i = 1 - \frac{0.0333}{\frac{0.0999}{3}+\frac{0.0001}{9}+\frac{0.0009}{3}+\frac{0.0001}{9}} \approx 0.01$$

We can see that the quality for the consensus allele A falls between $\omega_{i,1}$ and $\omega_{i,2}$.

By performing the estimation described above for all positions in the reads, we compute the consensus read

$$\hat{r} = (\hat{r}_i)_{i=0}^{m} \qquad \hat{r}_i = \hat{\theta}_i \tag{7.8}$$

with quality values

$$q = (q_i)_{i=0}^{m} = (1-\hat{\omega}_i)_{i=0}^{m} \tag{7.9}$$

where m is the length of the input reads. This read is written as output, alongside an annotation how many reads mere merged into this consensus read. Since at this point the DBR has served its purpose, it is not written to the new FASTQ file.

After each second order cluster has been processed, either by merging it into a consensus read or by writing it to file as a singleton, the pipeline terminates. In the next section we show, how the specified Levenshtein distances influence the results of the workflow.

7.4.3 Evaluation and Parameter Choices

While we have already shown in the previous section the positive influence our PCR deduplication has on ddRAD analysis by eliminating singletons, we will now evaluate its influence on cluster sizes. Additionally, we explain how we chose the default parameters. The analysis workflow described in this section[62] was published on Zenodo and GitHub.[63]

The main technological factors influencing our PCR deduplication workflow are the length of the DBR and genomic sequence as well as the expected error rate. Assuming the Illumina error model, which introduces substitution errors with a probability of $p_e \approx 0.01$, we can estimate the number of errors expected to affect DBR and sequence. For a PE read of length $2\cdot 100$ bp, with a DBR of length 13, we expect to see

$$|\text{DBR}|\cdot p_e = 0.13$$

[62] Timm, *PCR Deduplication Analysis Workflow*, 2021.

[63] github.com/HenningTimm/pcr_deduplication_analysis_workflow

sequencing errors within the DBR and

$$((2 \cdot 100) - |\mathrm{DBR}|) \cdot p_e = \frac{100 + 87}{100} = 1.87$$

errors within the genomic region. Consequently, we use $d = \lceil 0.13 \rceil = 1$ for the DBR and $d = \lceil 1.87 \rceil = 2$ for the genomic sequence as the default parameter choices for the Levenshtein distances allowed during the respective clustering steps.

To evaluate the impact of these parameters, we simulated reads with DDRAGE that contain a known amount of PCR duplicates. Simulated datasets did not contain any HRLs, singletons, ID, or dropout events, which could obfuscate the validation of the results. Mutations simulated for the dataset were all homozygous. Apart from that, we used default parameters, most notably a substitution error rate of $p_e = 0.01$, a coverage of 30, and the DBR sequence `NNNNNNMMGGACG` (which has 16 348 possible configurations).

Figure 7.30 shows the number of actual non-duplicate reads per locus before duplication, locus sizes including all reads, and locus sizes after deduplication. The first two values were derived from DDRAGE's ground truth. For this we employed our python package DINOPY,[64] which offers convenient access to optimized and compiled input and output classes for FASTA, FASTQ, and other file types for biological sequences. To compute coverage values after deduplication, we extracted the names of source loci for each read from the FASTQ annotation written by DDRAGE. Using these, we assembled a list of source loci for each second order cluster in the FASTQ annotation of the deduplicated reads. We then analyzed how many reads remained after deduplication and screened for erroneously merged reads using DINOPY. Across all parameter combinations, no erroneous merge (i.e. reads from different loci merged into a single consensus read) was performed.

As illustrated in Figure 7.30, an increasing number of permissible errors during first and second order clustering increases the number of deduplicated reads. It can be seen that the maximal Levenshtein distance permissible for the DBR clustering (1 in the left and 2 in the right column) does not have a strong influence on the distribution. This is to be expected, since the DBR takes up only a small fraction of the read and hence has a low probability to contain only one error. As shown above, the number of expected sequencing errors within the DBR used for this evaluation is 0.13, which is surpassed even by a Levenshtein distance of 1.

The number of permissible errors between sequences (max. sequence distance 1 – 4, 6, and 8, shown along the rows) has a stronger influence on the results. While the distribution of read sizes is reduced even for a Levenshtein distance of 1, a notable fraction of PCR duplicates remains in the data. For Levenshtein distances 2 – 4, the distributions further approximate the ground truth distribution. The locus size distributions for a maximal sequence distance of 6 and 8 most closely approximate the ground truth. This trend can be expected to continue with increasing val-

[64] Timm and Hartmann, *Dinopy — DNA input and output for Python and Cython*, 2020.

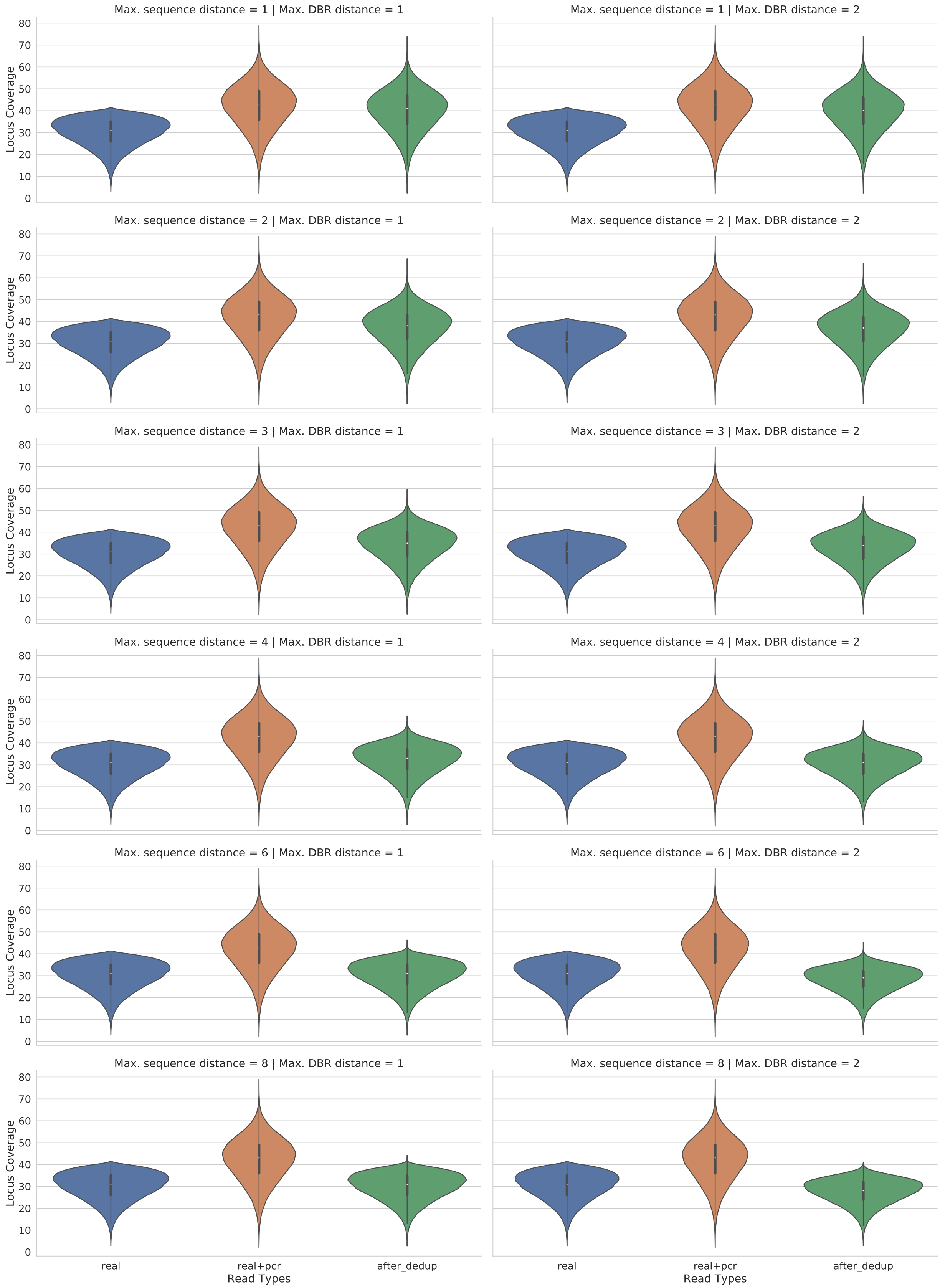


Figure 7.30: Size distribution of 100 000 loci with an expected coverage of 30 for one individual, simulated by ddRAGE. No HRLs, singletons, and dropout events were simulated and all simulated mutations were homozygous. For all remaining parameters, default values were used, including a substitution error rate of 0.01 and a read length of 100. The left (blue) distribution shows the real locus sizes without any PCR duplicates. In the middle (orange), the distribution including PCR duplicates is illustrated. The right (green) plot depicts the distribution of locus sizes after PCR deduplication. Each facet of the plot shows one combination of maximum DBR distance (x-axis, 1 – 2) and maximum sequence distance (y-axis, 1 – 4, 6, 8).

ues, until overmerging starts to create large clusters. Note however, that starcode restricts Levenshtein distances for clustering to be at most 8. However, as shown by the upper tailing of the violins (and especially the upper adjuncts, denoted as vertical gray lines), some loci with an increased number of reads remain. This can occur for single reads that receive a high number of sequencing errors and are not associated to the same cluster in the second clustering step.

Figure 7.31 plots the simulated coverage for a locus (x-axis) against its coverage after deduplication (y-axis). For a perfect deduplication, all points would need to align on the red diagonal line. Before any deduplication is performed, all loci are above or exactly on the diagonal, depending on the number of PCR duplicates simulated for the locus. This can be seen for low parameter values as in the upper left facet of Figure 7.31.

With increasing parameter values, loci coverages shift towards the simulated diagonal. For an increasing number of permissible sequence errors, there are values which closely resemble the simulated coverage. Most notably, a DBR distance of 1 with a sequence distance of 6 and a DBR distance of 2 with a sequence distance of 4. For these parameter combinations, coverages evenly scatter around the diagonal, minimizing their bias. Beyond this point, most loci show a lower coverage after deduplication than we originally simulated. This is a sign of overmerging. The effect of overmerging loci is stronger for a DBR distance of 2, resulting in a higher variance around the diagonal. This is due to the fact that a Levenshtein distance of 2 between two DBRs greatly increases the number of colliding DBRs. Since both DBRs are of length 13, with a Levenshtein distance of 1 there are only $13 \cdot (\sigma_{\text{DNA}} - 1) = 39$ possible colliding DBRs. These are those with a Levenshtein distance of 1, since with only one edit operation, we can only perform replacements while maintaining a length of 13. If we increase the permissible Levenshtein distance to 2, we can perform

$$\binom{13}{2} \cdot (\sigma_{\text{DNA}} - 1) = 78 \cdot 3 = 234$$

replacements and additionally up to

$$13 \cdot 12 \cdot \sigma_{\text{DNA}} = 624$$

combinations of deletions and insertions. This models removing one of the 13 bases and then adding one of the four possible bases at a different position. Note, that this is an upper limit, since depending on the DBR sequence compositions some of these sequences will be identical. Using a DBR distance of 2, the number of colliding DBRs is increased from 39 to 858, thus increasing the chance of collision by an order of magnitude.

Reads from the same locus with different DBRs, which are still within a Levenshtein distance of 2, are assigned to the same first order cluster and are subsequently merged into a single read after the second order clustering. This effect increases with higher

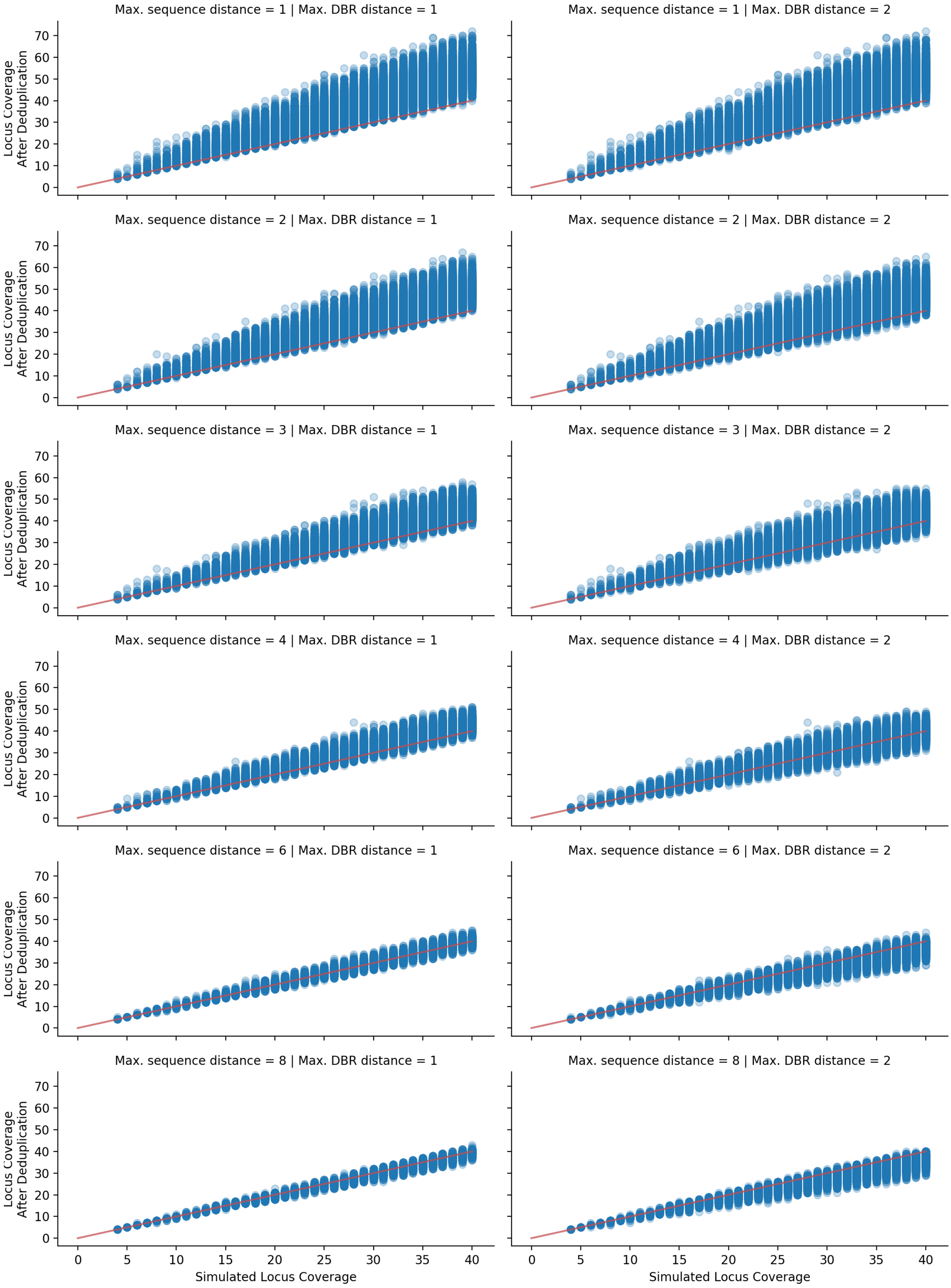


Figure 7.31: Illustration of simulated (x-axis) vs. deduplicated (y-axis) coverage for each simulated locus. As above, only one individual was simulated, so that locus and individual locus coverage are identical. Each subplot illustrates one parameter combination. Max. DBR distance increases by columns and max. sequence distance by rows.

permissible sequence distance as well. When reads from the same locus with different genotypes are assigned to the same first order cluster, we risk merging reads from different genotypes. Therefore, a conservative choice of parameters is advisable to avoid artificially reducing the coverage of certain events.

7.4.4 *Discussion*

We have implemented an approach for PCR deduplication based on a two-stepped clustering using STARCODE and using a maximum a-posteriori estimator to generate consensus sequences. As shown in the previous section, our approach is able to remove PCR duplicates from a simulated read dataset. Especially singleton reads that only appear in the clustering computed by STACKS due to PCR duplicates can be removed effectively.

We have chosen a maximal distance of 1 for DBR clustering and 2 for sequence clustering as default parameters as a conservative default. While our evaluation has shown, that more reads are merged using a higher permissible Levenshtein distance for sequence clustering, this approach mitigates overmerging. As overmerging, we denote incorrectly merging biological variants, instead of PCR duplicates. This is possible in the unlikely case, that reads with a heterozygous variant are added to the same cluster as reads from the same locus with the different allele. Using our default parameters, differences that surpass the expected 1.87 sequencing errors, like mutations introduced by the heterozygous allele, make merging these reads unlikely. On the other hand, if these reads were merged due to a high choice of permissible sequence Levenshtein distance, our quality value computation would reflect this. Consider an allele introducing a single SNP. Since the reads carrying the allele all support a different variant than is present in the consensus read, the computed base quality is low due to a high likelihood for both alleles.

A limitation to our approach is that it works best on SGS sequencing data due to their error model. Since (unaligned) TGS reads contain indel errors, we cannot assure that the bases of a cluster line up correctly using our current approach. Additionally, recently DBRs helped to discover a new type of PCR errors—PCR stutter—which are tandem repeats in low entropy regions.[65] This kind of errors is functionally identical to indel errors for the purpose of PCR deduplication. Through the use of Levenshtein distance for clustering, reads with small indels or PCR stutter errors are sorted into the same cluster. Our approach can be adapted to mitigate these errors by performing a multi sequence alignment within each second order cluster. Read positions with detected insertions or deletions can be excluded from the consensus read computation. To quickly reduce problem size and identify loci affected by this kind of error, we can pre-sort the multi alignment

[65] Sena et al., "Unique Molecular Identifiers Reveal a Novel Sequencing Artefact with Implications for RNA-Seq Based Gene Expression Analysis", 2018.

by categorizing reads that share the same beginning and ending q-grams.

Another problem is posed by the detecting of PCR substitution errors. In contrast to sequencing errors, PCR errors occur in a subset of reads, since once introduced, they are propagated through all derived copies. Additionally, they do not usually show a low Phred score, which further complicates their handling with our approach. Depending on the PCR cycle in which the error is introduced, the number reads presenting such an error can be used to distinguish them from SNPs. For one diploid individual, the expected allele frequency of a heterozygous mutation is 0.5, while the number of copies for a PCR error is expected to be in the order of 2^x, depending on the PCR cycle x in which it was introduced.[66]

As a possible optimization for our approach, the performance of the clustering could be optimized by using a locality sensitive hashing approach. In our current workflow, the first clustering step created |DBRs| cluster, which are then processed in a second clustering run. We have chosen this approach to control the memory use and performance of STARCODE. Using an alignment free approach for clustering would allow to create more candidate clusters based on similarity of both DBR and sequence. Additionally, alignment free approaches can be designed with a runtime independent of the chosen similarity. This is a limiting factor for STARCODE, which increases greatly with increasing distance.[67] An LSH approach would also not suffer from the restrictions STARCODE imposes on the maximum number of differences. In the section Outlook: Split Sketches for Chimera and Null Allele Detection (p. 228) we describe an LSH approach using split sketches for different parts of the same sequence, which could reduce the clustering step to one pass.

