Engineering of Algorithms for Personal Genome Pipelines

DISSERTATION

zur Erlangung des Grades

eines Doktors der Naturwissenschaften (Dr. rer. nat.)

am Fachbereich für Mathematik und Informatik

der Freien Universität Berlin

vorgelegt von Manuel Holtgrewe

Betreuer: Prof. Dr. Knut Reinert

Berlin 2015

Erstgutachter: Prof. Dr. Knut Reinert

Zweitgutachter: Prof. Dr. Alexander Goesmann

Tag der Disputation:

Dienstag, 20. Oktober, 2015

Abstract

Recent technical advances in high-throughput sequencing technologies and their commercial availability at low costs have paved the way for revolutionary opportunities in the life sciences. One milestone was reaching the $1000 genome, allowing to determine the genetic makeup of hundreds of human individuals within a week for less than $1000 each. This ongoing revolution of the life sciences creates new challenges for the software and algorithms that are processing this data. In my thesis, I consider a typical software pipeline for determining the genome of a human individual.

For the preprocessing pipeline step, I describe a method for error correction and consider the comparison of such methods. For the read mapping step, I provide a formal definition of read mapping and I present a software package implementing a benchmark for read mapping, based on my formal definition. I then describe the implementation, parallelisation, and engineering of a fully sensitive read mapper and evaluate its performance. For the variant calling step, I present a method for the prediction of insertion breakpoints and the assembly of large insertions.

Of course, such a pipeline is not limited to the processing of human data but it is also applicable to data from other mammals or organisms with smaller and less complex genomes. The presented work is available as an efficient open source C++ implementation, either as parts of the SeqAn library or as programs using SeqAn.

Zusammenfassung

Die technischen Fortschritte der letzten Jahre im Bereich der Hochdurchsatzsequenzierung und die kommerzielle Verfügbarkeit dieser Technik haben den Weg für revolutionäre Möglichkeiten in den Lebenswissenschaften geebnet. Ein Meilenstein dabei war das Erreichen des sogenanten 1000 Dollar Genoms. Es ist heute möglich, das Erbgut von hunderten von Menschen in unter einer Woche für weniger als je 1000 Dollar auszulesen. Diese Revolution der Lebenswissenschaften stellt auch neue Herausforderungen an die Software und Algorithmen für die Verarbeitung dieser Daten. In meiner Arbeit betrachte ich eine typischen Pipeline um ein menschliches Genom zu dekodieren.

Für den Vorverarbeitungsschritt beschreibe ich eine Methode zur Fehlerkorrektur und vergleiche verschiedene solcher Methoden. Für den Read Mapping Schritt entwickle ich zunächst eine formale Definition von Read Mapping und stelle dann ein Softwarepaket vor, dass den Vergleich von Read Mappern, basierend auf der formalen Definition, erlaubt. Danach beschreibe ich die Implementierung, Parallelisierung, und das Engineering eines vollsensitiven Read Mappers. Für den Schritt der Variantenanalyse präsentiere ich eine Methode für die Vorhersage von Bruchpunkten und eine Methode für das gezielte Assemblieren von langen Insertionen.

Eine solche Pipeline kann selbstverständlich auch für genomische Daten von anderen Säugetieren oder Lebewesen mit kürzerem, weniger komplexen Genom einsetzt werden. Die vorgestellte Arbeit ist als effiziente, quelloffene C++ Implementierung verfügbar, zum Teil als Bestandteil der SeqAn Bibliothek und zum Teil als Programme auf der Grundlage von SeqAn.

Acknowledgements

I want to thank everyone who supported me during my PhD work and the writing of my thesis. First and foremost, I have to express my gratitude to my supervisor Knut Reinert. My work and research would not have been possible without his continuous, patient, and constructive support. In particular, he always took the time to give helpful and constructive remarks whenever I knocked on his door.

I am thankful to Alexander Goesmann for his willingness to appraise this thesis and I am indebted to the German Research Foundation (DFG) and the German Federal Ministry of Education and Research (BMBF) for providing the funding for my research.

I was lucky to be involved in a number of successful collaborations. I thank Anne-Katrin Emde and David Weese for the work on Rabema, David Weese for collaborating on RazerS 3, Leon Kuchenbecker for joint work on Anise and Basil, and Marcel Schulz, David Weese, and Hugues Richard for the collaboration on Fiona.

Also, I would like to thank all current and previous members of the Reinert lab at Freie Universität Berlin for a friendly, productive, and stimulating work environment. I also thank the BioStore team (Björn, David, Jochen, and Sabrina) for the motivating joint work on said project. In addition, I am grateful to my colleagues at Nick Robinson's lab at Charité Universitätsmedizin for creating an equally friendly and productive work environment and interesting collaborations. I am grateful to Birte, Enrico, Jochen, Justin, Kathrin, Leon, and Max for helping me by proofreading my thesis and giving many helpful comments. Further, a big *thank you* to the SeqAn team (Anne-Katrin, Björn, Enrico, Birte, David, Hannes, Jochen, Kathrin, Knut, Rene, Sabrina, and Stephan) for creating a productive developer group.

Last but not least, I thank my family, my friends, and especially Jasmin for their continuous and unconditional support over the last years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2001, the draft sequence of the whole human genome was published simultaneously by the publically funded Human Genome Project (HGP) (Lander *et al.*, 2001) and a privately funded project at Celera Genomics (Venter *et al.*, 2001). This event marked a milestone in life sciences since having a reference genome facilitates important studies. Such studies include the analysis of individual whole genomes, *e.g.*, for the diagnosis of rare diseases but also also the analysis of tumors for facilitating more targeted chemotherapies.

Besides biotechnical and organizational challenges, decoding a genome as complex as the one of human also created computational challenges. For example, for a long time it was not clear which strategy for decoding the human genome was the better one. The two considered strategies were *hierarchical shotgun sequencing* and *whole genome shotgun sequencing*. Even in 1997, Green discarded the whole genome shotgun strategy proposed by Weber and Myers (1997) as not cost-effective and predicted that it would not work. Despite these doubts, whole genome shotgun sequencing has become *the* standard method for obtaining the sequence of full genomes. Notably, the challenges of whole genome shotgun sequencing were largely resolved using algorithmic advances by computer scientists developing a *whole genome shotgun assembler* program (Myers *et al.*, 2000).

Since 2001, there have been large advances in the area of *sequencing*, the process of reading out sequence information from genetic samples. The HGP ran from 1990 to 2003 and had a total cost of 3 billion US dollars (Collins and Hamburg, 2013). The cost of sequencing a single *base* (genomic sequence character) at the beginning of the HGP was $1 and reached 10 cents in 2003. Today, 1 million bases can be read for 4 cents (Quail *et al.*, 2012) with modern high throughput sequencing (HTS) machines. In 2014, HTS machine manufacturer Illumina announced the possibility to sequence a whole human genome for less than $1000, taking less than three days per genome (Hayden, 2014). Thus, individual whole human genomes can now be decoded for a cost six orders of magnitude below the cost of creating the initial draft sequence and in days instead of a decade.

Processing the output of one HTS machine can already be challenging and many laboratories have more than one such machine. Also, many more machines will be deployed in the future after the first FDA (Food and Drug Agency) authorization for a HTS machine for clinical use in 2013 (Collins and Hamburg, 2013). Further, it has previously been estimated (*e.g.*, by Mardis (2006)) that a price of less than $1000 per genome breaks a cost/benefit barrier. The individual genome will become so useful in relation to its cost, that whole genome sequencing will become standard practice as determining the blood type is today. Thus, the analysis of genomic information will become commonplace in clinics and will create computational challenges in many places. At the same time, solutions to these challenges will create equally wide-spread benefits.

The topic of this thesis is the engineering of algorithmic methods for determining the genome of a single individual. Decoding an individual genome with all features and differences to the reference sequence is computational challenging. Thus, new algorithmic methods have to be designed and carefully implemented such that they allow a robust and efficient analysis. The relevance of this topic will even increase further given the current developments in sequencing technologies and their distribution to clinics.

Structure of This Chapter. In this chapter, I give an introduction to the topic of this thesis by giving a rough overview of the biological background and outlining the pipeline of steps usually performed for obtaining an individual's genome. In Section 1.1 I give a short, non-exhaustive outline of the biological background and introduce relevant terms from molecular biology. Then, in Section 1.2 I present a short overview of sequencing technology and the bioinformatics applications in resequencing. In Section 1.3 I briefly describe the algorithm engineering approach to algorithmics. Finally, in Section 1.4 I give an outline of this thesis.

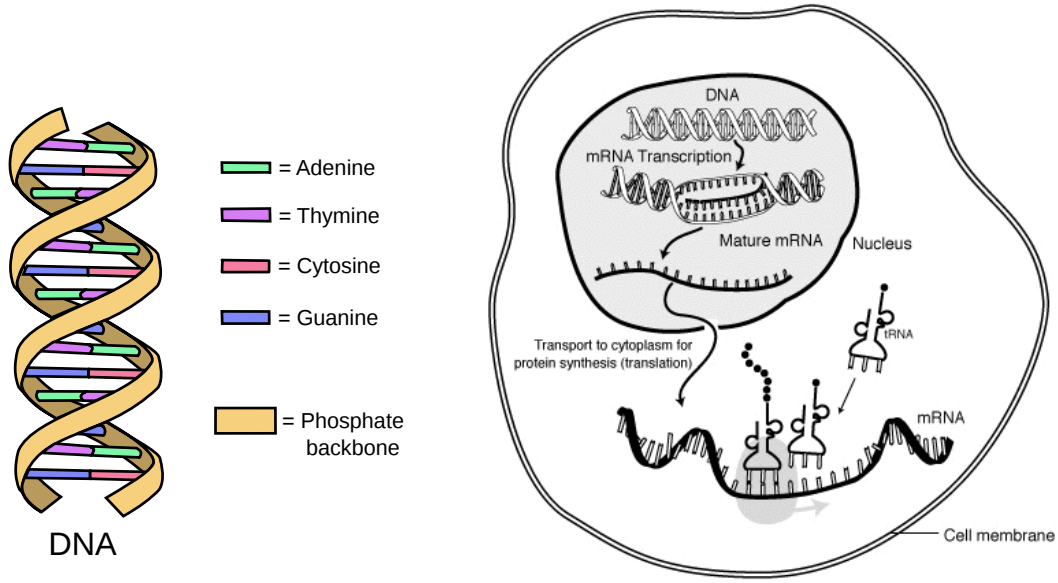
1.1 Biological Background

In this section, I introduce some fundamental terms from molecular biology and genetics. Readers who are already familiar with these topics can skip ahead to Section 1.2 on page 5.

DNA, RNA, and Proteins

Deoxyribonucleic acid (DNA) is an essential molecule in all known life forms that encodes the genetic information for an individual. Its double helix structure was discovered by Watson and Crick in 1953. Since then, a lot of effort has been invested in understanding the *genome* (the entirety of genetic information) of individual organisms and the relation to the genome of other organisms.

Usually, DNA molecules consist of two *strands* of *nucleotides* that are coiled around each other and form a double helix as can be seen in Figure 1.1a. A nucleotide consists of an organic base, a sugar, and phosphate groups and forms a *monomer*. In a DNA strand, the nucleotides are covalently linked into *polymers* having a sugar-phosphate backbone. The occuring bases are



(a) DNA structure

(b) fundamental dogma of biology

Figure 1.1: This figure shows (a) the double helix structure of DNA and (b) the "fundamental dogma of biology" of the transcription of DNA to mRNA followed by the translation of mRNA to proteins.
Source: National Institute of Health

adenine, cytosine, guanine, and thymine, often abbreviated as `A`, `C`, `G`, and `T`. Overall, a DNA molecule can be seen as a linear sequence of these bases (note that in most prokaryotes such as bacteria, the DNA molecules are cyclic). Cells of eukaryotic organisms such as human often have multiple DNA molecules, called *chromosomes*. The term *ploidy* describes the number of chromosome copies in each cell (for example, humans are diploid, having two copies of each of the 22 autosomal (non-sex) chromosomes). In summary, the genome of an individual consists of a number of DNA sequences, one for each chromosome copy.

The sugars in the backbone have five carbon atoms and are joined together by phosphate groups. A numbering from one to five of these carbon atoms has been established. The sugars are joined by phosphate groups between the third carbon atom of the one and the fifth carbon atom of the other (or next) nucleotide. Together with the asymmetry of the nucleotide molecules, this numbering leads to a 5' and a 3' end of each DNA strand.

In the double helix, each nucleotide is paired with a nucleotide from the other strand. Each nucleotide is always paired with its *complement* via hydrogen bounds: `A` is paired with `T`, and `C` is paired with `G`. The labels *forward strand* and *reverse strand* are assigned to the strands in an arbitrary but fixed order. As text, each strand is written from the 5' to the 3' end as a string of the characters `A`, `C`, `G`, and `T`. The reverse strand can be obtained from the forward strand by reversing the characters' order and complementing them (and vice versa).

One important function in organisms is the *replication* of DNA molecules to generate exact copies

of the chromosomes. In cells, this is performed by specialized enzymes called *DNA polymerase*. The hydrogen bonds between the nucleotides on opposite strands are relatively weak. During the process of replication, these bonds are broken. Then, the polymerase reads one strand of the DNA in the direction from the 5' to the 3' end and builds the reverse-complement from 3' to 5' end by adding the complementing nucleotides.

Another important function in organisms is the generation of *proteins*. Proteins are polymers of *amino acids* and perform a large part of an organism's functionality. For example, almost all enzymes are proteins. For creating proteins, an intermediate step called *transcription* is performed to obtain ribonucleic acid (RNA) molecules. An enzyme called *RNA polymerase* uses a DNA molecule as a template for synthesizing an RNA module. RNA molecules are similar to DNA molecules but are single-stranded, use uracil (often abbreviated as U) instead of thymine, and are less stable due to their structure. There are various kinds of generated RNA molecules and an important distinction is between *messenger RNA* (mRNA) and *non-messenger RNA* (also often called *non-coding RNA*).

After some processing, the then mature mRNA is used for *translation* into proteins. The translation is performed by complex molecular machines called *ribosomes*. The whole process is outlined in Figure 1.1b. For the translation, *triplets* (triples) of RNA nucleotides (called *codons*) are translated into one amino acid each. The translation can be seen as a function from the set of 64 RNA nucleotide triplets to the set of 20 standard amino acids while some triplets are reserved for encoding stop codons. The amino acid methionine corresponds to a start codon. In the standard genetic code used for human chromosomes, the number of codons for each amino acid varies from one to six codons per amino acid.

Protein-coding regions on the genome are examples of *genes*. The term *gene* is used to describe a region in a genome that encodes a functional unit for the organism. In eukaryotes such as human, protein-coding genes often consist of *exonic* and *intronic* regions, also called *exons* and *introns*. Both exons and introns are first transcribed from the genome into premature mRNA (the gene is *expressed*). Then, in the processing into mature mRNA, the introns are removed in the *splicing* step. The mature mRNA is then translated into proteins, such that the exons are translated into amino acids and *code* for the protein while introns do not. It is known that the expression level of genes can be increased or decreased *(regulated)* by the presence and action of certain proteins and non-coding RNA.

Genetic Variation

The precision of DNA replication is ensured by complex mechanisms that even include a proofreading step (Berg *et al.*, 2002, ch. 27). Nevertheless, errors might occur during replication at low probabilities and they might not be detected. In this context, an *error* means the deviation of a replicated DNA molecule from its template DNA polymer.

Such errors give rise to differences between individual genomes, and differences between two genomes are called *variations*. Such variations include the *substitution* of single nucleotides (*single nucleotide variants* or SNVs) and the insertion or deletion of short parts of the DNA polymer (*indels*). Larger variations are often called *structural variants (SVs)*. SVs include the *inversion*,

insertion, and *duplication* of DNA molecule segments. The borders of the segments that are subjected to structural variation are called *breakpoints*. It might not be possible to pinpoint them to a single base, so a breakpoint might also refer to a region of the genome.

Such variation gives rise to different forms of genes, their *alleles*: in general, in each human cell, the two chromosome copies differ. Thus, a given gene on one chromosome copy will be slightly different on the other copy, each being one allele of the gene. This also holds for non-coding segments *(loci)* on the chromosomes.

The *genotype* is the specific genetic make-up of an individual. In diploid organisms, such as human, the two copies of a chromosome can have different variants. Here, it makes sense to consider the genetic make-up of each copy separately, the *haplotype* (short for *haploid genotype*) for each copy. For example, there might be two SNV positions in a gene and the individual's genome has different values for both of them, say the characters `C` and `G` on the first haplotype and the characters `A` and `T` on the second haplotype. In general, this can lead to encoding two different proteins and one needs to know the actual haplotypes to know which proteins are encoded. A genotype that has the same allele for both copies is called *homozygous* and a genotype with different alleles for the copies (as in the example above) is called *heterozygous*.

There are also other sources of variation besides imperfect replication. For exampe, *retroviruses* have the ability to integrate virus DNA or RNA into the host genome, causing insertions into the DNA molecule in one cell with respect to another cell. Further, certain bacteria have the ability to exchange genetic material which can also be the cause for variation between two bacteria's genomes. In primates, such as human, *Alu* elements (short genome segments comprising about 10% of human DNA) are a source of further genomic variation. Together with *LINE-1* elements (short genome segments comprising about 17% of human DNA), Alu elements can insert themselves into DNA. Alu elements play an important role in the evolution of primates (Shen *et al.*, 2011).

Depending on their kind and location, variations can yield advantages or disadvantages for the affected cell or individual. For example, variations might increase or lower an individual's susceptibility to certain diseases. Variants might also lead to the resulting cell not being capable of survival. A change might also have no effect, for example when a SNV in a coding region does not alter the encoded amino acid.

The term *polymorphism* describes a variant that has no adverse effect on the individual, is thus able to spread throughout the population and be observed at a relatively high probability. In recent years, large-scale databases variants have been built for estimating variant frequencies. SNVs are the most common and best-studied type of variant and dbSNP (Sherry *et al.*, 2001) is a large-scale database of single nucleotide polymorphisms (SNPs), annotated with frequency. Commonly, a SNV is called a SNP if it occurs in at least 1% of the population.

1.2 Sequencing and Resequencing

The genome of an organism cannot be obtained directly from its chromosomes. Rather, due to technical limitations, only small sections of DNA molecules can be read out in a process called

sequencing. These can then later be jigsawed together in *de novo assembly* or a reference sequence can be used in *resequencing* to help obtain the genome. Contiguous sequence parts resulting from assembly are usually called *contigs*.

Genome Sequencing. The traditional sequencing technology is *Sanger sequencing* and yields long reads at a relatively low throughput. Sanger sequencing was used both by Celera Genomics and the HGP for obtaining the initial draft sequence of human, for example. In contrast, HTS platforms yield shorter reads at much higher throughput rates. Important examples for HTS platforms are the ones by Illumina, 454/Roche, and the IonTorrent technology by Life Sciences. In Appendix B, I describe these sequencing technologies in more detail. Here, the following description of technologies involved in DNA sequencing is sufficient.

DNA can be replicated at high efficiency using a process called *polymerase chain reaction (PCR)*. Further, DNA can be randomly cut into small pieces in a process called *shearing*. The sequence of these small reads can then be read out by *sequencing*. Using PCR, followed by shearing and sequencing, sequencing machines obtain sequence data in *reads* (the sequence of relatively short DNA polymers) such that each base in the genome is represented *(covered)* by multiple bases in the reads. For example, when it is known that a genome has a length of ≈ 5 mega base pairs (Mbp), as is the case for *E. coli*, and a total of 25 Mbp of read bases is generated, the genome is said to be *sequenced to a coverage of 5x*.

The literature contains a lot of work that deals with estimating the necessary coverage for obtaining an organism's genome, depending on the genome length, the read length, and other parameters. One of the important early works is the one by Lander and Waterman (1988). An extension of this model is due to Roach *et al.* (1995) who extend the results by Lander and Waterman (1988) to whole-genome sequencing (the original model was formulated for the *clone fingerprinting problem*). These articles provide mathematical models that allow to estimate the success of a *de novo* sequencing project depending on the chosen parameters for the sequencing process (and thus the necessary coverage).

Resequencing. When the *reference sequence* for an organism is available, *resequencing* experiments can be performed for obtaining the sequence of an individual of the same or a similar species. The aim is to identify similarities and differences to the selected reference sequence.

An important example for resequencing is obtaining the full genomic sequence of a human individual (the *donor*) using HTS. The sequenced information can then be used to determine the donor's genomic variations, *e.g.*, the nucleotides at SNV positions. One aim of ongoing research is to understand genomic predisposition for diseases. Studies aiming at correlating genomic features with diseases or features of the individual (such as hair or eye color) are called *genome-wide association studies (GWAS)*.

Another example of resequencing has been performed by Izutsu *et al.* (2012). A strain of *D. melanogaster* was kept in a dark environment for 57 years. The authors obtained a sample of this strain's DNA, sequenced it, and identified genetic adaption of this fruit fly strain to its dark environment. Thus, resequencing allows the observation of evolution in terms of the genomes

of individuals or populations.

While the reference sequence can be the genome of any particular individual, often a *consensus* sequence of some sort is used. That is, after obtaining the sequence of a collection of individuals, the most prevalent genotype (the *wild-type*) of a genomic location is chosen as the reference for this location. Often, such a reference sequence is accompanied by a database of observed polymorphisms, *e.g.*, dbSNP. This allows to determine the genotype at a SNP position even if the individual has the same nucleotide as the reference and no variant would be called without the database information.

The current standard coverage for the resequencing of whole human genomes with Illumina reads is 30-35x. The redundancy is required for covering most parts of the human genome and also for allowing the reliable detection of variants. There is some discussion on which sequencing depth should be used. For example, Ajay *et al.* (2011) found that at a coverage of 40x, reliable genotyping is possible for 95% of the genome. These authors argue against using only a 30x coverage for resequencing which, by their findings, leads to reliable genotyping of 90% or less of the genome.

A typical software pipeline for the analysis of HTS resequencing data consists of the following steps:

- The input consists of the considered organism's reference genome and a set of HTS reads.
- The reads are first subjected to the **preprocessing** step: the quality of the reads is analyzed in the *quality control* sub step and the *correction* of sequencing errors can be attempted.
- This is followed by the **read mapping** step: for each read, the sample location is searched for in the reference genome. That is, this step attempts to find the location in the reference genome that corresponds to the location where the read was sampled from in the donor genome.
- After read mapping, the **variant analysis** step follows. In this step, the mapped reads are analyzed for variants with respect to the reference genome.

1.3 Algorithm Engineering

Algorithm engineering (Sanders, 2009) is an approach for practical algorithm research and development that is inspired by the scientific method by Popper (1934). An overview is shown in Figure 1.2. This section describes the algorithm engineering approach as it applies to bioinformatics and the work described in this thesis.

The aim of algorithm engineering is to develop practical solutions for algorithmic problems and to analyze and understand their performance on real world input data sets. This aim is congruent with the aims of this thesis. The algorithm engineering development process resembles a cycle of the steps (1) design, (2) analysis, (3) implementation, (4) experiments. In Appendix C, I describe

my contributions to software and algorithm engineering for the SeqAn algorithm library (Döring *et al.*, 2008).

Design Step. In the first step, the algorithm is designed, *e.g.*, using existing algorithmic building blocks and data structures. In traditional algorithmics, algorithms are designed to have low running time and memory usage. The considered problems are either solved exactly or quality guarantees are sought for. Often in traditional algorithmics, running time and memory usage are considered in big-$\mathcal{O}$ notation only, *i.e.*, ignoring constant factors and summands, and on theoretical machine models.

In contrast, algorithm engineering seeks to create more practical solutions. For example, constant factors are important since real machines have limited resources and doubling memory usage might increase the cost by an even larger factor and make an algorithm economically infeasible to use. Furthermore, algorithms do not just have to work on paper, but their implementation must be feasible. Ideally, practical algorithms allow for simple implementation which often leads to fewer bugs and good performance.

Also, for many $\mathcal{NP}$-hard problems, no constant factor approximations are possible in general. However, the real inputs might not be worst case scenarios, so heuristics might yield better running times and sufficiently good results in practice.

Analysis Step. The next step is an analysis of the previously designed algorithm's resource requirements. The result of this step are bounds on the running time and memory consumption of the algorithm. Further, realistic models should be used, *e.g.*, using realistic models for disk I/O

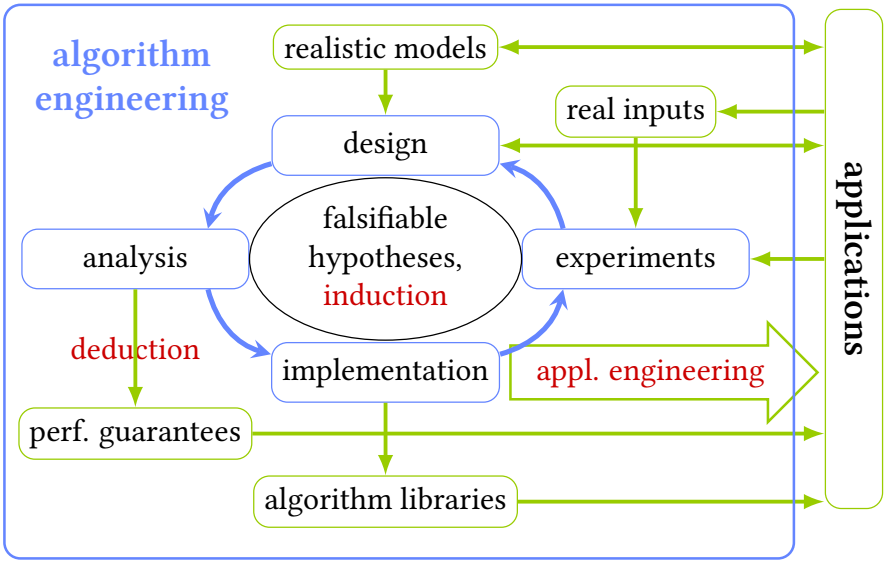


Figure 1.2: Overview of the algorithm engineering approach to algorithmics. Adapted from (Sanders, 2009).

when processing large inputs instead of assuming constant time memory access costs. This gives important hints at whether the algorithm is practical for real input data sets at all. Ideally, better performance guarantees can be deduced for the expected kinds of inputs than in the general case.

Implementation Step. Here, the algorithm is carefully implemented. To be of practical use, implementations must be correct and have a low resource consumption. Conversely, incorrect results or too high resource requirements (either by bugs or bad software engineering) can lead to being impractical. If available, previously deduced guarantees can help as checks to determine whether an implementation is of good quality.

In this step, there is a big overlap with software engineering and the best practices from this field should be used. For example, comprehensive automated tests help with writing implementations with fewer errors and not introducing regression errors in existing ones. Good documentation helps sustainable software engineering and reuse. Ideally, the implemented algorithmic solutions are made available as algorithm libraries such as MCSTL (Singler *et al.*, 2007), STXXL (Dementiev *et al.*, 2008), or SeqAn (Döring *et al.*, 2008) do.

Experiment Step. The implementation should then be tested on real data sets. In bioinformatics, the comprehensive evaluation on real data is crucial since many important features of biological data are not fully understood or hard to simulate. In many cases, data is available in public databases such as GenBank (Benson *et al.*, 2000), the Sequence Read Archive SRA (Leinonen *et al.*, 2011b), or the European Nucleotide Archive ENA (Leinonen *et al.*, 2011a).

Despite the shortcomings, simulated data is very useful when there is no ground truth available as is often the case in bioinformatics. Simulations are useful to complement the evaluation with real world data. The results of the experiments can then be used to improve the current algorithm design or create a new one.

The experiments should be driven by falsifiable hypotheses. For example, when an algorithm is designed to work well for repetitive genomic regions then care has to be taken to create appropriate experiments and check whether this design goal was actually achieved.

1.4 Thesis Outline

In this thesis, I give an overview of the state of the art in the algorithmic aspects of resequencing and describe my contribution to this field. The thesis is roughly structured after the typical resequencing pipeline outlined above.

In this chapter, I gave an introduction to the biological background. In Chapter 2 I give formal and mathematical preliminaries and describe some important fundamentals from the area of bioinformatics. In Chapter 3 I give an overview of the currently used approaches and methods in read preprocessing for quality control and read error correction. I discuss approaches for the evaluation of read error correction tools and present an experimental study for the comparison of read error correction methods. In Chapter 4 I give an introduction to and a formal definition

for the problem of read mapping and describe the state of the art for read mapping approaches and methods. Then, I present the RazerS 3 read mapping method, the Rabema method for the comparison of read mapping methods, and close with the description of an experimental study of read mapping methods. In Chapter 5 I give an overview of current methods for the detection and analysis of genomic variants. I then proceed to the description of the methods Basil and Anise for large genomic insertion analysis and close with an experimental evaluation of these two methods. Finally, in Chapter 6 I give a conclusion to the thesis.

Chapter 2

Preliminaries

In this chapter, I first give a description of the formal notation and terms used in this thesis in Section 2.1. This is followed by a short description of linear scanning algorithms and sequence alignment in Section 2.2 since these are two recurring topics in the thesis.

2.1 Mathematical Preliminaries

2.1.1 Logic

As usual, the logical operations *and* between two predicates p and q is denoted as $p \wedge q$ and the operation *or* is denoted as $p \vee q$. Also, if p implies q, this is denoted as $p \to q$.

Often, one needs an *indicator function* for a predicate. Given a predicate p, the indicator function is defined as

$$[p] := \begin{cases} 1, & \text{if } p \\ 0, & \text{else.} \end{cases}$$

2.1.2 Sets and Relations

Sets. Sets are denoted using curly brackets (*e.g.*, $\{0, 1, 2\}$ denotes the set with three elements 0, 1, and 2). Dots are used to denote that a sequence continues with (*e.g.*, $\{0, 1, \ldots, 9, 10\}$) or without (*e.g.*, $\{0, 1, \ldots\}$) a bound. As usual, $\mathbb{N}$ is the set of the non-negative (*natural*) numbers $\{0, 1, 2, \ldots\}$ and $\mathbb{Z}$ is the set of all integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

The Cartesian product of two sets S_1 and S_2 is denoted as $S_1 \times S_2$. S_1 being a (true) subset of S_2 is denoted as $S_1 \subset S_2$. S_1 being a subset of or equal to S_2 is denoted as $S_1 \subseteq S_2$.

The size of set S is denoted as $|S|$.

Relations. A relation R over a set S is a set of ordered pairs ($R \subseteq S \times S$). Given a set S and a relation R over S, the membership of a pair in the relation $(a, b) \in R$ for $a, b \in S$ is denoted as aRb.

In this thesis, the following properties of relations are of interest. Given a relation R over set S and $a, b, c \in S$. The relation R is *reflexive* if aRa holds for all $a \in S$. The relation R is *symmetric* if $aRb \rightarrow bRa$ holds for all $a, b \in S$. The relation R is *transitive* if $(aRb \wedge bRc) \rightarrow aRc$ holds for all $a, b, c \in S$. An *equivalence relation* is defined as a relation R that is reflexive, symmetric, and transitive.

2.1.3 Sequences, Strings and Alphabets

A sequence *(string)* S over a set A (the *alphabet*) of length k is denoted as $S = \langle s_0, s_1, \ldots, s_{k-1} \rangle$ ($s_i \in A$). The length of S is denoted as $|S|$.

The *concatenation* of two sequences S and T is written as $S \cdot T$.

Given a finite set A and a sequence $S = \langle s_0, s_1, \ldots, s_{k-1} \rangle$ of length k with $s_i \in A$ for all i. $S[i] = s_i$ is called the *i-th character of* S for $0 \leq i < |S|$ (0-based string indices are used as is common in the C programming language, for example). The empty string is denoted as $\mathcal{E}$.

In this thesis, I use the alphabets $\textsc{Dna} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and $\textsc{Dna5} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$.

Substrings. It is often convenient to have access to contiguous substrings. Given a string S and an integer i ($0 \leq i < |S|$).

The *prefix* of length n of S is defined as: $\text{prefix}(s, n) := \langle s_0, \ldots, s_{n-1} \rangle$. The *suffix* of length n of S is defined as: $\text{suffix}(s, n) := \langle s_n, \ldots, s_{|S|-1} \rangle$. The *infix* from i to j in S is defined as: $\text{infix}(s, i, j) := \langle s_i, \ldots, s_{j-1} \rangle$.

2.1.4 String Distance Metrics

Given two strings H and V over an arbitrary alphabet. Often, it is of interest to measure how different or similar H and V are. Two common and simple ways of measuring such *string distances* are *Hamming distance* (Hamming, 1950) (see also (Gusfield, 1997, p. 403)) and *edit distance* (Levenshtein, 1965) (see also (Gusfield, 1997, p. 215)).

Hamming Distance. Given two strings H and V of equal length ℓ. The Hamming distance δ_h is defined as the number of *mismatches* between two strings:

$$\delta_h(H, V) := \sum_{i=0}^{\ell-1} [H[i] \neq V[i]].$$

Edit Transcript. One common way of defining edit distance is using *edit transcripts*. An *edit transcript* is a series of operations describing a transformation from one string into another. The operations are the *deletion* of a character, the *insertion* of a character, and the *substitution* of a character. The set of the first two operations is often abbreviated as *indel*. Further, a *match* describes that a character remains unmodified. These operations are denoted as *D*, *I*, *S*, and *M*. The first three operations are called *edit operations*.