Another possible optimization would be to restrict the clustering steps to a Hamming distance of 1 instead of using Levenshtein distance. Based on the error model for Illumina sequencers, most sequencing errors are substitution errors, which can be modeled well by Hamming distance, which can be computed more efficiently.

[66] More precisely, the number of copies is expected to be $(1+\eta)^x$, where η is the efficiency of the employed PCR process and described by the efficiency of the four PCR phases—denaturing, annealing, polymerase binding, and target elongation—depending on the PCR cycle (Booth et al., "Efficiency of the Polymerase Chain Reaction", 2010; Louw et al., "Experimental Validation of a Fundamental Model for PCR Efficiency", 2011).

[67] Zorita, Cusco, and Filion, "Starcode: Sequence Clustering Based on All-pairs Search", 2015.

8
Conclusion and Outlook

In this thesis, we provided an overview of similarity-based approaches for sequence analysis and presented as well as analyzed new techniques. To close out our work with this chapter, we aggregate our conclusions and sketch out potential future applications of the techniques we described.

8.1 Conclusions

Locality Sensitive Hashing and MinHashing strategies in particular have been used in the field of bioinformatics since their inception. Nonetheless, in recent years this class of techniques experienced a notable influx of biological applications. As we have shown in Chapter 5, similarity and containment estimation through LSH and MinHashing techniques are applied to a variety of biological challenges in state-of-the-art analysis software. This is due in part to the existence of large and ever growing amounts of sequencing data, which lend themselves to reduced representation analysis using sketches.

In Chapter 6, we analyzed the length distribution of segments obtained with compressed winnowing, a sketching technique that condenses repetitive regions, which are prevalent in biological sequences. In contrast to the (robust) winnowing techniques as presented by Schleimer et al.[1] and Roberts et al.,[2] our approach is able to mitigate the influence of repetitive regions by representing them with one sketch entry only. This is especially useful when using winnowed segments to create reference indices, where many identical segments result in a high number of similar and uninformative alignment targets. To analyze the influence of this modification, we presented a recursive formula to compute the expected segment length distribution of uniformly independently chosen random hash values. Based on the expected distributions for random hash values, we have shown that minimal hash values for q-grams computed with compressed winnowing behave similar to randomly chosen hash values for our experiments. Comparing the expected distribution to simulated DNA sequences with different GC-content as well as a selection of reference genomes, we could show, that our

[1] Schleimer, Wilkerson, and Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", 2003.

[2] Roberts et al., "Reducing Storage Requirements for Biological Sequence Comparison", 2004.

approach is able to reduce the number of reported segments and thereby the number of alignment targets.

Additionally, we evaluated different combinations of hash functions and canonization strategies to judge their influence on the generated segment length distribution. Our analysis has shown that using most hash functions, the distribution of segment lengths behaves as predicted by our expected distribution. While trivial hash functions like unmodified 2-bit integer-encoded q-grams or swap-mixed q-grams performed poorly, even a simple invertible multiplicative hash function yielded results behaving similar to randomly chosen hash values. We obtained the best results using twisted tabulation hashing and `mmh3`. Considering canonization, using max-canonical q-grams instead of min-canonical q-grams did not offer an obvious improvement of segment length distribution. While using max-canonical q-grams with trivial hash functions reduced the number of segments with length 1 and adhered better to the expected distribution than with min-canonical q-grams, this coincided with a reduced number of segments with lengths less than q. Across all experiments, segment lengths obtained with non-canonical q-grams did show a behavior closer to the prediction than with min- and max-canonical q-grams. This is due to the fact that canonizing q-grams reduces the space of possible hash values. Hence, especially in scenarios where short q-grams are required, it can be beneficial to forego canonization when using MinHashing techniques when this is feasible for the analysis.

As an application that could benefit greatly from this technique, we outlined an index built upon multiple compressed winnowing of a reference database. Using multiple winnowings in combination with segment length information allows to narrow possible alignment locations, resulting in a smaller input size for the subsequent local or semiglobal alignment computation.

Further, we presented in Chapter 4 a cache efficient hash table—the bit-packed hopscotch hash table (BPHT)—which can be used to realize such an index. By combining bit-packing and quotienting, we were able to reduce the number of compulsory cache misses caused by accessing a separate hop bit array. We have shown that our implementation can speed up the runtime for lookups with respect to a reference implementation using two separate arrays. While we made restrictive assumptions for the implementation presented in this work, for example restricting hash table sizes to powers of 2 and hash values to $[2^{32}]$, these limitations are not inherent to the described architecture. Through the use of word-packing we can break up the reliance on entries to fit into 64-bit integer values which allows the use of arbitrary size address, remainder, and value spaces. While this technique introduces a small overhead of computations to access array entries, we can assume this will not significantly impact the runtime of our data structure. Since our evaluation has shown that even the use of computationally expensive hash functions did not influence the runtime in a notable way, we can assume the computation to be dominated by input and output operations.

Finally, in Chapter 7 we described ddRADseq analysis as an application of similarity based analysis techniques. For this, we presented our simulation software DDRAGE, an analysis workflow for ddRADseq data, and the PCR deduplication approach used for this workflow. Analyzing ddRADseq data relies on similarity-based approaches, since most of the time no reference genome is available for the analyzed species. First, we described DDRAGE, our simulation software that can generate ddRADseq datasets that model the biological and technological effects prevalent in ddRADseq reads.

Using data simulated with DDRAGE as ground truth, we implemented and evaluated a ddRADseq analysis workflow employing the STACKS software for the main analysis. We have shown that the results yielded by STACKS vary by a large amount based on the provided parameters. A core feature of our pipeline is the ability to explore multiple sets of these parameters in parallel, while reusing input data from a single run of its preprocessing phase. Through the analysis of simulated reads, we were able to identify patterns that emerge for certain read configuration when comparing runs with different parameters. For example a decreasing number of detected loci with increasing minimal required coverage m points towards a low coverage data set, while a drastic reduction in the number of detected loci between raw and deduplicated analysis runs can indicate a dataset with high genetic diversity. This allows us to pick parameters that best suit the dataset at hand.

Merging loci that were identified as distinct during the USTACKS and CSTACKS phases of the STACKS analysis has proved to be the most influencing task. Through the modular architecture of our workflow, we can replace single parts of the pipeline with different software, as long as it generates compatible output files. This would allow us to replace the CSTACKS clustering with a custom clustering approach leveraging a MinHashing technique to avoid the merging step by joining loci in the first place.

Finally, we implemented a PCR deduplication software, which we have shown to improve the results of the STACKS analysis across all runs. We have shown that through the early removal of PCR duplicates, we were able to greatly reduce the number of candidate loci that need to be processed in the analysis pipeline. To be as effective for longer reads than for the SGS short reads we analyzed in our evaluation, we need to replace the clustering approach, which can only deal with a limited Levenshtein distance. An approach that could incorporate the detection and removal of PCR chimeras into PCR deduplication is outlined as future research in the following section. Additionally, this approach is also able to solve an open challenge in ddRADseq analysis: the detection of alternative sequence null alleles.

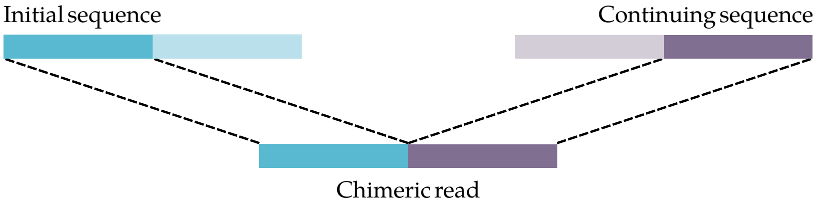


Figure 8.1: Structure of conjoined reads caused by PCR chimeras. The combined read shown below comprises a prefix of the teal (left) and a suffix of the purple (right) read. For conjoined reads caused by ddRADseq NAs (or incomplete digestion), we would observe only one of the matches (depending on the location of the NA—p5 or p7).

8.2 Outlook: Split Sketches for Chimera and Null Allele Detection

Over the course of this work, we introduced two biological effects that are characterized by the recombination of sequence suffixes and prefixes. Both chimeric reads (cf. Section 2.4.3) and alternative sequence Null Alleles (NA, cf. Section 7.1.3) exhibit the following pattern:

- Reads share a common prefix, but a different suffix (or vice versa).
- The split occurs exactly (for ddRADseq NAs in paired-end reads), or approximately (for certain kinds of chimeric reads) in the middle of read sequences.
- We want to identify read clusters for which at least one of the two halves are identical.

An illustration of such reads is shown in Figure 8.1. We collectively refer to this kind of reads as conjoined reads and to a set of reads caused by conjoined read events as conjoined read clusters.

For ddRADseq reads with an alternative sequence NA, either the p5 or the p7 sequence of a paired-end read are identical to other reads from the same locus (within the same or across multiple individuals). To retain the information for at least the unaffected part of the read pair, we require to identify a locus matching only one of the read parts. An example of reads showing this pattern is illustrated in Figure 8.2.

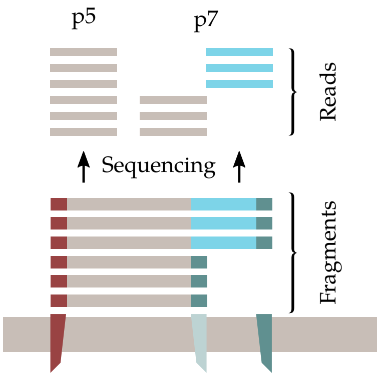


Figure 8.2: Structure of conjoined reads caused by an alternative sequence p7 null allele. Reads generated from the affected fragments show a different p7 sequence (shown in teal) than their unaffected counterparts. Both groups of reads share the same p5 read sequence.

Chimeric reads occur during PCR amplification when a prematurely terminated PCR copy reanneals with another sequence and continues with that sequence.[3] For a specific class of reads used for taxonomic classification, this break point is located in the middle of a genomic fragment, resulting in chimeric reads. Reads obtained from amplicon sequencing of the latter part of the 18S rRNA gene and the first part of the internal transcribed spacer (ITS) comprise two variable regions flanking a strongly conserved region (see Figure 8.3).

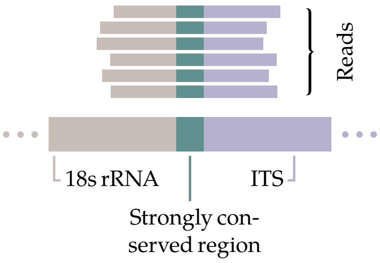


Figure 8.3: Structure of reads obtained from 18S amplicon sequencing. Reads span the suffix of the 18S rRNA, a strongly conserved region, and an internal transcribed spacer (ITS). During PCR amplification, the elongation of the sequence can abort at the conserved region and finish in a later cycle with a different sequence.

The two variable sequence parts—the 18S and the ITS sequences—allow taxonomic classification, while the strongly conserved region enables primers to bind. Due to this, there is a high chance that an incomplete copy is continued during the next PCR cycle combining

[3] Haas et al., "Chimeric 16S rRNA Sequence Formation and Detection in Sanger and 454-Pyrosequenced PCR Amplicons", 2011.

two 18S and ITS sequences that are not contained in the input data, thus creating a chimeric read.

For both cases, we are interested in identifying the deviant reads and relate them back to their *parent* reads, i.e. the unaffected ddRADseq locus or the two similar other DNA fragments for PCR chimeras. We propose identifying two classes of similarity matches to identify this relation: weak matches, which have a high similarity to only one half of another read, and strong matches, which are similar to both halves of the other sequence. A high similarity, in this case, refers to a similarity threshold taking the specific sequencing technology, sequence length, etc. into account.

Solving this with plain k-mins or bottom-k sketching is not reliable, since a resemblance $r(x_i, x_j) = 0.5$ does not offer any information about which parts of the reads x_i and x_j are similar. Using winnowing techniques mitigates this problem, but we still cannot completely rule out that minimizers of the first and the second part cross over.

Hence, we propose explicitly encoding the locality information using a split sketch

$$S^{\bullet\lrcorner\llcorner\bullet}(x) = \left(S^{\bullet\lrcorner\cdots}(x), S^{\cdots\llcorner\bullet}(x)\right) = \Big(S(x_{[0:b]}), S(x_{[b:\ell]})\Big)$$

where x is a read sequence, $\ell = |x|$, and $b = \lfloor \ell/2 \rfloor$ denotes the break point within the sequence. Sketches for the prefix and suffix of x are denoted by $S^{\bullet\lrcorner\cdots}(x)$ and $S^{\cdots\llcorner\bullet}(x)$ respectively.[4] For the remainder of this section we will assume them to be bottom-k sketches. Note that when referring to paired-end ddRAD reads in this context, we assume x to be the concatenated p5 and p7 sequence which both have a length of b.

Using such split sketches allows confidently identifying matches from which the structure of conjoined read clusters can be derived. If, for example, only one half of the sketch of a sequence x_i shows a high similarity to a sequence x_j, e.g. $r(S^{\bullet\lrcorner\cdots}(x_i), S^{\bullet\lrcorner\cdots}(x_j)) \approx 1$ but $r(S^{\cdots\llcorner\bullet}(x_i), S^{\cdots\llcorner\bullet}(x_j)) \approx 0$, we can be sure that x_i and x_j have a matching prefix. Based on this, we can define a set of rules (shown in Table 8.1) to derive weak and strong links between reads from their split sketch similarity. From now on, we focus on chimeric reads since these express more complex behavior than NA reads. We will return to NA reads later in this section.

A chimeric read cluster is characterized by weak and strong links, where a weak link denotes a match of only $S^{\bullet\lrcorner\cdots}$ or $S^{\cdots\llcorner\bullet}$, and a strong link denotes a match of both. Computing weak and strong links allows the identification of read cluster groups that follow the pattern illustrated in Figure 8.4. Identical reads fall into strongly linked connected cliques, which are interconnected by weak links.

Notice, that in practice we are are not interested in identifying cliques (a problem that is NP-complete for general graphs). We are actually interested in connected components in the graph $G = (\mathcal{R}, E^{\mathrm{s}} \cup E^{\mathrm{w}})$, where E^{s} and E^{w} denote edge sets that model strong and weak links respectively. From this information, we derive a

[4] The two parts of the sketch denoted by S without any superscript can be realized by k-mins or bottom-k sketches.

Table 8.1: Different links for chimeric reads that can be derived from similarity patterns in split sketches of two reads. The entries in the similarity columns, e.g. $S^{\bullet\lrcorner\cdots}\uparrow$, are short hand notation for a high similarity of $r(S^{\bullet\lrcorner\cdots}(x_i), S^{\bullet\lrcorner\cdots}(x_j)) > t$ for two sequences x_i, x_j and a given similarity threshold t.

Similarity		Link type		Interpretation
$S^{\bullet\lrcorner\cdots}\uparrow$	$S^{\cdot\cdot\llcorner\bullet}\uparrow$	▬	strong	chimeric read vs. chimeric read **or**
		▬	strong	unaffected read vs. unaffected read
$S^{\bullet\lrcorner\cdots}\uparrow$	$S^{\cdot\cdot\llcorner\bullet}\downarrow$	–	weak	Chimeric read vs. unaffected read
$S^{\bullet\lrcorner\cdots}\downarrow$	$S^{\cdot\cdot\llcorner\bullet}\uparrow$	–	weak	Chimeric read vs. unaffected read
$S^{\bullet\lrcorner\cdots}\downarrow$	$S^{\cdot\cdot\llcorner\bullet}\downarrow$		none	no similarity

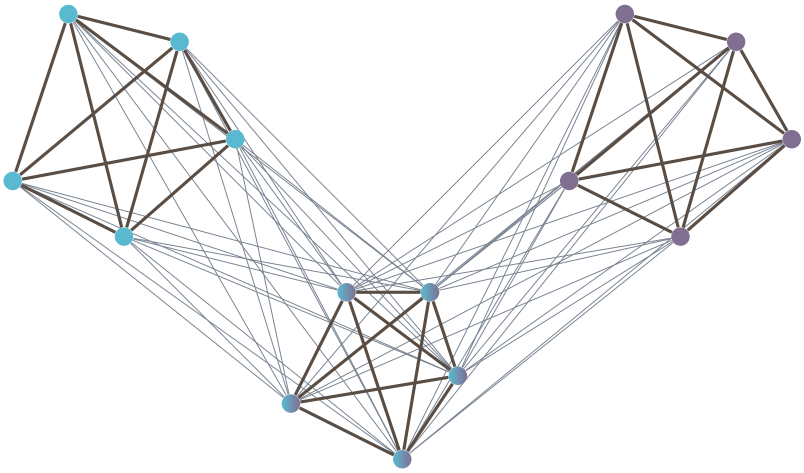


Figure 8.4: Clustering of chimeric reads. Reads are shown as circles with color denoting their sequence of origin. Thick lines ▬ symbolize strong links contained in the edge set E^s and thinner lines – symbolize weak links contained in E^w.

tiered clustering of reads as illustrated in Figure 8.5, i.e. a clique graph in which cliques act as nodes and weak connections between cliques as edges.

The presence of different kinds of original and chimeric reads manifests in different structures in the clique graph. We start from the fact that a clique with a weak connection to only one other clique has to comprise original reads, since a chimeric read requires two (similar, hence connected) parent reads: one read containing their prefix and another one containing their suffix sequence. In the simplest case, as illustrated in Figure 8.5, we can identify chimeras as cliques that possess weak links to two other cliques, which themselves are only weakly connected to this one clique. More complicated patterns arise for original reads that form chimeras with multiple other reads or even with other chimeras. An example for such a structure with three different reads is shown in Figure 8.6. For such instances, we need to identify which reads (i.e. cliques) are original and which are chimeric. This problem is similar to finding a vertex cover on the clique graph, where original read nodes are selected into the vertex cover. While the vertex cover problem without restrictions is NP-complete, we can impose the following restrictions:

- Nodes with a degree of one have to be selected into the vertex cover.