Edit Distance. Given two strings H and V, each of arbitrary length. The edit distance $\delta_e(H, V)$ of H and V is defined as the minimal number of edit operations (thus excluding *M*) to transform H into V.

A common visualization of a transcript is to write the two strings below each other, inserting *gaps* between the characters in the case of insertions. An insertion yields a gap in the first sequence and a deletion yields a gap in the second sequence. Gaps are often denoted as hyphens "-". For example:

transcript	M M M S M M I M M M D M M
H	C G A T G C T A G T − T A
V	C G A A G C − A G T A T A

Hamming Neighborhood. Given an integer d, the *Hamming neighborhood* of a string S is the set of strings that have Hamming distance $\leq d$ to S.

2.1.5 Alignments

Often, "writing two or more strings below each other with gaps" is formalized as *(tabular) alignments*. The alignment contains a row for each string and each row has the same number of characters. For characters, the alphabet A of the strings and a non-alphabet gap character "-" are used. Alignments of two strings are called *pairwise alignments*. When more than two strings are aligned, this is called a *multiple sequence alignment (MSA)*.

Alignment Scores. Alignments are often used in the context of sequence comparison and determining the similarity or distance of strings. For this, *(pairwise) alignment scores* can be used. In this thesis, I use the following definition.

Given an alignment A of two strings H and V, its pairwise alignment score is computed as

$$\text{score}(A) := \sum_j \delta(A[0, j], A[1, j]) + c \cdot \text{gap-opens}(A),$$

where $A[i, j]$ is the character from row i at position j in the alignment, δ is a function that assigns a numeric value to a pair of alignment characters, and gap-opens(A) is the number of gaps opened in A.

In the case of $c = 0$, the score is called *linear* and otherwise it is called *affine*. The score for alignment of characters against gaps and c is required to be smaller than zero so they are penalized.

In some cases, gaps are not penalized at the beginning or end (*begin gaps* or *end gaps*) of the first or second row and the score is called *begin (or end) gaps free in first (or second) sequence*. A *global alignment* is an alignment that has no free begin or end gaps. When the alignment is begin and end gaps free in the second sequence, it is called a *semi-global alignment of V against H*. When the alignment is begin and end gaps free in both sequences, it is called an *overlap alignment of H and V*.

Given two sequences H and V and a score function, I define the pairwise alignment problem as finding a pairwise alignment for H and V such that the alignment score is maximized.

MSA Scores. As often in the literature, the extension of alignment scores to multiple sequence alignments is defined in terms of pairwise alignments in a *sum of pairs score*. Each unordered pair of alignment rows is considered and I define the score of the MSA as the sum of the scores of the pairwise alignment created by using the selected MSA rows as rows for a pairwise alignment.

2.1.6 String Search

In many applications, it is of interest to find a small *needle* (or *query*) sequence in a long *haystack* (or *database*) sequence.

Exact String Search Problem. Given a needle N and a haystack H, a *match* of N in H is defined as a position p such that infix$(H, p, p+|N|) = N$. I define the *exact string search problem* to find the leftmost match of N in H. A natural extension is the *exact string search all* problem to find all matches of N in H.

Approximate String Search Problem. Given a needle N and a haystack H, a distance function δ, and a maximal distance k. Approximate string search is a relaxation of exact string search by allowing errors in the match.

In *approximate Hamming distance string search*, δ_h is used as the distance function. Again, a match is identified by its position p in H. I define the aim to find the leftmost match p of N in H such that $\delta_h(\text{infix}(H, p, p + |N|), N) \leq k$, that is the Hamming distance of the needle and the

match is at most k. Again, the extension to the *approximate Hamming distance string search all* problem is the natural one.

Similarly, in *approximate edit distance string search*, δ_e is used as the distance function. The definition of a match for edit distance matches is more involved and I will discuss it in Chapter 4. For now, let a match be identified by its position p and length ℓ. I define the aim to find the leftmost match p of length ℓ such that $\delta_e(\text{infix}(H, p, p + \ell), N) \leq k$.

It is one of the aims of Chapter 4 to give a precise formulation of approximate string search using edit distance, especially when considering multiple matches.

2.2 Algorithmic Background

The Terms Approach and Method. In this thesis, I use the terms *approach* to describe a class of algorithms for solving a specific problem. An example for an approach is the divide-and-conquer approach for comparison-based sorting.

I use the term *method* to describe a realization of one or more approaches. Thus, Quicksort and Mergesort are both methods for sorting following the divide-and-conquer approach for comparison-based sorting.

2.2.1 Alignment Algorithms

Efficient algorithms for finding optimal pairwise alignments are well-known early results of bioinformatics. Needleman and Wunsch (1970) described an efficient solution to compute general pairwise alignments with linear gap scores while Gotoh (1982) found an extension that also works with affine gap scores. Both solutions are based on dynamic programming and have running time and space consumption of $\mathcal{O}(n \cdot m)$ where n and m are the lengths of the first and second sequence.

Here, the Needleman-Wunsch algorithm is described for linear gap costs g and for sequences H and V. The used score function for scoring a pair of alignment characters is called s. The solution is based on a dynamic programming (DP) matrix F with $|H| + 1$ columns and $|V| + 1$ rows. Algorithm 2.1 shows the algorithm and Figure 2.1 gives an example.

$F[i, j]$ is the entry of matrix F in row i and column j. The DP solution is based on a recurrence relation. The base cases are given by $F[i, 0] := i \cdot g$ and $F[0, j] := i \cdot g$. The recurrence relation is $F[i, j] := \min(F[i-1, j]+g, F[i, j-1]+g, F[i-1, j-1]+s(V[i], H[j]))$. The arguments to $\min$ are the score for inserting a gap in V, inserting a gap in H, and aligning two string characters.

Besides storing the value of $F[i, j]$, each cell also stores a pointer (having a value from $\{\leftarrow, \uparrow, \nwarrow\}$) to indicate which one of the arguments to $\min$ was used. Note that there might be ambiguities, but picking any best cell for basing the next store upon yields an optimal result of the algorithm.

After computing F in quadratic time, the algorithm can now compute an optimal alignment in linear time in the *traceback* step. To compute an alignment with no free end gaps, the algorithm

Algorithm 2.1: The NEEDLEMAN-WUNSCH algorithm for computing global alignments.

```
Input  : H, V                 // sequences to align
         s                    // pairwise score function
         gap-cost             // gap cost of scoring scheme
Output : bestScore            // score of best alignment
         t                    // transcript representing alignment using
                              // M, I, D for match/mismatch, insertion, deletion

// allocate matrix and initialize first row and column
F ← new (m + 1) × (n + 1) matrix
F[i · gap-cost, 0] ← (i, ↑) for i ← 0 ... m
F[0, j · gap-cost] ← (j, ←) for j ← 0 ... n

// fill matrix in column-wise fashion
for i ← 1 ... m do
    for j ← 1 ... m do
        match ← (F[i − 1, j − 1] + s(H[i − 1], V[i − 1]), ↖)
        insert ← (F[i − 1, j] + gap-cost, ↑)
        delete ← (F[i, j − 1] + gap-cost, ←)
        F[i, j] ← min(match, insert, delete)                    // compared by score
    end
end

// compute traceback
bestScore ← F[m, n]
t, i, j ← ⟨⟩, m, n
while (i, j) ≠ (0, 0) do
    switch pointer (F[i, j]) do
        case ↑
            prepend(t, I)
            i ← i − 1
        case ←
            prepend(t, D)
            j ← j − 1
        case ↖
            prepend(t, M)
            i, j ← i − 1, j − 1
    endsw
end

return bestScore, t
```

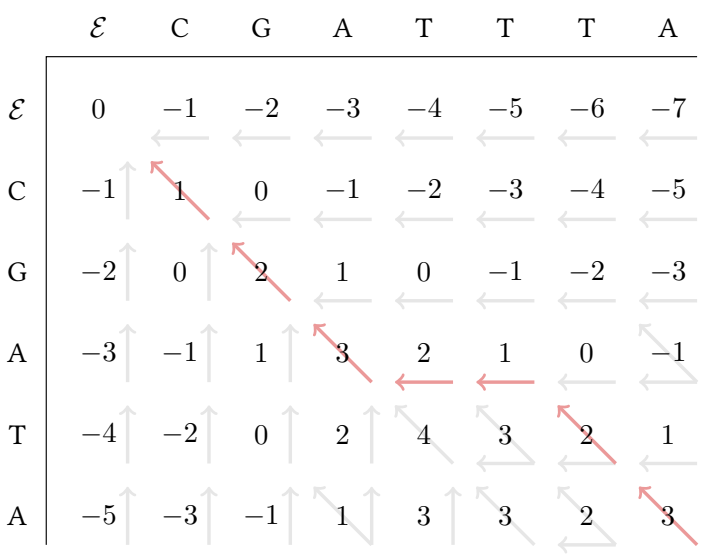


Figure 2.1: Example of a dynamic programming matrix for aligning two sequences. The match score is 1, mismatch and gaps are penalized with -1. The scores and *all* possible pointers are shown for each cell. The selected pointers are highlighted in red and give a chosen trace.

starts in cell (m, n) and follows the stored pointers to cell $(0, 0)$. A traversal of a pointer $\uparrow$ inserts a gap in the H, $\leftarrow$ inserts a gap in V, and $\nwarrow$ aligns two characters.

When gaps at the end of H are to be ignored in the score, the traceback starts from the highest-scoring matrix cell in row m and traceback starts at the highest-scoring matrix cell in column n for free gaps in V. A simple extension is to search both column n and row m for the highest-scoring cell if end gaps are free in both sequences.

Since the NEEDLEMAN-WUNSCH algorithm is of practical importance, there is a number of robust and efficient implementations, including the NEEDLE program in the EMBOSS (Rice *et al.*, 2000) package and functions in the SeqAn (Döring *et al.*, 2008) library as well as the SeqAn program PAIR_ALIGN.

2.2.2 Alignment Kinds

As mentioned in Section 2.1.5, allowing different begin and end gaps to be free leads to different kinds of alignments. In this section, I enumerate those of interest in this thesis and introduce a nomenclature to make the description more precise. Figure 2.2 shows examples for the different kinds of alignments.

Global Alignments. For global alignment, both begin and end gaps are not free which makes most sense if the sequences are expected to be similar or originate from a common ancestor. The

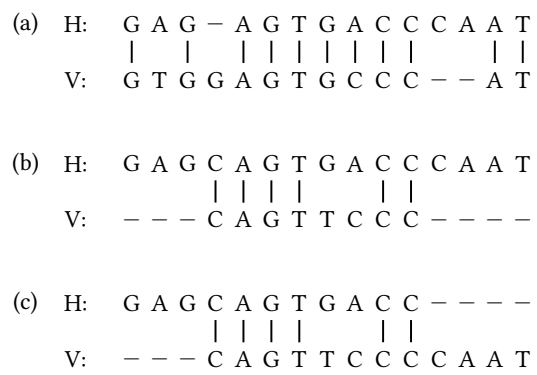


Figure 2.2: Examples of (a) a global alignment, (b) a semi-global alignment, and (c) an overlap alignment.

edit distance between two sequences is an example for a global alignment with match score of 0, and mismatch and gap scores of -1. Figure 2.2a shows an example for a global alignment.

Semi-global Alignment. Without loss of generaliy, in the case that sequence H is longer than sequence V and V is expected to be similar to only a part of H, it makes sense to align V to H in a *semi-global alignment*. Begin and end gaps are free in the row of V but not of H. Figure 2.2b shows an example of a semi-global alignment.

Overlap Alignments. Consider the case that the suffix of a sequence H is similar to the prefix of another sequence V. In this case, the sequences can be aligned with free end gaps in the row of H and free begin gaps in the row of V. Figure 2.2c shows an example of an overlap alignment. Here, H *hangs over to the left* (and aligns to gaps at the beginning of the row of V) and V *hangs over to the right* (and aligns to gaps at the end of the row of H).

Containment Alignments and Overlaps. When no sequence in a pairwise alignment hangs over the side of the other (at most one alignment row has leading and/or trailing gaps) then this is called *containment alignment*. When considering overlap alignments, this is called a *containment overlap*. Figure 2.2a and 2.2b show examples of containment alignments.

2.2.3 Linear Scanning Algorithms

Linear scanning is an algorithmic technique for efficiently analyzing points and intervals on the line. Such objects may be genomic coordinates or regions on the genome. The approach is similar to the *sweep-line* technique from computational geometry (Shamos and Hoey, 1976) where one dimension is swept in a linear fashion to simplify the problem at hand.

This technique is best explained with an example. Given a set of integer points and intervals, let the task be to count the highest number of overlapping objects along the line.

Without loss of generality, the intervals are given as half-open intervals $[a, b)$ and points are represented as $[a, a)$. For applying the technique, the intervals $[a, b)$ are converted into a sorted list of events, one for the start of the interval and one for the end of the interval. Each event is represented by a pair (p, e) where p is a point or genomic position and e is a boolean flag that indicates whether the event is an open or a close event. The pairs are sorted lexicographically (the flag values compare OPEN $<$ CLOSE) and then processed in their lexicographic order. The algorithm considers each event only once and only performs a local operation using this information. When designed carefully, such algorithms can run with low memory requirements and be very fast.

In Algorithm 2.2, I give a solution to the problem of computing the maximal depth (coverage) of intervals and points on the line. The input is a lexicographically sorted list of *events* containing a start and end event each for each point and interval. For example, this can be used to compute the maximal coverage given a list of read alignments. It is easy to extend this to writing out all alignments that are in regions below a certain coverage. Also, it is possible to extend the description above by allowing to attach arbitrary data to each interval and event. Further, it is not necessary that all events are known before starting processing, but they can be generated by another process as long as the linear scanning algorithm processes them in sorted order.

Algorithm 2.2: An algorithm for computing the maximal coverage of a multi-read alignment.

Input : *events* `// sorted list of events` (p, e)
Output : $M \leftarrow 0$ `// highest coverage`

```
// initialize counter
```
$m \leftarrow 0$
```
// process all events
```
for (p, e) *in events* **do**
  if $e =$ *OPEN* **then**
    $m \leftarrow m + 1$
  else
    $m \leftarrow m - 1$
  end
  $M = \max(M, m)$
end
return M

2.2.4 Suffix Trees

The *suffix tree* is a versatile data structure for representing the structure of a string. A detailed description and discussion of suffix trees can be found in (Gusfield, 1997, part II). In this section,

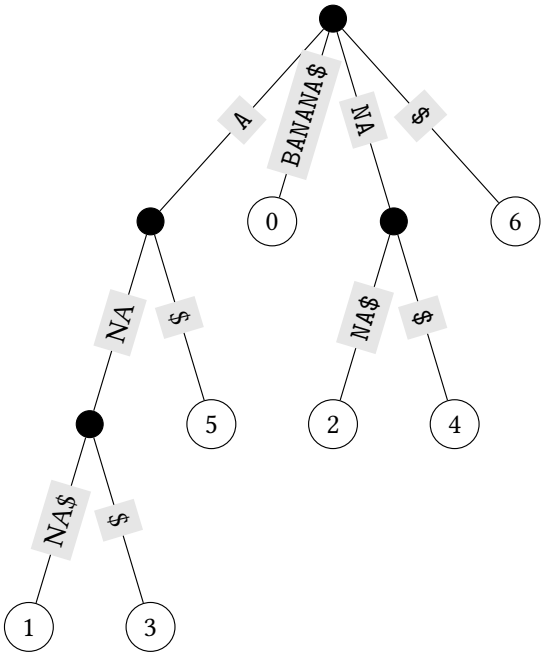


Figure 2.3: The suffix tree for the word BANANA.

I give a short description and some simple properties of suffix trees that apply to the contents of this thesis.

Definition 2.1 (suffix tree (Gusfield, 1997)). *A suffix tree for a string S of length ℓ is a rooted directed tree with the following properties.*

- *The tree has ℓ leaves numbered 0 to $\ell - 1$.*
- *Each internal node except the root has ≥ 2 children.*
- *Each edge is labeled with a nonempty substring of S.*
- *No two edges of a node can have edge labels beginning with the same character.*
- *For any leaf i, the concatenation of the edge labels of the path from the root to i exactly spells out the suffix of S that starts at position i.*

To ensure that a suffix tree exists, a non-alphabet character $ is appended to the text. Figure 2.3 shows a suffix tree for the word BANANA. Suffix trees have a wide number of applications, including exact and approximate string search. *Generalized suffix trees* are a variant of suffix trees that allow to store the sequence of a set of strings. Suffix trees can be represented by different data structures that improve on the time and memory characteristics of the pointer-based suffix tree data structure and algorithms building and using it.

Chapter 3

Data Preprocessing

The read error correction method Fiona was developed in collaboration with Marcel Schulz, David Weese, Victoria Dimitrova, Sijia Niu, Knut Reinert, and Hugues Richard. We presented the method at the European Conference on Computational Biology 2014 and published it in Bioinformatics (Schulz et al.*, 2014):*

M. H. Schulz, D. Weese, M. Holtgrewe, V. Dimitrova, S. Niu, K. Reinert, and H. Richard. Fiona: a parallel and automatic strategy for read error correction. *Bioinformatics*, 30, 17:i356-63.

My contribution was mainly to the experimental evaluation that was continuously performed during the development of Fiona. In this chapter, I describe the state of the art in data preprocessing and read error correction and then focus on the aspects of the experimental evaluation of read error correction.

Modern sequencing machines produce reads in a highly automatized fashion and vendors often market their technology as "push button" products. Nevertheless, sequencers still perform biochemical experiments and rely on visual or electronic components such as CMOS camera chips for measuring light intensity signals. Neither are these experiments perfect nor do the visual and electronic components produce noise-free signals.

The sequencing machines themselves incorporate means to reduce the noise in these signals. Despite this, the downstream analysis has to deal with errors in the sequencing data (*sequencing errors*). In Appendix B, I give a general overview of important sequencing technologies. In this chapter, I focus on the important sequencing technologies from 454/Roche (Margulies *et al.*, 2005), Illumina (Bentley *et al.*, 2008), and on IonTorrent/IonProton by Life Technologies (Rothberg *et al.*, 2011).

During sequencing, errors do not occur uniformly at random. Rather, the errors follow certain

platform-specific profiles and there exist biases depending on the used sequencing platform. The process of determining the base at a position in the read is called *base calling*. Generally, the probability of an error increases over time during the sequencing process. Thus, bases called earlier generally have a lower error probability than those called at a later stage. Furthermore, the chemistry and other technology used can cause biases in the data.

In the remainder of this section, I give a summary of known error profiles and biases to set the scene for read data quality control and read error correction described later in this chapter.

In Illumina data, the dominant kind of error are substitutions while indel errors occur at a lower probability. Dohm *et al.* (2008) and Quail *et al.* (2012) found that sequence-dependent errors and biases are also present in Illumina data. In 454 data, indels are the dominating kind of error and are caused by calling wrong lengths of long *homopolymers* (stretches of the same nucleotide). Gomez-Alvarez *et al.* (2009) found that 454 data exhibits biases for reads starting at the same or very similar positions lead to artificial replicates. Bragg *et al.* (2013) found that data generated by IonTorrent machines from Life Technologies also has indels as its dominating error kind. However, IonTorrent data was found to be more prone than 454 data to deleting existing homomonopolymers and inserting nucleotides that do not exist in the sequenced molecule.

Ross *et al.* (2013) compared Illumina, IonTorrent, and Pacific Biosciences data and found biases depending on GC content (the percentage of `G` and `C` in the genome sequence). They report that Pacific Biosciences data is least prone to such biases, followed by Illumina, then IonTorrent. The authors conclude that the bias depends on the used chemistry and also changes when the chemistry is updated.

During base calling, the HTS machines also compute estimates on the probability of a base call being correct and annotate each base with a *quality value*. Today, the *de facto* standard is using the FASTQ file format (Cock *et al.*, 2010) and annotating bases with phred qualities (Ewing and Green, 1998). Phred qualities are base-wise error probabilities encoded into a printable ASCII character, originally introduced in the Phred (Ewing and Green, 1998) base caller for capillary sequencing. This quality information can be used to ignore low-quality bases in variant calls or ignore reads that have too many low-quality bases.

Structure of This Chapter. The topic of this chapter are strategies for dealing with erroneous sequencing data in a preprocessing step. The first step in quality control usually is getting metrics summarizing the sequencing data quality. This is eventually followed by removing possibly erroneous reads or read ends based on read base qualities. In Section 3.1 I describe approaches for these two steps. It is desirable to detect errors from the raw read data and correct them instead of removing reads or read suffixes. In Section 3.2 I describe approaches for read error correction and list some important methods. This includes Fiona, the read correction method that I have contributed to. In Section 3.3 I discuss methods for the evaluation of read error correction approaches and in Section 3.4, I give the results of an experimental study of read error correction. In Section 3.5 I give a conclusion to this chapter.

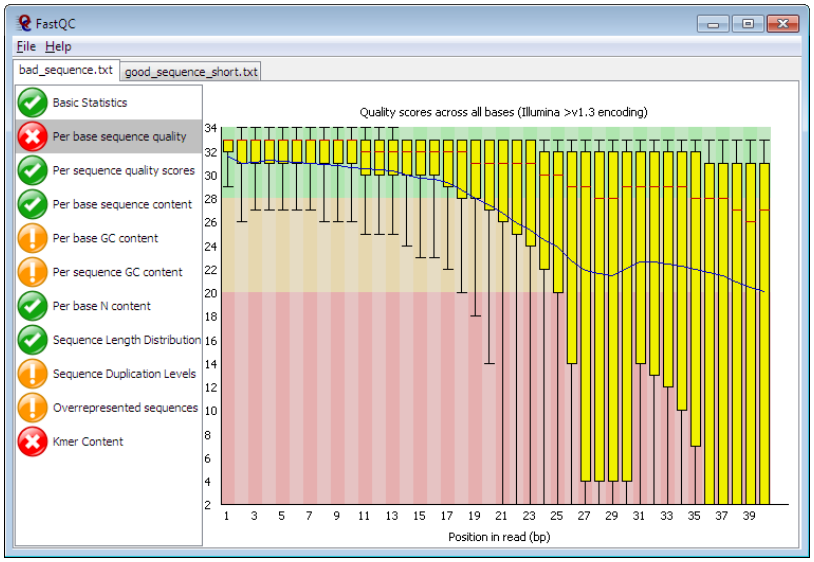


Figure 3.1: Bar chart of the per-base quality values for a 40 bp Illumina data set as visualized by FastQC.
Source: `http://www.bioinformatics.babraham.ac.uk/projects/fastqc/`

3.1 Quality Control Measures

Read Quality Metrics. In any environment, having a good measure on the quality of data is the requirement for any profound action to ensure or possibly to improve the quality thereof. FastQC (Andrews, 2010) is a popular tool for computing and visualizing summary metrics for HTS read data. The results produced by this program include statistical values on the distribution of base quality values for each position in the read, the GC content, content of `N` (non-called) bases, and k-mer content shared with known contaminant sequences. Figure 3.1 shows a screenshot of the FastQC report window.

FastQC is a good representative of quality control (QC) tools in that it provides a comprehensive list of features. Other packages for QC, such as the FastX Toolkit (Gordon and Hannon, 2010), provide similar functionality.

Read Trimming, Read Removal, and Contaminant Removal. As mentioned earlier, the probability for an error increases over the sequencing process and thus along the length of the reads. This is also visible in the base quality values as can be seen in Figure 3.1 for a set of Illumina reads.

Isolated errors are not as problematic as multiple consecutive errors (*stretches* of erroneous bases). In applications such as *de novo* sequencing or read mapping, single base miscalls can usually be resolved using the information from neighboring bases and redundancy in sequenced bases. A common approach (*e.g.*, as taken by Bolger *et al.* (2014)) is to detect regions of low quality bases by using sliding windows of the read and to consider the average quality value in the window.

Depending on the specific method, the position of the stretch in the read, and the overall read quality, the read is then *trimmed* (everything right of and including the low quality stretch is removed) or the whole read is removed completely from further consideration.

Another source of errors are contaminants, *i.e.*, sequence that is present in the read but not in the sample of interest. This kind of errors includes the sequence of adapters and barcodes. Adapters and barcodes are both short sequences at the ends of the reads. They are not part of the sequenced sample but are attached to the reads for technical reasons. Since the sequence of adapters and barcodes is known, a search for them can be performed in the reads and the corresponding sequence can be trimmed away. Popular tools for read trimming, read removal, and adapter removal include the FastX Toolkit (Gordon and Hannon, 2010), Trimmomatic (Bolger *et al.*, 2014), and Flexbar (Dodt *et al.*, 2012).

A further example for sample contamination is sequence from the Eppstein–Barr virus that is commonly used in cell cultures for technical reasons (Laurence *et al.*, 2014). The Univec database (Kitts *et al.*) contains the sequence of many known contaminants and can be used for removing reads containing contaminant sequence.

3.2 Read Error Correction Approaches

While the removal and trimming of reads based on base quality is an effective measure, it leaves undetected and isolated errors remaining in the reads. It would be beneficial to *correct* these errors after quality-based read quality control to remove remaining errors or even before so that less bases need to be removed in QC. In this section, I describe the approaches for *read error correction* and their underlying ideas.

Genomes are usually sequenced at a sufficiently high coverage, such that the read data contains redundant information. The fundamental idea behind read error correction is to use this redundancy for first identifying errors and then correcting them. If the true alignment of the reads to the genome sequence was known, one could easily identify errors as singular (or very low frequency) deviations from the other reads overlapping with the erroneous base. However, read error correction should also work when the reference genome is unavailable.

Note that this fundamental idea is limited to scenarios where low and high coverage can be used for discriminating between correct and erroneous sequence (Kelley *et al.*, 2010). In the case of RNA-seq, for example, the abundance of sequence in the sequencing run depends on the abundance of the sequence in the sample. The abundance of sequence in the sample, however, cannot be assumed to be uniformly distributed. In fact, in RNA-seq, one is often interested in measuring the difference in abundance levels. However, the fundamental idea can be applied to the important cases of exome and whole genome sequencing.

Read error correction is an important topic in sequence assembly. Thus, early assemblers for Sanger reads include error correction modules based on the two older approaches *spectral alignment* and *multiple sequence alignment*. For example, Euler (Pevzner *et al.*, 2001) provided error correction based on spectral alignment and Arachne (Batzoglou *et al.*, 2002) uses error correction

based on multiple sequence alignment. Because of the high data volumes, the first assemblers for short HTS reads relied on the computationally less expensive spectral alignment approach, *e.g.*, Euler-SR (Chaisson *et al.*, 2004). Recently, a new approach based on suffix trees has been introduced by Schröder *et al.* (2009) in the context of HTS read correction. Also recently, the multiple sequence alignment approach has been revived by Salmela and Schröder (2011) for HTS data.

I close this introduction with the note that there is no common formal definition of a Read-Error-Correction Problem. Rather pragmatically, the aim of read error correction is to obtain the actually sequenced nucleic sequence from the possibly erroneous reads. Within the implementation of read correction methods, however, the selection of a specific correction from a set of candidate correction is often formulated as an optimization problem. Examples are the spectral alignment problem described below and our formulation for optimal error correction selection in (Schulz *et al.*, 2014).

In the remainder of this section, I describe the basic idea behind the three approaches.

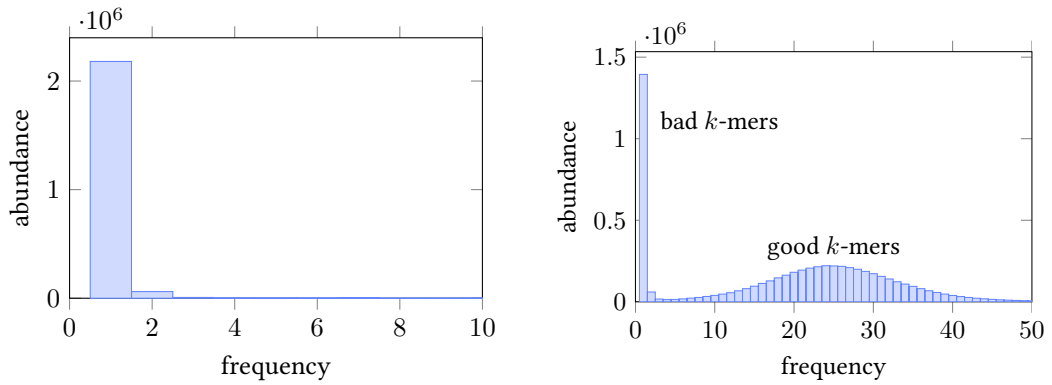
3.2.1 The Spectral Alignment Approach

A rough approximation of a window over the true read alignment against the reference is to consider the k-mers in the read data set. Assuming approximately uniform coverage of the donor, a k-mer without an error will in general not be unique since other reads will overlap and contribute the same k-mer. However, a k-mer that has an error will in general have a very low frequency or even be unique.

Figure 3.2 shows the k-mer spectra of the genome of *E. coli* and a 30x Illumina read data set of *E. coli* for $k = 15$. The spectrum shows the abundance of k-mers with a certain frequency. Since *E. coli* (Figure 3.2a) has a fairly simple genome with few repeats, most k-mers are unique. When sequencing without errors and bias, one would expect a coverage of 30x for all unique k-mers. In contrast, a typical k-mer spectrum for read data sets looks as shown in Figure 3.2b. Uneven coverage leads to the bell shape to the right (*good k-mers*). Sequencing errors are unique to one read or only occur in few reads and thus cause many k-mers with low frequency (*bad k-mers*).

The first step in spectral alignment–based methods is to find a threshold to separate the k-mers that have a low frequency (bad k-mers) and are tentatively erroneous from those that have a higher frequency and are tentatively correct (good k-mers). The second step is then to correct the erroneous k-mers. Note that the set of good k-mers can contain erroneous k-mers (and vice versa for the bad k-mers), but they tend to contain mostly correct k-mers (respectively erroneous k-mers for the set of bad k-mers).

This first step is often solved by using prior knowledge of a sequencing platform's error profile. For example, Quake (Kelley *et al.*, 2010) and Echo (Kao *et al.*, 2011) fit distributions to the k-mer spectrum of the input read data set. These distributions are selected based on knowledge about Illumina data which they focus on. Other methods such as Blue (Greenfield *et al.*, 2014) write out a histogram for the k-mer coverage similar to the one in Figure 3.2b and let the user select the threshold manually.



(a) genome k-mer spectrum **(b)** 30x read set k-mer spectrum

Figure 3.2: The k-mer coverage of (a) the *E. coli* genome and (b) a 30x coverage Illumina read data set for *E. coli*. In both cases, k was set to 15. For both (a) and (b), there exist k-mers with higher frequencies than 10/50 but the x axes were limited to interesting intervals since the abundance of the number k-mers was very small.

The spectral alignment approach obtained its name from the idea for performing the second step: by modeling the read error correction problem as a *spectral alignment problem*. The following definition follows the one by Pevzner *et al.* (2001).

Definition 3.1 (Spectrum, T-string). *Given an alphabet A and an integer k. A set T of k-mers over A is called a* spectrum. *A string S is a T-string if all of its k-mers belong to T.*

Definition 3.2 (Spectral Alignment Problem). *Given a string S and a spectrum T, find the minimal number of substitutions for S such that S becomes a T-string.*

In the setting of error correction, the set of good k-mers is used as T and the aim is to update all reads containing bad k-mers such that each of these reads becomes a T-string. A naïve solution is to enumerate the Hamming neighborhood for each bad k-mer until a good k-mer is found. However, most modern methods just use the spectral alignment problem as an inspiration and implement more sophisticated means than a simple Hamming neighborhood enumeration. For example, Quake uses base quality values to guide the correction by base substitution and Echo even includes platform specific substitution error biases in its internal statistical models.

Further, practical spectral alignment–based methods often include other heuristics for removing errors or include additional features that attempt to make the method more suitable for biological data. For example, Quake also trims reads using base qualities and Echo tries to recognize heterozygous SNVs from diploid genomes so that they are not treated as sequencing errors.

Note that it is crucial to select an appropriate value for k. Too small values lead to high frequencies just by chance and thus a high noise-to-signal ratio while too large values would make too many (or even all) k-mers unique. Also, errors in repeat regions are harder to detect. A substitution error in a k-mer with high frequency might change it to another k-mer with high frequency.

This is more probable for k-mers from repeat regions than from regions with relatively unique sequence (Kelley *et al.*, 2010).

3.2.2 The Substring Tree Approach

The most recent approach for read error correction is based on substring trees and was pioneered by Schröder *et al.* (2009). The tree is built from the reads and their reverse-complements. This approach is based on the following observation when considering the *generalized suffix tree* (see (Gusfield, 1997, p. 116) and Section 2.2.4) of HTS read data sets.

Let each edge of the tree be weighted with the number of leaves below it. Below a certain level, the tree becomes sparser and the methods following the substring tree approach consider this sparser region. In the spectral alignment approach, substitution errors lead to k-mers with a low frequency. In the substring tree approach, there is a similar correspondence: substitution errors lead to sparse branches while having siblings with a high density. This can be used to detect substitution errors and there are similar motifs in the tree for indel errors in the reads. Figure 3.3 shows an example. Similarly to the spectral alignment approach, corrections in unique regions show a stronger signal than those in repeat regions.