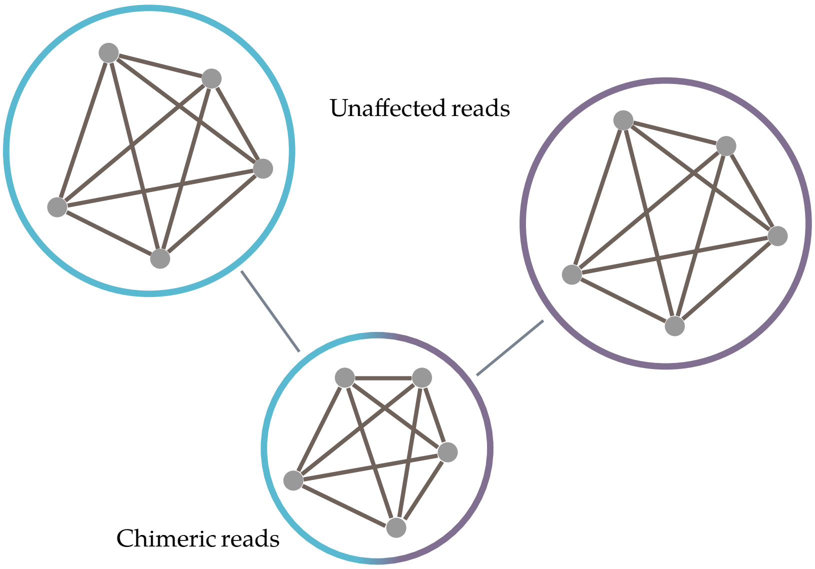


Figure 8.5: Illustration of a conjoined read cluster. Mirroring the structure of Figure 8.1, the lower cluster contains strongly connected chimeric reads, while the upper clusters contain strongly connected non-chimeric reads that served as parents for the chimeric read. The circles and interconnecting weak links (shows as thin lines) describe the induced clique graph.

- Nodes selected into the vertex cover must have a distance of at least two.
- Every chimeric read node is connected to at most two original read nodes.

However, the influence of these restrictions, as well as the size and benevolence of the instances resulting from actual datasets, remain open. Depending on the presence of all chimeric combinations of such reads, resolving which reads are original and which are chimeric might not be possible based solely on graph structure. However, secondary information like the size of cliques—original reads can be expected to possess more duplicates than PCR chimeras—could allow a heuristic approach.

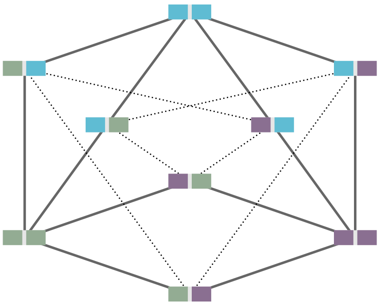


Figure 8.6: Illustration of a complete clique graph for three reads and all their chimeric combinations. Each pair of colored blocks denotes a clique, with the color of blocks denoting a specific sequence. Pairs with matching colors denote original reads, those with mismatched colors denote chimeric reads. Solid edges between nodes all denote weak similarity, with edges between original and chimeric reads drawn as thick lines. The edges shown as dashed lines denote weak similarity between chimeric reads.

More formally, we propose the following approach to detect chimeric reads: Consider an input file containing a set of reads $\mathcal{R}$. To avoid all-vs-all comparison, we propose computing link information using the following process:

1. Iterate through $\mathcal{R}$ and compute the split sketch $S^{\bullet|\bullet}(x_i)$ for all reads $x_i \in \mathcal{R}$. Store the sketches in two separate hash tables $T^{\bullet|\cdots}$ and $T^{\cdots|\bullet}$ for the head and the tail sketch respectively.
2. Create two linking arrays L^{s} and L^{w} with $|L^{\mathrm{s}}| = |L^{\mathrm{w}}| = |\mathcal{R}|$. These will be used to map read number (used as array index for L) to a cluster representative (the entry stored in that slot of L) using a leader clustering approach. Sketch $x_i \in \mathcal{R}$ again (or reuse the previously computed sketches) to retrieve all similar reads for x_i (using a similarity threshold of t). Query both $T^{\bullet|\cdots}$ and $T^{\cdots|\bullet}$ with each part of $S^{\bullet|\bullet}(x_i)$ and collect reads that are strongly and weakly linked to x_i into a candidate list $\mathcal{L}$ using the rule set shown in Table 8.1. Check if an entry in $\mathcal{L}$ has already been assigned to a cluster by strong or weak links. If yes, then assign

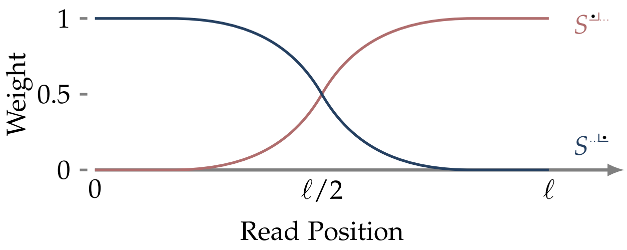


Figure 8.7: Exemplary weight functions for head and tail sketch.

all identified candidates to this cluster, if not create a new cluster with x_i as representative and assign the remaining reads to this cluster. If conflicts arise, i.e. if entries of $\mathcal{L}$ have been assigned to different clusters, either merge these clusters, mark them as conflicting and revisit them later to resolve these conflicts, or reject the assignments all together. The best strategy to resolve these conflicts remains as future research.

3. Using L^{s} and L^{w}, identify connected components within the read graph $G = (\mathcal{R}, E^{\mathrm{s}} \cup E^{\mathrm{w}})$. Within each connected component, identify strongly connected subsets (i.e. connected components using only strong edges) and subsequently identify the structure of weak links between these subsets to construct a clique graph. Cf. Figure 8.5 for an example.

4. For each connected component of the clique graph, distinguish original and chimeric reads, as described above.

To adapt this workflow to the detection of alternative sequence NAs in ddRAD data, we only need to adapt the last step of the analysis. For this case, we only expect pairs of strongly connected read subsets. While a locus with both a p5 and a p7 alternative sequence NA is possible, it is highly unlikely and the case where only the p7 read is affected can be expected to be the most prevalent). Consequently, the resulting clique graphs can be expected to only contain small connected components, resulting in problem instances that can be solved quickly.

Notice that this approach can also be modified to additionally perform PCR deduplication. To achieve this, we need to exclude the degenerate base region (DBR) from the reads. This can either be done explicitly by cropping this sequence and using it in a subsequent step to discern PCR duplicates within one cluster. Alternatively the DBR can be incorporated into a third part of a split sketch that is only evaluated within a cluster. Since clusters are already selected to share similar sequences, they can be expected to be small problem instances for which an explicit comparison is valid. Note that this is different to the approach that we followed in Section 7.4, where we first grouped input reads by their DBR to achieve smaller subproblems. However, based on current read and DBR lengths, the benefit from using a MinHash technique for DBR comparison is limited.

For cases where the break point is not as clear as with the examples given above, we can employ a weighted MinHash approach for the sketch generation (cf. Section 5.10 p. 116f). Using the distance to the read start and read end as weights as illustrated in Figure 8.7.

While we already implemented a working prototype using this approach for chimera detection, its further development and evaluation as well as its application for NA detection and PCR deduplication remain for future research.

A
Hash Function Code Samples

```
//! Integer Encoding

const ENCODING_2BIT: [u32; 256] = [
    // 0b00, 0b01, 0b10, and 0b11 for
    // 65 (A), 67 (C), 71 (G) and 84 (T)
    // respectively. Zero for all other values.
];

pub struct IntegerEncoding<'a> {
    seq: &'a [u8],
    q: usize,
    pos: usize,
    value: u32,
    mask: u32,
    bits_per_char: usize,
}

impl<'a> IntegerEncoding<'a> {
    // Initialization [...]

    fn next_encoding(&mut self) -> u32 {
        self.value <<= bits_per_char;
        let enc = ENCODING_2BIT[
            self.seq[self.pos + self.q - 1] as usize
        ];
        self.value |= enc;
        self.value &= self.mask;
        self.value
    }
}
```

Listing 2: Rust implementation of 2-bit integer encoding for DNA sequences. The `ENCODING` variable holds a stack-allocated integer array mapping byte values to integer values specific for the alphabet. This code omits the initialization of the first full q-gram in the constructor.

```rust
//! Hlin Hashing

use rand;
use rand::Rng;

#[derive(Clone)]
pub struct HlinParams {
    a64: u64,
    b64: u64,
    a128: u128,
    b128: u128,
}

impl HlinParams {
    pub fn new () -> Self {
        HlinParams{
            a64: rand::random::<u64>(),
            b64: rand::random::<u64>(),
            a128: rand::random::<u128>(),
            b128: rand::random::<u128>(),
        }
    }

    pub fn with_params(a64: u64, b64: u64, a128: u128, b128: u128) -> Self {
        HlinParams{
            a64: a64,
            b64: b64,
            a128: a128,
            b128: b128,
        }
    }
}


/// H^{lin} hash function for 32-bit keys and hash values
pub fn df_32(x: u32, params: &HlinParams) -> u32 {
    ((params.a64.wrapping_mul(x as u64).wrapping_add(params.b64)) >> 32) as u32
}

/// H^{lin} hash function for 64-bit keys and hash values
pub fn df_64(x: u64, params: &HlinParams) -> u64 {
    ((params.a128.wrapping_mul(x as u128).wrapping_add(params.b128)) >> 64) as u64
}
```

Listing 3: Rust implementation of the $\mathcal{H}^{\text{lin}}_{2^{32},2^{32},2^{64}}$ and $\mathcal{H}^{\text{lin}}_{2^{64},2^{64},2^{128}}$ subfamilies of hash functions. An instance of the `HlinParams` struct represents one hash function $h \in \mathcal{H}^{\text{lin}}$.

```
//! Twisted Tabulation Hashing

/// Split 32-bit value into four 8-bit integers
/// This could also be realized using std::mem::transmute
fn byte_chunks (x: u32) -> [u8; 4] {
    [(x & 0x000000FF) as u8, ((x & 0x0000FF00) >> 8) as u8,
     ((x & 0x00FF0000) >> 16) as u8, ((x & 0xFF000000) >> 24) as u8
    ]
}

/// 32-bit simple tabulation hashing
pub fn tab32_simple(x: u32, T: &[[u32; 256]; 4]) -> u32 {
    let mut h: u32 = 0;  // initialize hash value as 0

    // iterate over all chunks
    for (i, c) in byte_chunks(x).iter().enumerate() {
        h ^= T[i as usize][*c as usize];
    }

    return h
}

/// 32-bit twisted tabulation hashing
pub fn tab32_twisted(x: u32, T: &[[u64; 256]; 4]) -> u32 {
    let mut h: u64 = 0;  // initialize hash value as 0

    // iterate over first three chunks
    let chunks = byte_chunks(x);
    for (i, c) in chunks[0..3].iter().enumerate() {
        h ^= T[i as usize][*c as usize];
    }

    // factor in last chunk
    let c = chunks[3] ^ (h & 0xFF) as u8;
    h ^= T[3][c as usize];
    h = h.overflowing_shr(32).0;

    return (h as u32)
}
```

Listing 4: Rust implementation of simple and twisted tabulation hashing. Generating and filling the table T is not shown.

```
//! Swap Mixing

/// Bit masks to extract high and low words of a 64-bit integer
const LOW: u64 =  0b_00000000_00000000_00000000_00000000_11111111_11111111_11111111_11111111;
const HIGH: u64 = 0b_11111111_11111111_11111111_11111111_00000000_00000000_00000000_00000000;

/// Swap low and high words of a u64 integer value.
pub fn swap_words_q (hash: u64, q: u64) -> u64 {
    let low;
    let high;
    if q == 32 {
        low = LOW;
        high = HIGH;
    } else {
        low = 2_u64.pow(q as u32) - 1;
        high = (2_u64.pow(2 * q as u32) - 1) ^ 2_u64.pow(q as u32) - 1;
    }
    let low = hash & low;
    let high = hash & high;
    return (low << q) | (high >> q)
}
```

Listing 5: Swap mixing, a simple hash function used for the evaluation of segment length distributions in Chapter 6.

B

Additional Figures for Segment Length Distribution

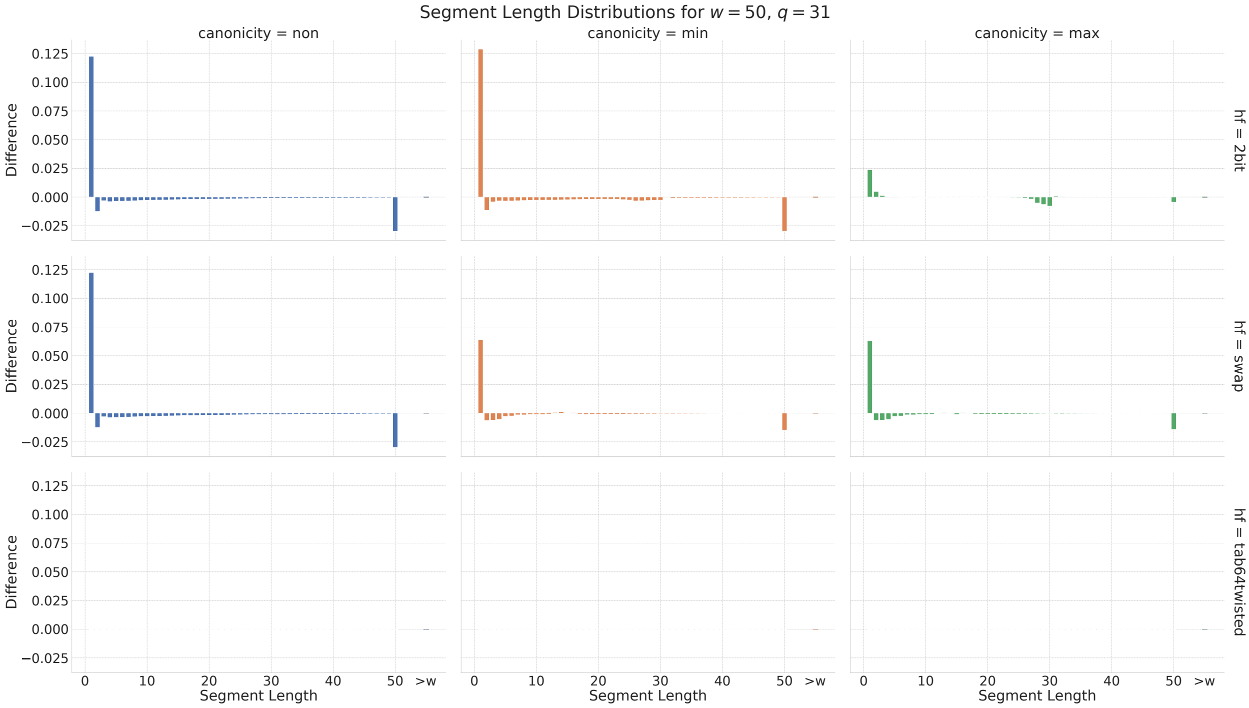


Figure B.1: Differences between an empirically computed segment length distribution and the predicted segment length distribution $\Psi^{2\,000\,000}_{50,150}$. This plot shows the differences for the first three rows of Figure 6.14. Empiric values were computed on 31-grams with a window of size $w = 50$, generated from 10 simulated genomes with a GC-content of 0.5 and a length of 100 000 000. All plots in one column share the same canonicity (non, min, max), while all plots in one row share the same hash function. Bars denote the difference between the empiric and the predicted distribution, i.e. bars above the x-axis denote that more segments than expected were found for the specific segment length. Both for the empirical and the predicted distribution, all values $> w$ are collected into a single bar on the right side of the plot. In this case, no such values are present.

For both 2-bit encoding and swap mixing, the number of segments with length 1 is notably higher than predicted. Additionally, for 2-bit encoding with max-canonical q-grams, while the number of segments of length 1 is less pronounced, this effect is also present for segments of length 2 and 3. For the remaining segment lengths computed using these hash functions, most segment lengths occur less frequently than predicted. Note, that for the twisted tabulation hash function, deviations from the predicted distribution were too small to render on this scale, leaving this plot virtually empty.

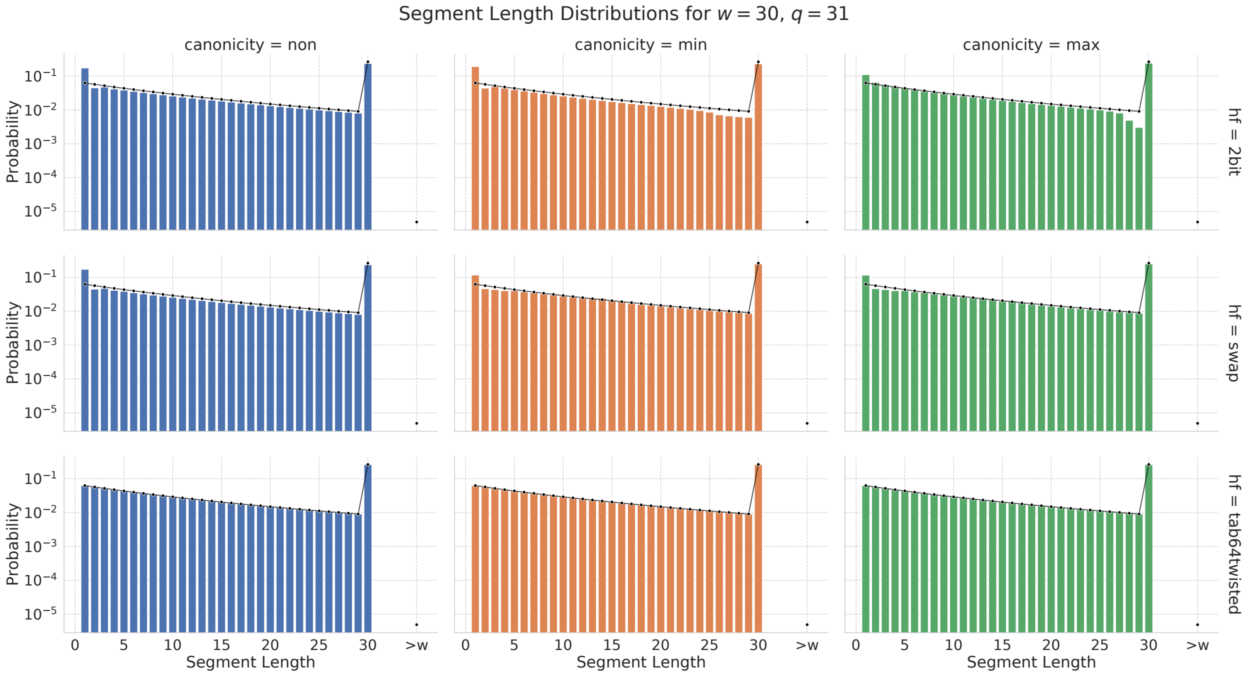


Figure B.2: Length distribution of segments (colored bars) on 31-grams with a window of size $w = 30$, generated from 10 simulated genomes with a GC-content of 0.5 and a length of 100 000 000. The y-axis is logarithmically scaled. All plots in one column share the same canonicity (color), while all plots in one row share the same hash function. The expected distribution $\Psi^{2\,000\,000}_{30,90}$ is shown as black points. Both for the empirical and the expected distribution, all values $> w$ are collected into a single, darker bar (or point for the expected value) on the right side of the plot. In this plot, no such segments are present.