After identifying a branch as potentially erroneous, methods following the substring tree approach perform a local search in the branches of the tree around the error. In the case of Shrec (Schröder *et al.*, 2009), the read sequence and the neighboring nodes are considered to select a correction. This correction is then applied to the read sequence and the tree is updated accordingly.

Shrec (Schröder *et al.*, 2009) introduced the substring tree approach for correcting substitution errors on Illumina data. Salmela (2010) published an improved version of Shrec, called Hybrid-Shrec, that works on reads of different length, can correct ABI SOLiD reads, and can also correct indel errors. This makes the method suitable for 454 and IonTorrent data as well.

Both Shrec and Hybrid-Shrec use the suffix trie data structure. To lower the memory requirement, Hitec (Ilie *et al.*, 2011) improves on Shrec by using the *suffix array* data structure (Gusfield, 1997, p. 149) and also determines more parameters automatically.

Fiona (Schulz *et al.*, 2014) is another recent method for read error correction based on substring trees. We introduced a criterion for optimal error correction and a statistical framework for detecting errors. Further, the tool features auto-tuning of parameters which make it easier to use, an efficient parallel tree traversal to save running time, and uses partial suffix arrays for saving memory. Fiona is applicable to both Illumina and indel-prone data such as generated by 454 and IonTorrent technology.

3.2.3 The Multiple Sequence Alignment Approach

The multiple sequence alignment approach was originally conceived for Sanger reads. The underlying idea is to compute multiple sequence alignments for similar reads and use them for

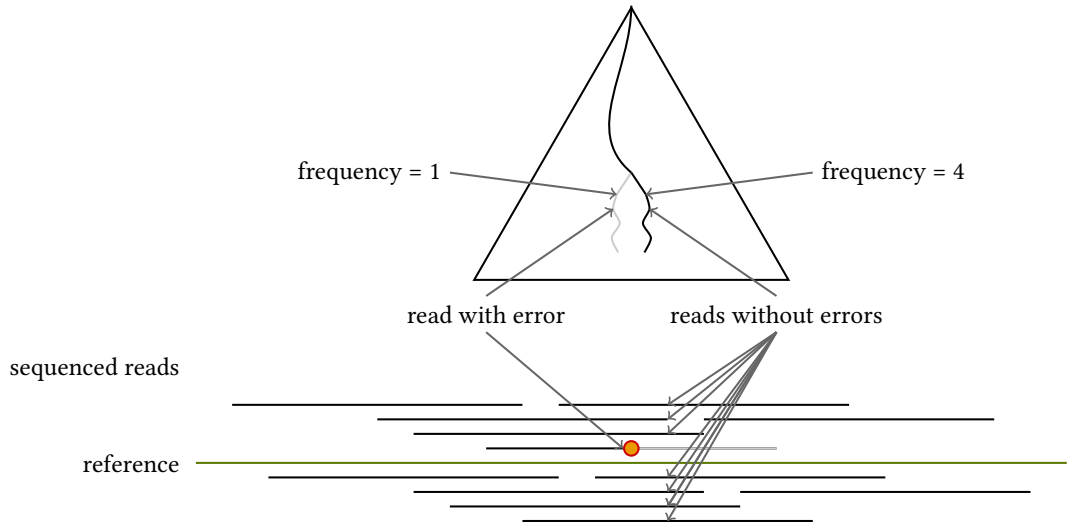


Figure 3.3: Example of a substitution error and its effect on the substring tree (shown above) and the actual read alignment against the reference (shown below, reference in green). The substring tree is shown schematically as a triangle, the paths through the tree are shown as lines.
Adapted from (Schröder *et al.*, 2009, Fig. 3).

detecting and correcting errors. Recently, it has been scaled to the data volumes of HTS in the Coral method (Salmela and Schröder, 2011).

Coral uses k-mers for rapidly identifying similar reads. The reads are then aligned to each other and an approximate MSA is obtained by iteratively building the consensus of the reads selected as similar. Errors are then detected by looking at deviations in alignment columns and also at the base quality values. The detected errors are corrected using a column's consensus character. Coral supports the correction of both substitution and indel errors and its authors claim that the method is very flexible and can be applied to future sequencing platforms by "parameter choice on the command line" (Salmela and Schröder, 2011).

3.3 Read Error Correction Evaluation

Originally, read correction was considered as a preprocessing step for whole genome assemblers. Similar to the setting of assembly, the evaluation of read error correction is complicated by not having a perfect ground truth for real data. Each publication describing a read error correction method contains an experimental evaluation, often using different metrics for the evaluation. In this section, I give an overview of the evaluation methods found in the literature and an assessment of the metrics themselves.

3.3.1 Evaluation Approaches

Simulated vs. Real Data. Earlier studies used simulated data to evaluate the performance of the read error correction (Kelley *et al.*, 2010; Salmela, 2010; Schröder *et al.*, 2009). The advantage of simulated data is that the ground truth is known. As in many areas of bioinformatics, simulated data can be used to complement the evaluation of error correction tools with real data but it cannot replace it. The error profiles and biases of HTS platforms are complex, as also described by Dohm *et al.* (2008) and Quail *et al.* (2012). Thus, it is disputable how much the performance of a read error correction method on simulated data correlates with its performance on real world data.

Downstream Analysis Result Evaluation. One way to circumvent this issue is to subject the corrected and uncorrected data set to downstream analysis where the result comparison might be more straightforward.

The studies by Schröder *et al.* (2009), Kelley *et al.* (2010), Salmela (2010), and Salmela and Schröder (2011), pass the corrected and uncorrected read data set to a *de novo* assembler. The number of errors in a read data set has an influence on the resource requirements of assemblers and their quality (Kelley *et al.*, 2010). To measure the influence of the correction on these values, the authors measure the running time and memory usage of the assembler on the data sets before and after read correction. Further, they consider common assembly evaluation metrics such as the N50 score and the number of contigs. The N50 score for an assembly that consists of a set of contigs is similar to a median and computed as follows. Sort the contigs descendingly by length and pick the longest contig C such that the sum of the length of all contigs longer than C is at least half the sum of all contigs in the assembly.

Another application of read error correction is before the read mapping step in a variant calling pipeline. The studies by Salmela (2010) and Kelley *et al.* (2010) subject the read set before and after correction to read mapping and (in the second case) the mapped reads to SNV calling. The authors could show that the read mapper could align more reads and that SNV calling improved after error correction.

The evaluation of assemblers is a non-trivial task itself (Salzberg *et al.*, 2012). Read mapping is a simpler problem than *de novo* assembly but counting alignable reads itself is not an ideal metric (Holtgrewe *et al.*, 2011). Thus, the evaluation depends on the quality of the assembly, read mapping, and SNV calling software. Each of these methods also has means implemented to deal with sequencing errors itself that might cause further biases in the evaluation (see also (Ilie and Molnar, 2013)).

Kelley *et al.* (2010) use two well-known bacterial genomes and reads from one of the bacteria for the evaluation using SNV calling. While such an evaluation is very useful and interesting, it is limited to the cases where high-quality genomes are available for two closely related organisms.

Obtaining a Gold Standard by Read Mapping. In the evaluation of their error correction method Reptile, Yang *et al.* (2010) introduced the *gain* metric that is explained below in Section 3.3.2. This metric requires some ground truth and obtains an approximation thereof by first

aligning the reads from the evaluation to the genome of the same organism or a closely related one. The reference sequence at the read alignment location is then used as the ground truth for each read.

On the one hand, this has the disadvantage that the sequenced individual has at least some small variation with the reference. Thus, the read mapping locations might not be completely correct and variation will be called as errors remaining in the data in the evaluation. Also, reads that could not be aligned before correction are subsequently ignored in the evaluation.

On the other hand, a good reference is known for many important organisms. Thus, ground truth (or *gold standard*) is easy to obtain. At the time of writing, the gain metric has been established as the *de facto* standard metric for the evaluation in read error correction and used in studies by Yang *et al.* (2013, 2010), Salmela and Schröder (2011), Schulz *et al.* (2014), and Greenfield *et al.* (2014).

Exact Search of Corrected Read in Reference. Ilie and Molnar (2013) introduced an alternative approach. After correction, the reads are searched for in the used reference genome without allowing any errors. The authors then report the percentage of reads that could be found in this search.

While this approach circumvents possible ambiguity problems by read mapping before correction, it also has its limitations. One limitation is that the reference is taken as an absolute gold standard and no genomic variation is allowed. This requires a reference genome that is very close to the sequenced organism. Secondly, only corrections that lead to reads identical to the gold standard are counted.

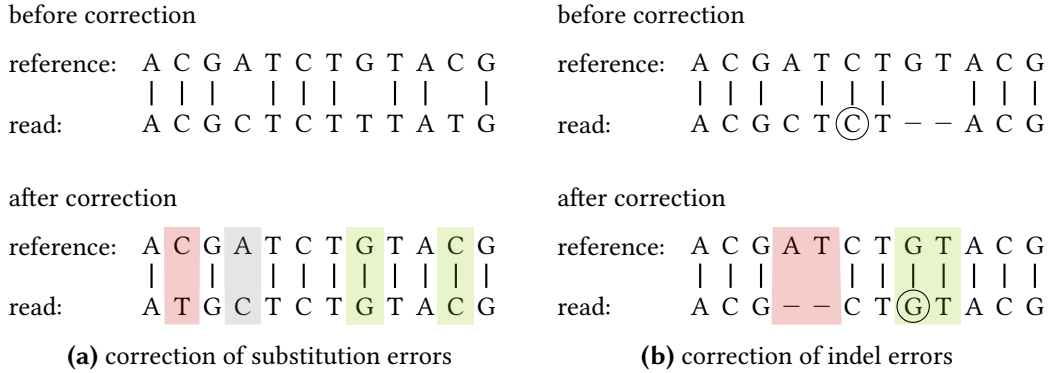
Because of these limitations, I do not consider this evaluation approach further below. Instead, I build the gold standard from the read mapping results as described above.

3.3.2 Evaluation Metrics

In this section, I will first introduce the common metrics from the literature in the case that only substitution errors are corrected. Then, I extend the definitions to edit distance and discuss limitations.

Note that the study by Schröder *et al.* (2009) counts correct and incorrect reads before and after correction. While a possibly useful measure for short 36 bp Illumina reads that rarely have more than two errors, it is of less use for present longer $\geq$ 100 bp Illumina reads. Thus, I will not consider it further in this exposition.

Substitution Error Metrics. In read error correction, the ground truth for a read is the true sequence of nucleotides. This ground truth is used to compute error metrics by computing the distance between the read and its ground truth. A gold standard for the ground truth can be obtained by aligning reads against the reference of the organism as explained above. When only



(a) correction of substitution errors **(b)** correction of indel errors

Figure 3.4: Example of read alignments before and after correction, (a) only in the presence of substitution errors and (b) also in the presence of indel errors. In (b), the corrected base is circled.

considering substitution errors, Hamming distance is used to compute the errors of the read before and after correction.

Figure 3.4a shows an example for this. The columns of the alignment after correction have been assigned a background color based on one of four cases: (1) white for *unchanged* bases that are correct before and after correction, (2) gray for *uncorrected* bases that are incorrect before and after correction, (3) green for *corrected* bases that were erroneous before correction but have been corrected afterwards, and (4) red for *introduced errors* on bases that were correct before correction but are erroneous afterwards.

Read error correction methods can be interpreted as classifiers. A changed base is interpreted as being classified as erroneous, an unchanged base as being classified as correct. Using the gold standard, one then counts (1) unchanged bases as *true negatives (TN)*, (2) uncorrected bases as *false negatives (FN)*, (3) corrected bases as *true positives (TP)*, and (4) introduced errors as *false positives (FP)*. From this, the common statistical metrics *sensitivity*, *specificity*, and *false discovery rate (FDR)* are derived:

$$\begin{aligned} \text{sensitivity} &= \text{TP}/(\text{TP}+\text{FN}), \\ \text{specificity} &= \text{TN}/(\text{FP}+\text{TN}), \\ \text{FDR} &= \text{FP}/(\text{TP}+\text{FP}). \end{aligned}$$

Also based on these values, Yang *et al.* (2010) introduced the *gain* metric as:

$$\text{gain} = (\text{TP}-\text{FP})/(\text{TP}+\text{FN}).$$

The counts for TP, FP, and FN are determined over all reads. That is, the gain is the net number of errors removed, normalized by the sum of the errors before correction.

It can first be observed that $(\text{TP} - \text{FP})$ is equal to the sum of the differences of the Hamming distance between all reads and their gold standard after and before correction. Further, the sum $(\text{TP} + \text{FN})$ can be computed as the sum of the Hamming distance between all reads and their gold standard after correction. Thus, the gain can be computed from the Hamming distance before and after correction alone.

Two further useful metrics are the *base-wise error rate* before and after correction. These metrics can be computed by dividing the sum of the Hamming distances between reads and their gold standard by the total number of bases in the read set. It can be secondly observed that the (base-wise) error rate before correction is $(\text{TP} + \text{FN})/n$ where n is the total number of read bases and the error rate after correction is $(\text{FN} + \text{FP})/n$. Since the difference in error rate before and after correction is $(\text{TP} - \text{FP})/n$, the gain metric is a summary of the error rates before and after correction.

Error Metrics in the Presence of Indels. The first observation also holds for edit distance, *i.e.*, with indel errors present. That is, the gain can be computed from the difference of the edit distance between reads before and after correction to a gold standard. However, the computation of the other metrics is not straightforward when indels are present.

Yang *et al.* (2010) originally used the gain metric in the context of Hamming distance only and extended it to edit distance in (Yang *et al.*, 2013). Their approach for this is as follows. They consider the alignment of the read before correction against the reference. For each erroneous base, they collect a quadruple (p, o, c, w) where p is the position in the genome, o is an offset in gaps in the reference, c is the correct base (the reference base, can be "$-$" for gaps in the reference), and w is the wrong base in the read. Algorithm 3.1 computes these quadruple sets.

The sets B and A contain the quadruples for the errors before and after correction. The true positive count is then computed as $|B \setminus A|$, the false positive count as $|A \setminus B|$, and the number of false negatives as $|A \cap B|$. The number of true negatives is the number of reference positions before correction that are neither in A nor in B.

This approach works for the majority of cases. In manual inspection of random alignments before and after correction, I observed only very few cases where the counting in Algorithm 3.1 yields incorrect results and correct results otherwise. I found that these cases stem from the following shortcomings of the counting approach.

First, the error correction method might choose to correct a substitution error towards the ends of the read by an indel operation. In this case, the true alignment would need to be extended to the left or to the right in the reference to get an optimal semi-global alignment against the gold standard mapping location. This problem could be alleviated by aligning the read semi-globally within an environment of the original mapping location. This in turn would introduce a problem with the counting described in Algorithm 3.1 in the case that the alignment shifts to either side, leading to incorrect counts.

Second, even when ignoring such problems of changing begin and end positions of the best alignment in the reference using edit distance there are ambiguities that can lead to errors. Consider Figure 3.4b. The edit distance before correction is 3 and it is 2 afterwards, yielding a gain

of 1. Also, only one correction is made (the base is circled before and after correction) but Algorithm 3.1 counts two corrected bases and two introduced errors. This is caused by ambiguity of alignments with gaps as explained in Section 2.2.1. In this case, it is caused by the alignment algorithm placing gaps in the second sequence at the rightmost position in case of ambiguities. For this read and reference, the problem would not appear if the gaps were placed at the leftmost position. It is easy to verify that reversing both read and reference would lead to a corresponding case for an alignment algorithm that places gaps at their leftmost position.

The base-wise error rate can be used in the presence of indels without problems. The computation remains the same, but instead of Hamming distance, the edit distance metric is used.

Assessment of Metrics. Despite its shortcomings, the gain metric is probably the most useful metric in my judgement since it is simple and thus robust to compute with real data. A number of high-quality reference sequences for model organisms of various genome length is available together with real read sets in databases such as the European Nucleotide Archive (ENA) (Leinonen *et al.*, 2011a). Thus, it is possible to perform a comprehensive benchmark on realistic data using a robust metric. The base-wise error rates before and after correction give a good impression of the quality of the raw data and the improvement after correction.

I also find the metrics based on TP, FP, etc. to be useful for estimating whether a method corrects too few errors *(undercorrection)* or introduces too many errors *(overcorrection)*. As I outlined above, there are some issues in pathological cases but overall, these metrics should give another perspective on the performance of a read correction method. The sensitivity describes the percentage of corrected errors of all errors and thus is a measure of how well errors are recognized.

Algorithm 3.1: Error counting for alignments of a read R against its gold standard G.

Input : R, G // alignment row for read and gold standard
Output : $E \leftarrow \emptyset$ // error quadruple set

$p \leftarrow 0$ // position in reference
$o \leftarrow 0$ // offset counter
for $i \leftarrow 0 \ldots |R| - 1$ **do**
  if $R[i] \neq G[i]$ **then** // include error tuple in result
    $E \leftarrow E \cup (p, o, R[i], G[i])$
  end
  if $R[i] =$ "-" $\wedge\, G[i] \neq$ "-" **then**
    $o \leftarrow o + 1$
  else // assume that there is no all-gap column in the alignment
    $p \leftarrow p + 1$
    $o \leftarrow 0$
  end
end
return E

The specificity describes the percentage of non-erroneous bases that are left uncorrected of all non-erroneous bases. Thus, it is a measure for how well non-erroneous bases are not falsely corrected. Since the number of non-erroneous bases is much larger than the number of erroneous bases, the sensitivity will be very close to 100% in most cases. Two methods will thus only have a small absolute difference specificity. For read error correction, I thus found the FDR to be a better substitute for sensitivity. The FDR describes the percentage of non-erroneuos bases in all modified bases. Thus, it is the percentage of bad corrections.

In recent studies, the metrics based on the downstream analysis are of lower importance. While giving a more application-driven perspective, they rely on one or more further downstream analysis steps that make them more indirect. I see their main purpose in showing that read error correction is important since it improves downstream analysis and the earlier studies cited above have already established this.

The Program Compute-Gain

Included in the distribution of Fiona (Schulz *et al.*, 2014) is my program Compute-Gain that allows to perform the metric computation. As input, it uses a reference FASTA file, the aligned reads before correction as a SAM/BAM (Li *et al.*, 2009a) file, and the corrected reads as a FASTA or FASTQ file. Depending on the parameters, the program performs an evaluation employing multi-core parallelism. Compute-Gain supports both Hamming and edit distance using banded DP alignment.

We implemented the program in C++ using SeqAn (Döring *et al.*, 2008) for aligning reads and OpenMP (Dagum and Menon, 1998) for the parallelization. By using common file format, it facilitates experimental studies on read error correction such as the one we performed for (Schulz *et al.*, 2014) and the one I present in Section 3.4.

3.4 Read Error Correction Results

In this section, I describe the results of an experimental study regarding read correction. This study extends the one I performed in (Schulz *et al.*, 2014) together with my coauthors. Appendix F contains additional tables supporting this section.

Evaluation Methods. For the study, I obtained various read data sets for different organisms. I then mapped the Illumina reads with BWA (Li and Durbin, 2009) and BWA-SW (Li and Durbin, 2010) to the reference genome to obtain a gold standard for gain computation as described in Section 3.3. I aligned the Illumina reads semi-globally using BWA and then considered semi-global alignments. For 454 and IonTorrent data, I obtained local alignments with BWA-SW.

I used Compute-Gain for computing the gain and FDR for each corrected read set. The error rate is not shown in these results since the difference of the base-wise error rate before and after correction is a linear function of the gain. I used a match filtration schema similar to the one

in (Yang *et al.*, 2013): I ignore reads where fewer than 30 bases are aligned by the mapper, as well as alignments with more than 10 edit distance errors or with an error rate of more than 20%.

I ran most the experiments presented here on a server with an 8-core Intel Xeon X5550 2.67 Ghz processor, having 72 GB of RAM and on Debian Linux 6.0.6. For the largest data set of human data, I used a machine with 16 physical and 32 virtual cores having 370 GB of RAM.

Used Data Sets. Table F.1 (p. 208) shows details on the genomes used for the evaluation and Table F.2 (p. 208) shows details of the read sets used in the evaluation. In total, I used 10 references, their length ranging from 4 Mbp to 2.8 Gbp, including those of bacteria, insects, and human: *B. pertussis* (abbreviated as *B. pert.*), *C. elegans* (*C. el.*), *D. melanogaster* (*D. mel.*), *E. coli* (*E. col.*), *H. sapiens* (*H. .sap.*), *P. falciparum* (*P.falc.*), *P. syrigae* (*P. syr.*), *S. aureus* (*S. aur.*), and *S. cerevisae* (*S. cer.*). I used a total of 19 read sets, including short and long Illumina, 454, and IonTorrent data, their coverages range from 5x to 490x.

Evaluated Read Error Correction Methods. Most tools for read error correction focus on Illumina data since the prevalent substitution errors are easier to correct. Thus, I chose a subset of applicable methods for the evaluation of these data sets. Quake (Kelley *et al.*, 2010) is a popular method based on spectral alignment that is fast and has low resource requirements. Echo (Kao *et al.*, 2011) is a recent method also using spectral alignment that includes a sophisticated error model for Illumina data. Allpaths-LG (Gnerre *et al.*, 2011) is a popular whole genome assembler that allows the assembly of mammalian genomes with moderate resource consumption. It includes an error correction module that is efficient and yields good results. The module is also available as a standalone program and I used it as a representative for a modern read error correction module from an assembler program. I evaluated Coral (Salmela and Schröder, 2011) as a representantive of the multiple alignment approach. I used Hitec (Ilie *et al.*, 2011) in the evaluation as a representantive of the substring tree approach, just as I used Fiona (Schulz *et al.*, 2014) here in its mode for Illumina data Fiona-H. Other recent tools are Blue (Greenfield *et al.*, 2014) and Racer (Ilie and Molnar, 2013) that promise fast and efficient error correction.

For indel prone 454 and IonTorrent data, fewer tools are available. I evaluated Fiona in its mode for 454 and IonTorrent data. I evaluated HybridShrec (Salmela, 2010) as another representantive of the substring tree approach. Allpaths-LG, Blue, and Coral are also applicable to indel prone 454 and IonTorrent data.

Parameter Choice. One of the main problems when applying read error correction tools is the choice of parameters. I tried to run the evaluated tools with default parameters or to select parameters guided by the tool authors, *i.e.*, as a practitioner would choose them. This yielded good results for most tools, with the exception of Coral and HybridShrec. Thus, I performed a search for good parameters for these tools. Coral yields good results with default parameters on Illumina so only the results with default settings for this kind of data are reported. For indel data, I tried several parameters and report the best results as Coral*. For HybridShrec, the default parameters did not yield good results, so I report the results tha I obtained when varying the parameters as HybridShrec*. Further, I tried to set the *from* and *to level* in HybridShrec as

they were chosen automatically by Fiona and report the results as HybridShrec^F. Tables F.5 (p. 211) and F.6 (p. 212) show the performance of the Coral and HybridShrec parameterizations used for the selection of best parameters. For Blue, I used the instructions in the manual. I chose the value for the threshold to be at the first local minimum in the histogram generated by the Tessel subprogram. I describe the exact program version and program calls in Appendix D.

3.4.1 Results on Illumina Data

Correction Performance. Table 3.1 shows the gain and FDR value of the corrected reads on Illumina data for Allpaths-LG, Blue, Coral, Echo, Fiona, Hitec, Quake, and Racer. I will first consider the results in terms of gain value and then consider the FDR.

Overall, Allpaths-LG, Blue, Fiona, and Quake achieve the best results and Fiona has a slight edge on the three other methods. The results of Allpaths-LG are most competitive when the coverage is $\approx$ 30x. The result quality of Quake varies somewhat. Both Blue and Fiona are competitive on all data sets. Coral has problems on the *C. elegans* data set and yields a negative result. Echo is competitive on the smaller data sets but has problems with the larger ones and crashes or runs longer than 24 hours. Hitec also has problems with larger data sets and requires more than 75 GB of memory, as does Echo. Racer yields good results for the *E. coli* data sets but creates inferior (even negative) gains on five of seven data sets.

The methods achieve the best results on the *E. coli* data sets with gain values of up to 99.29%. There are two possible explanations for this. First, the genome of *E. coli* is the shortest and simplest considered in these experiments and the lack of many repeats makes correction relatively easy. Second, the reads used for the evaluation were sampled from the same strain of *E. coli* which leads to a relatively low variance between the reference genome and the genome of the sample organisms.

In terms of FDR, Allpaths-LG, and Quake are the best methods. Blue and Echo also achieve low FDR values, followed by Fiona. Hitec and Racer are worst in terms of FDR rates. Again, the tools achieve the lowest FDR rates on the *E. coli* data sets which can be explained with the

	ALG		Blue		Coral		Echo		Fiona		Hitec		Quake		Racer	
data set	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR
C. el. (30x)	**28.4**	**4.4**	25.9	8.9	−3.7	52.5	−[1]		25.2	13.9	−[2]		17.0	6.0	−88.6	80.4
D. mel. (5x)	32.8	3.7	45.1	6.3	47.0	9.7	42.2	4.4	**50.3**	5.4	23.9	29.1	47.0	**1.5**	−1.3	50.8
D. mel. (30x)	32.1	4.2	31.5	4.0	23.0	23.7	−[1]		31.4	7.8	−[2]		**68.0**	**1.3**	18.7	31.6
E. col. (30x)	**99.3**	**0.1**	97.7	0.8	91.4	7.2	91.1	1.9	97.8	1.1	93.2	1.3	95.6	**0.1**	94.5	2.9
E. col. (490x)	98.2	**0.1**	92.2	0.5	96.0	1.3	−[2]		**98.6**	0.6	93.8	1.0	88.1	**0.1**	96.1	2.0
P. syr. (21x)	74.4	**1.2**	86.5	1.8	79.6	5.7	**91.1**	**1.2**	**91.0**	4.6	77.4	15.3	79.7	1.8	13.1	46.3
S. cer. (22x)	56.0	3.4	**66.7**	3.6	44.8	20.9	32.3	7.5	58.6	19.9	−23.0	56.9	52.1	**3.0**	−201.6	79.0

Table 3.1: Read error correction results on Illumina data. Both gain and FDR are shown in percent. The best metric values ($\pm$ 0.1% points) for each data set are highlighted in bold. Allpaths-LG is abbreviated as ALG.
[1] The program ran too long. [2] The program crashed.

	ALG		Blue		Coral		Coral*		Fiona		HybridShrec		HybridShrec^F	
data set	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR	gain	FDR
D. mel. (18x)	8.9	5.8	45.8	10.6	38.8	12.4	53.3	17.7	**65.2**	**4.6**	−279.5	87.2	38.2	7.5
E. col. (13x)	54.5	3.0	77.7	2.6	46.9	17.5	51.9	24.3	**90.7**	**2.4**	41.8	22.4	40.3	6.0
S. aur. (34x)	23.8	11.0	59.0	10.6	0.0	100.0	**74.9**	16.2	70.3	**4.8**	9.6	43.3	20.5	7.0
S. cer. (16x)	18.4	6.9	30.8	**5.8**	0.6	15.6	3.0	42.2	**36.1**	9.1	5.5	44.7	23.1	7.3
B. pert. (85x)	40.1	**3.1**	68.3	4.4	0.0	12.8	30.6	24.4	**73.0**	6.0	−235.5	89.8	−9.7	70.1
E. col. (8x)	30.7	**3.2**	67.1	6.2	49.4	17.1	63.8	18.3	**76.8**	8.1	40.1	25.2	34.3	8.6
E. col. (153x)	32.0	18.4	67.2	4.1	61.1	25.4	**74.7**	19.7	74.2	**3.4**	−24.1	86.1	0.0	100.0
E. col. (160x)	16.0	11.2	69.9	5.0	59.7	16.0	73.7	20.1	**81.4**	**4.0**	−18.9	84.7	0.0	100.0
H. sap. (11x)[1]	11.5	18.0	29.7	29.2	–[2]		–[2]		**63.0**	**9.2**	–[2]		–[2]	
P. falc. (12x)	21.4	**4.6**	54.0	17.0	0.0	15.7	24.9	31.7	**54.2**	18.2	−51.3	72.5	8.5	30.7
S. aur. (109x)	14.9	7.0	55.5	9.2	0.2	17.8	56.9	18.6	**65.8**	**4.0**	−13.4	83.3	0.3	44.2

Table 3.2: Read error correction results on 454 (top) and IonTorrent (bottom) data. Both gain and FDR are shown in percent. The best metric values for each data set are highlighted in bold. This table shows the same metrics as Table 3.1.
[1] The programs were run on machine with 16 physical and 32 virtual cores and 370 GB of RAM. [2] Out of memory.

same arguments as the good gain results on such data sets.

Running Time and Memory Consumption. Table F.3 (p. 209) shows the running time and memory consumption during the experiments from Table 3.1. Remarkably, there is no single fastest method. Blue, Quake, and Racer each achieve the lowest running time for two or three data sets. Notably, Racer needs more than six times the running time for the *C. elegans* data set than the next slowest method on this data set. Allpaths-LG achieves fairly low running time, without being fastest anywhere. Fiona is faster than Coral on more data sets than it is slower on. Both Fiona and Coral are faster than Hitec and Echo.

Blue and Quake have the lowest memory requirements, followed by Racer and then by Allpaths-LG. Afterwards, Fiona and Hitec are tied, the Coral variants require the most memory. The memory requirements of Echo and Hitec are moderate for the smaller data sets but the programs fail to execute on larger data sets because they try to allocate more memory than available.

3.4.2 454 and IonTorrent data

Correction Performance. Table 3.2 shows the results of read error correction on 454 and IonTorrent data. The results are first considered in terms of gain and then in terms of FDR.

Overall, Fiona has the best correction results, only beaten twice by Coral*. Coral* always outperforms Coral. Coral* and Blue are the second best methods, each beating the other on some data sets. Allpaths-LG beats Coral* in two cases, but Blue is consistently better than Allpaths-LG, the third best tool. HybridShrec* only yields good gains for low-coverage data sets of *E. coli*. Generally, HybridShrec^F yields better results than HybridShrec*. Notably,

Allpaths-LG, Blue, and Fiona are the only methods able to process the human data set within 370 GB of RAM, where Fiona shows the best results.

Fiona is the tool with the lowest FDR in most cases. However, on *P. falciparum*, the FDR is four times higher than the lowest FDR for this data set. Allpaths-LG is the next best method in terms of FDR, followed by Blue and then by Coral*. Remarkably, Coral* is always worse in terms of FDR than Coral. The HybridShrec variants have very high FDR values on some data sets and competitive ones on other data sets.

Running Time and Memory Consumption. Table F.4 (p. 210) shows the running time and memory consumption of the experiments from Table 3.2. Overall, Allpaths-LG and Blue have the lowest memory requirements and are fastest. The next best method in terms of memory usage is Fiona, followed by the Coral and HybridShrec variants. Remarkably, Fiona is faster than Blue on the largest *H. sapiens* data sets.

Discussion

In the experiments described in this section, I used a large number of reference organisms and read data sets with diverse properties. For the interpretation, I consider the Illumina and the indel data sets separately. Many methods focusing on substitution errors are not applicable to indel data sets or the results are different. It is to be noted that a gain of 100% is generally not possible because of variations between the donor and the reference genome.

For Illumina data, I found that Allpaths-LG and Fiona are the best tools in terms of gain. The spectral alignment based methods Blue, Quake, and Racer have the lowest running time and memory requirements. Allpaths-LG has moderate to low running times and memory requirements. Blue has good results in terms of gain. Coral yields relatively good results at a higher running time than the two fastest methods and competes with Fiona for being the third fastest tool. Echo, the tool with the most advanced model of Illumina errors achieves good but not the best results. Its running time prohibits its use with larger data sets. Hitec also achieves good results but has problems with three larger data sets. Racer is a fast tool (except on the largest Illumina data set) but does not achieve particularly competitive results in terms of gain. The FDR values are lowest for Allpaths-LG and Quake, followed by Blue and Fiona.

When low running time and memory usage are most important for a user, Allpaths-LG and Quake are good choices for an Illumina error correction tool. Both achieve good running times and do not need tuning of parameters. Blue is also a good choice in terms of running time and quality but requires manual interaction for the choice of the threshold value. In a setting where more memory is available and running time is not a first priority, Fiona can be good alternative.

The tools Allpaths-LG, Blue, and Quake have a consistently low FDR on Illumina data. Thus, they can be seen as being very conservative in their corrections. On the other hand, Fiona achieves very good results with slightly higher FDR values, thus it is a more aggressive correction tool.

For 454 and IonTorrent data, I found that Fiona is the best choice, yielding consistently good or best results in terms of gain with consistently low FDR values. When running time is a first priority, I recommend Blue for its error correction performance but it requires manual interaction in setting the value of the parameter k. Another recommendation is Allpaths-LG for correcting some errors in short time and with lower memory usage, but it is not competitive with the best error correction tools for indel data. Coral does also achieve good results but high memory usage and running time together with the problem of parameter choice make it hard to use in practice and on large data sets.

It remains as future work to improve the specificity of Fiona on Illumina data. This should be possible and could also make Fiona the single best tool on such data.

3.5 Chapter Conclusion

In this chapter, I described

- general sources and properties of errors and biases in sequencing,
- base quality values and their application to metrics and quality control,
- the basic idea behind read error correction,
- the three approaches to error correction (spectral alignment, substring tree, and multiple alignment based),
- different evaluation approaches and metrics,
- a benchmark that I used for the comparison of various read error correction methods on a large and varying input data set, and
- the results of the benchmark.

In this chapter, I discussed

- different properties of the read correction evaluation approaches as well as their advantages and disadvantages,
- properties and shortcomings of counting true positives, false negatives, etc. in the presence of indels, and
- the advantages and disadvantages of the evaluated read error correction methods.

I observed that

- counting true positives, false negatives etc. correctly is not possible in some corner cases because of ambiguities in edit distance alignments,
- Allpaths-LG and Fiona are the best available methods for Illumina read correction, where Allpaths-LG has the lowest resource consumptions,
- Fiona is the best available method for 454 and IonTorrent read correction, with Allpaths-LG and Blue being good alternatives when time and memory consumption is an issue, and
- several methods are less suitable or unsuitable for practical use because it is infeasible to pick optimal parameters in practice or they are too slow or they have too high memory requirements.

I conclude that

- the metric *gain* is a robust metric that correlates with good correction quality,
- the *false discovery rate* is a good metric for judging whether a method overcorrects or undercorrects and it might be good to prefer more conservative (*i.e.*, undercorrecting) methods,

- Fiona and Allpaths-LG are the currently best available overall error correction tools.

My main contributions in the area of read preprocessing are

- the discussion regarding corner cases when counting true positives, false negatives etc.,
- the comprehensive assessment of current read error correction methods, which enabled
- the data-driven development and and tuning of Fiona, and
- devising and implementing the benchmark tool Compute-Gain that can be of use for future benchmarks.

Chapter 4

Read Mapping

I developed the read mapping benchmark Rabema *in collaboration with Anne-Katrin Emde, David Weese, and Knut Reinert and we published it in BMC Bioinformatics (Holtgrewe* et al.*, 2011):*

Holtgrewe, M., Emde, A.-K., Weese, D., and Reinert, K. (2011). A novel and well-defined benchmarking method for second generation read mapping. *BMC Bioinformatics*, 12(1):210.

The read mapping method RazerS *3 was developed in collaboration with David Weese and Knut Reinert and we published it in Bioinformatics (Weese* et al.*, 2012):*

Weese, D., Holtgrewe, M., and Reinert, K. (2012). RazerS 3: faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599.

In resequencing pipelines, the step following the read preprocessing is *read mapping*. Roughly speaking, given a reference genome sequence for an organism and a set of HTS reads, the aim of read mapping is finding the sample location in the reference for each read. I give a precise definition and a discussion of the topic of read mapping later in this chapter.

Read mapping has many applications. These range from variant detection in resequencing projects of whole genomes (Bentley, 2006) or exomes (Ng *et al.*, 2009) to variant detection in whole populations (Abecasis *et al.*, 2010; Mills *et al.*, 2011). Other applications include the analysis of metagenomic samples (Qin *et al.*, 2010) and interactions of proteins with DNA in ChIP-seq (Valouev *et al.*, 2008), methylation analyses in epigenetics (Deng *et al.*, 2009), and cancer genome analyses (Campbell *et al.*, 2008).

I limit my considerations in this chapter to reads generated by the HTS platforms described in the introduction of Chapter 3 and I put the focus on Illumina reads. Generally, these platforms

generate shorter reads than previous Sanger-based sequencing machines at comparatively higher error and much higher throughput rates. Because of these different properties and the large number of possible applications, there has recently been a large interest in the development of efficient and accurate read alignment software.

Structure of This Chapter. The topic of this chapter is the read mapping problem and its solution. First, in Section 4.1 I give a formal definition of read mapping and discuss the differences between the problem in the biological application and the formal definition. Then, in Section 4.2 I describe my method Rabema benchmark for read mapping based on this formal definition. Third, in Section 4.3 I give an overview of read mapping approaches and some technical properties of sequencing as it relates to read mapping. This is followed by the detailed description of RazerS 3, a fully sensitive yet practically usable read mapper, in Section 4.4. In Section 4.5 I present an experimental evaluation of read mapping software (including RazerS 3) in which I also use Rabema for evaluating the read mapping programs in its formal framework. Finally, in Section 4.6 I give a conclusion to this chapter.

4.1 A Formal Definition of Read Mapping

The read mapping problem arises for different applications and for different sequencing platforms. Figure 4.1 shows examples for three different cases. Figure 4.1a shows a semi-global alignment as is usually performed for Illumina reads when aligning them to a reference sequence. Because of the high error rates towards the 3' end of 454 reads, they are often aligned locally to the reference as shown in Figure 4.1b. However, using read trimming, the ends with higher error rates in 454 reads could be removed before mapping, reducing them to the first case. In the case of deletions in the donor with respect to the reference or when aligning RNA-seq reads to the reference, spliced mapping is performed as shown in Figure 4.1c.

I do not mean the list above to be exhaustive but to give an impression about the different variants and to differentiate semi-global read mapping from other variants. Arguably, the case of semi-global alignment is the most important one since Illumina reads are most predominantly used in practice. Also, 454 and IonTorrent reads could be trimmed in the precomputation step such that their semi-global alignment is possible.

My aim in this section is to give a *formal definition* for semi-global read mapping. Since I limit the considerations in this chapter to the semi-global case, I use the term *read mapping* to exlusively refer to this case. First, in Section 4.1.2 I will give an intuition of what I consider one *match* in read mapping and when I consider two *alignments* to belong to the same match. Then, in Section 4.1.3 I will introduce a formal definition for matches in read mapping which allows a formal definition of the *single-end* read mapping problem in Section 4.1.3.

Note that in this section I will give a formal definition for and focus on *single-end* read mapping. I will give a discussion on an extension of this to *paired-end* read mapping in Section 4.1.5.

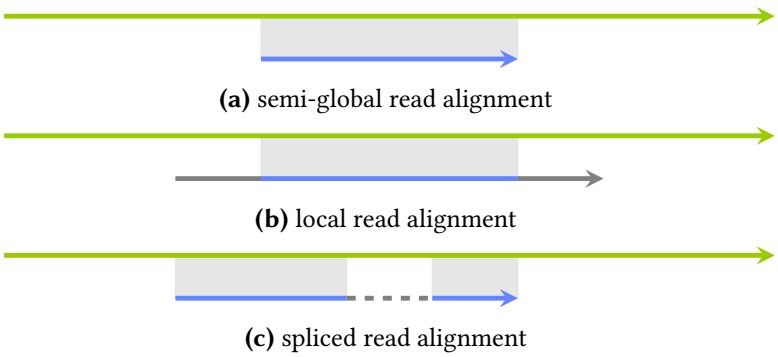


Figure 4.1: Three examples of different kinds of read alignments: (a) semi-global read alignment as in the case of Illumina reads against a reference sequence, (b) local read alignment as in the case of unprocessed 454 reads, and (c) spliced read alignment as is the case of aligning RNA reads to a reference, for example. The sequence is shown in green while the aligned parts reads are shown in blue and the unaligned parts in gray. Unaligned sequence is shown in gray while gaps between splicing points are shown as dashed lines.

4.1.1 Differences to Solving the Biological Problem

Before describing the formal definition for read mapping, a discussion regarding solving the biological problem is in order. Actually, it is desired to find the corresponding sample location in the reference for each read. However, solving this problem is challenging and it is not possible to find the correct solution in every case.

First, many genomes contain repetitive sequence, *i.e.*, DNA stretches that occur a second time with few errors or even perfectly (recent estimates as the one by de Koning *et al.* (2011) give 50–69% of the human genome as repetitive content). Thus, a read may align equally well to more than one position leading to ambiguity in read mapping. Such repeat regions cannot be simply ignored. For example, the simple repeat region 6,534,027–6,534,174 on chr. 19 of the human genome (build hg19) overlaps with gene TNFSF9, and Sokol and Atagun (2010) state that "there are known disease genes in this range".

Second, there are differences between the donor and the reference (in resequencing, these variants are what biological practitioners are after) and the sequencing process itself introduces errors and thus differences to the reference genome. This requires read mappers to search reads approximately in the genome, which can lead to finding many more alignments for a read. Some read mappers such as Bowtie (Langmead *et al.*, 2009) and BWA (Li and Durbin, 2010) try to incorporate base quality values into their mapping considerations and prefer mismatches of low-quality bases over those in high-quality bases. However, David *et al.* (2011) found that this can mislead such read mappers in the presence of SNVs and indels since they are mistaken for sequencing errors. Here, quality-aware read mappers can be misled by high-quality bases. Such read mappers often do not align a reads at locations overlapping with SNVs but with those explainig mismatches with lower qualities.

Thus, all authors of read mappers make some assumptions about the underlying chemistry and

biology. These assumptions are then (at least) implicitely formalized by writing a program and optimizing for some criterion. To the best of my knowledge, the formal definition for single-end read mapping that I give below is the only formal definition of the read mapping problem at the time of writing. In particular, my definition provides a formal criterion for a read mapper being fully sensitive or not. I published it together with my coauthors in (Holtgrewe *et al.*, 2011).

The biological problem cannot be solved directly. Thus, such formal models are only useful if they closely model the relevant parts of the biological problems, especially if it is then possible to check for the quality and possibly optimality of a solution.

4.1.2 An Intuition for Read Mapping Matches

A possible definition of read mapping is as follows. Note that I will replace some of the definitions given in continuous text further down in highlighted *Definition* sections. Given a reference sequence G, a read sequence R, a distance function δ, and a maximal distance d. The term $\delta(g, R)$ assigns a numeric distance value to an infix g of G and the read R. The domain of δ defines whether its two arguments must have the same length (as in the case of Hamming distance) or whether indels are allowed in the alignment of R to G. Note that δ could also be the score of a general alignment (*e.g.*, using a Smith-Waterman score) but I do not explore this case further and limit my considerations to Hamming and edit distance.

For each read R, the task in read mapping is to find a set of *matches* of R in G. The precise definition of the term *match* is surprisingly involved and I will give it in Section 4.1.3. For now, let a match be a *location in the reference where the read was aligned.* A *feasible* match is a match where the read aligns with distance $\leq k$. The task could then be to find all feasible matches for a read in the reference genome.

The difficulties for defining the term *match* arise when trying to decide when two close alignments belong to *distinct matches* and when they should be considered to belong to the *same match.* The taken approach for this decision will fundamentally influence the definition of the term *match.*

Alignments Beginning or Ending in Reference Gaps. First of all, I exclude alignments beginning or ending in reference gaps in their semi-global alignment from all subsequent considerations. Such alignments can always be replaced by ones with the same or a smaller distance by aligning the first or last base to the reference. Figure 4.2 shows an example. Aligning the last base of the read against the reference base after the gap would preserve the edit distance of 1. If the next base was `C` instead of `A`, the distance would decrease from 1 to 0.

When Do Two Alignments Belong to Different Matches? Figure 4.3 shows possible alignments of a read against a reference. Consider the case where the best two matches of the read against the reference are to be found with a distance of up to 3. The row *alignments* shows two alignments of the read to the reference sequence that somehow "naturally" belong to two distinct matches. The alignments in the rows $\star$ and $\star\star$ have a lower edit distance than the right

```
reference:  C C A G C G − A G A T
                    |   |   |
read:               G C G C
```

Figure 4.2: A semi-global alignment of a read to the genome ending in a gap.

```
reference    C A G A C T C C C A A C T G T C A  ...  C A G A C T C C C C C C A A C T C C A
alignments             T C C C A A C                              T C C C − − − A A C
   ⋆                T − C C C A A C
   ⋆⋆                 T C C C A A − C
```

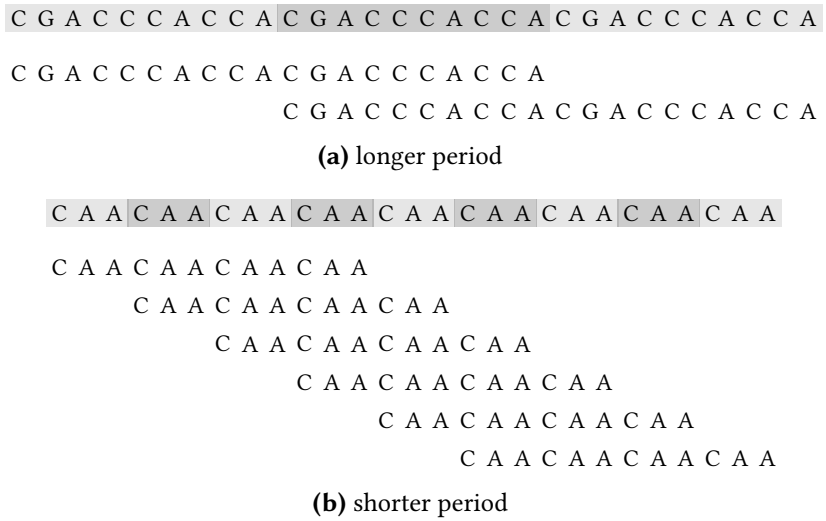
Figure 4.3: Alignments of the read `TCCCAAC` against two locations in the reference sequence.

match from the row *alignments*. Common sense tells that the alignments in the left column are not "significantly different", though: each alignment with distance k induces alignments with distance of $\leq k + 2$ by aligning the leftmost/rightmost base one more position to the left/right and introducing a gap. Note that for the sake of simplicity, the figures in this section use reads that are shorter and a higher error rate is allowed than usually found in practice (where reads are mostly ≥ 36 bp and a base-wise error rate of $\leq 2\%$ is observed for Illumina reads).

Repeats are another issue. Figure 4.4 shows two tandem repeats (with long period in Figure 4.4a and with short period in Figure 4.4b). In the case of the long period, intuition identifies the two distinct alignments shown in Figure 4.4a. In the case of the short tandem repeat, enumerating repeats as for the long repeat would lead to the six distinct alignments shown in Figure 4.4b.

However, using such a counting scheme for matches would require a read mapper to find many alignments in repeat regions. This is not desirable for two reasons. First, a lot of time and memory would be spent for finding many very similar alignments. Second, when counting matches in benchmarks, reads from short tandem repeat regions would get a higher weight with this counting scheme than reads from long tandem repeat regions or from non-repetitive regions. Weighting each found match with $1/n$ (where n is the number of positions the read aligns at) is deficient as well. It is preferable to find a way to naturally *merge similar alignments into matches* (*e.g.*, the one from the left column of Figure 4.3) and merge matches that are very close to each other (see also Figure 4.4b) as well as to *create separate matches for alignments that are sufficiently distinct* (see also Figure 4.4a).

The Definition of Trace Trees. Consider a dynamic programming (DP) matrix for semi-global alignment (see also Section 2.2.1). Each such alignment corresponds to a path from the top row to the bottom row. Horizontal and vertical movements between cells correspond to indels, diagonal movements to matches and mismatches. Standard DP alignment algorithms yield the



(a) longer period

(b) shorter period

Figure 4.4: Examples for alignments of reads from tandem repeats with (a) longer period and (b) shorter period. In the reference, the repeated copies are highlighted in alternating shades of gray.

smallest distance for each alignment end position. From an end position, the start position(s) can be found by performing a traceback search backwards and upwards in the matrix as outlined in Section 2.2.1. Given a value for k and an end position, all start positions yielding a feasible alignment can be computed by enumerating the traces.

In these considerations, only deterministic DP algorithms are of interest, *i.e.*, those that always perform the same choice in case of ambiguities. For example, having the choice between moving vertically, diagonally, and/or horizontally, such an algorithm could always take the rightmost choice. In this case, the preference in movements is (in descending order) vertical, diagonal, and horizontal. The Needleman-Wunsch algorithm is an example of a deterministic DP alignment algorithm. When plotting the traces for all feasible alignments, this yields a picture as shown in Figure 4.5.

The Simpler Case of Hamming Distance Alignments. A simple observation when considering the alignments in the case of Hamming distance is that each alignment unambiguously corresponds to exactly one diagonal in the alignment matrix. Thus, using the Hamming distance model, the term *match* for R in read mapping can be defined as an interval of G of length $|R|$. Since the length is fixed, Hamming distance match can be identified with either begin or end position. For consistency with the match definition for edit distance below, I identify Hamming distance matches with their end position.

Identifying Matches With Alignment end Positions. Considering all combinations of start and end positions is not desirable for edit distance. In Figure 4.5, there are $5 \times 6 = 30$ such alignments in the green tree alone, possibly many feasible ones. Other means have to be used for counting matches in the case of edit distance.

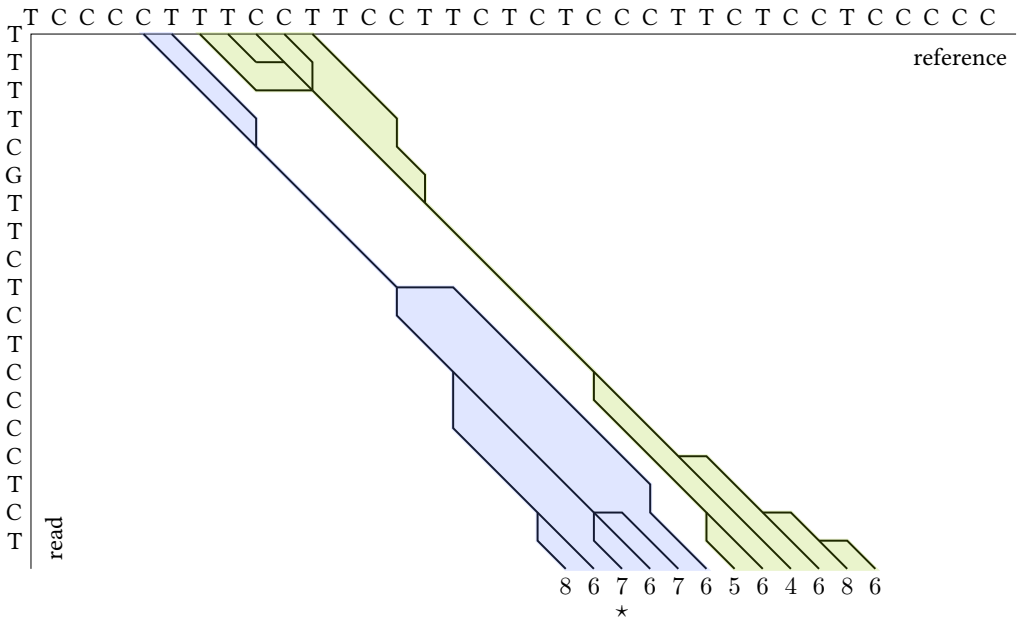
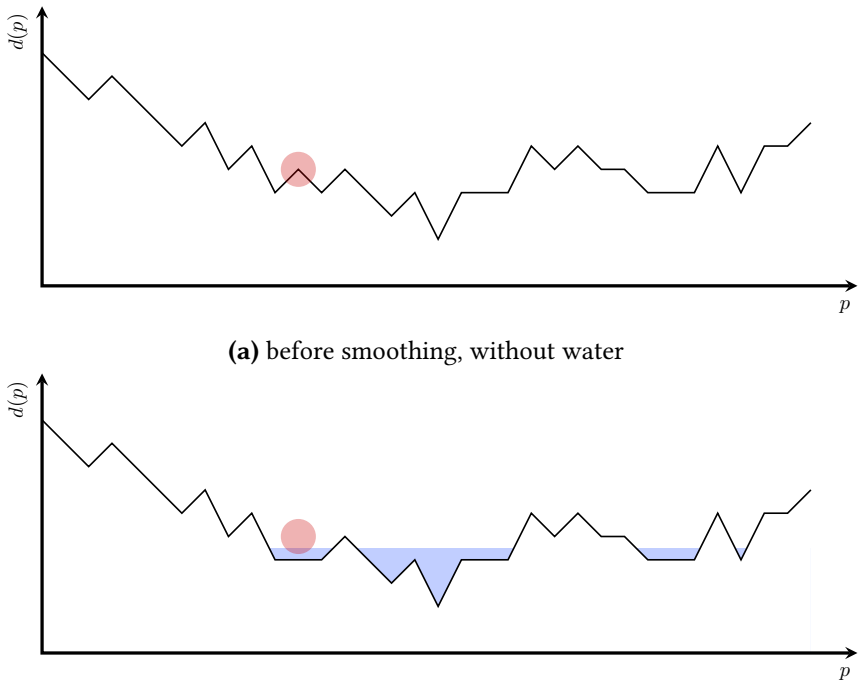


Figure 4.5: Two neighboring trace trees for $R =$ `TTTTCGTTCTCTCCCCTCT` in the reference $G =$ `TCCCCTTTCCTTCCTTCTCTCCCTTCTCCTCCCCCCCATCTCCA`. The numbers below the cells at the bottom show the edit distance of the alignment.

I observe that for alignments with a relatively low distance, the shared trace is often longer than the branching parts. This means that large parts of the alignment are the same and differing alignments can have the same distance. To avoid counting these as separated matches, I proceed as follows. Each match is identified with its end position e. The leftmost start position s that leads to an alignment of minimal distance is chosen as the *canonical start position*. The choice of s as the canonical start position is arbitrary. Picking the leftmost such position, however, has the advantage that the interval between s and e contains the start positions of all alignments of minimal score ending in e. In the example from Figure 4.5, this reduces the possible number of matches for the green tree from 30 to 6.

Error Landscapes. Using the definitions above, I now introduce the term *error landscapes* to capture the intuition of the definition of a *match* that I will formalize further in Section 4.1.3. Given a read R and a reference sequence G. I define the function d to assign a distance value to each genomic position p, such that $d(p)$ is the distance of the best alignment of R ending in p. Plotting and connecting the points $(p, d(p))$ for each position in G, yields a curve that can be interpreted as a two-dimensional height profile of a landscape as shown in Figure 4.6a.

Now, imaginary ground water rises in the landscape to a level of $k + 0.5$. This is shown in Figure 4.6b. In this example, this yields four lakes. Each lake represents a class of alignments with sufficiently low distance. The metaphor of the landscape with lakes corresponds to the natural merging of similar alignments.



(a) before smoothing, without water

(b) after smoothing, with water

Figure 4.6: Example of the error landscape (a) before smoothing, without water and (b) after smoothing, with water. The end position is plotted on the horixontal axis, the distance is plotted on the vertical axis. In (b), the raised ground water is shown and the separating alignment has been smoothed. The point between the lines marked in red is a separating position as described in Definition 4.4. The read and reference used is the same as for Figure 4.5 but with a larger portion of the repeat from this figure.

Each lake corresponds to a possible match and its lowest point corresponds to the distance of this match. Each match corresponds to an interval on the genome and all alignments ending in this interval are counted as belonging to the match. When building a gold standard for read mapping, these intervals are the matches that are to be found and a read mapper is expected to find one alignment ending in each match interval.

Landscape Smoothing. In the description of error landscapes above, I left out the detail of *landscape smoothing*. When using $k = 6$, the marked ($\star$) position from Figure 4.5 separates two lakes left and right of it. However, I wrote above that alignments sharing a part of their trace should belong to the same match. Thus, this position has to be *smoothed* away to the larger neighboring lake level as shown in Figure 4.6b.

4.1.3 A Formal Match Definition

Let G be a reference sequence of DNA5 characters and R be a read sequence over the same alphabet. Further, let a and b be positions on the reference representing semi-global read alignments, that is they are end positions of read alignments, w.l.o.g., $a \leq b$. As described above, the longest alignment is considered for each end position, *i.e.*, the alignment with the leftmost start position and best score for the given end position. These alignments are identified with their end positions and the two terms are used interchangeably. Let k be the largest allowed distance and $\delta(a)$ the edit distance of an alignment a.

In the following section, I will define the *neighbor*, *trace*, and *k-trace equivalence relations*. Together, they allow to define *match equivalence*. I will give proofs that each of the equivalence relations is well-defined. Because of the assumption that $a \leq b$, I do not prove symmetry below but this can easily be derived by the exchange of roles between a and b.

Definition 4.1 (feasible alignments). *An alignment a is feasible if $\delta(a) \leq k$.*

Definition 4.2 (neighbor equivalence). *Two feasible alignments a, b are* neighbor equivalent *(aNb) if $\delta(x) \leq k$ holds for all x with $a \leq x \leq b$.*

Proof that N is a well-formed equivalence relation. The reflexivity of N follows directly from the definition. Let a, b, c be alignments, w.l.o.g. $a \leq b \leq c$. Now, let aNb and bNc. Since b is feasible as well, $\delta(x) \leq k$ holds for all $a \leq x \leq c$ and N is an equivalence relation. □

Definition 4.3 (trace equivalence). *Two alignments a, b are* trace equivalent *(aTb) if their traces share a part. It is easily observed that a and b are trace equivalent if their canonical start position is the same.*

For example, for $k = 6$, the second and fourth alignment in Figure 4.5 from the blue tree are trace equivalent but not neighbor equivalent. In the same figure, the last alignment of the blue tree and the first match of the green tree are neighbor equivalent but not trace equivalent.

Observation 4.1. *When two positions a, b are trace equivalent then they are also trace equivalent to all x with $a \leq x \leq b$.*

Any canonical trace from x must cross the canonical traces from a or b. Therefore, the canonical alignment for x must have the same canonical start position s.

Proof that T is a well-formed equivalence relation. Reflexivity and transitivity follow directly from the definition of the trace. □

Definition 4.4 (separating position). *Given two alignments a, b. A separating position (or separating alignment) ζ is a position with $\delta(\zeta) > k$ and there exists α, β with $\alpha < \zeta < \beta$ such that $\delta(\alpha) \leq k$ and $\delta(\beta) \leq k$.*

Definition 4.5 (k-trace equivalence). *Two alignments a, b are k-trace equivalent (aKb) if one of the following holds: (1) the alignments are feasible, neighbor equivalent, and trace equivalent, or (2) there exist feasible, trace-equivalent alignments α, β, and a separating position ζ, such that $\alpha \leq a \leq \zeta \leq b \leq \beta$.*

Proof that K is a well-formed equivalence relation. Reflexivity is easy to see in both cases (1) and (2).

For transitivity, a case distinction is performed. Given aTb and bKc. There are now four cases, depending which case (1 or 2) holds for the left and right relation.

Case (1, 1). The transitivity follows since aNb and bNc holds.

Cases (1, 2) and (2, 1). I show the proof for case (1, 2), case (2, 1) follows analogously. a, b are feasible, neighbor equivalent, and trace equivalent. For b, c, there exist feasible, trace equivalent matches α, β and a separating match ζ such that $\alpha \leq b \leq \zeta \leq c \leq \beta$. Because of Observation 4.1 and $\alpha \leq b$ and aTb, a and α have to be trace equivalent. Transitively, a and β have to be trace equivalent. a is feasible and can take the role of α from before: $a \leq a \leq \zeta \leq b \leq \beta$ and thus transitivity is shown.

Case (2, 2). For a, b, c exist feasible matches $\alpha, \beta, \alpha', \beta'$ and separating matches ζ, ζ', such that $\alpha \leq a \leq \zeta \leq b \leq \beta$, and $\alpha' \leq b \leq \zeta' \leq c \leq \beta'$. It follows that $\alpha \leq a \leq \zeta \leq b \leq \zeta' \leq c \leq \beta'$ and thus $\alpha \leq a \leq \zeta \leq c \leq \beta'$. α and β are trace equivalent; the same holds for α' and β'. Because of Observation 4.1 and $\alpha' \leq \beta$ it follows that α and β' have to be trace equivalent, too. Thus, of k-trace equivalency holds for a and c in the case of (2). □

Definition 4.6 (match equivalence). *Two matches a, b are equivalent ($a \equiv b$) if there exist $\ell \geq 0$ feasible* connecting *matches $a \leq m_1 \leq \ldots \leq m_\ell \leq b$ such that:*

$$(aTm_1 \vee aNm_1) \wedge \ldots \wedge (m_{i-1}Tm_i \vee m_{i-1}Nm_i) \wedge \ldots \wedge (m_\ell Tb \wedge m_\ell Nb).$$

For $\ell = 0$, two matches are equivalent if $aTb \vee aNb$.

The motivation for the disjunction in the definition of match equivalence is to join neighboring k-trace equivalence classes/intervals when being directly adjacent to other k-trace equivalence classes or overlapping with neighbor equivalence classes. Such intervals/classes I, J (w.l.o.g. I left of J) are (in some sense) minimally joined by the rightmost match of I being neighbor equivalent to J or by the overlap.

Proof that $\equiv$ is a well-formed equivalence relation. Reflexivity follows from the case where $k = 0$. Transitivity for $a \equiv b$, $b \equiv c$ follows from using b as a connecting match if it is feasible. If b is not feasible then there exist feasible matches, α, β that are trace-equivalent to b and they can be used as connecting matches. □

Definition 4.7 (read mapping match). *The match equivalence relation creates a partition on the set of all positions of G. Each of these partition entries is a distinct* read mapping match *consisting of match equivalent alignments. The* distance $\delta(m)$ *of a match m is defined as $\delta(m) = \min_{a \in m} \delta(a)$. A match is* feasible *if it contains a position with a feasible alignment.*

4.1.4 A Formal Definition of Single-End Read Mapping

In the previous section, I gave a formal definition of a match in single-end semi-global read mapping. Using this definition, I can now give formal definition of several variants of the single-end read mapping problem.

Definition 4.8 (All-Mapping). *Given a reference G, a read R, and a maximal distance k, the All-Mapping problem is defined to find at least one alignment for each match from the set of all feasible matches M.*

Definition 4.9 (match stratum). *For each $d = 0, \ldots k$, the stratum S_d of matches for R in G is the subset of matches in M that have distance d. Formally:*

$$S_d := \{m \in M \text{ such that } \delta(m) = d\}.$$

This definition of a match stratum is similar to the one introduced in the manual of Bowtie (Langmead *et al.*, 2009).

Definition 4.10 (All-Best-Mapping). *Given a reference G, a read R, and a maximal distance k, the All-Best-Mapping problem is defined to find at least one alignment for* each *match from the stratum with the smallest distance.*

Definition 4.11 (Any-Best-Mapping). *Given a reference G, a read R, and a maximal distance k, the Any-Best-Mapping problem is defined to find at least one alignment for* one *match from the stratum with the smallest distance.*

4.1.5 Single-End Versus Paired-End Mapping

Above, I limited the considerations to *single-end* mapping. Another important variant of read mapping is *paired-end* mapping. *Paired reads* are generated by a different protocol when sequencing.

The general approach underlying most sequencing protocols is to first isolate the DNA from the sample that is to be sequenced. The DNA is then *sheared*, *i.e.*, split into smaller fragments (also called *templates*). For example, in the popular *paired-end protocol* by Illumina, these fragments have a size of 200–500 bp (in the popular *mate-pair protocol* by Illumina, the fragment size ranges from 2000 to 3000 bp).

Depending on the protocol used, the templates are then sequenced from one end or from both ends. In the case of single-end sequencing this yields a single read and in the case of paired-end

sequencing this yields two reads whose relative orientation is known. In the case of the Illumina paired-end protocol the reads are *inwards-facing* (in Illumina's mate-pair protocol the reads are *outwards-facing*).

The fragments generated prior to sequencing are not generated to all have the same length. Instead, the used biochemical technique leads to random sizes. The fragments are then *size-selected*. Often, the technique *gel electrophoresis* is used to distribute the fragments spatially depending on their mass (and thus size). Afterwards, the person preparing the sample selects an area from the gel that presumably contains fragments of desired size. Often, this makes the *template lengths* roughly follow a normal distribution with approximately known parameters.

Paired reads have interesting and important properties for variant detection that I will explore further in Chapter 5. Here, however, I will only consider them in the context of read mapping.

Thus, an expected template size range can be part of the input when performing read mapping. Consequently, a read mapping program could be limited to only return pairs of matches for pairs of reads such that their template inferred from the alignment lies within the expected range.

However, such a read mapper could not possibly map all paired reads spanning breakpoints of structural variation. For example, consider the case of a read pair with template size of 500 bp that is expected to have a template size between 400 and 600 bp but spans a deletion in the donor of 200 bp. In this case, the template size when mapping is 700 bp and the read mapper would ignore it.

Also, consider the following case where the expected template size is between 400 and 600 bp as well. Let one read align uniquely with distance 0 to the reference. The other read has two alignments, one leading to an inferred template size of 400 bp with distance 0 and one to an inferred template size of 500 bp with distance 2. It is unclear which alignment of the second read is to be preferred.

Thus, mapping pairs of reads leads to a two- or even multi-dimensional optimization problem. There is no clear objective function for combining the inferred template length and alignment distances of two read alignments. It follows that it is infeasible to create a generally useful definition of paired-end matches and paired-end read mapping.

However, my formal definitions above allow for the evaluation of a read mapper's capability to solve the *core* read mapping problem: finding alignments for single reads. Paired-end mapping can be interpreted as finding and combining single-end alignments into paired alignments. Thus, I argue that the formal definition is also useful for paired-end mapping to combine the definition of matches with other criteria for specific formal definitions of paired mapping. Also, when a read mapper's performance for single-end mapping is bad, this is a good indicator that its performance for paired-end mapping is not optimal either.

4.2 Rabema — A Method for Benchmarking Read Mappers

The formal definition of read mapping matches in Section 4.1.3 and the formal definition of read mapping in Section 4.1.4 originally appeared in (Holtgrewe *et al.*, 2011). I developed the for-

mal framework for the method Rabema, the Read Alignment BEnchMArk, together with my coauthors, and wrote the software package implementing the method. This method allows the comparison of read mapping methods based on the well-defined definition of read mapping. We developed it to solve problems with the common approaches of read mapping benchmarks.