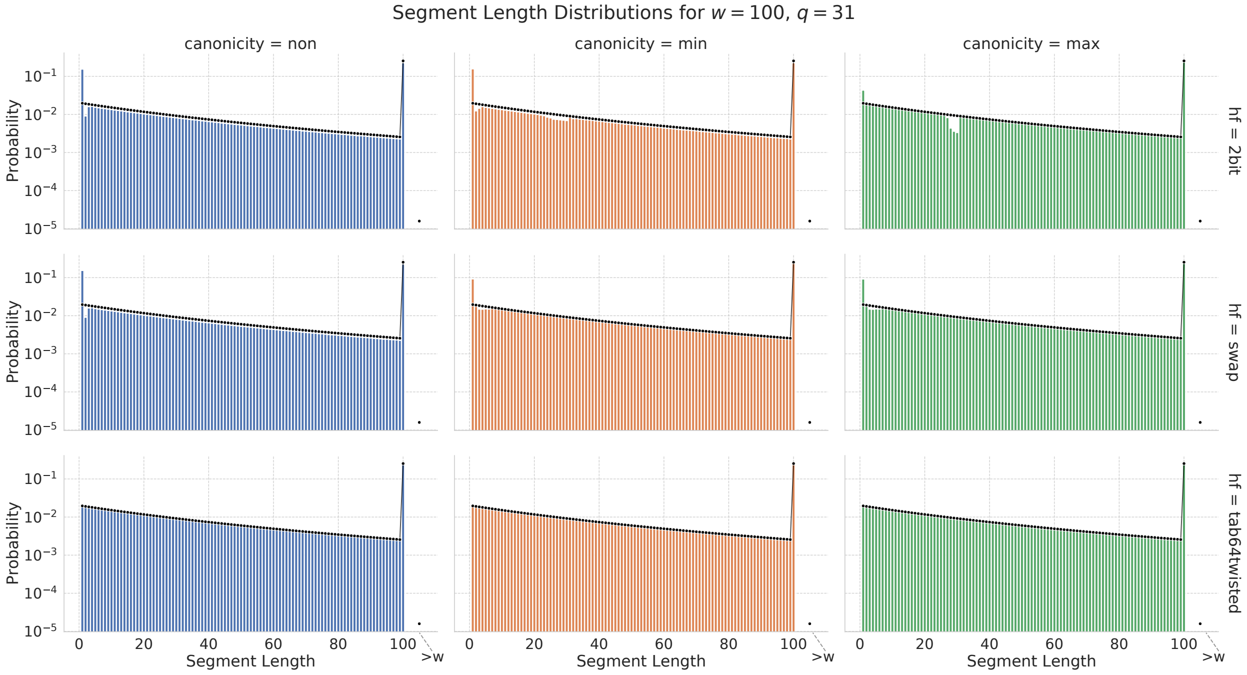


Figure B.3: Length distribution of segments (colored bars) on 31-grams with a window of size $w = 100$, generated from 10 simulated genomes with a GC-content of 0.5 and a length of 100 000 000. The y-axis is logarithmically scaled. All plots in one column share the same canonicity (color), while all plots in one row share the same hash function. The expected distribution $\Psi^{2\,000\,000}_{100,300}$ is shown as black points. Both for the empirical and the expected distribution, all values $> w$ are collected into a single, darker bar (or point for the expected value) on the right side of the plot. In this plot, no such segments are present.

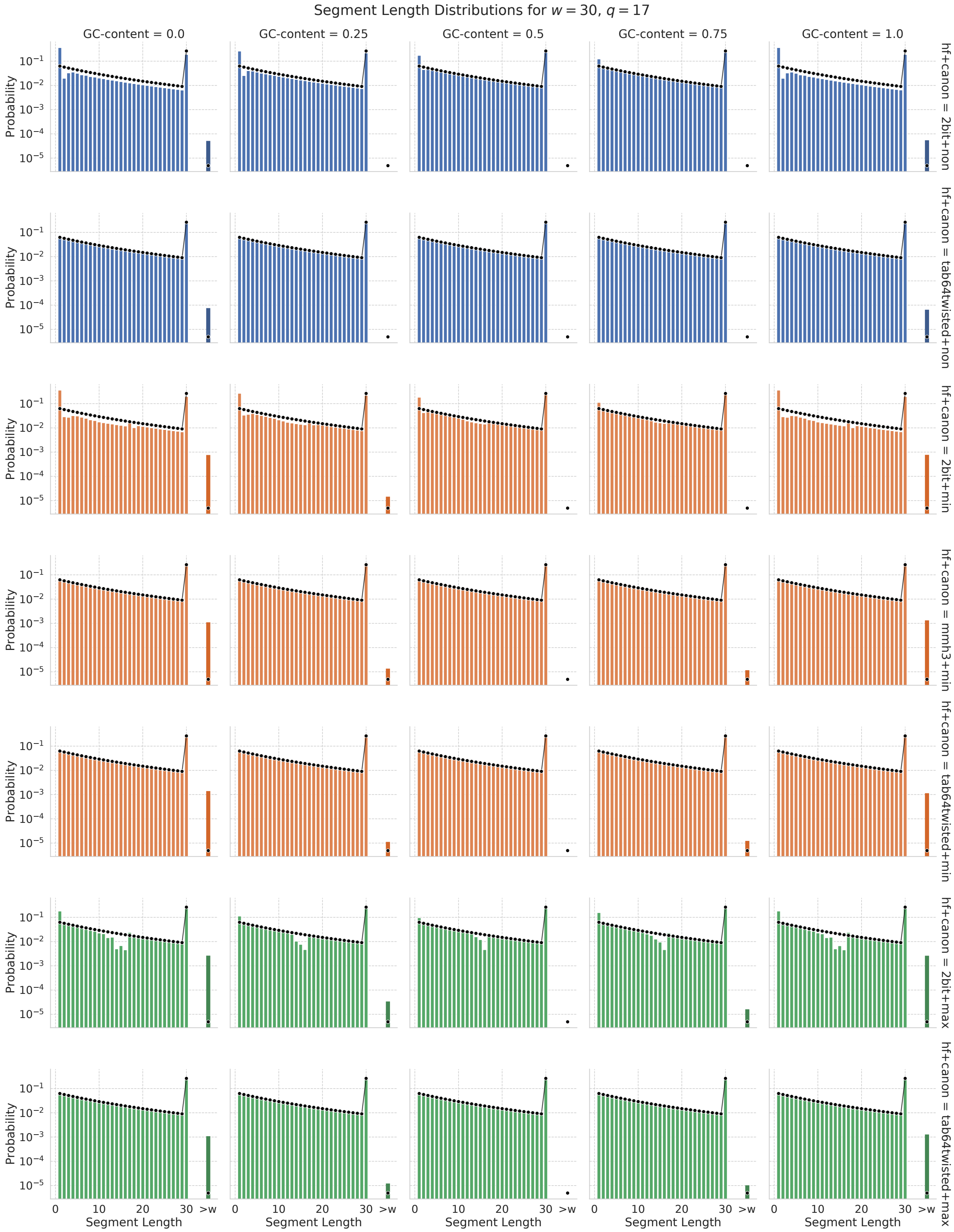


Figure B.4: Distribution of segment lengths for random sequences of length 100 000 000 for multiple GC-contents (columns) with $w = 30$ and $q = 17$. Each row contains plots for a combination of hash function and canonization strategy, with rows using the same canonization strategy grouped by color. As before, all segments above length w are aggregated into one bar.

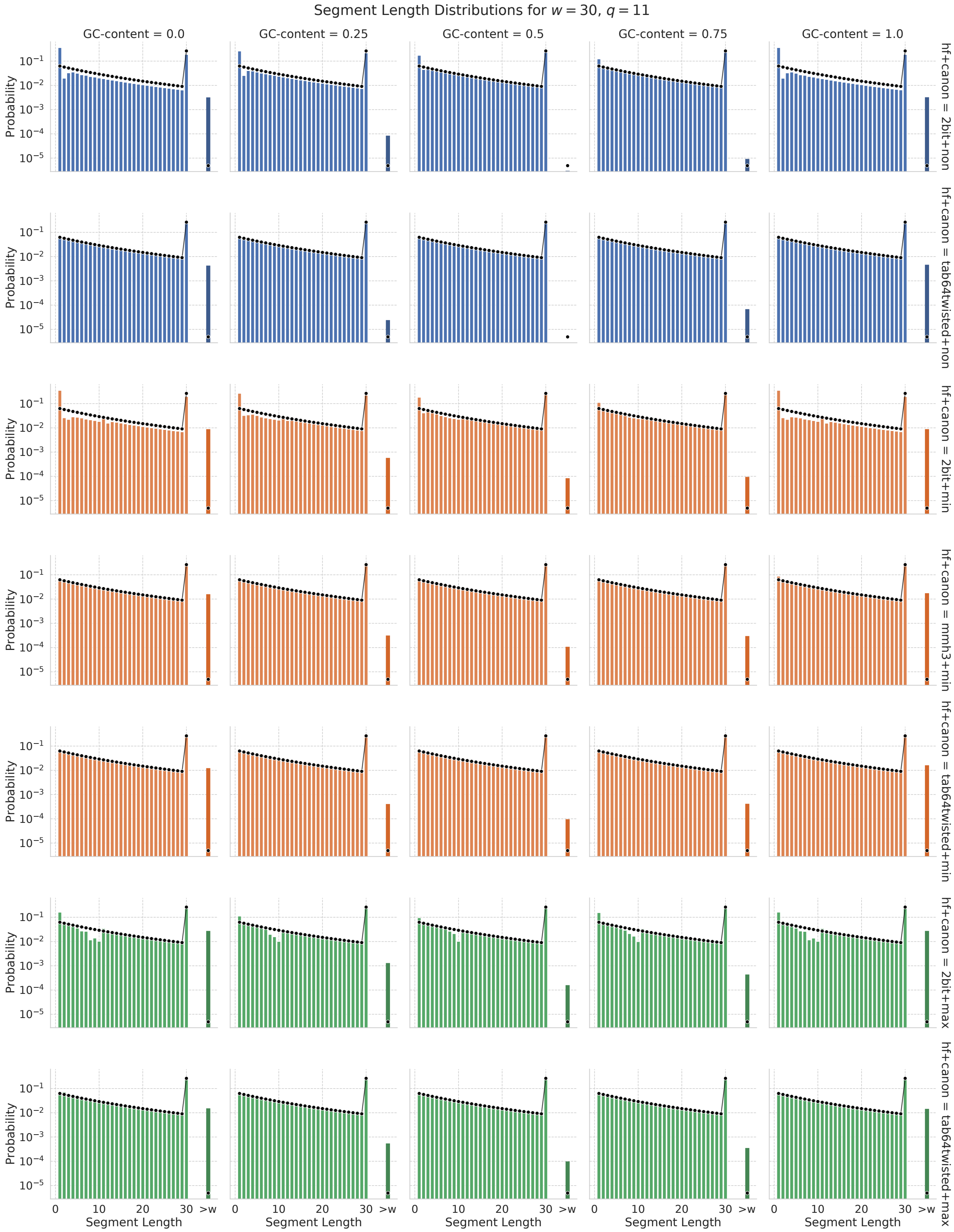


Figure B.5: Distribution of segment lengths for random sequences of length 100 000 000 for multiple GC-contents (columns) with $w = 30$ and $q = 11$. Each row contains plots for a combination of hash function and canonization strategy, with rows using the same canonization strategy grouped by color. As before, all segments above length w are aggregated into one bar.

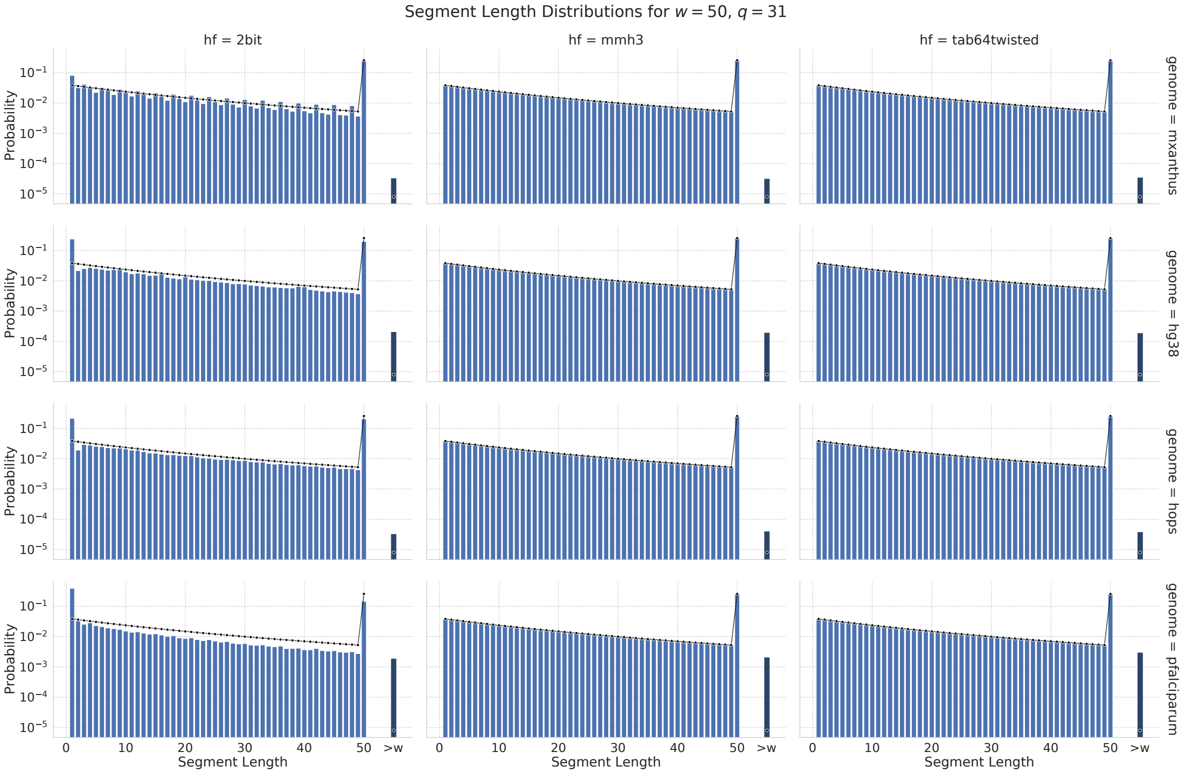


Figure B.6: Segment length distributions for reference genomes (rows) using different hash functions (columns) and non-canonical 31-grams. The empirical segment length distribution is shown as bars with the expected distribution shown by black points. All segment lengths larger than w are aggregated into a single bar with darker shade on the right of each facet.

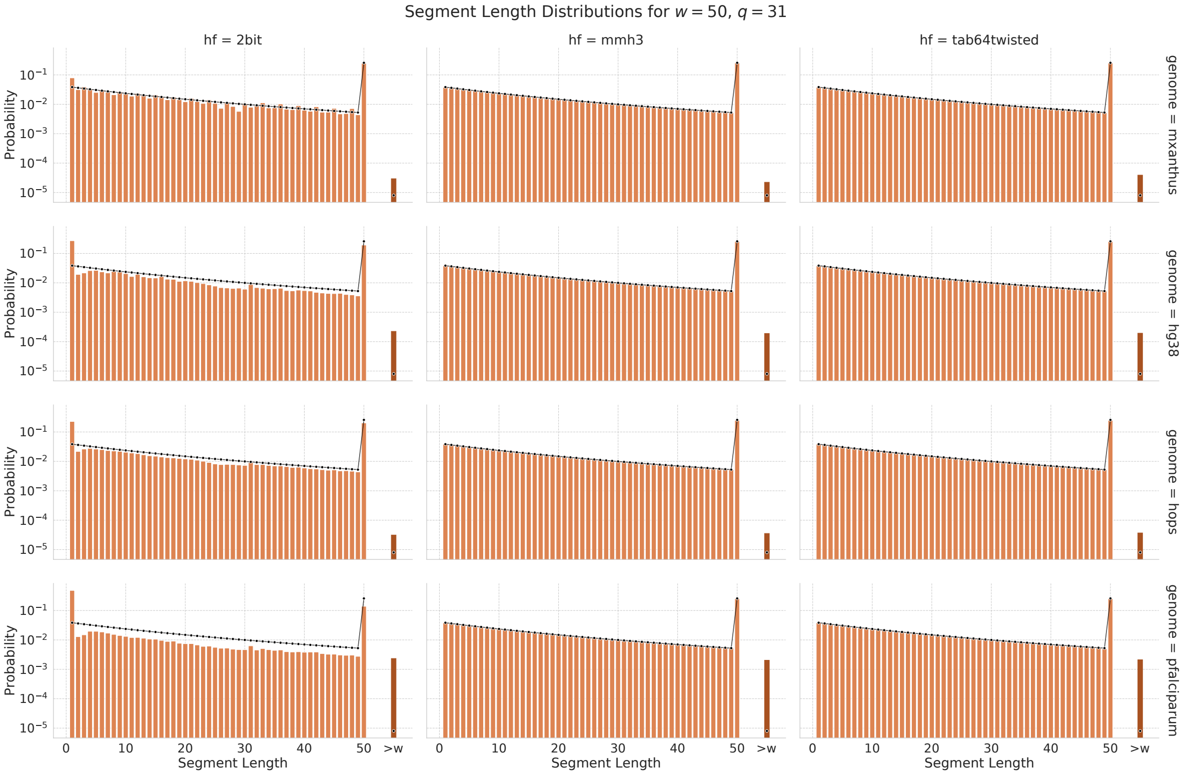


Figure B.7: Segment length distributions for reference genomes (rows) using different hash functions (columns) and min-canonical 31-grams. The empirical segment length distribution is shown as bars with the expected distribution shown by black points. All segment lengths larger than w are aggregated into a single bar with darker shade on the right of each facet.