Many authors of read mapping software count aligned reads when comparing their mapper with existing approaches and call the fraction of mappable read the *mapping efficiency*. This only allows for a rough estimation of the quality of a read mapper since some read mappers such as Bowtie (Langmead *et al.*, 2009) treat non-called characters (`N` in the read) as wildcards. Such read mappers will align more reads with `N` characters than read mappers always counting `N` characters as mismatches such as BWA (Li and Durbin, 2009).

Additional care has to be taken when counting uniquely matching reads. If a read mapper does not have full sensitivity then it could miss a possible second match and report a non-unique match as a unique one. Another read mapper could find both matches and discard the read as non-uniquely mapping. In such an evaluation, a less sensitive read mapper gets a higher rating when not controlling for this. To solve this problem, the set of reads reported as uniquely mapping has to be compared to the reads discarded since they are not aligning uniquely. One can then compute a set of false positives and report this number as well. This comparison is rarely, if ever done. Also, when no read mapper with full sensitivity is used, this set of false positives can only be seen as an approximation since the ground truth is not available.

Further, a definition of "full sensitivity" requires a formal definition of the read mapping problem as first given in the paper introducing Rabema.

4.2.1 Gold-Standard Generation

The first step when using Rabema is to build a gold standard, consisting of the equivalence classes resulting from the match equivalence relation. Rabema uses error rates instead of error counts to make the method also applicable in the case of using reads with varying length (*e.g.*, for IonTorrent or 454 reads or when using trimmed Illumina reads). Consequently, Rabema also defines the term strata and read mapping in terms of error rates and not error counts. I only consider Hamming and edit distance for Rabema. The error rate percentage of a read alignment is computed as $\lceil 100 \cdot e/\ell \rceil$ where ℓ is the length of the read and k is the number of errors in (distance of) the alignment. For each read, the gold standard consists of a set of intervals on the genome sequence for each stratum $S_0, \ldots, S_e$ where e is the largest allowed error rate in percent.

For each read, the gold standard is generated from a set of alignments for this read given as the input and a maximal error rate e. The generator then takes each alignment and performs an online string search around the position to first build the error landscape and then create the intervals representing the matches. The maximal error rate e is used for limiting the string search. The search to the right can be stopped if a position is reached where the corresponding alignment has an error rate greater than e and the alignment's start position is different from the start position of the previous alignment with an error rate $\leq e$. In this case, the interval is trimmed to the rightmost point with error rate $\leq e$. The search to the left is handled accordingly. The online string search uses the bit-vector algorithm by Myers (1999b).

The program building the gold standard expects the read alignments to be sorted by coordinate. This allows an optimization in the computation for redundant matches. By keeping the last found interval in memory for each read, alignments of a read are ignored if they end within the interval or left of it. In this case, the gold standard builder has already generated an interval for corresponding match. The resulting file contains the gold standard intervals (*GSI*) ordered by read name. For each read and error rate stratum, the file contains entries with the begin and end position of the intervals for all matches of the read in the stratum.

There are two modes of operation when generating a gold standard:

oracle mode In the case that the one true alignment is known for each read, as is the case when simulating reads, the user can provide a gold standard alignment file (in SAM or BAM format (Li *et al.*, 2009a)) for the read set. The gold standard will then be built as one match interval for each read. The interval will be extended by setting e to the error rate of the alignment in the gold standard alignment file.

real-world mode For real world data, the user can provide a gold standard alignment file (again in SAM or BAM format) generated by a fully sensitive read mapper. The maximal allowed error rate e is another input parameter to gold standard generation. The gold standard will then be generated by extending each alignment with the given e and any duplicate match caused by redundant alignments will be removed.

Our read mapper RazerS 3 (Weese *et al.*, 2012) is an example of a practical, fully sensitive read mapper for edit distance and is used in the evaluation presented in this thesis. In addition to the parameters mentioned above, the user can also specify whether to build the gold standard for edit or Hamming distance.

4.2.2 Read Mapper Result Comparison

Of course, Rabema also provides a program to compare the GSI file with the result of a read mapper. The read mapper result is given to the comparison program as a SAM or BAM file, sorted by read name. The program then compares the read mapper result according to its parameterization.

Besides the used metric (edit or Hamming distance) and the maximal error rate e, the user can configure whether to compare in the oracle or the real-world mode, the maximal error rate the gold standard was built for, and select one of the read mapping problem variants to perform the evaluation for.

In the real world mode, the user can compare between benchmarking for the All-Mapping, All-Best-Mapping, or Any-Best-Mapping problem. The program will generate a report that also breaks down how many of the required matches in each stratum are found by a read mapper. This allows to see how well a read mapper performs for reads that align with a low error rate in comparison to those that align with a high error rate.

It is important to choose the same error rate in the evaluation and when mapping reads as the one the gold standard was built for. If a lower error rate is chosen when building the gold standard than in the evaluation, this can lead to problems. In this case, error landscape lakes that appear

distinct at lower error rate when building the gold standard might be joined with the higher error rate in the read mapper. Consequently, a read mapper might only return one match in such a case where the evaluation would expect one match for each of the two lakes that it sees.

In the oracle mode, the evaluation program simply counts the number of reads in each stratum for which the alignment was found. A read is assigned to the stratum of its GSI entry.

4.2.3 A Peculiarity with Reverse-Strand Matches

I observed that there is a peculiarity with this definition when searching for reverse-complemented reads or reads on the reverse strand of the reference sequence. An example for this is shown in Figure 4.7 for $k = 5$. Because of the asymmetry of the search (forcing the alignment to end at each position while choosing the best begin position), there are two lakes in the first case but only one lake in the second case. Note that there is a truly separating alignment in Figure 4.7a, the alignment with distance 6 ($\star$). The alignments in the lake left and right of it are equivalent.

When building the gold standard, one can search for the reverse-complemented reads in the original reference sequence or search for the original reads in the reverse-complemented reference sequence. Depending on this choice, there might be more or less equivalence classes. In practice, I observed very few differences, mainly in low-complexity repeat regions and with very long (*e.g.*, 454) reads only.

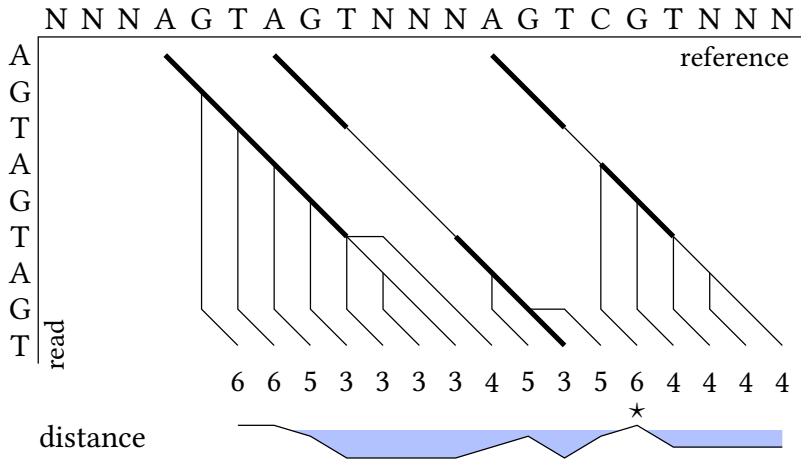
For Rabema, the gold standard is built using RazerS or RazerS 3. Both RazerS versions reverse-complement the reference sequence because this is computationally more efficient for its approach. The situation from Figure 4.7a might occur at some place. A read mapper that reverse-complements the reads (index-based read mappers such as Bowtie and BWA do so) will only see one lake, as depicted in Figure 4.7b.

However, since this almost only occurs in low-complexity regions and the number of occurrences is small, the arising difference of results will be neglegible. Furthermore, such read mappers can easily implement a special benchmark mode that reverse-complements the reference sequence instead of the reads. This might come at a higher computational cost but can then be used to assess the sensitivity and verify that the bias against the approach of reverse-complementing the contigs is small. For measuring computational costs, the normal mode can be used.

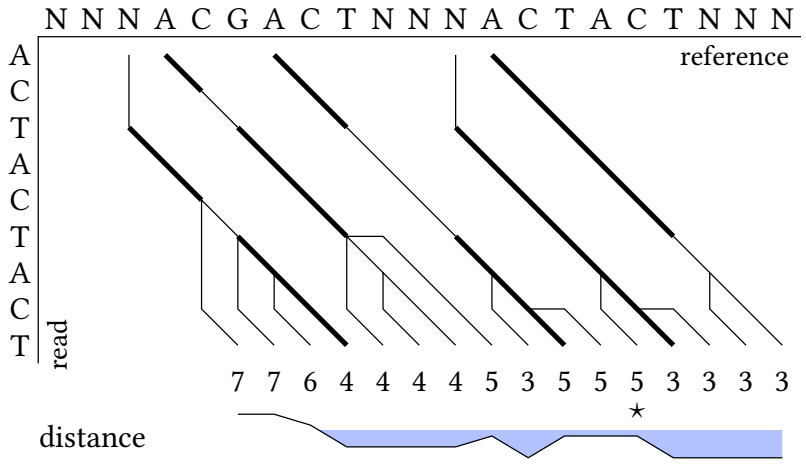
Note that one has to choose whether to reverse-complement the reads or the reference sequence. Each decision would slightly bias towards the chosen approach. I chose to reverse-complement the reads since this is the approach in RazerS and RazerS 3. These tools were used for generating the gold standard read alignments since they were the first fully sensitive read mappers.

4.2.4 Possible Extensions to Rabema

While Rabema offers comprehensive benchmarking for the read mapping problem as formalized in this Section 4.1.4, there are various possible extension in the future.



(a) forward strand



(b) reverse strand

Figure 4.7: This figure shows a peculiarity with matches on the reverse strand. (a) shows the alignment of a read on the forward strand and (b) shows the alignment of the reverse-complemented read sequence on the reverse-complement of the reference. Depending on whether the reference sequence or the reads are complemented, there might be an infeasible alignment that separates two match classes. Thick lines correspond to matches whereas thin lines correspond to mismatches or indels.

First, one could consider to include different scoring schemes for matches. Read mappers such as BWA use Needleman-Wunsch alignments for weighting matches, mismatches, and gaps differently in the alignment and also incorporate quality values in the alignment. Read mappers such as Bowtie separate their alignments into strata depending on the edit distance and then use base quality values on matching and mismatching bases to select a best alignment. Such scores could be supported by Rabema as well: using a sufficiently large e for creating a gold standard

mapping will find all biologically sensible alignments. When building the gold standard, the generated matches could be weighted using scoring schemes from the literature. However, at this point it becomes unclear what a good scoring scheme is, as can be seen by the various schemes used in the different read mappers.

Another possible extension is for paired reads. Currently, verification of paired-end performance can only be done using simulated reads in the oracle mode where the original sample location is known. One reason for paired-end sequencing is to lower the ambiguity in read mapping by requiring *concordant* (alinging to the genome in linear order, in the expected orientation, and within the expected template size) mappings. Thus, applying the oracle mode makes sense for such data.

A further extension of Rabema to paired data would lead to similar problems as using a non-edit distance scoring scheme. What comprises a "true" paired-end match? Does it make sense to enumerate all combinations of paired matches in the case of ambiguity? The answers to these questions remain unclear while using the oracle mode for evaluating paired-end mapping is a good alternative.

An alternative approach to the benchmarking of read mappers was considered by Hatem *et al.* (2013). These authors "define a read to be correctly mapped if it is mapped while not violating the mapping criteria," and consider multiple different mapping criteria. Their approach is more flexible than my formal definition but it lacks an important property of my formal definition.

Namely, the lack of a formal match definition and the consequent lack of a method for enumerating all formal matches does not allow the method to control for false negatives. First, a read might have a valid match but no read mapper is able to find this match. Second, their approach does not allow to estimate whether a read mapper is able to find all mapping locations. These are major drawbacks stemming from the fact that this definition depends on the performance of the used read mappers, leading to a catch 22.

4.3 Read Mapping Approaches and Methods

Obtaining the full genome sequence of organisms such as the fruit fly (Myers *et al.*, 2000) and humans (Lander *et al.*, 2001; Venter *et al.*, 2001) has unlocked the full potential of read mapping. The original full sequence of the human genome was published in 2001 and at this time, Sanger sequencing (*first generation sequencing*) was the prevalent sequencing technology. Since then, different technologies have been developed. These technologies include the technologies by 454, Illumina, and IonTorrent (often termed *second generation sequencing (2GS)*) and the recent technology by Pacific Biosciences (often termed *single-molecule* or *third generation sequencing (3GS)*). 2GS and 3GS machines are also often called *high-throughput sequencing (HTS)* machines. These technologies have different properties and Table 4.1 shows the most relevant ones for read mapping. Appendix B contains a more detailed description of the sequencing platforms.

Note that the properties of each technology have changed over time, that is read length and throughput have increased while the error rate has decreased. To summarize, the advantage of

second generation sequencing is the high throughput (at dramatically lower cost, not shown in table) with the drawback of shorter reads and higher error rate. The main innovation of third-generation sequencing is to perform *single-molecule sequencing*, *i.e.*, the sequence is not read out from multiple cloned templates but from one molecule only. In the case of current Pacific Biosciences (PacBio) sequencing technology, this allows very long reads at the cost of a high error rate.

At the time of writing, the dominant technology in many areas is Illumina sequencing since it offers high throughput at a low cost. IonTorrent and 454 are used when long read lengths are important, for example in metagenomic sequencing. Sanger sequencing is still used for validation, where very long reads with very low error rate are required, that is when the trade-off of higher coverage depth with shorter read length of second generation technology does not pay off. Also, Sanger sequencing still is the gold standard for verifying mutations found by HTS in clinical contexts. The current main application of long PacBio reads is for *de novo* assembly.

4.3.1 The Practical Setting of Read Mapping

As described earlier in this chapter, read mapping is the process of approximately searching relatively short reads in long reference sequences. The differences in read length, error rate, and throughput led to the development of specialized tools for the different kinds of reads. Reads from Illumina machines are usually aligned semi-globally while 454 and IonTorrent reads are either trimmed or aligned locally. With the advent of second generation sequencing, there has been a large increase in available read mapping software. Reinert *et al.* (2015) give an overview of the area of read mapping, in particular the algorithmic ideas and Fonseca *et al.* (2012) maintain an updated list of alignment software.

Read mapping software has to deal with long genomes and large sets of reads. At the same time, the sensitive and specific alignment of reads to the reference is important such that the quality of downstream analyses does not degrade because of missing or incorrect mappings.

	1st gen.	2nd gen.			3rd gen.
	Sanger	454	Illumina	IonTorrent	PacBio
read length	400–900 bp	200–800 bp	20-150 bp	$\approx$ 200 bp	up to 10 kbp
error rate per base	0.001%	0.01%	1%	2%	13%
throughput per day	0.7 Mbp	0.7 Gbp	600 Gbp	12 Gbp	2.4 Gbp
paired reads	yes[1]	yes	yes	yes	no

Table 4.1: Overview of sequencing technologies, the throughput is given for one machine. Sources: (Liu *et al.*, 2012) and (Quail *et al.*, 2012).
[1] Not directly supported by sequencers such as 3730xl DNA Analyzer by Applied Biosystems, but available through wet lab techniques.

Varying Length and Complexity of Genomes. Genomes of different organisms have different complexities. Bacterial genomes such as the one of *E. coli* (with a length of $\approx$ 5 Mbp) are usually quite short and have a low content of repetitive sequence. The genomes of the eukaryotic model organisms *S. cerevisae* ($\approx$ 12 Mbp), *D. melanogaster* ($\approx$ 120 Mbp), and *C. elegans* ($\approx$ 100 Mbp) are longer and have a varying degree of repeat content. The human genome has a length of $\approx$ 3 Gbp and a repeat content of $\approx$ 50–69%.

Sequencing Errors. The early Illumina sequencing machines generated reads with a length of 20–36 bp, a substitution error rates of about 1%, and very few indel errors (Dohm *et al.*, 2008). It is a reasonable assumption to only allow one or two substitution errors when mapping such reads to genomes for SNV discovery. For discovering small indels and when mapping the recent longer Illumina reads with 100–150 bp, however, indels have to be allowed in mapping as well. For 454 and IonTorrent, the primary types of errors are indels and the error rate rises towards the 3' end of the read. For 454 reads, a higher frequence of errors can be seen at the beginning of the read as well.

Paired-/Single-End Mapping Variations. Paired-end sequencing is an important technology for resolving ambiguities due to repetitive sequencing. The available protocols allow different insert sizes. For example, the Illumina paired-end protocol yields relatively short pairs (template size means of $\approx$ 200–500 bp), while the Illumina mate-pair protocol allows insert sizes of $\approx$ 3 kbp.

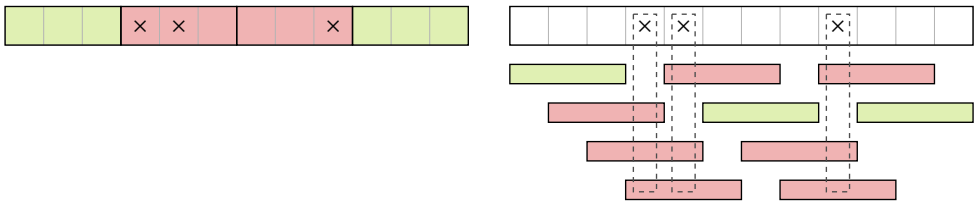
The insert size has various implications, for example a larger insert size increases the probability of covering SV breakpoints and thus such reads are often produced for SV analysis. Read libraries with lower insert sizes are often generated at higher coverages and are used for resolving ambiguities that are due to repetitive region and different small-scale variants.

4.3.2 Theoretical Insights

In this section, I present two classic, fundamental lemmas that are used in fully sensitive read mapping approaches and allow to explain the motivation behind heuristics. These lemmas give insight into the distribution of errors in reads but can also be applied to detecting other features in sequences, of course.

Pigeonhole Principle and k-mer Counting. First, the *pigeonhole principle* is the name for a simple counting argument. For example, when there are k pigeons in $k + 1$ holes then there has to be at least one hole without any pigeon. A similar argument can be made for errors in approximate string search (Navarro and Raffinot, 2002, p. 163), especially for read mapping as explained by Weese *et al.* (2009).

Lemma 4.1 (pigeonhole principle). *Given an alignment of two sequences g and R with edit distance k. When splitting R into $k + 1$ parts, at least one of these parts will have no error in the alignment to the corresponding part in g.*



(a) pigeonhole lemma **(b)** k-mer lemma

Figure 4.8: A read with three errors in the context of the (a) pigeonhole and the (b) k-mer counting lemma. In both cases, a read with ten characters is shown at the top, and errors are shown as crosses. The parts and k-mers affected by errors are colored red. In (a), the read was split into four parts of size three and the three errors affect two of these parts (three errors could affect at most three of the four parts). In (b), the ten k-mers are shown below the sequence.

An extension to this argument has been observed by Myers (1994):

Lemma 4.2 (generalized pigeonhole principle). *Given an alignment of two sequences g and R with edit distance k. When splitting R into j parts then at least one of these parts contains $\lfloor k/j \rfloor$ or fewer errors in the alignment to the corresponding part in g. Similarly, at least one of these parts will have $\lceil k/j \rceil$ or more errors in the alignment to the corresponding part in g.*

Thus, when performing approximate search of read R with up to k errors, *candidate regions* for the alignment of R can be obtained by searching for each of the $k+1$ infixes of R yielded by partitioning R. Often, these infixes are called *seeds*. While the length of each seed in the read is arbitrary, short seeds have the disadvantage of leading to many spurious occurrences in the reference. Also, using the same seed length can have technical advantages. Figure 4.8a shows an example.

A second fundamental insight is due to Jokinen and Ukkonen (1991):

Lemma 4.3 (k-mer lemma). *Given two sequences g and R such that their edit distance is k and $n = \max\{|g|, |R|\}$. Then, the number of shared k-mers between g and R is at least $t = n+1-(k+1)\cdot q$.*

Figure 4.8b shows an example. Such counting arguments can be used to design efficient algorithms for candidate region identification in approximate string search (Rasmussen *et al.*, 2006) and read mapping software (Weese *et al.*, 2009).

The candidate regions can then be *verified* by searching for a semi-global alignment of R to G around the match, for example using a variant of the Needleman-Wunsch algorithm.

Practically Efficient Heuristics. As described in Chapter 3, the positional error probability in NGS reads increases towards the 3' end of the reads. The error distributions observed in practice are in general not the worst cases assumed in the pigeonhole and k-mer lemma. This observation can be used to improve the running time of read mapping methods while keeping a good sensitivity.

It can be seen as a good heuristic to search for seeds from the 5' end of the reads where errors are less frequent. Another heuristic is to use larger values for k when using the pigeonhole lemma idea (*e.g.*, as done by Weese *et al.* (2012)) or to require more shared k-mers when using the k-mer lemma idea (*e.g.*, as done by Weese *et al.* (2009)).

4.3.3 Filtration-Based Methods

String filter algorithms are algorithms for approximate string search that allow to efficiently *exclude* parts of the database that cannot contain a match (see also Navarro and Raffinot, 2002, p. 162–170). They follow a general two-step approach. First, in the *filtration step*, candidate regions that can contain a match are located. Depending on the chosen filtration method, both false positives and negatives can be yielded. In the second, the *verification step*, the candidate regions are verified with an exact algorithm.

The number of verifications in filtration-based methods depends on the specificity of the used filter, that is the percentage of candidate regions that actually contain a match. Most such methods use the pigeonhole principle or the k-mer lemma for filtration. That is, they create seeds from the reads that are then searched in the reference or use k-mer counting. In some cases, the methods also create seeds from the reference and then search this in the set of reads. In the following, a few examples for filtration-based read mappers are given.

RazerS (Weese *et al.*, 2009) is an example of a read mapping program based on k-mer counting. It uses the SWIFT filter algorithm (Rasmussen *et al.*, 2006) for identifying candidate regions and Myers' bit-vector algorithm for the verification (Myers, 1999b) of candidates. For Hamming distance, it uses optimized *gapped* k-mers (some fixed positions in the k-mers are allowed to mismatch). The successor RazerS 3 (Weese *et al.*, 2012) uses a filtration algorithm based on the pigeonhole lemma and a banded version of Myers' bit-vector algorithm (similar to the one by Hyyrö (2003)) for verification. Using a command line switch, the k-mer counting filter from the previous program version can be used in RazerS 3 as well. RazerS 3 is described in more detail in Section 4.4. Both versions allow Hamming and edit distance alignments of reads. Shrimp and its successor Shrimp 2 also employ a filter based on k-mer counting with spaced seeds following a generalization of the k-mer lemma by Burkhardt and Kärkkäinen (2001) to such seeds. Shrimp also employs a bit-parallel verification of candidate regions and allows edit distance matches.

Soap (Li *et al.*, 2008b) and MAQ (Li *et al.*, 2008a) are early examples of read mappers using string filtration based on the pigeonhole lemma. Soap and MAQ use the heuristic described above and use the 5' end of the read for the seed. In the filtration step, the first 28 bp at the 5' end of each read are used as the seed. Notably, MAQ introduced a statistical model for single-end read alignment and formulas for computing read mapping qualities that were subsequently used in later read mappers. Other examples for string-filtration–based read mappers are mrsFAST (Alkan *et al.*, 2009) and mrFAST (Hach *et al.*, 2010).

Lately, a number of methods have been proposed that use substring-tree–like indices for searching for seeds in the filtration step. The most prominent one is Bowtie 2 (Langmead and Salzberg, 2012) that uses the Burrows-Wheeler-Transform (BWT) of the reference (Burrows and Wheeler, 1994) and ideas by Ferragina and Manzini (2001) for creating a substring-tree–like index of the

reference. It then creates slightly overlapping seeds from the reads and searches them exactly in the reference. The candidate locations are then verified using the Needleman-Wunsch algorithm. Another example is Masai (Siragusa *et al.*, 2013) that uses *approximate* seeds (*i.e.*, allowing errors in the seeds). In contrast to many other substring tree–based methods, Masai indexes both the genome and the read set and then uses backtracking to search for matches in both indices. Another example of a string filtration–based read mapper is Gem (Marco-Sola *et al.*, 2012).

While in theory string filtration–based read mapping algorithms allow the fully sensitive search of reads in the reference, many read mapping methods sacrifice sensitivity for lower running time. Notably, RazerS and RazerS 3 are examples of fully sensitive methods that also allow the user to select a lower bound on the sensitivity. The algorithm implemented for mrFAST allows full sensitivity for edit distance (and mrsFAST for Hamming distance), but as can be seen below in Section 4.5, mrFAST is not fully sensitive in practice. Notably, almost all other string-filtration–based alignment tools are heuristics with varying trade-offs regarding sensitivity.

4.3.4 Index-Based Methods

The earliest read mapping tools were based on string filtration and streaming k-mers of the reference against a k-mer index built for the reads (or vice versa). The reference is usually updated rarely while many different read sets are to be mapped against it. This led to using substring indices of the reference to speed up read mapping. An index is built in a precomputation step, saved to the hard-drive, and reused for each program call. This development predates the substring index–based methods with string filtration, however.

Bowtie (Langmead *et al.*, 2009) is a tool based on the Burrows-Wheeler Transform (BWT, Burrows and Wheeler, 1994), and ideas by Ferragina and Manzini (2001) (the FM index). Targeted at short (*e.g.*, 36 bp) Illumina reads, Bowtie only allows mismatch errors and thus mimics the search strategy used in Maq. Instead of using different fixed patterns that are searched exactly, Bowtie performs a search in the virtual substring tree represented by the index and uses backtracking to allow a limited number of errors in the seed.

Bwa (Li and Durbin, 2009) is a popular read mapper for the alignment of short Illumina reads. Bwa uses a similar approach as Bowtie but also allows gapped alignments. Bwa-SW (Li and Durbin, 2010) uses the BWT for the alignment of long reads with a speeded-up version of the Smith-Waterman algorithm. Soap 2 (Li *et al.*, 2009c) is another BWT-based tool for read mapping that only supports Hamming distance.

4.4 RazerS 3 — Parallel, Fully Sensitive Read Mapping

The read mapper program RazerS 3 (Weese *et al.*, 2012) is the successor of RazerS (Weese *et al.*, 2009) and was developed by me together with my coauthors. The program RazerS was the first fully sensitive read mapper according to the formal definitions of read mapping in Section 4.1.4. The original aim for RazerS 3 was to parallelize the program RazerS to adapt it to

current multi-core architectures. However, during the development, it became clear to us that further performance improvements were required to also adapt it to longer Illumina reads (*e.g.*, 100–150 bp) and unmasked large genomes. My contributions lie mostly in the parallelization, the match processing, and the evaluation.

In this section, I will first describe the underlying algorithmic approach of both RazerS variants in Section 4.4.1. Then, I will outline the changes to the program for the parallelization in Section 4.4.2 and in Section 4.4.3 I will describe further optimization steps taken in RazerS 3. In Section 4.4.6 I present the evaluation of the achieved speedup in the evaluation of RazerS 3. I conclude this section by describing trade-offs of using different filtration and verification strategies in Section 4.4.5.

4.4.1 Algorithmic Overview

Figure 4.9 shows the data flow and the algorithmic components in RazerS and Algorithm 4.1 gives an overview of the RazerS program. First, the reads are loaded and an appropriate value for the k-mer length in the SWIFT filter is chosen and the SWIFT filter is configured with precomputed gapped shapes. The program then maps the reads to each contig and its reverse complement (for simplicity, Algorithm 4.1 only shows mapping to one contig).

In the end, the matches are *compacted* in the function Compactify. This function is also used during the mapping. The function first sorts the read alignment information (tuples of the read ID, begin and end position, and the alignment score) lexicographically by read ID, end position, and score. Then, depending on the configuration of the program, the sorted list of tuples is processed. For example, when only the best matches for a read are to be returned, all suboptimal alignments are removed and the filter is configured to be stricter for this read if possible. As another example, when there is a limit on the number of alignments to return for each read and this limit is reached with perfect alignments then the filter is configured to return no more hits for this read. Further, duplicate alignments are always removed.

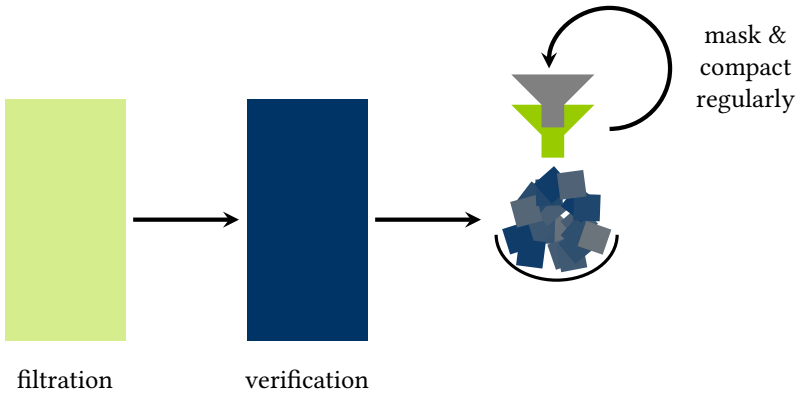


Figure 4.9: Overview of the data flow in the RazerS program.

Algorithm 4.1: The sequential RazerS algorithm.

```
Program RazerS(G, R)
    Input  : G, R                                  // genome and set of read sequences
    Output : A                                     // list of all read matches

    A ← Map-Reads(G, R)
    Reverse-Complement (G)
    A ← A · Map-Reads(G, R)
    Compactify (A)

    return A

Function Map-Reads(G, R)
    Input  : G, R                                  // genome and set of read sequences
    Output : A                                     // list of all read matches

    c ← 1024        // A is filtered and compactified if it has more elements
    A ← ⟨⟩

    // The filter hits m are intervals that candidate alignments' end
    // positions fall into.
    foreach filter hit m = (r, b, e) do                  // (read id, begin pos, end pos)
        // Verify returns one alignment for each match from Definition 4.7
        // whose end position falls into m.
        foreach m' = (r, b, e, s) ∈ Verify(m) do      // (read id, begin, end, score)
            A ← A · ⟨m'⟩
            if |A| > c then            // remove duplicate and superflous alignments
                Compactify (A)
                c ← c · 1.5
            end
        end
    end
```

The read mapping shown in the function Map-Reads in Algorithm 4.1 works as follows. The function builds a list of alignments A. Each SWIFT filter hit is an interval on the reference G that contains the end position of alignments for a read with identifier r. When configured for full sensitivity, the SWIFT filter provides the guarantee that it returns at least one interval for each match of a read. The filter hit is then verified in the function Verify. This function yields the begin and end position as well as the score for read alignments in the filter hit interval such that one alignment is created for each match of a read. In the implementation, the intervals are also annotated with whether they lie on the forward or reverse strand of the reference. This is not shown in Algorithm 4.1 for simplicity.

To save memory, the alignment list A is compacted if its length grows over a threshold c. This threshold grows by 50% after each compactification. This is a trade-off between allowing resizing of the list (which is implemented as an array) and compactification in amortized time $\mathcal{O}(n \cdot$

$\log n$) and not wasting too much memory. This compaction is necessary since the filter can yield multiple overlapping intervals for the alignment for a read. While the verification does not yield duplicate alignments in one of them, the verification of two or more such intervals can lead to duplicate alignments.

4.4.2 Parallelization Design

A naïve parallelization of RazerS with t threads could split the read set into blocks of $\lceil |\mathcal{R}|/t \rceil$ reads and let the algorithm run for each of these blocks independently. This, however, leads to problems with load balancing since the amount of work for each thread can differ greatly. The filter enumerates each k-mer of the genome and the necessary work at a genomic location depends on the frequency of the k-mer in the genome and the read. In this section, I discuss the constraints on the parallelization of RazerS 3 and the taken design decisions.

Reusing the Existing Implementation. Our first requirement was to reuse the existing sequential implementation. RazerS is a complex program and instead of rewriting the code from scratch, our aim was to base the parallel code on the existing implementation.

Restriction to One Reference Copy. To save memory, only one copy of the reference is held in memory. RazerS loads the reference contig by contig and reverse-complements the contigs instead of the reads. Reverse-complementing the reads would generate a larger overhead since usually, the set of reads is larger than the reference contigs. This leads to an implicit barrier for parallelism at the end of each contig and before reverse-complementing the current contig: all threads have to wait for the last thread to finish its processing of the contig.

Load Balancing in Filtration. The filter has to build a hash table of all k-mers for its reads. Moving reads from one hash table to another is expensive and either requires a complex implementation of the table (which lowers performance when reading) or rebuilding the table. Having more than one table for each thread also introduces an overhead. For the original implementation of RazerS 3, the number of tables for each thread was adjustable but assigning the reads statically to threads and only having one hash table was faster. Thus, we decided that the load balancing in the filtration was to be static.

Load Balancing in Verification. Conversely, we decided that the verification of the filter hits was to be used for load balancing between the threads. We adjusted the filtration to process the reference in windows of 500 kbp (by default) and to collect all filter hits. The collected filter hits are then split into packages and put into a thread-safe global queue. The threads then pick these packages and process them independently of each other. This allows the dynamic load balancing in verification.

4.4.3 Further Optimizations in RazerS 3

Besides the parallelization, we implemented the following optimizations in RazerS 3.

Open-Addressing k-mer Filter. In RazerS, a dense implementation of a k-mer index was used. Each k-mer of the alphabet Dna5 is assigned a number between 0 and 5^q. For small values of q, it is possible to create a hash table that uses this direct correspondence without collisions or probing. However, for larger values of q, such a hash table requires a large amount of memory. For RazerS 3, we implemented a k-mer index–based on an open addressing (Cormen *et al.*, 2001, p. 221 ff.) hash table.

A Banded Version of Myers' Bit-Vector Algorithm. Myers' bit-vector algorithm allows the bit-parallel computation of edit distance matches for query sequences up to the machine's word width (*i.e.*, 64 bit for modern machines). For 36 bp Illumina reads, which was the main target for RazerS, this worked very well. However, for modern 100–150 bp Illumina or even 454 reads, more than one machine word is used and bits have to be carried over machine word boundaries. For this, we developed a banded version of Myers' bit-vector algorithm.

A Pigeonhole-Based Filter as an Alterative to the SWIFT Filter. As an alternative to the SWIFT string filter, RazerS 3 implements a string filter–based on the pigeonhole principle: when finding a shared k-mer between a read an the reference, the filter hit is verified. As I describe in Section 4.4.5 below, the SWIFT filter is a good choice for long reads and high error rates in the case of Hamming distance. However, in the other cases (in particular for edit distance), the pigeonhole filter showed a better performance (when used together with the banded version of Myers' bit-vector algorithm).

Local Match Masking and Histogram-Based Setting of Thresholds. Both the SWIFT and the pigeonhole filter work by enumerating the k-mers in the reference sequentially from left to right. Although the position of these k-mers is strongly monotonically increasing, the position of the k-mer in the read can vary. Thus, neither the begin nor end position of the alignments after verification are guaranteed to be monotonically increasing. For example, an alignment can have a smaller begin position than the previous alignment for a read.

This non-monotonicity is limited, however, to some relatively small overlap. That is, after processing a window, the leftmost begin position of an alignment that can possibly be generated by the next window is limited by a bound that can be computed from the filter settings. Thus, the sorting for the masking step does not have to be performed on all matches. It is sufficient to perform the masking on a suffix of the list of found alignments:

First, all matches from the first window (the right border is shifted to the left by the overlap) are sorted by start, then by end position (as described above) and masked after each sorting. Then, all matches from the second window (the left border is shifted left by the overlap) are sorted

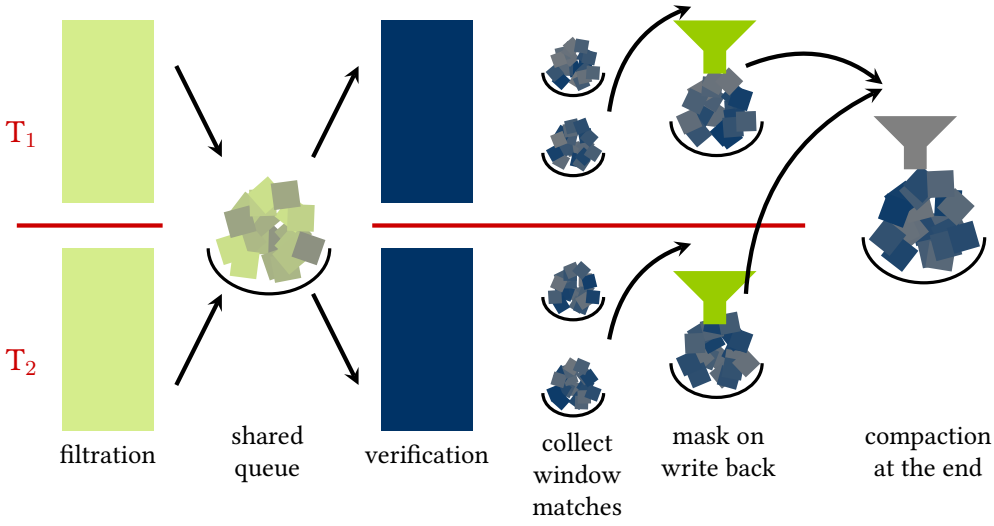


Figure 4.10: Overview of the data flow in the RazerS 3 program for two threads. The filter module and its hit packages are shown in green. The verification module and the packages of alignments are shown in blue.

by start, then by end position and masked after each sorting. This is repeated for all following windows.

This allows to manage the arrays in memory-mapped files and only access a relatively small part at the end of the file. Only this active part has to be kept in memory and the operating system will swap out the rest of the file if necessary. This strongly reduces the memory requirements of the program and makes them independent of the total number of matches. Rather, the memory required for matches is now bounded by the maximal number of matches of all reads that fall into one window. Note that the operating system will buffer as much of the file as possible in main memory for better performance. The sorting of the final compaction step can be done using external sorting for large result sets.

In the previous version of RazerS, the whole set of matches was sorted and masked for duplicates. The method also allows to specify a maximal number of matches for each read. When this number is reached for a read, the filter settings can be adjusted to return only matches that are better than the worst match found so far. In the previous version, the distribution of the error rate for each read could be computed during the masking step since all matches in A are available to the algorithm adjusting the thresholds. For the local match masking, I use the simple trick of storing a histogram of match distances for each read. The decision of adjusting the thresholds can be made after considering only the entries of the histogram and without having access to all matches.

Algorithm 4.2: The parallel RazerS 3 algorithm.

Program RazerS 3$(G, \mathcal{R})$
- **Input** : $G, \mathcal{R}$ // genome and set of read sequences
- **Output** : A // list of all read matches

$A \leftarrow$ Map-Reads$(G, \mathcal{R})$
Reverse-Complement (G)
$A \leftarrow A \cdot$ Map-Reads$(G, \mathcal{R})$
Compactify (A)
return A

Function Map-Reads$(G, \mathcal{R})$
- **Input** : $G, \mathcal{R}$ // genome and set of read sequences
- **Output** : A // list of all read matches

$Q \leftarrow \langle\rangle$ // thread-safe global queue of filter hits
fork threads and do in parallel
- $A \leftarrow \langle\rangle$ // each thread has its own copy of A
- // window length and package size are configurable
- **foreach** window W of length 50 kbp **do**
  - $Q' \leftarrow$ filter hits for W in packages of 10 k hits
  - $Q \leftarrow Q \cdot Q'$ // append atomically
  - $B \leftarrow \emptyset$ // mapping thread id $\rightarrow$ alignments
  - // While there is any work in Q or any thread can insert into Q.
  - **while** $(|Q| > 0) \vee$ (Num-Filtering-Threads() > 0) **do**
    - $p \leftarrow$ Pop(Q) // get next package atomically
    - **foreach** $m \in p$ **do**
      - $m' \leftarrow$ Verify(m)
      - $i \leftarrow$ Get-Thread(m) // id of thread for read in match
      - $B[i] \leftarrow B[i] \cup \{m'\}$
    - **end**
  - **end**
  - // match distribution
  - **thread-barrier** // wait for all threads
  - $C \leftarrow$ all matches from other thread's B for this thread
  - **thread-barrier** // wait for all threads
  - // masking
  - $A \leftarrow A \cdot C$
  - Mask-Matches(A)
  - **thread-barrier** // wait for all threads
- **end**

end
merge thread-local As into a global A
return A

filter	Myers variant	threads	avg. filtration time	avg. verification time	total time	candidates	matches
SWIFT	unbanded	1	14:14:04	34:33	14:49:06	489 M	29 M
SWIFT	banded	1	13:42:41	10:08	13:53:19	489 M	24 M
pigeonhole	unbanded	1	12:28	5:10:45	5:23:41	5 490 M	24 M
pigeonhole	banded	1	12:25	56:41	1:09:34	5 490 M	24 M
SWIFT	unbanded	8	1:01:13	4:27	1:05:47	489 M	29 M
SWIFT	banded	8	1:04:25	1:17	1:11:09	489 M	29 M
pigeonhole	unbanded	8	2:31	39:22	41:59	5 490 M	24 M
pigeonhole	banded	8	2:36	6:41	9:24	5 490 M	24 M

Table 4.2: Running time of the RazerS 3 mapping step in different variants, excluding I/O. The time is given as [h:min:s]. Note that the column "total time" also includes the I/O of references as well as the running time for match management.

4.4.4 The Parallelization in RazerS 3

Figure 4.10 gives an overview of the program RazerS 3 and shows the data flow between the threads, and Algorithm 4.2 gives pseudo code. Each thread is statically assigned approximately the same number of reads and builds the filtration state (including the hash table) for them. Then, each thread runs the filtration for its reads over the next window. The filter hits are collected in packages of a few thousand and written to a thread-safe shared queue.

After finishing the filtration, each thread then performs the verifications, using the packages from the shared queue. The resulting alignments are collected in local buckets, each of the t threads has t buckets, one for each other thread and itself. The read ID allows the mapping to the thread that is responsible for a given read. The alignments are written to the bucket for the responsible thread.

After the verification for the current window is complete, the locally collected alignments are written to a thread-local queue for each thread. Before writing back, the alignments are *masked*, *i.e.*, duplicates are removed and the threshold for each read is adjusted depending on the number of alignments. This is done in the local fashion described in Section 4.4.3.

When the whole genome has been processed, the alignments from all threads are collected and *compaction* is performed, *i.e.*, alignments that are not to be written out are removed. This can be caused by reaching a limit on the number of alignments for a read or by having found a better match in All-Best mode, for example.

4.4.5 Filtration and Verification Performance Trade-Offs

The effects of the filter and verification algorithm choice are complex and also interact with whether multi-thread is used or not. In this section, I describe and discuss these effects.

Influence of Banded Verification. Table 4.2 shows the total mapping time, average filtration time, and average verification time when running Razers 3 with one and eight threads with different combinations of filter and verification variants. The number of verifications (candidates) and successful verifications (matches) is also shown. The upper half of the table shows the results when RazerS 3 is run with one thread. I mapped ten million 100 bp reads from read set ERR012100 to chr. 2 of the human genome on an 8-core machine. I ran RazerS 3 with 99% sensitivity (the default setting) and an error rate of 4%.

When using one thread, the verification time using SWIFT and unbanded Myers is 34.5 minutes while it is 10.13 minutes when using banded Myers instead. When using the pigeonhole filter, the time decreases from 310.75 minutes to 69.56 minutes. Thus, the speedup of using banded Myers versus unbanded Myers is 3.4 for SWIFT and 4.47 for pigeonhole.

When using eight threads, the pigeonhole/unbanded variant uses 39.37 minutes for verification, pigeonhole/banded uses 9.3 minutes, a speedup of 4.2. The SWIFT/unbanded variant uses 4.45 minutes, SWIFT/banded uses 1.28 minutes, a speedup of 3.47.

Interestingly, the total mapping time does not decrease as greatly for SWIFT when changing from banded to unbanded while there is pronounced a speedup for pigeonhole. Thus, the influence of the pigeonhole filter has to be considered as well. I give a description of this in the next paragraph.

Influence of the Pigeonhole Filter. Using the SWIFT filter identifies 489 M candidates while the pigeonhole filter identifies 5 490 M candidates, more than 11 times as many (see Table 4.2). However, using the pigeonhole filter, RazerS 3 is faster than when using the SWIFT filter. Depending on whether using one or eight threads and banded or unbanded verification, the difference in mapping time is a factor between 1.56 (eight threads, unbanded) and 11.98 (one thread, banded). The factor is 2.75 for one thread, unbanded and 7.56 for eight threads, banded.

When changing the filter from SWIFT to pigeonhole, the filtration time decreases greatly at the cost of more candidates and thus more time is spent in verification. The time spent in verification can be greatly reduced by using the banded instead of the unbanded verification algorithm.

Another interesting effect of the faster verification phase is the impact on load balancing. I instrumented the RazerS 3 code to record the start and end time of each step and thus also the times where a thread waits for others. Figure 4.11 shows excerpts from the time charts of the first two threads from each run towards the end of the run. The x-axis indicates the time, where all charts have the same time scale. Each green bar represents the time spent for filtering candidate regions between an eighth of all reads and a reference window of length 500 kbp. The blue bars represent the time spent for verification (which is load balanced in packages between the blocks). Colored parts of the bar indicate that the thread was actively doing work whereas white parts indicate waiting.

As can be seen, when using the SWIFT filter the running time is dominated by filtration. Since filtration is not load balanced dynamically, multiple threads are idle when they have completed their filtration. When running with the pigeonhole filter, the running time is dominated by verification. Verification can be load balanced dynamically and consequently, the threads finish

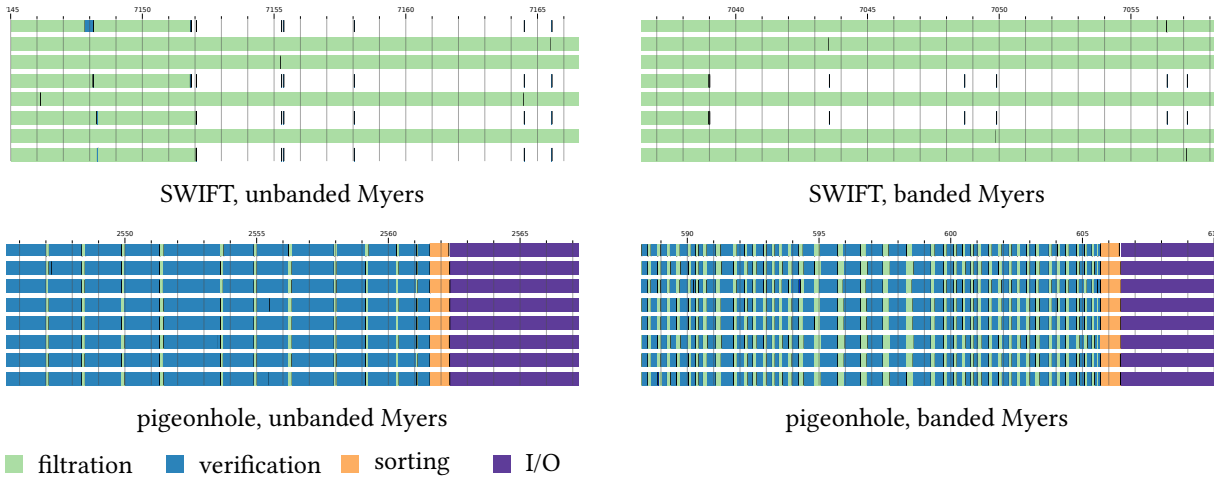


Figure 4.11: Running time profiles for RazerS 3 with different variants for the filtration and verification phase. The charts show the work performed by each of the 8 threads towards the end of the program run. In pigeonhole mode, all threads finish almost simultaneously because the time spent for verification dominates the running time and the verification can be load balanced dynamically. In SWIFT mode, many threads are idle from some point of time since the filtration time dominates and filtration is load balanced only statically. Note that while the charts all have the same scale and show an interval of the same length, they all show an interval starting at different points of time. For the pigeonhole filter case, the times are taken around the end of the mapping phase. In the case of the SWIFT filter this was not possible on this scale since the threads finish so irregularly.

almost simultaneously. This can be seen in the two charts on the bottom which show the last 25 s of the pigeonhole runs. Furthermore, it can be observed that the time spent on verification is shorter when using banded verification but the remaining verification work still allows good load balancing in terms of running time.

To summarize, both switching from unbanded to banded verification and from the SWIFT to the pigeonhole filter improves the running time. When using the pigeonhole filter, the improvement from unbanded to banded verification is even more pronounced (5x instead of 3.5x) because the lower specificity of the pigeonhole filter yields 10x more candidates. When using multiple threads, the shift of time spent in filtration to time spent in verification improves the overall running time by improved load balancing. The reason for this is that the verification can be load balanced while the filtration cannot.

Comparing Pigeonhole and SWIFT Filter. The only disadvantage of the banded verification compared to the unbanded variant is that the band size is limited by the machine word width in our implementation because of performance considerations. Thus, on a 64 bit machine, the verification is limited to a bandwidth of 64 and can only guarantee to find alignments with 32 errors (for an error rate of 5%, this allows reads of up to 650 bp). The banded verification is faster than the unbanded variant and also does not need the precomputation required for the unbanded

variant of Myers' bit-vector algorithm. Thus, for Illumina reads, it is always preferable to use the banded verification.

However, it is not clear for which cases the SWIFT or the pigeonhole filter is better. Thus, I performed several benchmark runs. I used RazerS 3 to map ten million Illumina reads from the organisms *E. coli*, *C. elegans*, and human to the organism's reference sequence (for human, chr. 2 was used as the reference). I varied the error rate from 0–10%, for sensitivity, I used 99% and 100%, and I used both Hamming and edit distance for mapping. I ran the benchmarks on an 8-core machine and using 8 threads. Figure 4.12 shows the results and Table E.1 (p. 199) shows the names and details of the read data sets.

Generally, SWIFT shows its largest advantage using gapless (Hamming) alignment for error rates over 6%. For edit distance and error rates smaller than 6%, the pigeonhole filter is faster. It is interesting to note that there are running time ratios of 1:32 or more for both variants, *i.e.*, for both filters there are settings in which one greatly outperforms the other.

It is also remarkable that in some cases the superiority of one method is not continuous. For example, for edit distance on *C. elegans* with 100% sensitivity and a read length of 50 bp, the SWIFT filter is slightly better than the pigeonhole filter for error rates of 4–5%. A possible explanation for this is that the SWIFT filter uses precomputed gapped shapes and counter threshold values. For some settings, good shapes can be found while for neighboring configurations, the found settings might be worse. For Hamming distance the SWIFT filter uses gapped shapes, which gives it an edge over the pigeonhole filter. Thus, the SWIFT filter tends to be better than the pigeonhole filter for high error rates when mapping in Hamming distance.

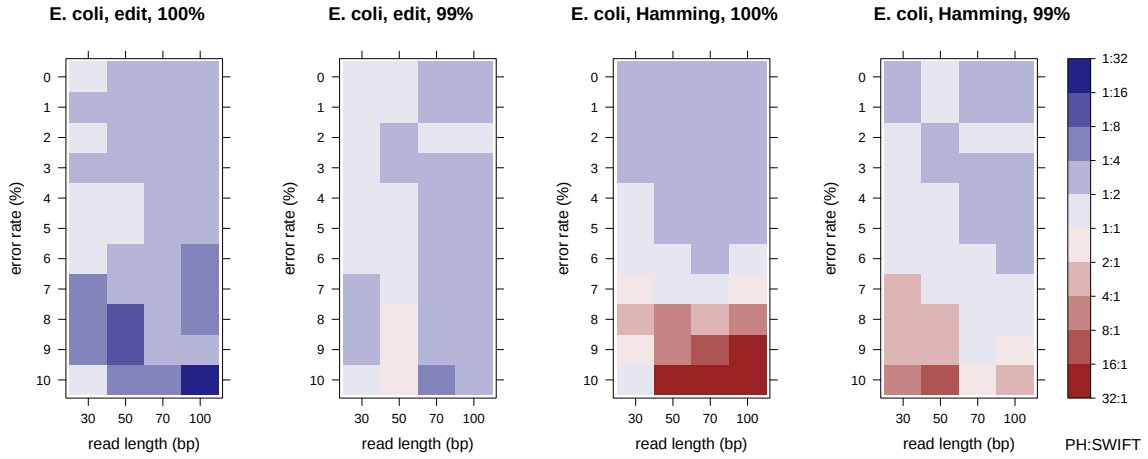
Our general recommendation is to use the pigeonhole filter for edit distance for all error rates and for Hamming distance for error rates up to 6%. The SWIFT filter is a good choice for Hamming distance with error rates above 6%.

4.4.6 Evaluation of the Parallelization

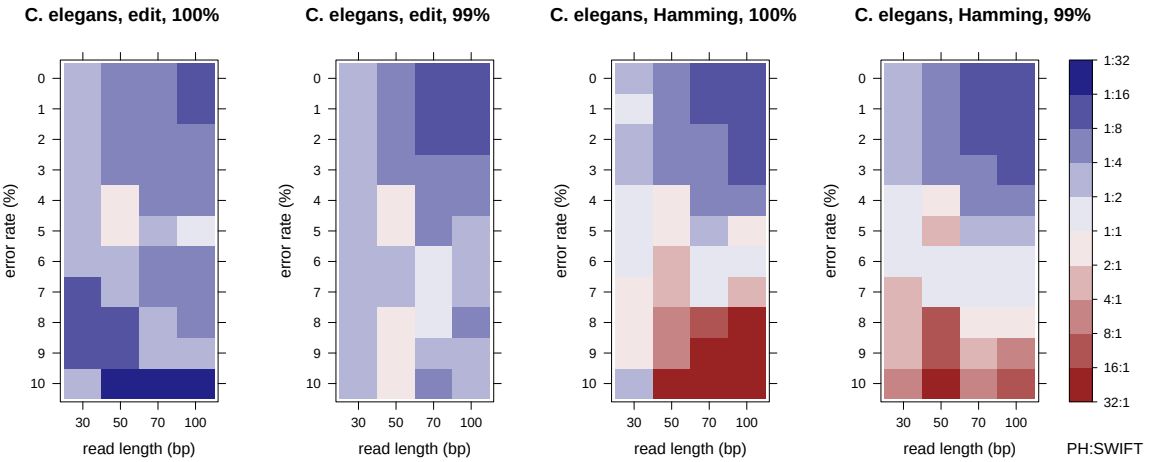
In this section, I analyze the parallelization of RazerS 3 in detail.

Achieved Speedup. To evaluate how much the implementation benefits from widely available multi-core architectures, I used a relatively large dataset (10 M reads of set ERR012100) and mapped it against chr. 2 of the human genome. I ran RazerS 3 with 1, 2, 4, and 8 threads (*dynamic* load balancing). I compared the results with the trivial parallelization (*static* load balancing) of splitting the read set into t parts of equal size and running t separate RazerS 3 processes in parallel that use one thread each.

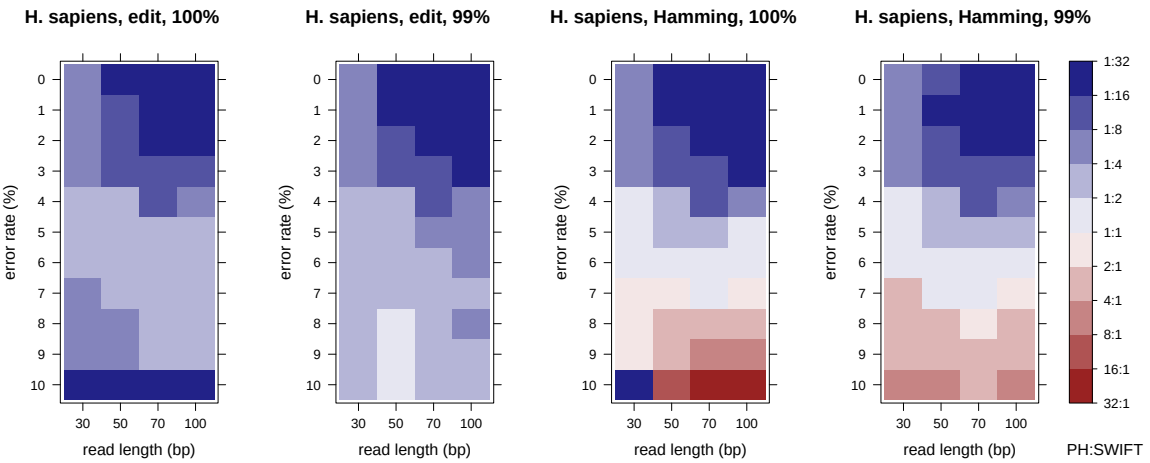
Both the runs with dynamic and static load balancing required about 89.5 min with one thread. Mapping reads with dynamic load balancing scaled almost linearly with speedups of 1.95, 3.95, and 7.46 for 2, 4, and 8 threads. Static load balancing scaled worse. The speedups were 1.90, 3.63, and 6.61. With 8 threads one effectively gains one more processor core with the dynamic balancing scheme compared to the static load balancing.



(a) running time ratios for *E. coli*



(b) running time ratios for *C. elegans*



(c) running time ratios for human chr. 2

Figure 4.12: Ratios of running time with pigeonhole (PH) and SWIFT filter. RazerS 3 was run with 8 threads using reads of different length and different genomes as well as different error rates and sensitivities.

Alternative Load-Balancing Methods. Above, I compared static and dynamic load balancing schemes. Another possible load balancing scheme would split the read set into many packages which are then distributed through a queue to a group of worker threads. Each thread processes one package in a sequential fashion and the next package is given to the earliest idle thread.

This approach has a major drawback. The filtration step of RazerS 3 has to scan over the text for each package. While the time for verification is linear in the number of matches and the number of verifications does not change, this is not the case for the filtration: doubling the number of packages, leads to an increase by a factor of 1.7–1.8 for the total CPU time spent in filtration in the performed experiments.

I mapped ten million reads against the *D. melanogaster* genome. For this, I split the reads into 8, 16, 32, and 40 packages. I then mapped each package using a separate RazerS 3 process. I employed the work queue program TS[1] for running the mapping processes such that 8 were run in parallel. The CPU time spent for filtering was 216 s, 388 s, 694 s, and 1025 s using a splitting scheme of 8, 16, 32, and 40 packages. When using dynamic load balancing, the time for filtration was 243 s (the difference to the time for 8 packages can be explained by the better isolation and fewer synchronizations using separate processes).

When using 8 packages, the load balancing is the same as for the static scheme described above. While there is no additional overhead for the filtration, the load balancing is inferior to the dynamic load balancing implemented in RazerS 3. Using more packages, the overhead for the additional filtration makes this approach even worse than static load balancing.

In conclusion, I found the employed load balancing strategy to be superior to the more naïve approaches. Thus, the work invested in the fairly complex parallelization paid off.

4.5 Rabema and RazerS 3 Results

In this section, I present an experimental study comparing RazerS 3 with other read mappers and its results. Together with my coauthors, I previously published the study in (Weese *et al.*, 2012).

We chose Bowtie 2 (Langmead and Salzberg, 2012) and BWA (Li and Durbin, 2009) as representative of recent BWT-based read mappers. Soap 2 (Li *et al.*, 2009c) is another popular BWT-based tool although it does not support indel errors. Hobbes (Ahmadi *et al.*, 2012), mrFAST (Alkan *et al.*, 2009), and Shrimp 2 (David *et al.*, 2011) are popular read mappers based on k-mer counting or k-mer indices. We used RazerS 3 (Weese *et al.*, 2012) with its pigeonhole filter and the banded verification. RazerS 3 has a parameter for setting the estimated sensitivity and we used the sensitivities 100% and 95% (*R3-100* and *R3-95*).

For the read sets, I give the SRA/ENA identifiers. As reference sequences, we used the whole genomes of *E. coli* (NCBI NC_000913.2), *C. elegans* (WormBase WS195), *D. melanogaster* (Fly-

[1] `http://vicerveza.homeunix.net/~viric/soft/ts/`

Base release 5.42), and *H. sapiens* (GRCh37.p2). We measured the running times on a cluster of nodes with 72 GB RAM and 2 Intel Xeon X5650 processors (each with 6 cores) per node running Linux 3.2.0.

4.5.1 Evaluation Methods

Generally, one would expect the BWT-based tools to be best at locating one or a few matches with a low number of errors and to perform well for solving the Any-Best-Mapping and All-Best-Mapping problems. At the same time, these methods are expected to be fast for this task. On the other hand, k-mer-based methods can be expected to be very sensitive for all three mapping problems, and especially outperform the BWT-based tools for the All-Mapping problem. However, these methods are expected to be slower, given that they can generate large amounts of data (depending on the parameters and input).

For this reason, we group the read mappers into *best-mappers* and *all-mappers*. That is, programs designed and parameterized to either quickly find one best match for each read or to enumerate all matches for a read up to a given error rate. The BWT-based tools are *best-mappers* while the k-mer-based methods are *all-mappers*. RazerS 3 is also considered as a *best-mapper* here (and placed in both categories), using a parameterization that returns only one best alignment for each read. The exact parameterizations can be found in Appendix D.

Benchmarking with Rabema. We used Mason (Holtgrewe, 2010) for the simulation of 100k reads of length 100 bp from the whole human genome. We chose a typical Illumina error profile for simulating sequencing errors. As in the Rabema (version 1.1) manual, we used the program RazerS 3 for generating a fully sensitive mapping. From this, we generated a gold standard intervals file that we used for the subsequent evaluation. We generated the gold standard for up to 5% errors.

Then, we used Rabema to perform an evaluation in the categories All-Mapping, All-Best-Mapping and Any-Best-Mapping. We report the result of the evaluation in the oracle mode as *recall*, *i.e.*, the percentage of reads that could be aligned to the simulated sample location. We did not consider running time and memory consumption in this benchmark, since it is only meant to assess the sensitivity of the read mappers in terms of the Rabema benchmark.

For the Rabema benchmark, we also tried to configure Bowtie 2 and BWA as *all-mappers*. We did this to see how well the BWT-based methods work for enumerating all matches (see Appendix D).

Variant Detection Benchmark. Similarly to the evaluation of SHRiMP 2 (David *et al.*, 2011), we subjected the read mappers to an analysis of their performance in the presence of small variants, *i.e.*, SNVs and small indels. The BWT-based tools Bowtie 2 and BWA use quality values for trying to discriminate tentative errors from correct bases. Small variants introduce alignment errors but without lowering the base quality values. Thus, these read mappers could prefer alignments at wrong locations if mismatches at such locations can be explained by low base quality values.

We simulated read pairs of length 2×100 bp from the whole human genome. We simulated SNVs and small indels into the genome and the reads also contain simulated sequencing errors following a typical Illumina error profile. We used MASON for simulating these reads as well. We filtered the reads such that they aligned with up to 5 errors to the reference sequence and we randomly selected 5 million from the filtered read set.

Then, we mapped read sets to the reference both in single-end and paired-end mode to assess the performance of all read mappers and parameterizations in both modes. We count a read (or pair) as aligned *correctly* if we find an alignment that falls within 10 bp of the sampled origin. We count it as *mapping uniquely* if the mapper only reports one alignment. We group the reads (or pairs) by the number of SNVs and indels spanned in the simulated donor genome. We label each group with the pair (s, i), having s SNVs and i indels. For each group, we compute *recall* as the percentage of all contained reads (or pairs) and *precision* is the percentage of uniquely mapped reads (or pairs) that we found to be mapped correctly.

Performance Comparison Benchmark. Last, we compare the performance of the read mappers in terms of mapped reads as well as running time and memory consumption. For this, we used four different sets of ten million real-world Illumina read pairs of length 2×100 bp for the organisms *E. coli*, *C. elegans*, *D. melanogaster*, and *H. sapiens*. Further, to assess the read mappers' performances on future long reads, we mapped six simulated datasets consisting of 1 million simulated read pairs of lengths 2×200 bp, 2×400 bp, and 2×800 bp from *D. melanogaster* and *H. sapiens* to the reference genomes. We simulated the reads with a typical Illumina error profile using MASON. We mapped all reads both in single-end and paired-end mode with an error rate of 4%. We measured the running time and peak memory consumption as well as the number of mapped reads (or pairs) and read (or pairs) with minimal edit distance.

For the evaluation, we used the common measures of percentage of mapped reads (or pairs), *i.e.*, the fraction of reads (or pairs) that are reported as aligned in the result file of the mapper. However, as some mappers report alignments without constraints on the number of errors, we determined the fraction of reads (or pairs) whose best match has an error rate of at most $0\%, \ldots, 4\%$. We call a read (or pair) ε-*mappable* if it can be aligned with an error rate of ε (by any mapper). As a more stringent measure for edit distance mappers, we call an ε-mappable read (or pair) *correctly* mapped if at least one (paired) alignment has been found with an error rate of ε. For each mapper, we measured the percentage of correctly mapped reads (or pairs), *i.e.*, the fraction of ε-mappable reads (or pairs) for $\varepsilon \in [0, 4\%]$ that are correctly mapped. For a more detailed analysis, we give the percentages for sets of $\varepsilon = 0, \varepsilon \in (0, 1\%], \ldots, \varepsilon \in (3, 4\%]$ separately.

When possible, we ran the read mappers with 12 threads. Since MRFAST does not support shared-memory parallelization, we split the reads into packages of 500 k reads and then mapped them with 12 concurrent processes using TS[2]. Hobbes had a large memory consumption such that we had to map the reads package-wise but otherwise used the built-in parallelism and 12 threads.