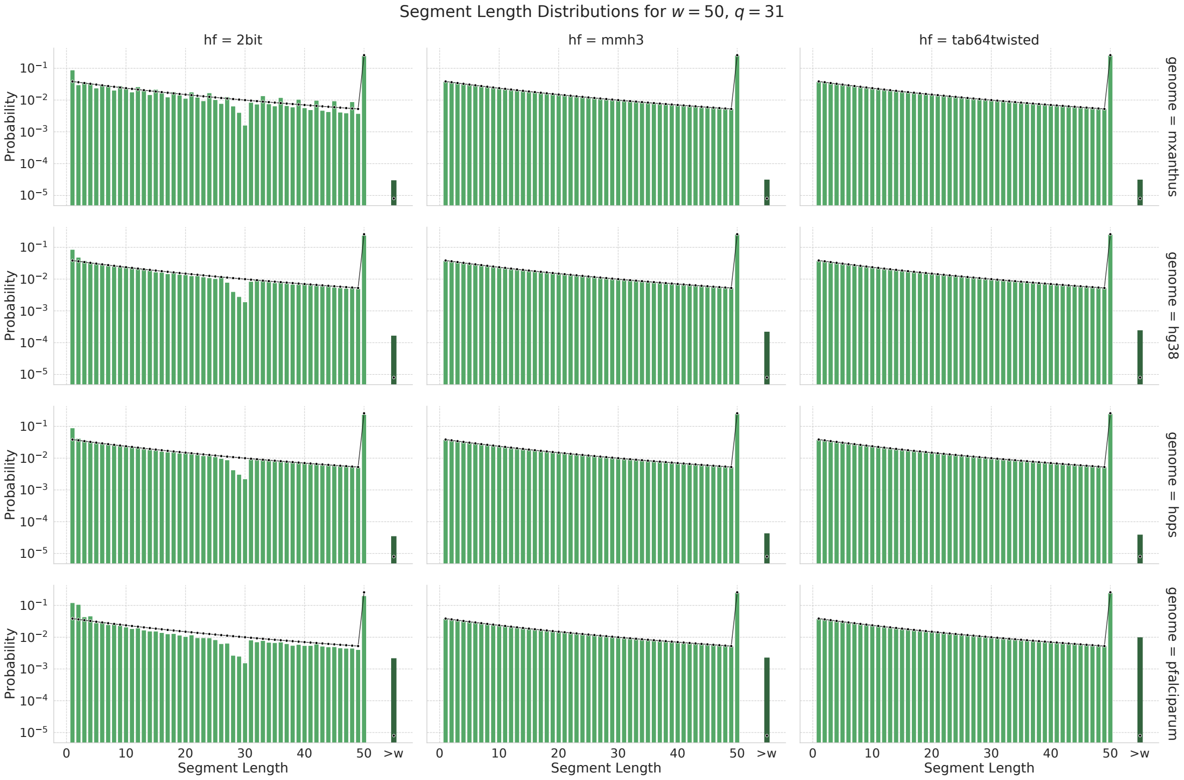


Figure B.8: Segment length distributions for reference genomes (rows) using different hash functions (columns) and max-canonical 31-grams. The empirical segment length distribution is shown as bars with the expected distribution shown by black points. All segment lengths larger than w are aggregated into a single bar with darker shade on the right of each facet.

C
File Graph for our ddRAD Analysis Workflow

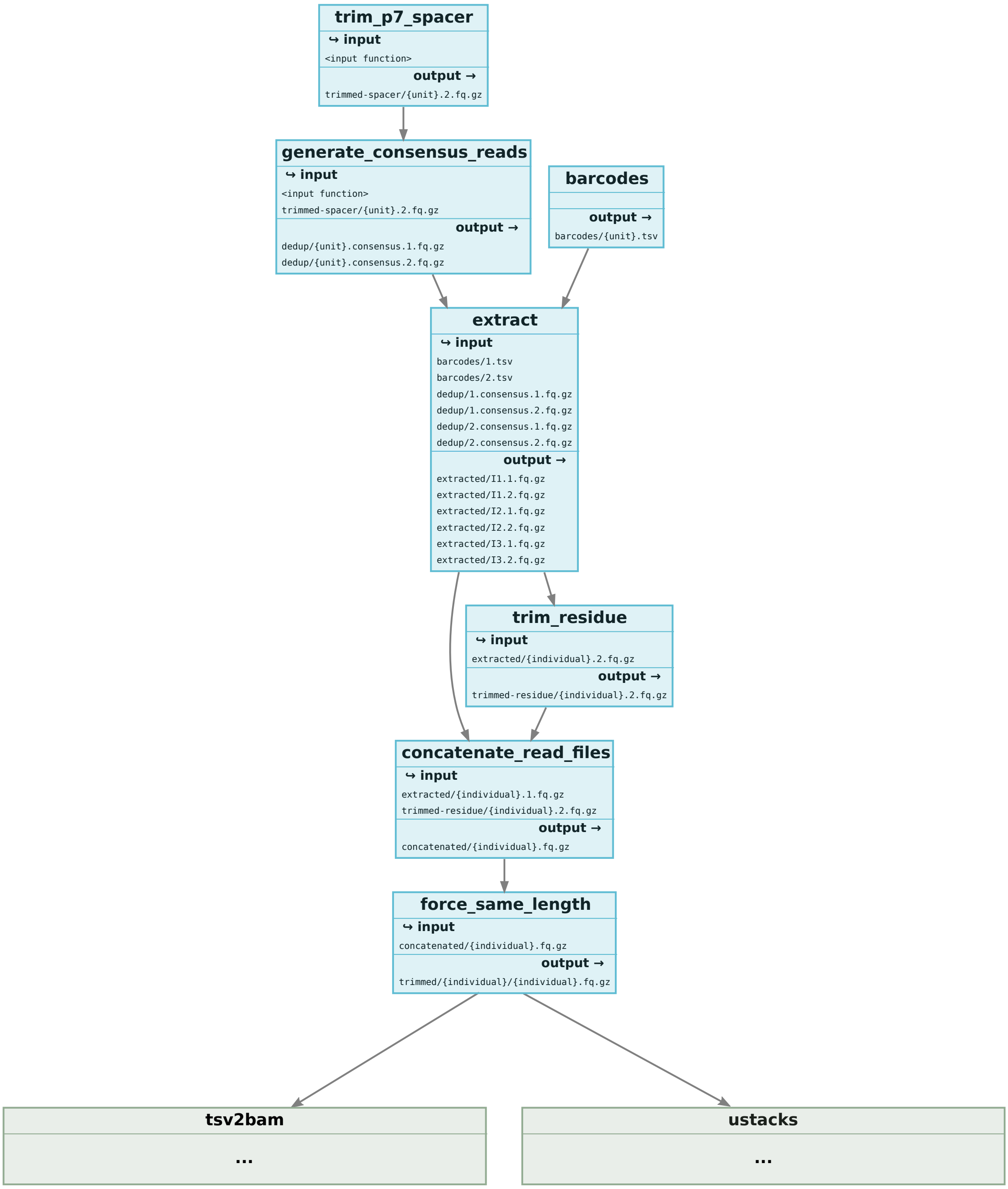

trim_p7_spacer
↪ input
<input function>
output →
trimmed-spacer/{unit}.2.fq.gz
generate_consensus_reads
↪ input
<input function>
trimmed-spacer/{unit}.2.fq.gz
output →
dedup/{unit}.consensus.1.fq.gz
dedup/{unit}.consensus.2.fq.gz
barcodes
output →
barcodes/{unit}.tsv
extract
↪ input
barcodes/1.tsv
barcodes/2.tsv
dedup/1.consensus.1.fq.gz
dedup/1.consensus.2.fq.gz
dedup/2.consensus.1.fq.gz
dedup/2.consensus.2.fq.gz
output →
extracted/I1.1.fq.gz
extracted/I1.2.fq.gz
extracted/I2.1.fq.gz
extracted/I2.2.fq.gz
extracted/I3.1.fq.gz
extracted/I3.2.fq.gz
trim_residue
↪ input
extracted/{individual}.2.fq.gz
output →
trimmed-residue/{individual}.2.fq.gz
concatenate_read_files
↪ input
extracted/{individual}.1.fq.gz
trimmed-residue/{individual}.2.fq.gz
output →
concatenated/{individual}.fq.gz
force_same_length
↪ input
concatenated/{individual}.fq.gz
output →
trimmed/{individual}/{individual}.fq.gz
tsv2bam
...
ustacks
...

Figure C.1: Snakemake filegraph for the preprocessing phase of our ddRADseq analysis workflow. The test dataset shown here contains three individuals I1, I2, and I3, from two units unit1 and unit2. Detailed descriptions of the single steps can be found in Section 7.3.1. Continued on page 247.

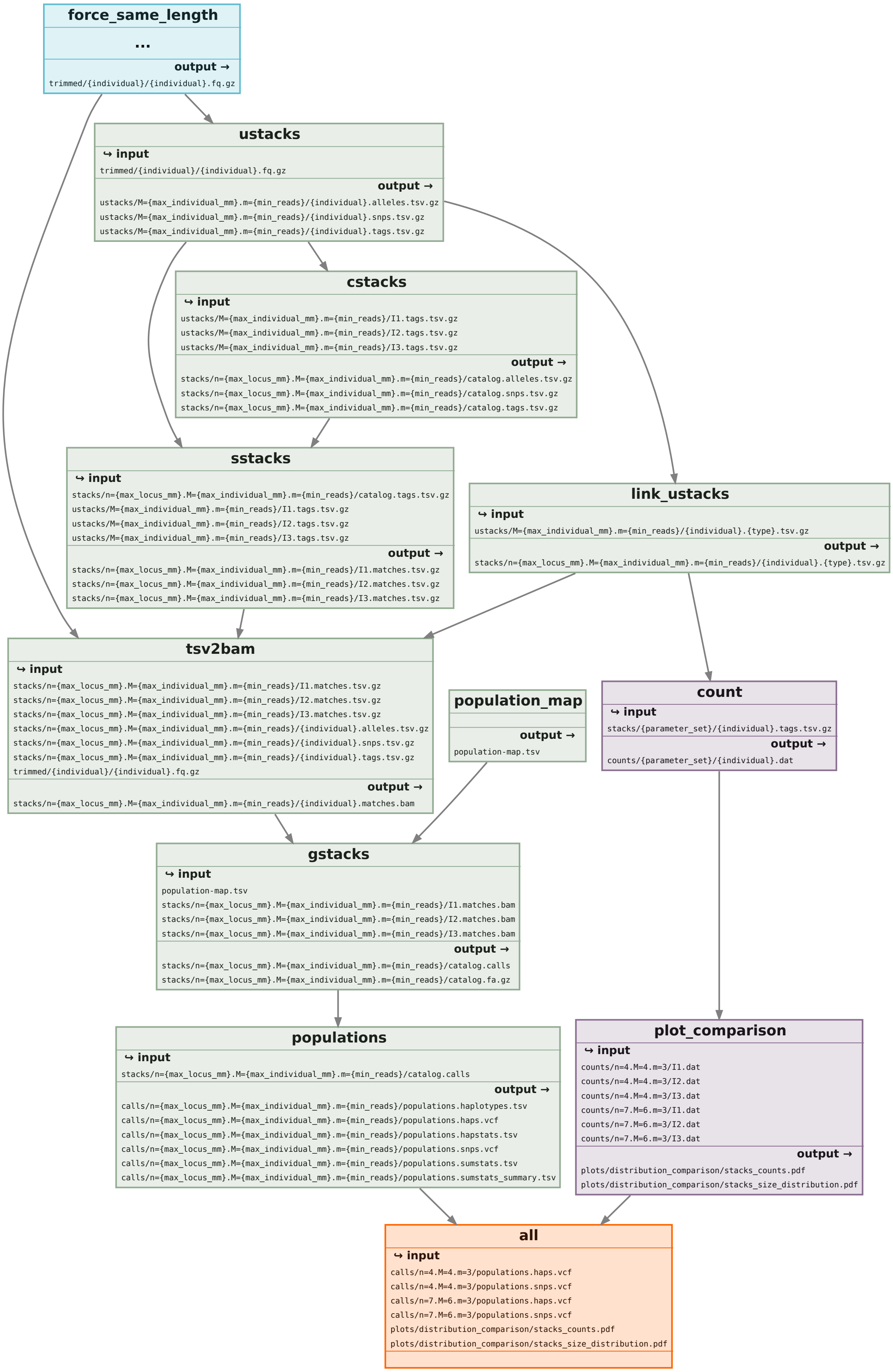

force_same_length
...
output →
trimmed/{individual}/{individual}.fq.gz
ustacks
↪ input
trimmed/{individual}/{individual}.fq.gz
output →
ustacks/M={max_individual_mm}.m={min_reads}/{individual}.alleles.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/{individual}.snps.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/{individual}.tags.tsv.gz
cstacks
↪ input
ustacks/M={max_individual_mm}.m={min_reads}/I1.tags.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/I2.tags.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/I3.tags.tsv.gz
output →
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.alleles.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.snps.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.tags.tsv.gz
sstacks
↪ input
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.tags.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/I1.tags.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/I2.tags.tsv.gz
ustacks/M={max_individual_mm}.m={min_reads}/I3.tags.tsv.gz
output →
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I1.matches.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I2.matches.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I3.matches.tsv.gz
link_ustacks
↪ input
ustacks/M={max_individual_mm}.m={min_reads}/{individual}.{type}.tsv.gz
output →
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/{individual}.{type}.tsv.gz
tsv2bam
↪ input
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I1.matches.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I2.matches.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I3.matches.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/{individual}.alleles.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/{individual}.snps.tsv.gz
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/{individual}.tags.tsv.gz
trimmed/{individual}/{individual}.fq.gz
output →
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/{individual}.matches.bam
population_map
output →
population-map.tsv
count
↪ input
stacks/{parameter_set}/{individual}.tags.tsv.gz
output →
counts/{parameter_set}/{individual}.dat
gstacks
↪ input
population-map.tsv
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I1.matches.bam
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I2.matches.bam
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/I3.matches.bam
output →
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.calls
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.fa.gz
populations
↪ input
stacks/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/catalog.calls
output →
calls/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/populations.haplotypes.tsv
calls/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/populations.haps.vcf
calls/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/populations.hapstats.tsv
calls/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/populations.snps.vcf
calls/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/populations.sumstats.tsv
calls/n={max_locus_mm}.M={max_individual_mm}.m={min_reads}/populations.sumstats_summary.tsv
plot_comparison
↪ input
counts/n=4.M=4.m=3/I1.dat
counts/n=4.M=4.m=3/I2.dat
counts/n=4.M=4.m=3/I3.dat
counts/n=7.M=6.m=3/I1.dat
counts/n=7.M=6.m=3/I2.dat
counts/n=7.M=6.m=3/I3.dat
output →
plots/distribution_comparison/stacks_counts.pdf
plots/distribution_comparison/stacks_size_distribution.pdf
all
↪ input
calls/n=4.M=4.m=3/populations.haps.vcf
calls/n=4.M=4.m=3/populations.snps.vcf
calls/n=7.M=6.m=3/populations.haps.vcf
calls/n=7.M=6.m=3/populations.snps.vcf
plots/distribution_comparison/stacks_counts.pdf
plots/distribution_comparison/stacks_size_distribution.pdf

Figure C.2: Snakemake filegraph for the Stacks workflow and Evaluation phases of our ddRADseq analysis workflow. Detailed descriptions of the single steps can be found in Section 7.3.1. Continuation from page 246.

D
Additional Plots for ddRAD Analysis Workflow Evaluation

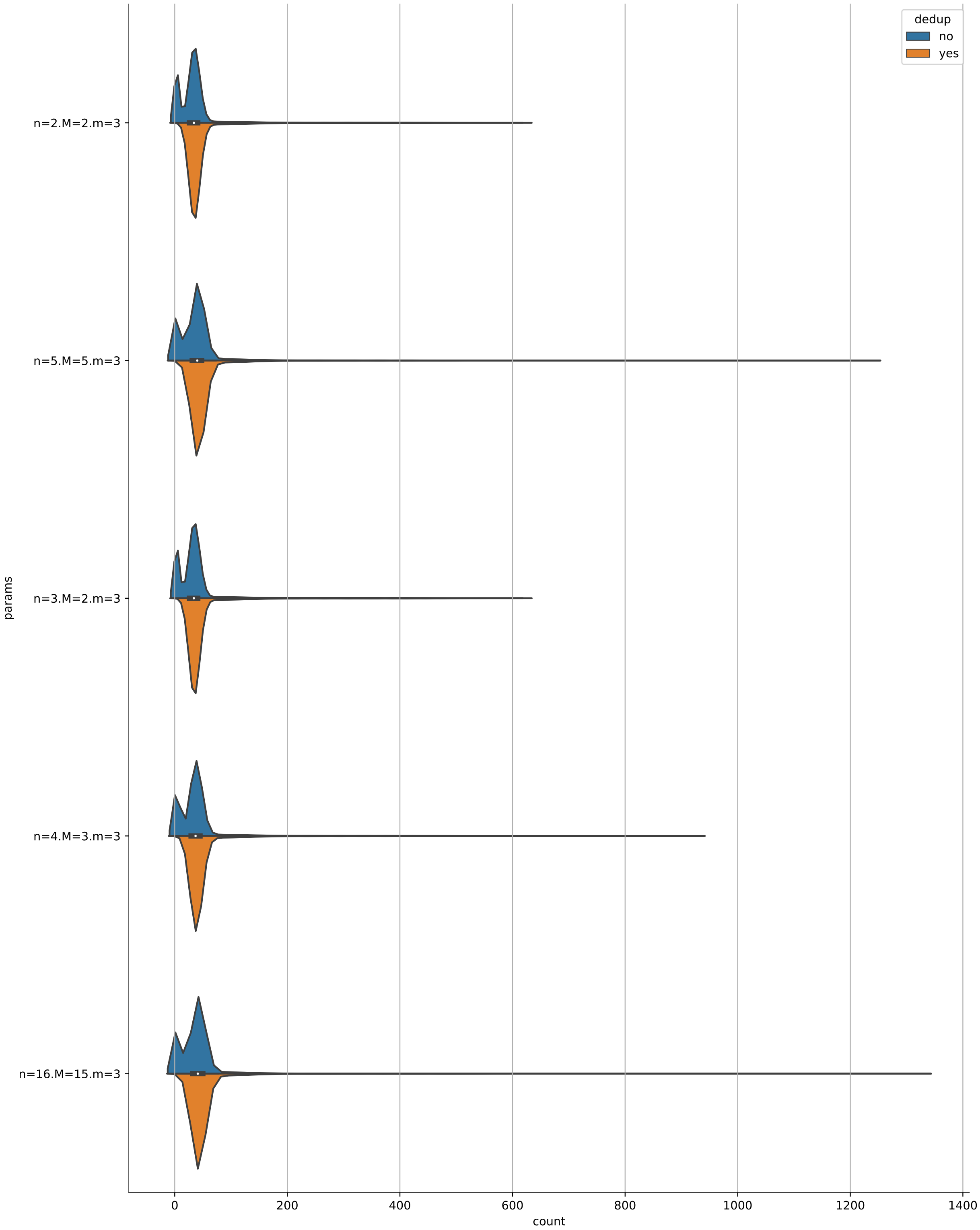


Figure D.1: Violin plots, comparing the distributions of locus sizes (x-axis in reads per individual and locus) for different parameter sets for Stacks (rows). (n: maximum distance between sample loci for merging, M: maximum distance of loci for merging within a sample, m: minimum stack stack size.) Each parameter set was run both with (orange) and without (blue) our PCR deduplication step. Note, that coverage values ≤ 0 shown in this plot are introduced by the kernel density estimation (KDE) used to generate the violin plots.