[2] `http://vicerveza.homeunix.net/~viric/soft/ts/`

	method	all				all-best				any-best				recall			
best-mappers	Bowtie 2	92.04	99.18	98.72	96.80	96.16	97.79	97.85	95.80	98.08	100.00	99.96	97.55	95.94	98.01	97.72	95.55
			93.44	81.94	40.19		94.83	93.37	88.86		96.62	94.93	90.46		94.24	92.79	89.52
	BWA	92.18	99.18	98.72	97.81	96.81	97.79	97.87	97.88	98.81	100.00	99.95	99.81	96.41	97.93	97.69	97.25
			94.25	80.92	37.65		96.59	92.63	83.47		98.55	94.28	85.37		95.77	91.98	84.61
	Soap 2	65.93	99.18	95.55	91.34	69.89	97.79	94.74	91.37	71.37	100.00	96.78	93.18	69.91	98.05	94.62	91.20
			8.67	0.70	0.00		8.98	0.79	0.00		9.21	0.81	0.00		11.85	1.41	0.36
	R3-100	93.30	99.18	98.73	97.93	97.96	97.79	97.88	98.03	100.00	100.00	100.00	100.00	97.80	98.00	97.85	97.75
			95.60	85.81	44.15		98.00	98.27	97.93		100.00	100.00	100.00		97.65	97.70	97.69
	R3-95	93.10	99.18	98.73	97.93	97.75	97.79	97.88	98.03	99.79	100.00	100.00	100.00	97.60	98.03	97.85	97.74
			95.49	84.76	42.82		97.88	97.03	94.97		99.89	98.74	97.00		97.52	96.56	94.99
all-mappers	Bowtie 2	95.69	99.98	99.91	99.45	98.85	99.74	99.79	98.61	99.16	100.00	99.98	99.01	98.54	99.74	99.58	98.27
			97.99	90.69	55.14		98.21	97.55	93.84		98.63	97.94	94.17		97.64	96.87	94.40
	BWA	95.89	99.96	99.88	99.49	97.98	98.81	99.01	99.02	98.82	100.00	99.95	99.82	97.80	99.03	98.96	98.75
			97.13	87.79	64.11		97.83	93.95	85.20		98.56	94.34	85.37		97.35	93.43	86.36
	Hobbes	96.56	99.41	99.00	98.76	97.08	97.23	96.59	97.01	98.01	97.92	97.51	97.96	96.41	95.49	95.84	96.54
			97.80	93.20	73.05		97.38	98.16	97.42		98.43	99.12	98.46		97.03	97.98	97.79
	mrFAST	99.97	100.00	100.00	100.00	99.97	100.00	100.00	100.00	99.97	100.00	100.00	100.00	99.97	100.00	100.00	100.00
			100.00	99.99	99.53		100.00	100.00	99.10		100.00	100.00	99.13		99.99	100.00	99.18
	SHRiMP 2	96.53	99.87	99.82	99.53	99.50	99.34	99.50	99.60	99.85	99.87	99.90	99.91	99.25	99.35	99.30	99.24
			98.37	92.58	64.63		99.64	99.65	98.32		99.89	99.84	98.57		99.30	99.09	98.48
	R3-100	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
			100.00	100.00	100.00		100.00	100.00	100.00		100.00	100.00	100.00		100.00	100.00	100.00
	R3-95	99.54	100.00	100.00	100.00	99.79	100.00	100.00	100.00	99.79	100.00	100.00	100.00	99.79	100.00	100.00	100.00
			99.89	98.67	95.11		99.89	98.71	96.96		99.89	98.74	97.00		99.89	98.77	97.17

Table 4.3: Rabema scores in percent (sum of the fractions of found matches for each read, normalized by the number of reads). Large numbers are the total scores in each Rabema category and small numbers show the category scores separately for reads with $\left(\begin{smallmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{smallmatrix}\right)$ errors.

4.5.2 Results and Discussion

Rabema Benchmark Results. Figure 4.3 shows our results of the Rabema benchmark. The first five rows of the table show the results for the best-mappers, the remaining rows show the results for the all-mappers. The large numbers show the overall normalized found intervals score and the small numbers show the score for alignments having $0, 1, \ldots, 5$ errors. The results of R3-100 can be seen as the gold standard since RazerS 3 guarantees to find one best or all best alignments, depending on its configuration.

The most relevant column for the best-mappers are the columns *any-best* and *recall*. First, consider the column *any-best*. As expected, R3-100 is fully sensitive and yields a score of 100.0. R3-95 is the next best method, achieving an overall score of 99.8% and scores $\geq 97.0\%$ when considering the scores by error. BWA and Bowtie 2 perform similarly, yielding an overall score of 98.81% and 98.08. Notably, BWA achieves better scores for alignments 0–3 errors while Bowtie 2 is better for alignments of 4 and 5 errors. Soap 2 achieves the lowest score of all best-mappers and aligns few reads with more than 2 errors. Second, consider the column *recall*. The RazerS 3 variants achieve the best scores in term of recall as well as in the case of the any-best category, BWA is slightly better than Bowtie 2. Soap 2 achieves the worst results.

The most relevant column for the all-mappers are the columns *all*, *all-best* and *recall*. Again, R3-100 comes out with full sensitivity scores as is to be expected. In the category *all*, it is followed by mrFAST which claims full sensitivity but loses some alignments presumably due to implementation bugs. The next best method is R3-100, followed by Hobbes and Shrimp 2. Notably,

Bowtie 2 and Bwa achieve quite good results in their all-mapper configuration, at the cost of higher running times. All methods except R3-100 lose most significantly in the matches with higher error rate. In the category *all-best*, the picture is quite similar but the achieved scores are higher than in the category all. For the categories *all-best*, *any-best*, and *recall*, Bowtie 2 is better than Bwa (both in their all-mapper configuration).

To summarize, the sensitive all-mapper RazerS 3 achieves good results in single-end best mapping and surprisingly also in terms of recall despite its lack of base quality support. Bwa is slightly better than Bowtie 2 in terms of any-best mapping and Soap 2 is only a good choice when alignments with more than two errors are not important. The category *any-best* mapping is very important because almost all common tools for small variant detection expect their input to be read alignments from an any-best mapper. For all-mapping, RazerS 3 shows the highest scores in all categories when configured with full sensitivity and it is still competitive or superior to all-mappers such as Hobbes and Shrimp 2 when configured with 95% sensitivity. mrFAST is the second best all-mapper but does not show full sensitivity. Surprisingly, Bowtie 2 and Bwa can be configured to outperform the all-mapper Hobbes in the category *all-mapping*.

Variant Detection Benchmark Results. Table 4.4 (p. 81) shows our results of the variant detection benchmark. Extended results are given in Tables E.2 (p. 200) and E.3 (p. 201).

Considering the all-mapper results, R3-100 shows the highest recall and precision value, both in single-end and paired-end mode. mrFAST is also fully sensitive on the single-end dataset but shows a low recall value of 8% pairs with 5 bp indels. Shrimp 2 shows full precision in all classes and experiments but misses some non-unique alignments. Hobbes appears to have problems with indels and shows the lowest sensitivities in the all-mapper comparison.

Surprisingly, R3-100 is the most sensitive best-mapper even in the non-variant class $(0, 0)$ where the simulated qualities should give quality-based mappers an advantage. For paired-end reads where matches are also ranked by their deviation from the template size, it is even more sensitive than the all-mappers Hobbes and mrFAST. As also observed by David *et al.* (2011), quality-based mappers like Bowtie 2, BWA, and Soap 2 are not suited to reliably detect the origin of reads with variants. Their recall values deteriorate with more variants as they prefer alignments where mismatches can be explained by sequencing errors instead of natural sequence variants. The low sensitivities of Soap 2 are due to its limitation to at most 3 mismatches and no support for indels.

Performance Comparison Benchmark Results. Table 4.5 shows our results of the real-world performance benchmark for the real-world *D. melanogaster* and *H. sapiens* data sets as well as for the simulated 800 bp *D. melanogaster* data set. Extended results can be found in Tables E.4–E.7 (p. 202–205).

For the single-end data, R3-100 achieves the best results in terms of correctly mapped reads of all best-mappers, followed by R3-95. Bowtie 2 is the next best tool, followed by BWA. Remarkably, the performance of BWA deteriorates for the long reads which can be explained by the fact that BWA is not designed for such long reads. Soap 2 achieves the worst results, aligning few reads with more than 2% errors. R3-100 also is the best tool of the all-mappers. mrFAST is the

		(0,0)		(2,0)		(4,0)		(1,1)		(1,2)		(0,3)	
	method	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	97.6	97.3	94.6	92.0	92.6	82.5	95.3	93.3	93.5	92.3	96.1	95.4
	BWA	98.2	97.9	97.6	95.3	94.9	85.1	97.4	90.9	97.1	80.3	96.3	66.5
	Soap 2	98.1	82.9	97.4	31.0	0.0	0.0	90.6	6.2	0.0	0.0	0.0	0.0
	R3-100	98.4	98.4	98.2	98.2	96.3	96.3	98.1	98.1	97.9	97.9	97.6	97.6
	R3-95	98.4	98.3	98.2	97.3	96.1	91.7	98.2	97.6	97.9	97.6	97.5	97.5
all-mappers	Hobbes	99.9	99.9	99.9	99.9	100.0	100.0	100.0	99.8	100.0	93.6	99.6	90.5
	mrFAST	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	SHRiMP 2	100.0	99.4	100.0	99.7	100.0	99.7	100.0	99.5	100.0	99.2	100.0	99.6
	R3-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R3-95	100.0	99.9	100.0	99.0	100.0	95.4	100.0	99.4	100.0	99.6	100.0	99.9

(a) single-end

		(0,0)		(4,0)		(8,0)		(2,2)		(2,4)		(0,5)	
	method	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	98.8	98.8	98.3	96.4	100.0	92.0	98.5	97.6	98.3	97.5	98.5	98.3
	BWA	99.0	98.4	99.2	93.5	100.0	80.0	98.5	79.9	100.0	65.2	99.1	69.0
	Soap 2	98.7	95.5	98.9	53.2	0.0	0.0	97.3	53.3	96.6	46.3	98.5	79.7
	R3-100	99.0	99.0	99.0	99.0	100.0	100.0	99.7	99.7	99.6	99.6	99.0	99.0
	R3-95	99.0	98.9	99.1	96.3	100.0	88.0	99.1	97.9	100.0	99.2	98.7	98.6
all-mappers	Hobbes	97.5	93.2	98.0	94.6	100.0	100.0	96.5	80.1	99.5	86.0	97.7	85.2
	mrFAST	98.8	98.8	98.9	98.9	100.0	100.0	99.1	99.1	98.8	98.8	91.8	7.8
	SHRiMP 2	100.0	99.7	100.0	99.9	100.0	100.0	100.0	99.7	100.0	99.6	100.0	99.7
	R3-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R3-95	100.0	99.9	100.0	97.3	100.0	88.0	100.0	98.8	100.0	99.2	100.0	99.9

(b) paired-end

Table 4.4: Results of the variant calling benchmark. Shown are the percentages of found origins (recall) and the fraction of unique reads mapped to their origin (precision), grouped by reads with s SNVs and i indels, labeled as (s, i).

	dataset	SRR497711 D. melanogaster			ERR012100 H. sapiens			simulated, $m = 800$ D. melanogaster		
	method	time [min:s]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	2:00	99.65	85.71	5:37	99.62	96.72	13:48	96.73	99.99
	BWA	5:35	98.96	79.37	13:45	99.66	93.53	5:38	74.96	68.09
	Soap 2	1:55	91.78	72.49	2:34	96.45	89.73	0:54	41.21	38.14
	R3-100	1:28	100.00	78.92	85:56	100.00	92.99	1:17	100.00	90.43
	R3-95	1:26	99.87	78.82	43:16	99.96	92.95	1:15	100.00	90.43
all-mappers	Hobbes	4:51	96.49	76.16	265:48	95.97	89.24	–	–	–
	mrFAST	4:01	100.00	78.92	413:40	100.00	92.99	5:16	65.25	69.32
	SHRiMP 2	23:40	99.83	89.91	1312:09	99.81	99.06	796:06	95.70	99.31
	R3-100	1:51	100.00	78.92	118:26	100.00	92.99	1:20	100.00	90.43
	R3-95	1:45	99.87	78.82	58:13	99.96	92.95	1:20	100.00	90.43

(a) single-end

	dataset	SRR497711 D. melanogaster			ERR012100 H. sapiens			simulated, $m = 800$ D. melanogaster		
	method	time [min:s]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	6:32	98.94	81.94	10:51	99.51	94.19	39:07	93.64	99.70
	BWA	13:33	97.47	73.41	34:35	98.84	88.06	11:26	56.28	46.44
	Soap 2	5:29	88.67	72.77	8:24	91.58	87.47	12:36	23.55	28.23
	R3-100	9:01	100.00	72.95	176:29	100.00	86.93	2:22	100.00	71.16
	R3-95	6:56	99.78	72.80	135:44	99.89	86.84	2:19	100.00	71.16
all-mappers	Hobbes	8:43	84.78	62.48	89:35	95.11	84.05	–	–	–
	mrFAST	8:26	100.00	73.16	779:12	99.94	87.79	10:47	44.19	49.69
	SHRiMP 2	47:07	99.67	87.36	2762:32	99.74	97.51	1617:26	91.64	98.62
	R3-100	7:59	100.00	72.95	184:27	100.00	86.93	2:30	100.00	71.16
	R3-95	7:36	99.78	72.80	166:22	99.89	86.84	2:29	100.00	71.16

(b) paired-end

Table 4.5: Mapping time and accuracy of single-end and paired-end mapping. Hobbes could not be run on reads longer than 100 bp. In large, the percentage of totally mapped reads and in small the percentages of reads with up to $\left(\begin{smallmatrix} 0\% & 1\% & 2\% \\ 3\% & 4\% \end{smallmatrix}\right)$ errors are shown. In case there are no reads in a class, this is denoted as "—".

second best method but has problems with long reads. R3-95 is the next best method, followed by Shrimp 2. Hobbes achieves the worst results of all all-mappers (caused by regular crashes) and cannot handle long reads at all.

Surprisingly, RazerS 3 achieves the best running time on the *D. melanogaster* data sets both for the all-mappers and the best-mappers. Otherwise, RazerS 3 always is the fastest all-mapper, only using half the time of Hobbes and a tenth of the time of Shrimp 2 on the *H. sapiens* data set, for example. Soap 2 is the fastest best-mapper (at the cost of achieving the worst quality result), followed by Bowtie 2 and BWA. RazerS 3 is the slowest best-mapper on the *H. sapiens* data set since it uses an online search strategy and no reference index such as the other three best-mapper tools.

For the paired-end data, the picture is similar. RazerS 3 achieves the best results in terms of correctly mapped reads, followed by mrFAST and Shrimp 2 for all-mapping and Bowtie 2 and BWA for best-mapping. Again, RazerS 3 is the fastest tool for all-mapping, except for Hobbes which, however, crashes regularly. The index-based tools benefit from the paired-end mapping mode in terms of running time for the 100 bp reads and RazerS 3 is again the fastest tool for mapping the simulated long reads.

In all experiments, the specificity of R3-95 is much higher than the configured 95% in terms of correctly mapped reads. However, RazerS 3 benefits greatly in terms of running time from less sensitive settings. In single-end mapping, R3-95 only takes half the time of R3-100. The differences in paired-end mapping are less pronounced. Here, the verification is only performed when two candidate matches occur within a window restricted by the expected concordant insert size. Considering both mates of a pair probably increases the specificity of the filter.

Note that the order in terms of mapped reads differs from the order of correctly mapped reads. For some read mappers, it is not possible to directly control the error rate and they can generate discordant alignments. For example, Shrimp 2 is able to align 99% of all reads for the single-end *H. sapiens* data set whereas the results of the fully sensitive read mapper RazerS 3 show that only 93% of the reads have an alignment with up to 4% errors.

In all-mode (best-mode), RazerS 3 requires 15 GB (9 GB) for mapping 10 M reads of length 100 bp to hg18. The memory requirement is proportional to the number of reads and matches, about 10 GB are required for each additional 10 M $\times$ 100 bp reads. The input read set can be separated into blocks that are mapped after each other to adapt to memory requirements. For the same input set, Bowtie 2 uses 3.3 GB, BWA uses 4.5 GB, Soap 2 uses 5.4 GB, Shrimp 2 uses 38 GB. Due to the lack of parallelization or a high memory consumption mrFAST and Hobbes were run on packages of 500 k reads where they required 11 GB and 70 GB of memory, respectively.

4.6 Chapter Conclusion

In this chapter, I described

- the state of the art in general approaches to read mapping,
- an intuitive interpretation of what a *match* in read mapping should be,
- a formal definition of when two read alignments are equivalent,
- a formal definition of read mapping matches,
- a practical software package Rabema for performing read mapping benchmarks based on the formal definition,
- a parallelization approach for the string filter–based read mapper RazerS 3,
- local match masking and histogram-based threshold setting as improvements to data flow management in string filter–based read mappers, and
- a comprehensive read mapper benchmark based on Rabema, evaluating the capability to align reads in the presence of genomic variation and using real-world data.

In this chapter, I discussed

- drawbacks of using "number of aligned reads" as a measure for the quality of a read mapper,
- trade-offs in the choice of filtration and verification algorithm for the string filter–based read mapper RazerS 3 that are applicable in future read mappers, and
- the performances of various read mapping software packages in terms of the formally defined benchmark, in the presence of genomic variation, and on real world data.

I observed that

- using a formal definition of read alignments can give a different picture than merely counting aligned reads,
- the read mapper RazerS 3 performs very well as a best-mapper in terms of correctly aligned reads,
- surprisingly, RazerS 3 performs better in the presence of genomic variation than read mappers interpreting base quality values,
- the sensitivity parameter to RazerS 3 is only a lower bound on the sensitivity of the program and consistently better results are achieved, and
- RazerS 3 is the best available all-mapper in terms of sensitivity (since it has full sensitivity) and overall running time.

I conclude that

- having a formal benchmark for read mapping, especially on real-world data, allows an objective evaluation of read mappers,

- among the tested tools, RazerS 3 is the best one for all-mapping, and
- further investigation in base quality value interpretation is required such that it helps the read mappers and does not lead them astray.

My main contributions in the area of read mapping are

- the formal definition of an equivalence relation on read alignments,
- the formal definition of *matches* in read mapping using this equivalence relation that has also an intuitive representation,
- the design and implementation of a read mapping benchmark based on these definitions,
- the implementations of the Rabema read mapping benchmark software package,
- the parallelization of the fully sensitive read mapper RazerS 3 and optimizations in the data flow that allow practical, fully sensitive read mapping, and
- a comprehensive study comparing state-of-the art read best- and all-mappers together with my coauthors.

Chapter 5

Variant Detection

I developed the methods for locating insertion breakpoints and assembling inserted sequences, Basil and Anise in collaboration with Leon Kuchenbecker and Knut Reinert. We published our work in Bioinformatics (Holtgrewe et al.*, 2015):*

Holtgrewe, M., Kuchebecker, L., and Reinert, K. (2015). Methods for the Detection and Assembly of Novel Sequence in High-Throughput Sequencing Data. *Bioinformatics*, btv051.

While the genomes of different individuals of the same species are similar, there is a certain amount of variation between them. For example, for humans, the base-wise variation is estimated to be 0.2%–0.4% (Tishkoff and Kidd, 2004). The aim of variant detection in a personal genome analysis pipeline is to determine the variants present in an individual genome. The variants of an individual are computed with respect to a reference sequence. The result of this can then be used in genome-wide association studies (GWAS) to determine which parts of a genotype cause certain features in the *phenotype* (the properties of the individual).

Genomic variants can be roughly categorized into *small variants* and *structural variants (SVs)*. Small variants include single nucleotide variants (SNVs) as well as short insertions and deletions *(indels)*. Traditionally, variants with a size > 1 kbp were called structural variants (Medvedev *et al.*, 2009). With advances in sequencing technologies, the resolution of SV detection has been improved and thus a definition as *variations that change the structure of the genome* is more appropriate today, at the time of writing. *Structural changes* means identifiably changing the order, number of occurrence, and/or orientation of genomic sequence segments.

Variant detection could be performed by whole genome alignment if the whole genome of the individual was available (*e.g.*, see the exposition by Kehr *et al.* (2014)). However, assembling complex whole genomes such as human ones reliably is still computationally challenging and

expensive. Thus, variants are often identified *(called)* from NGS read data (Medvedev *et al.*, 2009).

An important distinction in variant calling methods is between *de novo* calling of variants and using a database of known variants. In the latter case, the database can be used for facilitating the detection of known variants. Of course, hybrid methods can combine both approaches.

Generally, most *de novo* variant calling methods work by first attempting to locate the sample position of reads in the reference genome. Most do this by solving variants of the read mapping problem. The difference between the genome sequence of the donor and the reference then gives rise to certain read alignment patterns *(signatures)* in the read alignments. I will describe these signatures in more detail in Sections 5.1 and 5.2. Identifying these signatures can be seen as *signal identification*. From this point of view, incorrect realignments stemming from sequencing errors and the previously discussed ambiguities can be seen as *noise*.

The topic of this chapter is the identification of genomic variants from NGS data. The focus lies on genomic variants in individuals and not in populations. The 1000 Genomes Project (1KGP) is an important example for a large-scale variant detection project. Aiming at sequencing and analyzing 1000 genomes, the 1KGP performed a population-scale sequencing project and created a large database of small and large variants (*e.g.*, see Abecasis *et al.*, 2010, 2012; Mills *et al.*, 2011).

Mostly, variant calling in resequencing pipelines is performed after aligning the NGS reads to the reference using read mapper programs as described in Chapter 4. The read alignment is then often postprocessed by performing a *realignment* and then *recalibrating* the base qualities, *e.g.*, using GATK (DePristo *et al.*, 2011). In recalibration, the reference bases (and possibly other overlapping reads) are used to reestimate the base quality values of each based on observed errors (Nielsen *et al.*, 2011).

Structure of This Chapter. In this chapter, I give an introduction to variant calling in general and then describe my contribution to structural variant calling. First, in Section 5.1 I give a more detailed introduction to small variants and then list and shortly discuss approaches and methods for small variant calling. In Section 5.2 I do the same for structural variants. In Section 5.3 I describe Basil, a method for the detection of SV and large insertion breakpoints, and in Section 5.4 I describe Anise, a method for the assembly of large insertions. In Sections 5.5–5.7 I present the evaluation of Basil and Anise and the results thereof. Finally, I give a conclusion to this chapter in Section 5.8.

5.1 Small Variants

Small variants consist of SNVs and small indels (sometimes called *microindels*). Figure 5.1 shows a SNV, an insertion, and a deletion in a multi-read alignment (*pileup*). The top row shows the reference sequence and the row below shows the coverage at each genomic base as a bar chart. Below, the multi-read alignment is shown. Each gray box corresponds to a read and the triangles indicate the alignment direction. Deviations from the reference are shown by colored letters in the gray boxes. At the left side, there is a homozygous SNV from `A` to `C`. The violet vertical bars

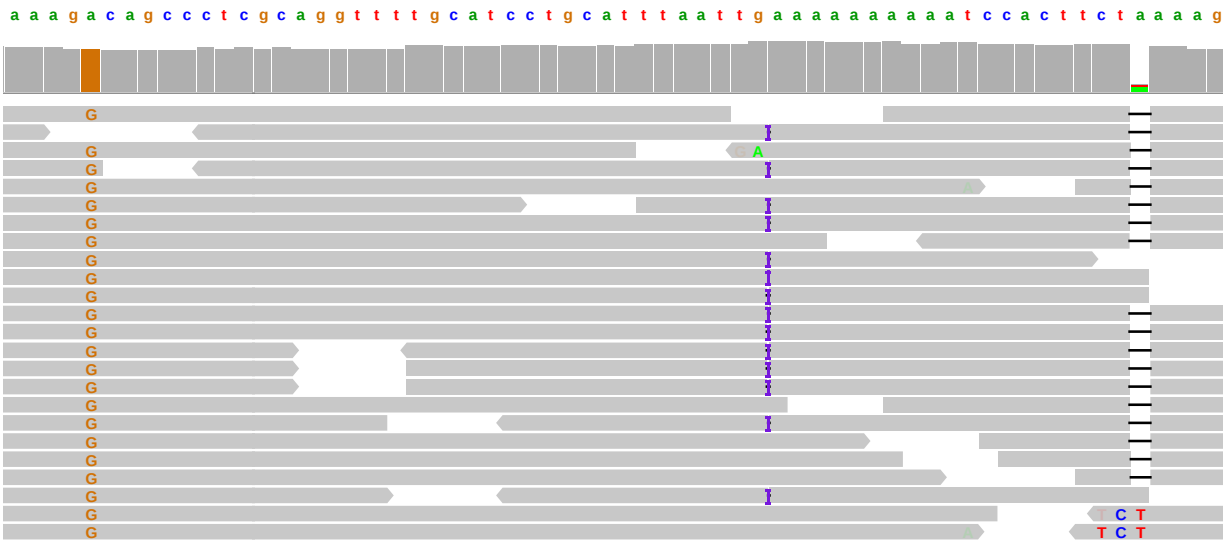


Figure 5.1: Example of a read alignment with a SNV, an insertion, and a deletion in the donor with respect to the reference. The figure was created from an image export from the program IGV (Robinson *et al.*, 2011). The main text describes the figure in more detail.

in the center show a heterozygous insertion of a base. The black horizontal lines to the right show a homozygous deletion of an `A`. Note that the incorrect alignments with `TCT` at the lower right are most probably due to alignment errors.

A few SNVs or microindels (or even a single one) can cause pronounced phenotypes. For example, Enattah *et al.* (2002) found single nucleotide variants in the MCM6 and LCT gene regions that are responsible for lactose (in-)tolerance in adult life. More drastic examples are found in the literature on Mendelian diseases, *e.g.*, by Keupp *et al.* (2013). In their study, the authors found small mutations in the WNT1 gene regions causing "autosomal-recessive *osteogenesis imperfecta*, a congenital disorder characterized by reduced bone mass and recurrent fractures." Most of the affected individuals in their study had a bone fracture within days after birth.

Approaches and Methods for Small Variant Calling

Nielsen *et al.* (2011) give a review of small variant calling and genotyping. Further, the authors give an introduction to approaches and methods for small variant calling from NGS data. After read mapping and possibly recalibration, the multi-read alignment is considered in a column-wise fashion. When columns with deviations from the reference occur, they can be subjected to further analysis.

Early studies used fixed thresholds to determine whether there is a SNV at a certain location in the read pileup and whether the SNV was heterozygous or homozygous in the individual. Later method use probabilistic models to improve the variant calling. They generate candidate genotypes at the individual positions and compute likelihoods for each genotype. The genotype

with the highest likelihood can then be chosen and a score expressing an error probability is assigned to it. Thus, a measure of a call's uncertainty is available and it is possible to judge whether a variant call can be considered reliable or not.

Examples for small variant calling methods are Maq (Li *et al.*, 2008a), the mpileup command of Samtools (Li and Durbin, 2009), SOAPsnp (Li *et al.*, 2009b), and VarScan (Koboldt *et al.*, 2009, 2012). Maq is a software package including a read mapping software for early, short Illumina reads and a small variant caller. Samtools is a versatile software package that can call small variants from the result of any read mapper supporting the BAM file format. SOAPsnp and VarScan are other tools for calling small variants. Small variant calling is such an important and elementary task in NGS analysis that most toolkits for the analysis of genomic NGS data (*e.g.*, GATK (DePristo *et al.*, 2011)) include SNV and microindel detection methods.

Most small variant calling tools expect their input to be the result of read mapping in any-best mode. An exception to this is Sniper (Simola and Kim, 2011), that implements a Bayesian method to incorporate the information of reads aligning at multiple locations.

Databases of known SNPs (*e.g.*, the database dbSNP) and microindels can be used to obtain the genotype at SNP positions in an organism where the donor has the same base as the reference. Also, they can be used to confirm whether a called variant is already known, as done in SOAPsnp, for example. When a read alignment column with a tentative substitution variant is close to a possible indel variant, a *refinement* of the read alignment (*e.g.*, using the methods by Anson and Myers (1997) or Homer and Nelson (2010)) can help to improve the variant calling.

Another important problem related to variant calling is that of *haplotype assembly*, *i.e.*, the assignment of variants called from read data to haplotypes. It is known that most practically interesting variants of this problem are $\mathcal{NP}$-hard (*e.g.*, see Bonizzoni *et al.*, 2003). However, efficient heuristics have been developed that work well for practical inputs, *e.g.*, the ones by Aguiar and Istrail (2012), Mazrouee and Wang (2014), and Kuleshov (2014).

5.2 Structural Variants

In contrast to small variants, structural variants can change the linear order of the genome. Traditionally, the term *structural variant* was used for variants with a size larger than 1 kbp (Medvedev *et al.*, 2009). However, with advances in sequencing technology and computational methods, structural variants can be identified for sizes starting at 50 bp (Alkan *et al.*, 2011). Figure 5.2 shows illustrations of the five basic SV types insertion, deletion, duplication, inversion, and translocation. Medvedev *et al.* (2009) and Alkan *et al.* (2009) each give an overview over approaches and methods for structural variant calling from HTS data.

For insertions, one can differentiate between the insertion of *novel* sequence and the insertion of *mobile elements*. Novel sequence is sequence that is present in the donor but not present anywhere in the reference. Mobile elements are genetic sequences present in the genome that can copy themselves to other locations in the genome. A prominent example for mobile elements is the Alu sequence in primates. Alu sequence and other mobile elements comprise a significant

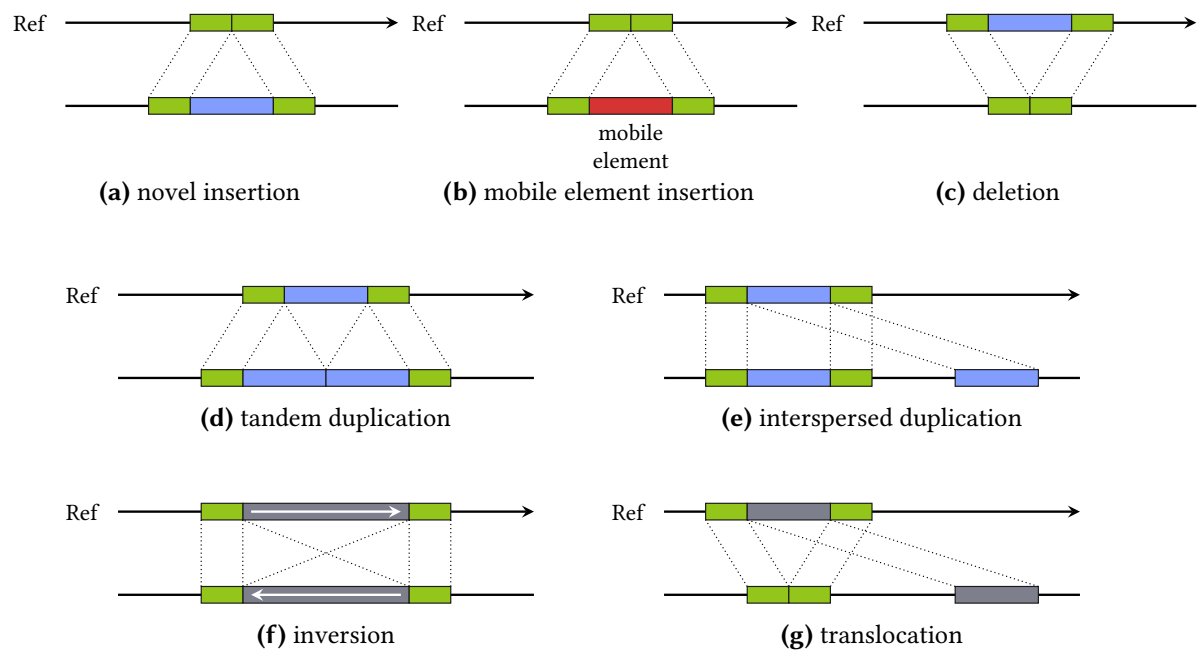


Figure 5.2: Basic SV types insertion, deletion, duplication, inversion, and translocation. The segments that are subjected to structural variants are shown in blue if they change their copy number, they are shown in gray if they retain it, and mobile elements (that are contained in corresponding data bases) are shown in red. The segments that are adjacent to the reference breakpoints are shown in green. The insertion is shown as the basic insertion and mobile element insertion; and for duplication, the two cases tandem duplication and interspersed duplication are shown.
Adapted from (Alkan *et al.*, 2011)

part of the human genome (Lander *et al.*, 2001). Even more pronounced, the maize genome consists to 90% of transposable elements, a kind of mobile elements (SanMiguel *et al.*, 1996).

Structural variants can be categorized by whether they change the *copy number* of a sequence segment, *i.e.*, the number of occurrences in the genome. Note that in this context, the term *copy of a segment* in this context means a genomic segment that has high identity to the original one but does not have to be an exact copy. Insertions, deletions, and duplications change the copy number while the inversion and translocation of segments does not.

For duplications, one can differentiate between *tandem* duplication, where one copy follows the other, and *interspersed* duplication, where both copies are separated by a segment of different sequence. In contrast to mobile elements (that generally have a *high* number of occurrences), duplication events occur for unique segments or segments with a *low* number of occurrences.

5.2.1 Approaches and Methods for Structural Variant Calling

The relatively short read lengths of NGS reads make the detection of structural variants challenging. However, the high coverage and low cost provided by NGS platforms allow SV detection at a larger scale than was possible with previous technology, *e.g.*, as done in the 1KGP (Abecasis *et al.*, 2010, 2012).

The remainder of this section describes approaches and methods used for the detection of structural variation in NGS data. The enumeration of the approaches follows the expositions by Medvedev *et al.* (2009) and Alkan *et al.* (2011). Figure 5.3 contains an illustration of the detection approaches to the six types of structural variants described above. Table 5.1 (p. 94) lists some methods for the detection of structural variants.