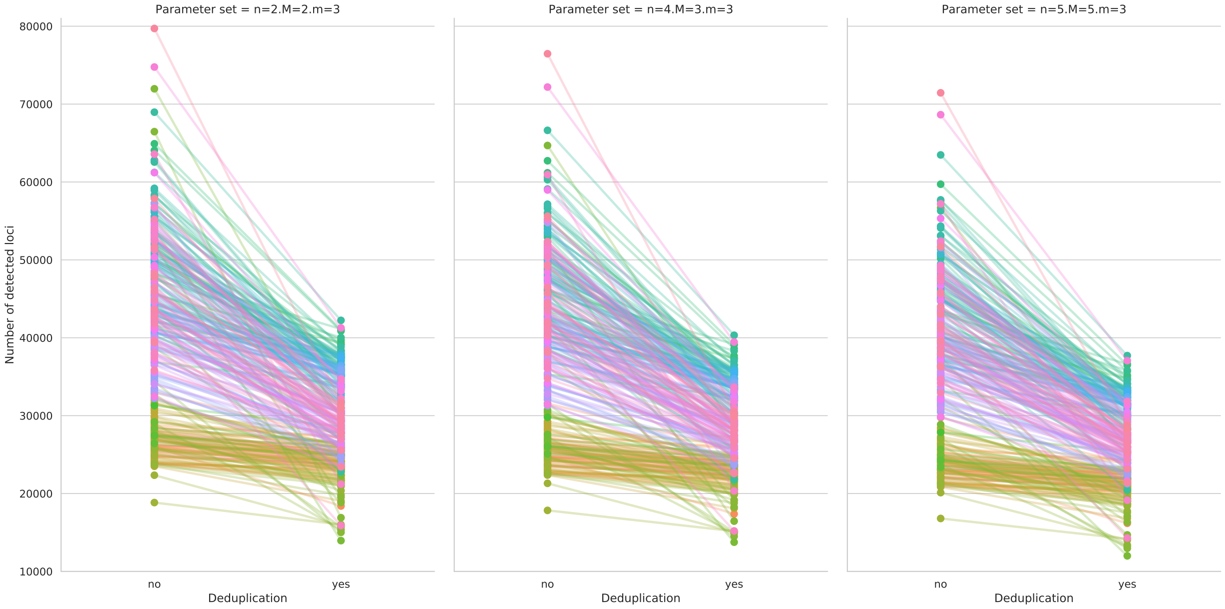


Figure D.2: Point plots comparing the number of loci identified in the *G. fossarum* dataset for each individual (y-axis) for runs without (left) and with (right) PCR deduplication. Connected points (of the same color) denote the same individual, owever, we omitted labels for specific individuals to increase the readbility of the figure. Individuals share a color and are connected.

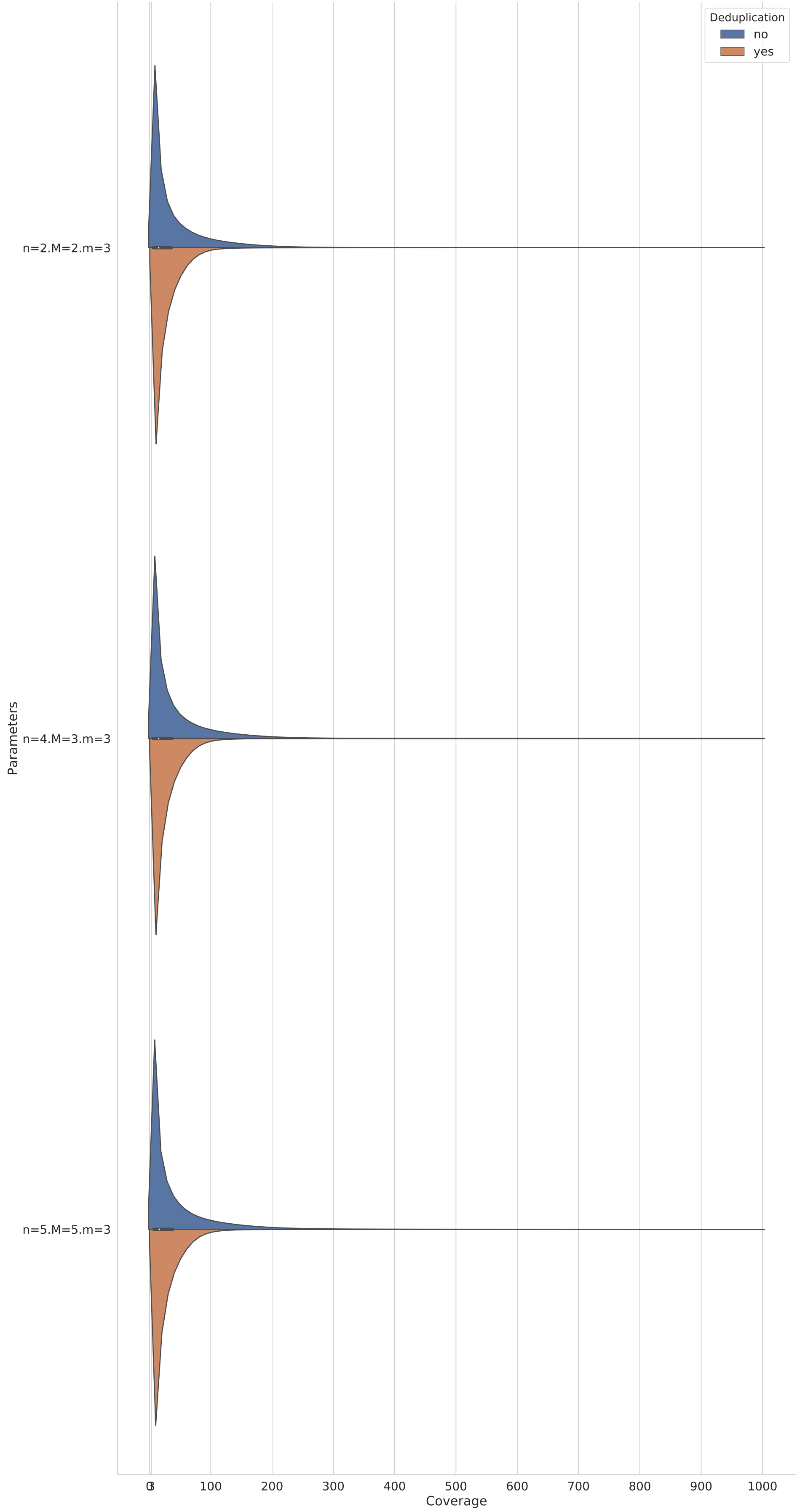


Figure D.3: Violin plots, comparing the distributions of locus sizes (x-axis in reads per individual and locus) for different parameter sets for Stacks (rows). (*n*: maximum distance between sample loci for merging, *M*: maximum distance of loci for merging within a sample, *m*: minimum stack stack size.) Each parameter set was run both with (orange) and without (blue) our PCR deduplication step. Note, that coverage values ≤ 0 shown in this plot are introduced by the kernel density estimation (KDE) used to generate the violin plots.

Abbreviations

AC	Average Case
BAM	Binary Alignment Map; File format to store sequence alignments
BBD	Beta-Binomial Distribution
BD	Binomial Distribution
BLOSUM	Blocks Substitution Matrix
bp	Base Pairs
BPHT	Bit-packed Hopscotch Hash Table
CLT	Central Limit Theorem
CNV	Copy Number Variation
CRAM	File format to store sequence alignments
CTR	Cyclic Reversible Termination
DBR	Degenerate Base Region
ddRADseq	Double Digest RADseq
DNA	Deoxyribonucleic Acid
DP	Dynamic Programming
DUD	Discrete Uniform Distribution
FASTA	File format to store DNA and amino acid sequences
FASTQ	File format to store DNA sequences and their associated (sequencing) quality values
FGS	First Generation Sequencing
GPHF	General Purpose Hash Function
GRCh37	Genome Reference Consortium Human Build 37
HDD	Hard Disk Drive
HF	Hash Function
hg38	Genome Reference Consortium Human Build 38
HRL	Highly Repetitive Locus
HT	Hash Table

i.i.d.	Independent and identically distributed
ID	Incomplete (Enzymatic) Digestion
Indel	Insertion and Deletion Mutations
ITS	Internal Transcribed Spacer
IUPAC	International Union of Pure and Applied Chemistry
KDE	Kernel Density Estimation
KVS	Key-Value Store
LoLN	Law of Large Numbers
LSB	Least Significant Bit
LSH	Locality Sensitive Hashing
M10, M15	Murphy Alphabets with 10 and 15 characters; Reduced Representation Amino Acid Alphabets
MD5	MD5 Message-Digest Algorithm
mmh3	Murmur Hash 3
MPHF	Minimal Perfect Hash Function
mRNA	Messenger Ribonucleic Acid
MSB	Most Significant Bit
NA	Null Allele
NCBI	National Center for Biotechnology Information
NP	Complexity class; Nondeterministic Polynomial Time
OMH	Order Min Hash
ONT	Oxford Nanopore Technologies
ORF	Open Reading Frame
p5 Read	Forward Read
p7 Read	Reverse Read
PacBio	Pacific Biosciences
PCR	Polymerase Chain Reaction
PD	Poisson Distribution
PE Read	Paired-end Read
PLHT	Plain Hopscotch Hash Table
RAD	Restriction Site Associated DNA
RADseq	Restriction Site Associated DNA sequencing
RNA	Ribonucleic Acid
RNG	Random Number Generator
rRNA	Ribosomal RNA

SAM	Sequence Alignment Map; File format to store sequence alignments
SBS	Sequencing by Synthesis
SE Read	Single-end Read
SFT	Six Frame Translation
SGS	Second Generation Sequencing
SHA	Secure Hash Algorithm
SMRT Sequencing	Single Molecule Real Time sequencing
SNP	Single Nucleotide Polymorphism
SNV	Single Nucleotide Variant
SSD	Solid-state Drive
SV	Structural Variation
TGS	Third Generation Sequencing
tmRNA	Transfer-messenger RNA
tRNA	Transfer RNA
UMI	Unique Molecular Identifier
VCF	Variant Call Format
VS	Value Store
WC	Worst Case
XOR	Exclusive (Bitwise) OR
ZMW	Zero Mode Waveguide
ZTPD	Zero-Truncated Poisson Distribution

Bibliography

Alberts, Bruce, Alexander Johnson, Julian Lewis, David Morgan, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell*. 6th Edition. New York, USA: Garland Science, 2017. DOI: 10.1201/9781315735368.

Alkan, Can, Bradley P. Coe, and Evan E. Eichler. "Genome Structural Variation Discovery and Genotyping". *Nature Reviews Genetics* 12.5 (2011), pp. 363–376. DOI: 10.1038/nrg2958.

Andrews, Kimberly R., Jeffrey M. Good, Michael R. Miller, Gordon Luikart, and Paul A. Hohenlohe. "Harnessing the Power of RAD-seq for Ecological and Evolutionary Genomics". *Nature Reviews Genetics* 17.2 (2016), pp. 81–92. DOI: 10.1038/nrg.2015.28.

Appleby, Austin. "Murmurhash3". https://github.com/aappleby/smhasher/wiki/MurmurHash3. Accessed on 17.01.2019. 2016.

– "SMHasher". https://github.com/aappleby/smhasher/wiki/SMHasher. Accessed on 27.06.2019. 2016.

Askitis, Nikolas. "Fast and Compact Hash Tables for Integer Keys". In: *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*. Australian Computer Society, Inc. 2009, pp. 113–122.

Bashiardes, Stavros, Gili Zilberman-Schapira, and Eran Elinav. "Use of Metatranscriptomics in Microbiome Research". *Bioinformatics and Biology Insights* 10 (2016). DOI: 10.4137/BBI.S34610.

Batu, Tugkan, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. "A Sublinear Algorithm for Weakly Approximating Edit Distance". In: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*. ACM. 2003, 316–324. DOI: 10.1145/780542.780590.

Beisser, Daniela, Nadine Graupner, Lars Grossmann, Henning Timm, Jens Boenigk, and Sven Rahmann. "TaxMapper: An Analysis Tool, Reference Database and Workflow for Metatranscriptome Analysis of Eukaryotic Microorganisms". *BMC Genomics* 18.1 (2017). DOI: 10.1186/s12864-017-4168-6.

Berlin, Konstantin, Sergey Koren, Chen-Shan Chin, James P. Drake, Jane M. Landolin, and Adam M. Phillippy. "Assembling Large Genomes with Single-Molecule Sequencing and Locality-Sensitive Hashing". *Nature Biotechnology* 33.6 (2015), pp. 623–630. DOI: 10.1038/nbt.3238.

Bertoni, Guido, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Keccak Sponge Function Family Main Document". *Submission to NIST (Round 2)* 3.30 (2009).

Bloom, Burton H. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". *Communications of the ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692.

Bonomi, Flavio, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. "An Improved Construction for Counting Bloom Filters". In: *European Symposium on Algorithms*. Springer. 2006, pp. 684–695. DOI: 10.1007/11841036_61.

Booth, Christine S., Elsje Pienaar, Joel R. Termaat, Scott E. Whitney, Tobias M. Louw, and Hendrik J. Viljoen. "Efficiency of the Polymerase Chain Reaction". *Chemical Engineering Science* 65.17 (2010), pp. 4996–5006. DOI: 10.1016/j.ces.2010.05.046.

Bresch, Carsten and Rudolf Hausmann. *Klassische und molekulare Genetik*. Springer Berlin Heidelberg, 1972. DOI: 10.1007/978-3-642-87168-9.

Broder, Andrei and Michael Mitzenmacher. "Using Multiple Hash Functions to Improve IP Lookups". In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 3. IEEE. 2001, pp. 1454–1463. DOI: 10.1109/INFCOM.2001.916641.

Broder, Andrei Z. "On the Resemblance and Containment of Documents". In: *Proceedings. Compression and Complexity of Sequences 1997*. IEEE. 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

Broder, Andrei Z., Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. "Min-Wise Independent Permutations". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 3. ACM. 1998, pp. 630–659. DOI: 10.1006/jcss.1999.1690.

Buchfink, Benjamin, Chao Xie, and Daniel H. Huson. "Fast and Sensitive Protein Alignment using DIAMOND". *Nature Methods* 12.1 (2015), pp. 59–60. DOI: 10.1038/nmeth.3176.

Buhler, Jeremy. "Efficient Large-Scale Sequence Comparison by Locality-Sensitive Hashing". *Bioinformatics* 17.5 (2001), pp. 419–428. DOI: 10.1093/bioinformatics/17.5.419.

Burkhardt, Stefan and Juha Kärkkäinen. "Better Filtering with Gapped q-Grams". In: *Combinatorial Pattern Matching*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 73–85. DOI: 10.1007/3-540-48194-X_6.

Carter, J. Lawrence and Mark N. Wegman. "Universal Classes of Hash Functions". *Journal of Computer and System Sciences* 18.2 (1979), pp. 143–154. DOI: 10.1016/0022-0000(79)90044-8.

Catchen, Julian, Paul A. Hohenlohe, Susan Bassham, Angel Amores, and William A. Cresko. "Stacks: an Analysis Tool Set for Population Genomics". *Molecular Ecology* 22.11 (2013), pp. 3124–3140. DOI: 10.1111/mec.12354.

Chafin, Tyler K., Bradley T. Martin, Steven M. Mussmann, Marlis R. Douglas, and Michael E. Douglas. "FRAGMATIC: in silico Locus

Prediction and Its Utility in Optimizing ddRADseq Projects". *Conservation Genetics Resources* 10.3 (2017), pp. 325–328. DOI: 10.1007/s12686-017-0814-1.

Chao, Kun-Mao, William R. Pearson, and Webb Miller. "Aligning Two Sequences Within a Specified Diagonal Band". *Bioinformatics* 8.5 (1992), pp. 481–487. DOI: 10.1093/bioinformatics/8.5.481.

Chum, Ondrej, James Philbin, and Andrew Zisserman. "Near Duplicate Image Detection: min-Hash and tf-idf Weighting". In: *Proceedings of the British Machine Vision Conference*. Vol. 810. BMVA Press, 2008, 50:1–50:10. DOI: 10.5244/C.22.50.

Cock, Peter J. A., Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants". *Nucleic Acids Research* 38.6 (2009), pp. 1767–1771. DOI: 10.1093/nar/gkp1137.

Cohen, Edith and Haim Kaplan. "Summarizing Data Using Bottom-k Sketches". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, 225–234. DOI: 10.1145/1281100.1281133.

Cohen, Jeffery S. and Daniel M. Kane. "Bounds on the Independence Required for Cuckoo Hashing". Manuscript. Accessed on 28.12.2020. 2009. URL: https://cseweb.ucsd.edu/~dakane/cuchkoohashing.pdf.

Collet, Yann. "xxHash–Extremely Fast Hash Algorithm". http://cyan4973.github.io/xxHash/. Accessed on 26.06.2019. 2016.

Conroy, Matthew M. "A Collection of Dice Problems". Manuscript. Accessed on 01.11.2020. 2018. URL: https://msekce.karlin.mff.cuni.cz/~nagy/NMSA202/dice1.pdf.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd Edition. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

Cornish-Bowden, Athel. "Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984." *Nucleic Acids Research* 13.9 (1985), pp. 3021–3030. DOI: 10.1093/nar/13.9.3021.

Dahlgaard, Søren and Mikkel Thorup. "Approximately Minwise Independence with Twisted Tabulation". In: *Scandinavian Workshop on Algorithm Theory – SWAT 2014*. Springer International Publishing, 2014, pp. 134–145. DOI: 10.1007/978-3-319-08404-6_12.

Damiani, Ernesto, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. "An Open Digest-based Technique for Spam Detection". In: *International Conference on Parallel and Distributed Computing Systems (PDCS)*. HANDLE: 2434/180377. ISCA, 2004, pp. 559–564.

Danecek, Petr, Adam Auton, Goncalo Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, Richard Durbin, and 1000 Genomes Project Analysis Group.

"The variant call format and VCFtools". *Bioinformatics* 27.15 (2011), pp. 2156–2158. DOI: 10.1093/bioinformatics/btr330.

Davey, John W. and Mark L. Blaxter. "RADSeq: Next-Generation Population Genetics". *Briefings in Functional Genomics* 9.5-6 (2010), pp. 416–423. DOI: 10.1093/bfgp/elq031.

Davies, Heledd M., Stephanie D. Nofal, Emilia J. McLaughlin, and Andrew R. Osborne. "Repetitive Sequences in Malaria Parasite Proteins". *FEMS Microbiology Reviews* 41.6 (2017), pp. 923–940. DOI: 10.1093/femsre/fux046.

DePristo, Mark A., Eric Banks, Ryan Poplin, Kiran V. Garimella, Jared R. Maguire, Christopher Hartl, Anthony A. Philippakis, Guillermo Del Angel, Manuel A. Rivas, Matt Hanna, Aaron McKenna, Tim J. Fennell, Andrew M. Kernytsky, Andrey Y. Sivachenko, Kristian Cibulskis, Stacey B. Gabriel, David Altshuler, and Mark J. Daly. "A Framework for Variation Discovery and Genotyping Using Next-generation DNA Sequencing Data". *Nature Genetics* 43.5 (2011), pp. 491–498. DOI: 10.1038/ng.806.

Deza, Michel M. and Elena Deza. *Encyclopedia of Distances*. Springer, 2009, pp. 1–583. DOI: 10.1007/978-3-642-30958-8.

Dietzfelbinger, Martin. "Universal Hashing and k-Wise Independent Random Variables via Integer Arithmetic without Primes". In: *Annual Symposium on Theoretical Aspects of Computer Science — STACS 96*. Springer. 1996, pp. 567–580. DOI: 10.1007/3-540-60922-9_46.

Dietzfelbinger, Martin, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. "A Reliable Randomized Algorithm for the Closest-Pair Problem". *Journal of Algorithms* 25.1 (1997), pp. 19–51. DOI: 10.1006/jagm.1997.0873.

Dietzfelbinger, Martin and Christoph Weidling. "Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins". *Theoretical Computer Science* 380.1–2 (2007), pp. 47–68. DOI: 10.1016/j.tcs.2007.02.054.

Drepper, Ulrich. "What Every Programmer Should Know About Memory". http://people.redhat.com/drepper/cpumemory.pdf. Accessed on 16.08.2020. 2007.

Drmota, Michael and Reinhard Kutzelnigg. "A Precise Analysis of Cuckoo Hashing". *ACM Transactions on Algorithms (TALG)* 8.2 (2012), 11:1–11:36. DOI: 10.1145/2151171.2151174.

Eaton, Deren A. R. "PyRAD: Assembly of de novo RADseq Loci for Phylogenetic Analyses". *Bioinformatics* 30.13 (2014), pp. 1844–1849. DOI: 10.1093/bioinformatics/btu121.

Eaton, Deren A. R. and Isaac Overcast. "ipyrad: Interactive Assembly and Analysis of RADseq Datasets". *Bioinformatics* 36.8 (2020), pp. 2592–2594. DOI: 10.1093/bioinformatics/btz966.

Elbayoumi, Mahmoud A. M. S. "Strategies for Quality and Performance Improvement of Hardware Verification and Synthesis Algorithms". HANDLE: 10919/51221. PhD thesis. Virginia Polytechnic Institute and State University, 2014.

Ensemble Genomes. “M. xanthus DK 1622”. ftp://ftp.ensemblgenomes.org/pub/bacteria/release-39/fasta/bacteria_0_collection/myxococcus_xanthus_dk_1622/dna/. Accessed on 24.02.2020. 2018.

Erlingsson, Ulfar, Mark Manasse, and Frank McSherry. “A Cool and Practical Alternative to Traditional Hash Tables”. In: *Proc. 7th Workshop on Distributed Data and Structures (WDAS’06)*. 2006.

Ewing, Brent and Phil Green. “Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities”. *Genome Research* 8.3 (1998), pp. 186–194. DOI: 10.1101/gr.8.3.186.

Fan, Li, Pei Cao, Jussara Almeida, and Andrei Z. Broder. “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol”. *IEEE/ACM Transactions on Networking (TON)* 8.3 (2000), pp. 281–293. DOI: 10.1109/90.851975.

Fennel, Tim and Nils Homer. “fgbio”. http://fulcrumgenomics.github.io/fgbio/tools/latest/CallDuplexConsensusReads.html. Accessed on 21.01.2020. 2017.

Flajolet, Philippe, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. “Hyperloglog: The Analysis of a Near-optimal Cardinality Estimation Algorithm”. In: *Discrete Mathematics and Theoretical Computer Science*. 2007, pp. 127–146.

Flajolet, Philippe and G. Nigel Martin. “Probabilistic Counting Algorithms for Data Base Applications”. *Journal of Computer and System Sciences* 31.2 (1985), pp. 182–209. DOI: 10.1016/0022-0000(85)90041-8.

Fotakis, Dimitris, Rasmus Pagh, Peter Sanders, and Paul Spirakis. “Space Efficient Hash Tables with Worst Case Constant Access Time”. In: *Annual Symposium on Theoretical Aspects of Computer Science — STACS 2003*. Springer. 2003, pp. 271–282. DOI: 10.1007/3-540-36494-3_25.

Fredman, Michael L., János Komlós, and Endre Szemerédi. “Storing a Sparse Table with O(1) Worst Case Access Time”. *Journal of the ACM (JACM)* 31.3 (1984), pp. 538–544. DOI: 10.1145/828.1884.

Gallager, Robert. “Low-density Parity-check Codes”. *IRE Transactions on Information Theory* 8.1 (1962), pp. 21–28. DOI: 10.1109/TIT.1962.1057683.

Gionis, Aristides, Piotr Indyk, and Rajeev Motwani. “Similarity Search in High Dimensions via Hashing”. In: *VLDB*. Vol. 99. 6. 1999, pp. 518–529.

Goodwin, Sara, James Gurtowski, Scott Ethe-Sayers, Panchajanya Deshpande, Michael C. Schatz, and W. Richard McCombie. “Oxford Nanopore Sequencing, Hybrid Error Correction, and de novo Assembly of a Eukaryotic Genome”. *Genome Research* 25.11 (2015), pp. 1750–1756. DOI: 10.1101/gr.191395.115.

Goodwin, Sara, John D. McPherson, and W. Richard McCombie. “Coming of Age: Ten Years of Next-Generation Sequencing Technologies”. *Nature Reviews Genetics* 17.6 (2016), pp. 333–351. DOI: 10.1038/nrg.2016.49.

Grüning, Björn, Ryan Dale, Andreas Sjödin, Brad A. Chapman, Jillian Rowe, Christopher H. Tomkins-Tinch, Renan Valieris, Johannes Köster, and The Bioconda Team. "Bioconda: Sustainable and Comprehensive Software Distribution for the Life Sciences". *Nature Methods* 15.7 (2018). Henning Timm is part of "The Bioconda Team"., pp. 475–476. DOI: 10.1038/s41592-018-0046-7.

Gudbjartsson, Daniel F., Hannes Helgason, Sigurjon A. Gudjonsson, Florian Zink, Asmundur Oddson, Arnaldur Gylfason, Soren Besenbacher, Gisli Magnusson, Bjarni V. Halldorsson, Eirikur Hjartarson, Gunnar Th. Sigurdsson, Simon N. Stacey, Michael L. Frigge, Hilma Holm, Jona Saemundsdottir, Hafdis Th. Helgadottir, Hrefna Johannsdottir, Gunnlaugur Sigfusson, Gudmundur Thorgeirsson, Jon Th. Sverrisson, Solveig Gretarsdottir, G. Bragi Walters, Thorunn Rafnar, Bjarni Thjodleifsson, Einar S. Bjornsson, Sigurdur Olafsson, Hildur Thorarinsdottir, Thora Steingrimsdottir, Thora S. Gudmundsdottir, Asgeir Theodors, Jon G. Jonasson, Asgeir Sigurdsson, Gyda Bjornsdottir, Jon J. Jonsson, Olafur Thorarensen, Petur Ludvigsson, Hakon Gudbjartsson, Gudmundur I. Eyjolfsson, Olof Sigurdardottir, Isleifur Olafsson, David O. Arnar, Olafur Th. Magnusson, Augustine Kong, Gisli Masson, Unnur Thorsteinsdottir, Agnar Helgason, Patrick Sulem, and Kari Stefansson. "Large-scale Whole-genome Sequencing of the Icelandic Population". *Nature Genetics* 47.5 (2015), pp. 435–444. DOI: 10.1038/ng.3247.

Haas, Brian J., Dirk Gevers, Ashlee M. Earl, Mike Feldgarden, Doyle V. Ward, Georgia Giannoukos, Dawn Ciulla, Diana Tabbaa, Sarah K. Highlander, Erica Sodergren, Barbara Methé, Todd Z. DeSantis, The Human Microbiome Consortium, Joseph F. Petrosino, Bob Knight, and Bruce W. Birren. "Chimeric 16S rRNA Sequence Formation and Detection in Sanger and 454-Pyrosequenced PCR Amplicons". *Genome Research* 21.3 (2011), pp. 494–504. DOI: 10.1101/gr.112730.110.

Hamming, Richard W. "Error Detecting and Error Correcting Codes". *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

Heather, James M and Benjamin Chain. "The Sequence of Sequencers: The History of Sequencing DNA". *Genomics* 107.1 (2016), pp. 1–8. DOI: 10.1016/j.ygeno.2015.11.003.

Heileman, Gregory L. and Wenbin Luo. "How Caching Affects Hashing". In: *Proceedings of the 7th Workshopon Algorithm Engineering and Experiments*. SIAM. 2005.

Henikoff, Steven and Jorja G. Henikoff. "Amino Acid Substitution Matrices From Protein Blocks". *Proceedings of the National Academy of Sciences* 89.22 (1992), pp. 10915–10919. DOI: 10.1073/pnas.89.22.10915.

Herlihy, Maurice, Nir Shavit, and Moran Tzafrir. "Hopscotch Hashing". In: *International Symposium on Distributed Computing*. Springer. 2008, pp. 350–364. DOI: 10.1007/978-3-540-87779-0_24.

Hunter, J. D. “Matplotlib: A 2D graphics environment”. *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Indyk, Piotr and Rajeev Motwani. “Approximate Nearest Neighbors: Towards Removing The Curse of Dimensionality”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing — STOC 98*. ACM. 1998, 604–613. DOI: 10.1145/276698.276876.

Jaccard, Paul. “Lois de distribution florale dans la zone alpine”. *Bulletin de la Société Vaudoise des Sciences Naturelles* 38.144 (1902), pp. 69–130. DOI: 10.5169/seals-266762.

Jain, Chirag. “Long Read Mapping at Scale: Algorithms and Applications”. HANDLE: 1853/61258. PhD thesis. Georgia Institute of Technology, 2019.

Jain, Chirag, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. “A Fast Approximate Algorithm for Mapping Long Reads to Large Reference Databases”. In: *International Conference on Research in Computational Molecular Biology*. Springer. 2017, pp. 66–81. DOI: 10.1007/978-3-319-56970-3_5.

Jain, Chirag, Sergey Koren, Alexander Dilthey, Adam M. Phillippy, and Srinivas Aluru. “A Fast Adaptive Algorithm for Computing Whole-Genome Homology Maps”. *Bioinformatics* 34.17 (2018), pp. i748–i756. DOI: 10.1093/bioinformatics/bty597.

Johnson, Norman L., Adrienne W. Kemp, and Samuel Kotz. *Univariate Discrete Distributions*. 3rd Edition. Hoboken, NJ: John Wiley & Sons, 2005.

Kim, Euihyeok and Min-Soo Kim. “Enhanced Chained and Cuckoo Hashing Methods for Multi-core CPUs”. *Cluster Computing* 17.3 (2014), pp. 665–680. DOI: 10.1007/s10586-013-0343-y.

Kirsch, Adam, Michael Mitzenmacher, and Udi Wieder. “More Robust Hashing: Cuckoo Hashing With a Stash”. *SIAM Journal on Computing* 39.4 (2010), pp. 1543–1561. DOI: 10.1137/080728743.

Knuth, Donald E. *The Art of Computer Programming: Sorting and Searching*. Vol. 3. Pearson Education, 1997.

Koren, Sergey, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. “Canu: Scalable and Accurate Long-Read Assembly via Adaptive k-Mer Weighting and Repeat Separation”. *Genome Research* 27.5 (2017), pp. 722–736. DOI: 10.1101/gr.215087.116.

Koslicki, David and Hooman Zabeti. “Improving MinHash via the Containment Index with Applications to Metagenomic Analysis”. *Applied Mathematics and Computation* 354 (2019), pp. 206–215. DOI: 10.1016/j.amc.2019.02.018.

Kourtis, Kornilios, Nikolas Ioannou, and Ioannis Koltsidas. “Reaping the Performance of Fast NVM Storage with uDepot”. In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 2019, pp. 1–15.

Köster, Johannes and Sven Rahmann. “Snakemake — A Scalable Bioinformatics Workflow Engine”. *Bioinformatics* 28.19 (2012), pp. 2520–2522. DOI: 10.1093/bioinformatics/bts480.

Köster, Johannes and Henning Timm. “snakemake-workflows/rad-seq-stacks”. Version v1.0.0. First released in 2018 on https://github.com/koesterlab/rad-seq-stacks. 2021. DOI: 10.5281/zenodo.4423333.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert. “Numba: A llvm-based Python Jit Compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC — LLVM ’15*. ACM. 2015, 7:1–7:6. DOI: 10.1145/2833157.2833162.

Lepais, Olivier and Jason T. Weir. “SimRAD: an R Package for Simulation-Based Prediction of the Number of Loci Expected in RADseq and Similar Genotyping by Sequencing Approaches”. *Molecular Ecology Resources* 14.6 (2014), pp. 1314–1321. DOI: 10.1111/1755-0998.12273.

Levenshtein, Vladimir I. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Soviet Physics — Doklady*. Vol. 10. 8. 1966, pp. 707–710.

Levy-Lahad, E., A. Lahad, S. Eisenberg, E. Dagan, T. Paperna, L. Kasinetz, R. Catane, B. Kaufman, U. Beller, P. Renbaum, and R. Gershoni-Baruch. “A Single Nucleotide Polymorphism in the Rad51 Gene Modifies Cancer Risk in BRCA2 but not BRCA1 Carriers”. *Proceedings of the National Academy of Sciences* 98.6 (2001), pp. 3232–3236. DOI: 10.1073/pnas.051624098.

Li, Heng. “Mathematical Notes on SAMtools Algorithms”. http://lh3lh3.users.sourceforge.net/download/samtools.pdf. Accessed: 2020-01-13. 2010.

– “Minimap and Miniasm: Fast Mapping and de novo Assembly for Noisy Long Sequences”. *Bioinformatics* 32.14 (2016), pp. 2103–2110. DOI: 10.1093/bioinformatics/btw152.

– “Minimap2: Pairwise Alignment for Nucleotide Sequences”. *Bioinformatics* 34.18 (2018), pp. 3094–3100. DOI: 10.1093/bioinformatics/bty191.

Limasset, Antoine, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. “Fast and Scalable Minimal Perfect Hashing for Massive Key Sets”. In: *16th International Symposium on Experimental Algorithms (SEA 2017)*. Vol. 75. 25. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, 25:1–25:16. DOI: 10.4230/LIPIcs.SEA.2017.25.

Louw, Tobias M., Christine S. Booth, Elsje Pienaar, Joel R. TerMaat, Scott E. Whitney, and Hendrik J. Viljoen. “Experimental Validation of a Fundamental Model for PCR Efficiency”. *Chemical Engineering Science* 66.8 (2011), pp. 1783–1789. DOI: 10.1016/j.ces.2011.01.029.

Luo, Yunan, Yun W. Yu, Jianyang Zeng, Bonnie Berger, and Jian Peng. “Metagenomic Binning Through Low Density Hashing”. *Bioinformatics* 35.2 (2018), pp. 219–226. DOI: 10.1093/bioinformatics/bty611.

Madigan, Michael T., John M. Martinko, David A. Stahl, and David P. Clark. *Brock Biology of Microorganisms*. 13th Edition. San Francisco, USA: Pearson Education, 2012.

Marcotte, Edward M., Matteo Pellegrini, Todd O. Yeates, and David Eisenberg. "A Census of Protein Repeats". *Journal of Molecular Biology* 293.1 (1999), pp. 151–160. DOI: 10.1006/jmbi.1999.3136.

Margulies, Marcel, Michael Egholm, William E. Altman, Said Attiya, Joel S. Bader, Lisa A. Bemben, Jan Berka, Michael S. Braverman, Yi-Ju Chen, Zhoutao Chen, et al. "Genome Sequencing in Microfabricated High-Density Picolitre Reactors". *Nature* 437.7057 (2005), pp. 376–380. DOI: 10.1038/nature03959.

Marçais, Guillaume, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. "Locality-sensitive Hashing for the Edit Distance". *Bioinformatics* 35.14 (2019), pp. i127–i135. DOI: 10.1093/bioinformatics/btz354.

Mastretta-Yanes, Alicia, Nils Arrigo, Nadir Alvarez, Tove H. Jorgensen, Daniel Piñero, and Brent C. Emerson. "Restriction Site-associated DNA Sequencing, Genotyping Error Estimation and de novo Assembly Optimization for Population Genetic Inference". *Molecular Ecology Resources* 15 (2015). DOI: 10.1111/1755-0998.12291.

Meyerhans, Andreas, Jean-Pierre Vartanian, and Simon Wain-Hobson. "DNA Recombination During PCR". *Nucleic Acids Research* 18.7 (1990), pp. 1687–1691. DOI: 10.1093/nar/18.7.1687.

Mills, Ryan E., Christopher T. Luttig, Christine E. Larkins, Adam Beauchamp, Circe Tsui, W. Stephen Pittard, and Scott E. Devine. "An Initial Map of Insertion and Deletion (INDEL) Variation in the Human Genome". *Genome Research* 16.9 (2006), pp. 1182–1190. DOI: 10.1101/gr.4565806.

Montgomery, Douglas C. and George C. Runger. *Applied Statistics and Probability for Engineers*. 6th edition. Hoboken, NJ: John Wiley & Sons, 2014.

Mora-Márquez, Fernando, Víctor García-Olivares, Brent C. Emerson, and Unai López de Heredia. "ddRADseqTools: a Software Package for in silico Simulation and Testing of Double Digest RADseq Experiments". *Molecular Ecology Resources* 17.2 (2017), pp. 230–246. DOI: 10.1111/1755-0998.12550.

Motwani, Rajeev and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. DOI: 10.1017/CBO9780511814075.

Murphy, Lynne R., Anders Wallqvist, and Ronald M. Levy. "Simplified Amino Acid Alphabets for Protein Fold Recognition and Implications for Folding". *Protein Engineering* 13.3 (2000), pp. 149–152. DOI: 10.1093/protein/13.3.149.

Myers, Gene. "A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming". *Journal of the ACM (JACM)* 46.3 (1999), 395–415. DOI: 10.1145/316542.316550.

Natsume, Satoshi, Hiroki Takagi, Akira Shiraishi, Jun Murata, Hiromi Toyonaga, Josef Patzak, Motoshige Takagi, Hiroki Yaegashi, Aiko Uemura, Chikako Mitsuoka, Kentaro Yoshida, Karel Krofta, Honoo Satake, Ryohei Terauchi, and Eiichiro Ono. "The Draft Genome of Hop (Humulus lupulus), an Essence for Brewing". *Plant and Cell Physiology* 56.3 (2015), pp. 428–441. DOI: 10.1093/pcp/pcu169.

NCBI. "Genome Reference Consortium Human Build 38". https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39. Accessed on 24.02.2020. 2019.

Needleman, Saul B. and Christian 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 (1970), pp. 443–453. DOI: 10.1016/0022-2836(70)90057-4.

Ondov, Brian D., Gabriel J. Starrett, Anna Sappington, Aleksandra Kostic, Sergey Koren, Christopher B. Buck, and Adam M. Phillippy. "Mash Screen: High-throughput Sequence Containment Estimation for Genome Discovery". *Genome Biology* 20.1 (2019), 232:1–232:13. DOI: 10.1186/s13059-019-1841-x.

Ondov, Brian D., Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. "Mash: Fast Genome and Metagenome Distance Estimation using MinHash". *Genome Biology* 17.1 (2016), 132:1–132:14. DOI: 10.1186/s13059-016-0997-x.

Orabi, Baraa, Emre Erhan, Brian McConeghy, Stanislav V. Volik, Stephane Le Bihan, Robert Bell, Colin C. Collins, Cedric Chauve, and Faraz Hach. "Alignment-free Clustering of UMI Tagged DNA Molecules". *Bioinformatics* 35.11 (2018), pp. 1829–1836. DOI: 10.1093/bioinformatics/bty888.

Ozsolak, Fatih and Patrice M. Milos. "RNA Sequencing: Advances, Challenges and Opportunities". *Nature Reviews Genetics* 12.2 (2011), pp. 87–98. DOI: 10.1038/nrg2934.

Pagh, Rasmus and Flemming F. Rodler. "Cuckoo Hashing". *Journal of Algorithms* 51.2 (2004), pp. 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

Paris, Josephine R., Jamie R. Stevens, and Julian M. Catchen. "Lost in Parameter Space: A Road Map for STACKS". *Methods in Ecology and Evolution* 8.10 (2017), pp. 1360–1373. DOI: 10.1111/2041-210X.12775.

Pătraşcu, Mihai and Mikkel Thorup. "On the k-Independence Required by Linear Probing and Minwise Independence". In: *International Colloquium on Automata, Languages and Programming*. Berlin, Heidelberg: Springer, 2010, pp. 715–726. DOI: 10.1007/978-3-642-14165-2_60.

Pătraşcu, Mihai and Mikkel Thorup. "The Power of Simple Tabulation Hashing". In: *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing — STOC '11*. ACM. 2011, 1–10. DOI: 10.1145/1993636.1993638.

– "Twisted Tabulation Hashing". In: *Proceedings of the 2013 Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2013, pp. 209–228. DOI: 10.1137/1.9781611973105.16.

Pearson, William R. and David J. Lipman. "Improved Tools for Biological Sequence Comparison". *Proceedings of the National Academy of Sciences* 85.8 (1988), pp. 2444–2448. DOI: 10.1073/pnas.85.8.2444.

Penard, Wouter and Tim van Werkhoven. "On the Secure Hash Algorithm Family". *Cryptography in Context* (2008), pp. 1–18.

Peterson, Brant K., Jesse N. Weber, Emily H. Kay, Heidi S. Fisher, and Hopi E. Hoekstra. "Double Digest RADseq: an Inexpensive Method for de novo SNP Discovery and Genotyping in Model and Non-Model Species". *PloS one* 7.5 (2012), e37135:1–e37135:11. DOI: 10.1371/journal.pone.0037135.

Pike, Geoff and Jyrki Alakuijala. "Introducing CityHash". https://opensource.googleblog.com/2011/04/introducing-cityhash.html. Accessed on 26.06.2019. 2011.

Popic, Victoria and Serafim Batzoglou. "A Hybrid Cloud Read Aligner Based on MinHash and Kmer Voting That Preserves Privacy". *Nature Communications* 8.1 (2017), 15311:1–15311:7. DOI: 10.1038/ncomms15311.

Quedenfeld, Jens and Sven Rahmann. "Analysis of Min-Hashing for Variant Tolerant DNA Read Mapping". In: *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*. Vol. 88. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2017, 21:1–21:13. DOI: 10.4230/LIPIcs.WABI.2017.21.

– "Variant Tolerant Read Mapping using Min-Hashing". *arXiv preprint arXiv:1702.01703* (2017).

Rang, Franka J., Wigard P. Kloosterman, and Jeroen de Ridder. "From Squiggle to Basepair: Computational Approaches for Improving Nanopore Sequencing Read Accuracy". *Genome Biology* 19.1 (2018), 90:1–90:11. DOI: 10.1186/s13059-018-1462-9.

Rasheed, Zeehasham and Huzefa Rangwala. "A Map-Reduce Framework for Clustering Metagenomes". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 549–558. DOI: 10.1109/IPDPSW.2013.100.

– "MC-MinH: Metagenome Clustering using Minwise Based Hashing". In: *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM. 2013, pp. 677–685. DOI: 10.1137/1.9781611972832.75.

Rees, David C., Thomas N. Williams, and Mark T. Gladwin. "Sickle-Cell Disease". *The Lancet* 376.9757 (2010), pp. 2018–2031. DOI: 10.1016/S0140-6736(10)61029-X.

Richter, Stefan, Victor Alvarez, and Jens Dittrich. "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing". *Proceedings of the VLDB Endowment* 9.3 (2015), 96–107. DOI: 10.14778/2850583.2850585.

Rivera-Colón, Angel G., Nicolas C. Rochette, and Julian M. Catchen. "Simulation with RADinitio Improves RADseq Experimental Design and Sheds Light on Sources of Missing Data". *Molecular*

Ecology Resources (2020), pp. 1–16. DOI: 10.1111/1755-0998.13163.

Rivest, Ronald. *The MD5 Message-Digest Algorithm*. Tech. rep. MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992. DOI: 10.17487/RFC1321.

Roberts, Michael, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. "Reducing Storage Requirements for Biological Sequence Comparison". *Bioinformatics* 20.18 (2004), pp. 3363–3369. DOI: 10.1093/bioinformatics/bth408.

Rochette, Nicolas C., Angel G. Rivera-Colón, and Julian M. Catchen. "Stacks 2: Analytical methods for paired-end sequencing improve RADseq-based population genomics". *Molecular Ecology* 28.21 (2019), pp. 4737–4754. DOI: 10.1111/mec.15253.

Rogaway, Phillip and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *International Workshop on Fast Software Encryption*. Berlin, Heidelberg: Springer, 2004, pp. 371–388. DOI: 10.1007/978-3-540-25937-4_24.

Ryynanen, Matti and Anssi Klapuri. "Query by Humming of Midi and Audio Using Locality Sensitive Hashing". In: *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2008, pp. 2249–2252. DOI: 10.1109/ICASSP.2008.4518093.

Sadowski, Caitlin and Greg Levin. *Simhash: Hash-Based Similarity Detection*. Tech. rep. Google, 2007.

Sanger, Frederick, Steven Nicklen, and Alan R. Coulson. "DNA Sequencing with Chain-Terminating Inhibitors". *Proceedings of the National Academy of Sciences* 74.12 (1977), pp. 5463–5467. DOI: 10.1073/pnas.74.12.5463.

Sboner, Andrea, Xinmeng Jasmine Mu, Dov Greenbaum, Raymond K. Auerbach, and Mark B. Gerstein. "The real cost of sequencing: higher than you think!" *Genome Biology* 12.1 (2011), 125:1–125:10. DOI: 10.1186/gb-2011-12-8-125.

Schleimer, Saul, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data — SIGMOD '03*. ACM. 2003, 76–85. DOI: 10.1145/872757.872770.

Schmitt, Armin O. and Hanspeter Herzel. "Estimating the Entropy of DNA Sequences". *Journal of Theoretical Biology* 188.3 (1997), pp. 369–377. DOI: 10.1006/jtbi.1997.0493.

Schweyen, Hannah, Andrey Rozenberg, and Florian Leese. "Detection and Removal of PCR Duplicates in Population Genomic ddRAD Studies by Addition of a Degenerate Base Region (DBR) in Sequencing Adapters". *The Biological Bulletin* 227.2 (2014), pp. 146–160. DOI: 10.1086/BBLv227n2p146.

Sena, Johnny A., Giulia Galotto, Nico P. Devitt, Melanie C. Connick, Jennifer L. Jacobi, Pooja E. Umale, Luis Vidali, and Callum J. Bell. "Unique Molecular Identifiers Reveal a Novel Sequencing

Artefact with Implications for RNA-Seq Based Gene Expression Analysis". *Scientific Reports* 8.1 (2018), 13121:1–13121:13. DOI: 10.1038/s41598-018-31064-7.

Shendure, Jay, Shankar Balasubramanian, George M. Church, Walter Gilbert, Jane Rogers, Jeffery A. Schloss, and Robert H. Waterston. "DNA Sequencing at 40: Past, Present and Future". *Nature* 550.7676 (2017), pp. 345–353. DOI: 10.1038/nature24286.

Sherry, Stephen T., M.-H. Ward, M. Kholodov, J. Baker, Lon Phan, Elizabeth M. Smigielski, and Karl Sirotkin. "dbSNP: the NCBI Database of Genetic Variation". *Nucleic Acids Research* 29.1 (2001), pp. 308–311. DOI: 10.1093/nar/29.1.308.

Siegel, Alan. "On Universal Classes of Extremely Random Constant-Time Hash Functions". *SIAM Journal on Computing* 33.3 (2004), pp. 505–543. DOI: 10.1137/S0097539701386216.

Smith, Temple F. and Michael S. Waterman. "Identification of Common Molecular Subsequences". *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. DOI: 10.1016/0022-2836(81)90087-5.

Smith, Tom, Andreas Heger, and Ian Sudbery. "UMI-tools: Modeling Sequencing Errors in Unique Molecular Identifiers to Improve Quantification Accuracy". *Genome Research* 27.3 (2017), pp. 491–499. DOI: 10.1101/gr.209601.116.

Smyth, Redmond P., Timothy E. Schlub, Andrew J. Grimm, Vanessa Venturi, Abha Chopra, Simon A. Mallal, Miles P. Davenport, and Johnson Mak. "Reducing Chimera Formation During PCR Amplification to Ensure Accurate Genotyping". *Gene* 469.1-2 (2010), pp. 45–51. DOI: 10.1016/j.gene.2010.08.009.

Stinson, Douglas R. "Universal Hashing and Authentication Codes". *Designs, Codes and Cryptography* 4.3 (1994), pp. 369–380. DOI: 10.1007/BF01388651.

Sudmant, Peter H., Tobias Rausch, Eugene J. Gardner, Robert E. Handsaker, Alexej Abyzov, John Huddleston, Yan Zhang, Kai Ye, Goo Jun, Markus Hsi-Yang Fritz, et al. "An Integrated Map of Structural Variation in 2,504 Human Genomes". *Nature* 526.7571 (2015), pp. 75–81. DOI: 10.1038/nature15394.

Tettelin, Hervé, Vega Masignani, Michael J. Cieslewicz, Claudio Donati, Duccio Medini, Naomi L. Ward, Samuel V. Angiuoli, Jonathan Crabtree, Amanda L. Jones, A. Scott Durkin, et al. "Genome Analysis of Multiple Pathogenic Isolates of Streptococcus Agalactiae: Implications for the Microbial "Pan-Genome"". *Proceedings of the National Academy of Sciences* 102.39 (2005), pp. 13950–13955. DOI: 10.1073/pnas.0506758102.

The 1000 Genomes Project Consortium. "A Global Reference for Human Genetic Variation". *Nature* 526.7571 (2015), pp. 68–74. DOI: 10.1038/nature15393.

Thorup, Mikkel. "Bottom-k and Priority Sampling, Set Similarity and Subset Sums with Minimal Independence". In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing — STOC '13*. ACM. 2013, 371–380. DOI: 10.1145/2488608.2488655.

Thorup, Mikkel. "Fast and Powerful Hashing Using Tabulation". *Communications of the ACM* 60.7 (2017), 94–101. DOI: 10.1145/3068772.

– "High Speed Hashing for Integers and Strings". *arXiv preprint arXiv:1504.06804* (2015).

Timm, Henning. "BPHT Evaluation Workflow". Zenodo. Version 1.1.0. 2020. DOI: 10.5281/zenodo.4041213.

– "BPHT Source Code". Zenodo. Version 1.0.0. 2020. DOI: 10.5281/zenodo.4065163.

– "PCR Deduplication Analysis Workflow". 2021. DOI: 10.5281/zenodo.4575278.

– "Rad-seq-stacks Evaluation Workflow". Version 1.0.0. 2021. DOI: 10.5281/zenodo.4420213.

– "Rust-tab-hash Source Code". Zenodo. Version 0.3.1. 2020. DOI: 10.5281/zenodo.3936766.

– "Segment Length Analysis Workflow". Version 1.0.0. 2021. DOI: 10.5281/zenodo.4448209.

Timm, Henning and Till Hartmann. "Dinopy — DNA input and output for Python and Cython". Zenodo. First released on 10.04.2015 on https://bitbucket.org/HenningTimm/dinopy. 2020. DOI: 10.5281/zenodo.4389306.

Timm, Henning and Sven Rahmann. "ddRAGE — ddRAD Data Generator (Source Code)". Zenodo. First released on 28.02.2017 on https://bitbucket.org/genomeinformatics/rage. 2020. DOI: 10.5281/zenodo.4390215.

Timm, Henning, Hannah Weigand, Martina Weiss, Florian Leese, and Sven Rahmann. "ddRAGE: A Data Set Generator to Evaluate ddRADseq Analysis Software". *Molecular Ecology Resources* 18.3 (2018), pp. 681–690. DOI: 10.1111/1755-0998.12743.

Tin, Mandy M.-Y., Frank E. Rheindt, Emilie Cros, and Alexander S. Mikheyev. "Degenerate Adaptor Sequences for Detecting PCR Duplicates in Reduced Representation Sequencing Data Improve Genotype Calling Accuracy". *Molecular Ecology Resources* 15.2 (2015), pp. 329–336. DOI: 10.1111/1755-0998.12314.

Turcatti, Gerardo, Anthony Romieu, Milan Fedurco, and Ana-Paula Tairi. "A New Class of Cleavable Fluorescent Nucleotides: Synthesis and Optimization as Reversible Terminators for Dna Sequencing by Synthesis". *Nucleic Acids Research* 36.4 (2008), e25:1–e25:13. DOI: 10.1093/nar/gkn021.

Ukkonen, Esko. "Finding Approximate Patterns in Strings". *Journal of Algorithms* 6.1 (1985), pp. 132–137. DOI: 10.1016/0196-6774(85)90023-9.

Wagner, Robert A. and Michael J. Fischer. "The String-to-String Correction Problem". *Journal of the ACM (JACM)* 21.1 (1974), 168–173. DOI: 10.1145/321796.321811.

Walzer, Stefan. "Load Thresholds for Cuckoo Hashing with Overlapping Blocks". In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Vol. 107. Schloss

Dagstuhl-Leibniz-Zentrum für Informatik. 2018, 102:1–102:10. DOI: 10.4230/LIPIcs.ICALP.2018.102.

Wang, Shi, Eli Meyer, John K. McKay, and Mikhail V. Matz. "2b-RAD: a Simple and Flexible Method for Genome-Wide Genotyping". *Nature Methods* 9.8 (2012), 808–810. DOI: 10.1038/nmeth.2023.

Waskom, Michael and the seaborn development team. "mwaskom/seaborn". 2020. DOI: 10.5281/zenodo.592845.

Webster, Arthur F. and Stafford E. Tavares. "On the Design of S-Boxes". In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Berlin, Heidelberg: Springer, 1986, pp. 523–534. DOI: 10.1007/3-540-39799-X_41.

Wegman, Mark N. and J. Lawrence Carter. "New Hash Functions and Their Use in Authentication and Set Equality". *Journal of Computer and System Sciences* 22.3 (1981), pp. 265–279. DOI: 10.1016/0022-0000(81)90033-7.

Wellcome Sanger Institute. "P. falciparum clone E5 Version 1". ftp://ftp.sanger.ac.uk/pub/project/pathogens/Plasmodium/falciparum/E5/Version1/. Accessed on 24.02.2020. 2017.

Westbrook, Anthony, Jordan Ramsdell, Taruna Schuelke, Louisa Normington, R. Daniel Bergeron, W. Kelley Thomas, and Matthew D. MacManes. "PALADIN: Protein Alignment for Functional Profiling Whole Metagenome Shotgun Data". *Bioinformatics* 33.10 (2017), pp. 1473–1478. DOI: 10.1093/bioinformatics/btx021.

Yang, Xiao, Jaroslaw Zola, and Srinivas Aluru. "Parallel Metagenomic Sequence Clustering via Sketching and Maximal Quasi-Clique Enumeration on Map-Reduce Clouds". In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE. 2011, pp. 1223–1233. DOI: 10.1109/IPDPS.2011.116.

Ye, Yuzhen, Jeong-Hyeon Choi, and Haixu Tang. "RAPSearch: A Fast Protein Similarity Search Tool for Short Reads". *BMC Bioinformatics* 12 (2011). DOI: 10.1186/1471-2105-12-159.

Zentgraf, Jens, Henning Timm, and Sven Rahmann. "Cost-optimal Assignment of Elements in Genome-scale Multi-way Bucketed Cuckoo Hash Tables". In: *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2020, pp. 186–198. DOI: 10.1137/1.9781611976007.15.

Zhao, Yongan, Haixu Tang, and Yuzhen Ye. "RAPSearch2: A Fast and Memory-efficient Protein Similarity Search Tool for Next-generation Sequencing Data". *Bioinformatics* 28.1 (2011), pp. 125–126. DOI: 10.1093/bioinformatics/btr595.

Zilversmit, Martine M., Sarah K. Volkman, Mark A. DePristo, Dyann F. Wirth, Philip Awadalla, and Daniel L. Hartl. "Low-complexity Regions in Plasmodium falciparum: Missing Links in the Evolution of an Extreme Genome". *Molecular Biology and Evolution* 27.9 (2010), pp. 2198–2209. DOI: 10.1093/molbev/msq108.

Zobrist, Albert L. *A New Hashing Method With Application for Game Playing*. Tech. rep. The DOI points to a 1990 reprint of this report by IOS Press. University of Wisconsin-Madison Department of Computer Sciences, 1970. DOI: 10.3233/ICG-1990-13203.

Zorita, Eduard, Pol Cusco, and Guillaume J. Filion. "Starcode: Sequence Clustering Based on All-pairs Search". *Bioinformatics* 31.12 (2015), pp. 1913–1919. DOI: 10.1093/bioinformatics/btv053.

List of Figures

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich, Henning Timm, dass die Dissertation von mir selbstständig angefertigt wurde und alle von mir genutzten Hilfsmittel angegeben wurden. Ich versichere, dass alle in Anspruch genommenen Quellen und Hilfen in der Dissertation vermerkt wurden.

Ort, Datum	Unterschrift