Note that the methods listed for each of the approaches below use the same approach but might differ greatly in the details, depending on the kind of data they are designed for. Consider the following examples. The methods might be designed for the analysis of individual donor genomes or variations in pedigrees or populations. The strategies for dealing with ambiguous read mappings differ greatly, from ignoring multi-reads over probabilistic sampling from such mapping positions to explicitly attempting to resolve the ambiguities. A method might be designed for explicitly genotyping individuals, *i.e.*, determining whether variants are homozygous or heterozygous or just detect the fact that a variation exists. The generation of many false positives should be prevented and the approaches implement different means to increase the specificity. One such mean for increasing specificity is to combine (*cluster*) the signals of multiple reads and read pairs supporting the same variant.

5.2.2 The Paired Read Approach

The *paired read (PR)* approach uses the information of read pairs for the detection of structural variants. The underlying idea is that SVs in the donor lead to discordantly mapping pairs from the donor read set. These discordant pair alignments can then be analyzed for characteristic signatures (see Figure 5.3 for examples). Complex rearrangements might make it difficult to identify the SVs and some tools only detect and write out locations of breakpoints without attempting to classify the underling variant.

Methods following this approach include PEMer (Korbel *et al.*, 2007), the first PR-based method using NGS data, and the popular methods BreakDancer (Chen *et al.*, 2009) and VariationHunter (Hormozdiari *et al.*, 2009, 2010). Hydra (Quinlan *et al.*, 2010) is an example for a PR-based method that only attempts to detect breakpoints. Most PR methods focus on deletions, novel insertions, and inversions. Note that using the PR approach alone, base-level resolution breakpoints cannot be identified.

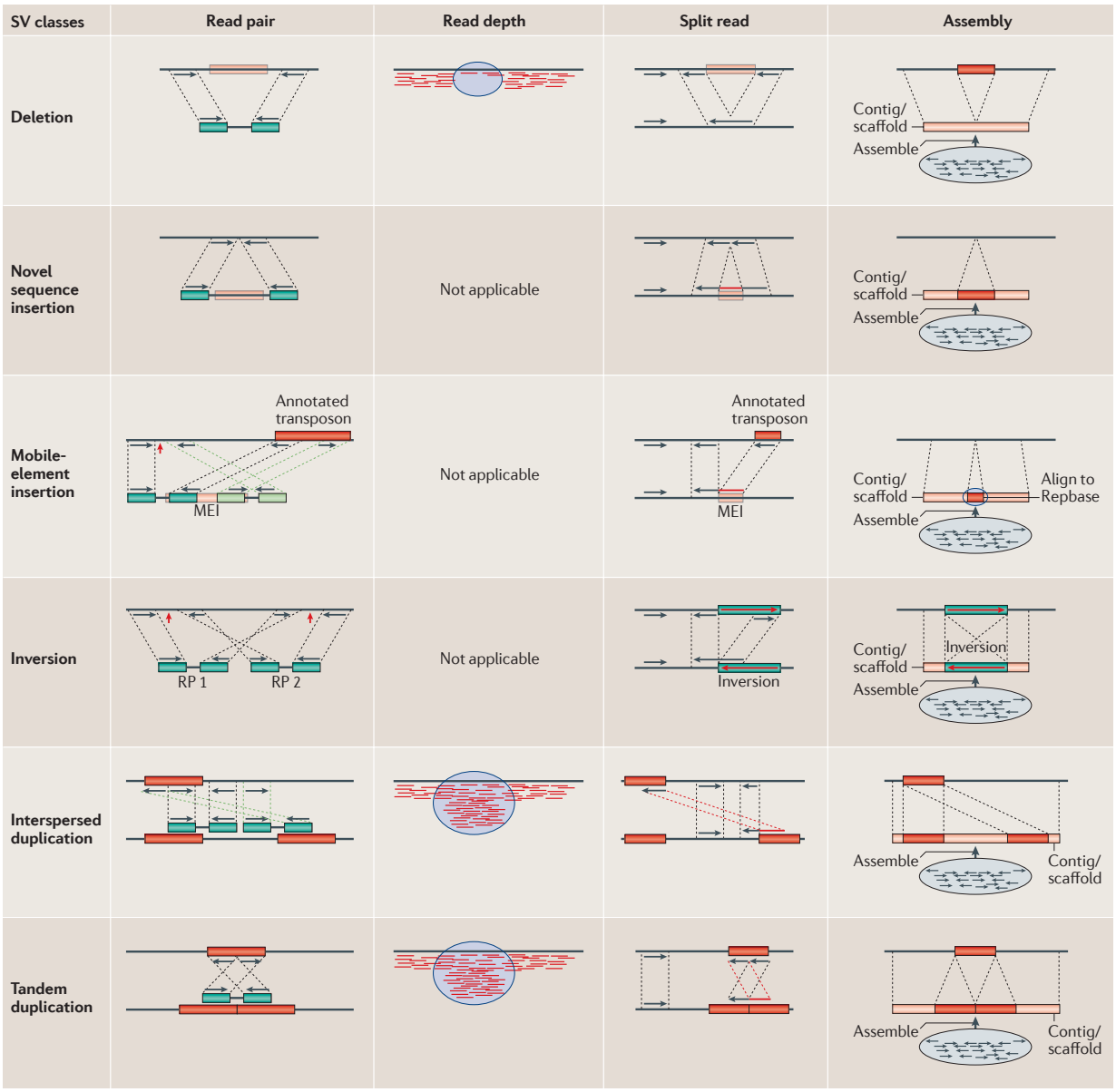


Figure 5.3: Overview of structural variants and their detection from NGS data with the four detection approaches *read pair*, *read depth*, *split read*, and *assembly*.
From (Alkan *et al.*, 2011), used with permission

5.2.3 The Split Read Approach

The *split read (SR)* approach, however, allows the detection of breakpoints at base pair resolution. In this approach, reads that could not be aligned semi-globally to the genome are considered. One possible reason for failing to find such an alignment is that the read spans a breakpoint of an SV in the donor with respect to the reference. In this case, the prefix and the suffix (left and right of

		variant type								approach						
	name	D	NI	MEI	INV	TDP	DP	TR	CNV	PR	SR	RD	AS	other	reference	comment
PR	BreakDancer	✓	✓		✓			✓		✓					Chen *et al.* (2009)	
	Clever	✓	✓							✓					Marschall *et al.* (2012, 2013)	
	GASV	✓	✓		✓					✓					Sindi *et al.* (2009)	
	GenomeSTRiP	✓								✓					Handsaker *et al.* (2011)	
	Hydra									✓					Quinlan *et al.* (2010)	only breakpoint detection
	PEMer	✓	✓		✓					✓					Korbel *et al.* (2009, 2007)	pioneering PEM approach for NGS data
	SVDetect	✓	✓		✓	✓	✓	✓		✓					Zeitouni *et al.* (2010)	
	VariationHunter	✓	✓	✓	✓					✓					Hormozdiari *et al.* (2009, 2010)	MEI in second version
SR	Breakpointer										✓				Sun *et al.* (2012)	only breakpoint detection, SVs by other tools
	ClipCrop	✓	✓		✓	✓		✓			✓				Suzuki *et al.* (2011)	not evaluated on real-world data
	Crest	✓	✓		✓			✓			✓				Wang *et al.* (2011)	
	Gustaf	✓	✓		✓	✓	✓	✓			✓				Trappe *et al.* (2014)	approach uses multi-split mapping
	Pindel	✓	✓								✓				Ye *et al.* (2009)	
	SplazerS	✓	✓								✓				Emde *et al.* (2012)	
RD	CNV-seq								✓			✓			Xie and Tammi (2009)	
	CNVnator								✓			✓			Abyzov *et al.* (2011)	
	RDXplorer								✓			✓			Yoon *et al.* (2009)	
assembly	Cortex_var	✓	✓		✓								✓		Iqbal *et al.* (2012)	allows multi-way comparison; from reads or genomes
	Tigra												✓		Chen *et al.* (2014)	assembly of previously detected breakpoints
	SOAPSV												✓	✓	Li *et al.* (2011)	*de novo* assembly followed by an analysis
hybrid	Anise/Basil		✓							✓	✓		✓		Holtgrewe *et al.* (2015)	assembly of long novel insertions, repeat resolution
	Delly	✓			✓	✓		✓							Rausch *et al.* (2012)	low false discovery rate
	ERDS								✓	✓		✓		✓	Zhu *et al.* (2012)	uses HMM to improve copy number inference
	GASVPro	✓	✓		✓					✓		✓			Sindi *et al.* (2012)	can incorporate multi-read alignments
	NovelSeq		✓									✓			Hajirasouliha *et al.* (2010)	assembly of long novel insertions
	Reprever						✓							✓	Kim *et al.* (2013)	detection and separation of duplicated copies
	SOAPindel														Li *et al.* (2013)	assembly of short and medium insertions
	SVMiner	✓	✓					✓		✓	✓				Pyon *et al.* (2011)	
	SVseq	✓								✓					Zhang and Wu (2011)	
	(Parrish *et al.*, 2011)		✓					✓		✓		✓	✓		Parrish *et al.* (2011)	assembly of long (also) non-novel insertions
other	BreakSeq													✓	Lam *et al.* (2010)	uses library of breakpoint sequence
	forestSV									✓	✓	✓		✓	Michaelson and Sebat (2012)	pre-trained machine learning approach
	SVMerger													✓	Wong *et al.* (2010)	merging of *de novo* SV caller results
	TakeABreak				✓									✓	Lemaitre *et al.* (2014)	calls from reads, no mapping or assembly

Table 5.1: A subset of the methods available for the detection of SVs from NGS data. The variant types are (D/deletion, NI/novel insertion, MEI/mobile element insertion, INV/inversion, TDP/tandem duplication, DP/interspersed duplication, TR/translocation, CNV/copy number variation. The field MEI is only checked when mobile element insertions are specifically targeted.

the breakpoint) of the read were adjacent in the donor but they correspond to different locations in the reference.

Consider the case of detecting deletions with SR in Figure 5.3. The prefix and suffix of the right read align with a distance in the reference that corresponds to the length of sequence deleted in the donor. This way, SR allows for the base-resolution detection of breakpoints.

The SR approach is most often used in conjunction with paired reads since this decreases the ambiguity and thus increases precision. In the case where one read aligns semi-globally against the genome, this read can be used as an *anchor* and the second read can be aligned locally (*i.e.*, partially) in the direction implied by the alignment of the anchor. Further, using paired reads decreases the search space for the local alignment since the method can limit this search by using the approximately known insert size of the read library.

Some split mapping methods are integrated into read mapping programs such as Pindel (Ye *et al.*, 2009) and SplazerS (Emde *et al.*, 2012). Others methods, such as ClipCrop (Wang *et al.*, 2011), CREST (Wang *et al.*, 2011), and Gustaf (Trappe *et al.*, 2014) use local alignments generated by read mappers (*e.g.*, BWA (Li and Durbin, 2009)) or local alignment tools (*e.g.*, BLAT (Kent, 2002) or STELLAR (Kehr *et al.*, 2011)) and work as a postprocessing step of these aligners.

5.2.4 The Read Depth Approach

The *read depth (RD)* approach considers the depth of the read coverage after mapping. This allows the detection of structural variants that change the copy number of a genomic segment *present in the reference*. For example, if the coverage is significantly lower in a region than expected from the deviation of coverage then this indicates a deletion.

However, the RD approach cannot be used to identify novel insertions or insertions of mobile elements: reads with novel sequence do not align to the reference at all and mobile elements occur too often in the reference to give a strong signal. Methods following the RD approach include CNV-seq (Xie and Tammi, 2009), RDExplorer (Yoon *et al.*, 2009), and CNVnator (Abyzov *et al.*, 2011).

5.2.5 The Assembly Approach

A fourth approach is the assembly of genomic sequence that can then be used for a direct comparison to the reference. There are two options for this. First, the global *de novo* assembly of the whole donor genome and second, the local assembly around breakpoints detected in an earlier step.

The *de novo* assembly of the whole genome is computationally challenging itself, as is the comparison of whole genomes. Nevertheless, it is a viable method and might become more popular when longer NGS reads have become widely available at low costs. An example for this is SOAPSV (Li and Durbin, 2009). In the study presenting this approach, the authors used the assembler SOAPdenovo (Li *et al.*, 2010) and then performed a comparison of the assembly with a reference genome.

The method Cortex_var (Iqbal *et al.*, 2012) builds *de Bruijn* graphs for multiple genomes, one for each, either from read sets or from a previously assembled reference genome. *De Bruijn* graphs are a common data structure used in the assembly of genomes from short read data. The graph construction is followed by an analysis and a comparison of the structures of the graphs. Cortex_var is then able to call variants from these graphs.

An example for a method performing the targeted assembly around previously detected breakpoints is Tigra (Chen *et al.*, 2014).

5.2.6 Hybrid Methods and Others

Considering the drawbacks of the approaches described above, it appears natural to combine them. Thus, recent methods often use hybrids of these approaches to improve their performance.

NovelSeq (Hajirasouliha *et al.*, 2010) uses PR signals for the detection of tentative large novel insertion breakpoints and then uses assembly methods for obtaining the inserted sequence. MindTheGap (Rizk *et al.*, 2014) is a recent method for the detection and assembly of inserted sequence, directly from the read sequence without a reference sequence and read mapping. Delly (Rausch *et al.*, 2012) combines the PR and SR approach for calling structural variations at base-level resolution with high specificity. SVMiner (Pyon *et al.*, 2011) is a method for detecting inversions and deletions that combines the PR, SR, and RD approaches. SOAPindel (Li *et al.*, 2013) combines split read mapping with assembly for the detection and assembly of deletions and small insertions. My methods Anise and Basil (Holtgrewe *et al.*, 2015) combine the PR and SR approaches for the detection of tentative large insertion breakpoints and then use an assembly method for assembling large breakpoints. Further, Anise includes means for the explicit handling of repetitive sequence in the assembly step.

Other methods use the calls generated by existing SV calling tools and merge these results (*e.g.*, SVMerger by Wong *et al.*, 2010) or use machine learning to learn patterns indicating SVs (*e.g.*, forestSV by Michaelson and Sebat, 2012). The method BreakSeq (Lam *et al.*, 2010) uses a database of breakpoint sequences to quickly identify known breakpoints in samples. The method TakeABreak (Lemaitre *et al.*, 2014) attempts to call inversion breakpoints directly from the read set without read mapping or assembly.

In the remainder of this chapter, I describe the methods Basil and Anise for the detection and assembly of large novel insertions and their evaluation. At the time of writing, there is only the MindTheGap method available, which is explicitely aimed at detecting and assembling large novel insertions (Section 5.5.1 goes into detail of problems with other methods). As I will show in the experimental evaluation in Sections 5.5– 5.7, Anise and Basil are successful in locating and assembling large insertions, even in the presence of repeats. Anise shows superior performance to MindTheGap and is competitive to state-of-the art general purpose assemblers for assembly in the presence of repeats.

5.3 Basil — Insertion Breakpoint Detection

Basil (Holtgrewe *et al.*, 2015) is a method for the detection of breakpoints from paired NGS data combining the PR and the SR approach. I developed it in conjunction with Anise, my method for the assembly of large novel insertions that is described in Section 5.4. Basil is meant to predict breakpoint locations for which Anise can then attempt an assembly.

Figure 5.4 shows an overview of the Basil program. As the input, Basil takes the reference genome sequence as a FASTA file and a BAM file with the paired read alignments, sorted by their genomic coordinates. The method consists of three steps. First, it preprocesses the read alignments so as to be able to efficiently and easily analyze them later using PR and SR signals. This preprocessing includes the removal of invalid pairs, performing local alignments of previously unaligned reads, and ensuring the correct order after all other processing steps. Second, Basil searches for PR and SR signals and generates candidate breakpoints from them. Finally, it attempts to combine the two kinds of signals. The aim of this third step is to use the base pair–resolution of the SR signals to refine the positions of the more reliable PR signals.

In this section, I describe Basil and give some background information on the PR and SR signatures that Basil uses. In Section 5.3.1 I first identify and discuss three types of insertions and their signatures. In Section 5.3.2 I describe the filter pipeline steps in Basil that are used for filtering and processing the read alignments. This is followed by the description of the signature detection steps in Section 5.3.3 and Section 5.3.4. Finally, I describe the combination of the PR and SR signals in Section 5.3.5.

5.3.1 Insertions Types and their Signatures

In the context of detecting and assembling insertions, I define three kinds of insertions: *small*, *medium*, and *large* insertions. The definition depends on the properties of the available sequencing technology.

In my design of Basil, I focus on the usage of Illumina paired-end read pairs since they allow for obtaining a cost-efficient coverage of large (*e.g.*, human) genomes. Of course, reads with different lengths and libraries with larger template size can be used. However, Basil does not exploit PR signatures specific to read pairs with large insert size, such as deviations from the

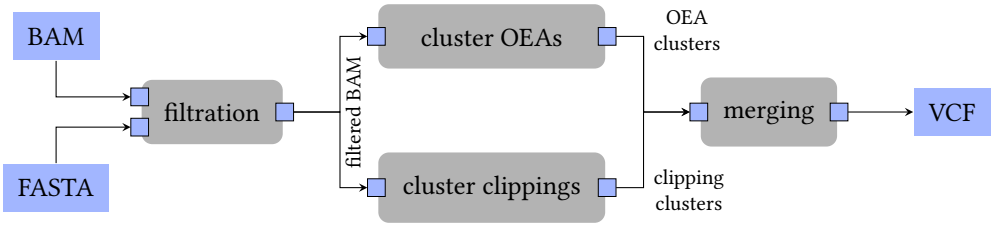


Figure 5.4: Overview of the steps in the Basil method.

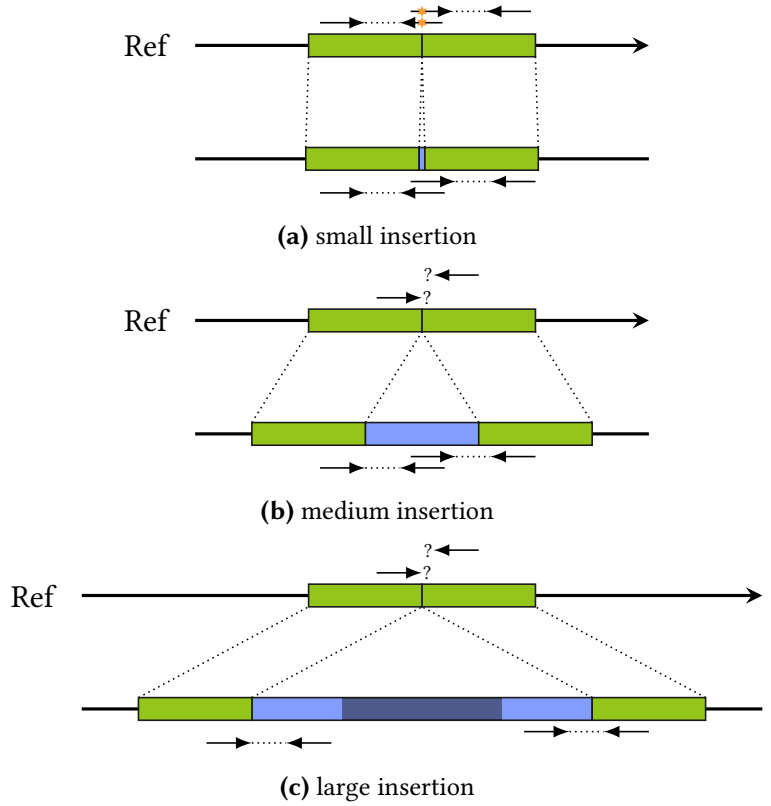


Figure 5.5: Examples of paired read signatures of (a) small, (b) medium and (c) large insertions. The inserted sequence in the donor is shown in blue, flanking segments are shown in green. The sample position of the read pairs is shown below the donor and the mapping locations are shown above the reference. In (a), the location of each read breakpoint after split alignment is indicated by an orange star. The question marks in (b) and (c) indicate that the mate of the aligned read could not be aligned to the reference since it stems from the inserted sequence. In (c), the part of the inserted sequence that cannot be reconstructed from assembling shadow sequence is shown in darker color.

expected inferred insert size of pairs spanning large insertions.

Figure 5.5 shows examples for the three kinds of insertions. In each case, the corresponding subfigure shows two paired reads that can be used for detecting the inserted sequence. For medium and large insertions, the case of pairs spanning *into* the insertion is important. Here, one read is sampled from the inserted sequence while the other is sampled from sequence that is also present in the reference. This gives rise to the particular kind of PR signatures, *one-end anchored* mapping signatures (this term was coined by Kidd *et al.* (2008)).

Definition 5.1 (One-end anchored (OEA) read pairs). *The read pairs for which one read can be aligned semi-globally and the other cannot are called* one-end-anchored (OEA) *pairs. The alignable read is called* anchor *and the other read is called* shadow. *Shadow reads are* anchored *by the other read in the pair.*

Small Insertions are short enough such that they are spanned by single reads. SR methods, such as Pindel (Ye *et al.*, 2009) or SplazerS (Emde *et al.*, 2012), can efficiently compute their location and sequence. Figure 5.5a shows an example of a small insertion and two reads that span over the insertion in so-called *split alignments*. For small insertions, the read prefix and suffix are aligned in such split alignments and an infix in the center remains unaligned. Specialized DP alignment algorithms are used for obtaining split alignments. These alignments are then used for locating the insertion breakpoint in the sequence and the unaligned part of the read is used for determining the insert sequence.

The maximal insert sequence length for which such methods are applicable depends on the available read length. For example, under the assumption of a read length of 100 bp and that a suffix and a prefix with a length of each 25 bp has to be aligned to the reference to detect the insertion, the maximal detectable insert size is 50 bp.

Note that small insertions can be detected by single-end reads but that using paired reads decreases the mapping ambiguity of the suffix and the prefix (as also explained by Emde *et al.* (2012)). Further, having a second pair can be used to decrease the search space and thus the running time.

Medium Insertions are those that cannot be covered reliably from one read with sufficient suffix and prefix alignment size (see Figure 5.5b). Given the assumptions above, their size is ≥ 50 bp.

However, the length of the inserted sequence must be short enough, such that they are completely covered by anchored shadow reads. That is, the coverage of the inserted sequence has to be of sufficient depth, such that the insertion can be assembled from the anchored shadows alone. When considering reads of the same length, this is the case when the length of the inserted sequence is less than approximately $2 \cdot (t - r)$ where t is the maximal expected template size and r is the read length. Under the assumption that the read length is 100 bp and the insert size follows the normal distribution $\mathcal{N}(400, 40)$, medium insertions are not longer than approximately $2 \cdot (400 + 3 \cdot 40 - 100) = 840$ bp (99.7% of all template sizes will fall into three standard deviations around the mean). However, this bound is only a rough approximation and will depend on sequencing depth in practice.

SOAPindel (Li *et al.*, 2013) is an example of a method designed for the detection and assembly of medium insertions. Li *et al.* (2013) give a (relatively conservative) bound of approximately $(t \cdot 1.2 - r)$ for the maximal detectable insertion length. For the values in the example above, this yields a bound of approximately 380 bp.

Large Insertions are those that are greater than medium insertions (see Figure 5.5c). The PR signature is the same for medium and large insertions. Both insertion types only differ in the fact that medium insertions can be covered using shadows alone while large insertions cannot. In Figure 5.5c, the shaded area in the inserted sequence shows the part of this sequence that cannot be covered by shadow reads because of limitations due to the template size.

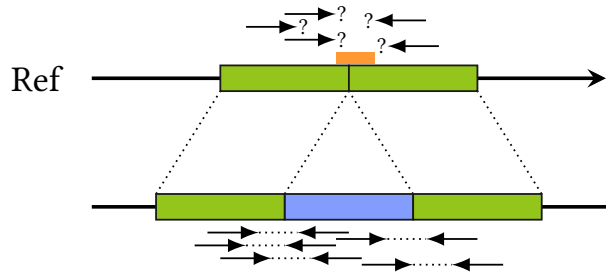


Figure 5.6: PR signature for an insertion breakpoint position. The inserted sequence in the donor is shown in blue, the flanking segments are shown in green in both the donor and the reference. The sampled read pairs are shown below the donor and the anchors are shown aligned to the reference. The question marks beside a read indicate that no alignment could be found for their mate. Here, the flanking sequence also corresponds to the left and right window around the true breakpoint. The orange rectangle above the reference indicates the range for the possible insert breakpoint, as can be called from the PR signal.

Large and Medium Insertion PR Signals. Basil searches for the same type of signatures as the program mrCAR from the NovelSeq method (Hajirasouliha *et al.*, 2010). A breakpoint at a genomic position p with sufficient coverage generally leads to a signature as shown in Figure 5.6. Left of p, there are OEA anchors on the forward strand and right of p, there are OEA anchors on the reverse strand. More formally, I define the *support* of these alignments as below, in consistency with Hajirasouliha *et al.* (2010).

Definition 5.2 (OEA forward and reverse support for a position). *Given a genomic position p and reads with an insert size that approximately follows a normal distribution $\mathcal{N}(\mu, \sigma)$. The* OEA forward support *is the number of OEA alignments on the forward strand in a window of length $\mu + 3\sigma$ left of p. Accordingly, the* OEA reverse support *is the number of OEA alignments on the reverse strand in a window of the same length right of p.*

The PR signature for large and medium insertions at a position p is a high OEA forward and reverse support *(OEA signature)*. In general, the generated PR signals will not have base pair resolution. Instead, there will be an interval of positions with the same supporting OEA forward and reverse alignments. It is sufficient to report the center of the interval.

Figure 5.6 shows an example for this. Here, the insertion breakpoint in the reference could be anywhere between the innermost anchors on the forward and reverse strand. The orange rectangle above the breakpoint indicates this interval and its center position can be used as an approximation for the breakpoint.

Ambiguities in PR Signals. Depending on the strategy employed by the read mapper used for generating the input, SVs of a different type than insertions can yield the same PR signature in the read mapper's result. For example, for the Illumina *paired-end* protocol, the read mapper BWA (Li and Durbin, 2009) only attempts to find concordant read alignments as described below.

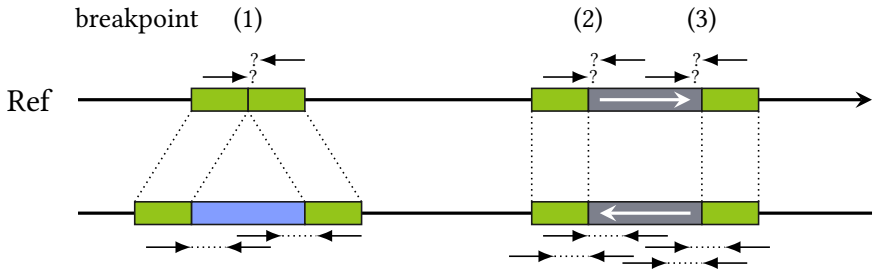


Figure 5.7: An example for ambiguous PR signatures of breakpoints. The donor sequence has a large insertion and an inversion with respect to the reference. The sampled reads are shown below the donor and the mapped reads are shown above the reference. The read alignments are depicted as a read mapper such as BWA could generate them. From the OEA signature alone, the type of the SV for each breakpoint is unclear.

The paired-end protocol yields template sizes of 200–400 bp and is often used in resequencing project because of its lower cost per base pair, when compared to the mate-pair protocol that yields template sizes of 2–5 kbp.

Figure 5.7 shows an example. For paired-end reads, BWA first tries to find a semi-global alignment of one read in each pair and then attempts to find a concordant alignment for the read's mate. It does not consider alignments at higher distances or on other chromosomes. If no such concordant alignment can be found, the program then attempts to find a local alignment for the still unaligned read. When this fails, the read mapper generates an OEA alignment with a shadow read. Thus, other kinds of SVs (*e.g.*, the inversion shown in Figure 5.7) can yield the same OEA and clipping signatures and Basil will generate candidate breakpoint locations for them as well.

However, as I will describe in Section 5.7.3, this does not lead to problems downstream in the insert assembly. These non-insertion breakpoints can be filtered out in a postprocesing step after running Anise. Roughly, Anise attempts to fill an insertion from each breakpoint side towards the other one simultaneously. It does so by repetitively assembling the shadow reads and then mapping the *orphan* (unmapped) reads against the newly assembled sequence. In the case of breakpoint (2) and (3) in Figure 5.7, the iterative assembly processes from the left and right side will not meet. This means that the assembly attempt will stop without succeeding and such partial assembly results can be easily removed in a postprocessing step.

Clipping Signatures. Some read mappers such as BWA can also generate local alignments. BWA does this when it cannot find a concordant pair alignment with both reads aligning semi-globally. As explained above, BWA will first try to find a good semi-global alignment of one read (the anchor) in this case. Then, it attempts to perform a local alignment of sufficient size and score in the direction indicated by the anchor. The unaligned prefix and suffix of a read is called *clipped* sequence. The positions in the read where the clipping starts and the corresponding positions in the genome are called *clipping positions*. The *clipping support* of a position is defined as follows.

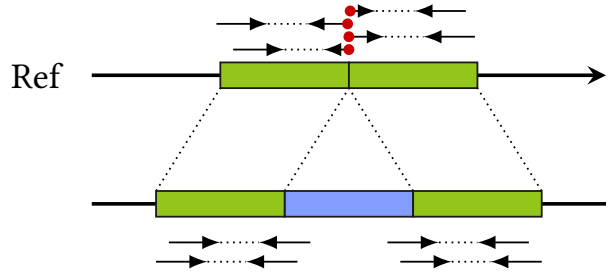


Figure 5.8: Clipping signature for an insertion breakpoint. The reads spanning into the insertion are aligned locally to the reference. The clipping position is shown as a red circle in the clipped alignment.

Definition 5.3 (clipping support of a window). *Given a genomic position p and a* radius r*, the* clipping support *of p is the number of clipping positions that fall into the window of length $\ell = 2r$ around p.*

Figure 5.8 shows an example of such clipped alignments around a breakpoint. The locally aligned reads are treated as shadows in the PR signature detection and clusters of clipping signatures can be treated as a second type of signal. The clipping positions can also be used to get a base pair resolution position for the breakpoints detected by the PR signal.

Basil only considers clipped alignments when a read prefix or suffix can be aligned that has a minimal length (defaults to 15 bp). Thus, true infixes or too short prefix and suffix alignments are ignored. I classify using such clipped alignments as following the SR approach, even though only one partial alignment is performed instead of attempting to align the remainder of the read.

Note that using such clipped alignments differs from using the SR approach in that only one part of the read is aligned to the reference. The original SR approach (*e.g.*, as used by SplazerS) is not possible in the case of medium and large insertions. Because of their length, reads cannot span over such insertions and have a prefix and suffix of sufficient length aligning to the reference. Also, in the case of novel insertion, the clipped-away part of the read sequence does not have a corresponding location in the reference.

5.3.2 Filter Pipeline Steps

I implemented Basil as a pipeline with several steps as also shown in Figure 5.4 (p. 97). Each pipeline step is executed in a separate thread, allowing parallel insert site prediction. The input is a file with alignments in BAM format, sorted by coordinate. That is, the read alignments are sorted lexicographically by (c, p) where c is the number of the chromosome (in an arbitrary but fixed numbering scheme) and p is the begin position of the alignment. Further, the reference is given as the input in FASTA format.

Each record from the BAM file includes the name and sequence of the read as well as the position and orientation of the read mate's alignment. Each record also describes the pairwise alignment of the read to the reference as well as information about the clipping of a record. Thus, the records can be read sequentially, one by one, and the distance to the other mate's alignment is known. Note that the BAM record does not contain information on whether the other mate's alignment is clipped or not.

In a step before the preprocessing, the first few thousand records are read from the BAM file. These records are used to guess the read orientation and to estimate the mean and standard deviation of the normal distribution $\mathcal{N}(\mu, \sigma)$ that the template size follows. The records are then read sequentially from the beginning and subjected to the following pipeline.

Remove High Coverage Reads. In the first step, reads from regions with too high coverage are removed. Excessive coverage is an indicator for mobile elements in these regions. Mobile elements are not targeted by the downstream method Anise for the assembly of *novel* sequence. Regions with too high coverage are detected using a simple incremental linear scanning algorithm (see Section 2.2.3).

Here, *incremental* means that records with alignments that do not overlap with high-coverage regions are written out as soon as possible and subjected to the next pipeline step. All subsequent pipeline steps also process the alignments in such an incremental fashion to keep the memory requirements low.

Remove Discordant Pairs. In this step, reads aligning with an unexpected relative orientation or with a too large inferred template size are removed. This removes alignment artifacts and allows for optimizations in the downstream pipeline steps. By default, the maximally allowed inferred template size is $\mu + 8\sigma$. Consequently, downstream pipeline steps only have to buffer a certain range of BAM records to have all records in memory whose read pairs can overlap with the currently processed position.

Remove Invalid Pairs. In this step, records with missing mates are removed. This removes artifacts either caused by the removal of records in the previous two steps or erroneous read mapper output. I occasionally observed such erroneous output with older versions of BWA.

Treat Clipped Alignments as Non-Aligned. In this step, records with clipped alignments are updated such that they appear as shadows. In the BAM format, shadows are marked as being unaligned by a flag in the record and the shadows are assigned the same position as their mate. This step performs such an update of the flags and the alignment position for the clipped records. In the case that both reads of a pair contain clipping, the pair is removed. A permanent mark is then added as a BAM tag to the anchoring reads, such that they can be later identified as anchors after the local alignment step.

This step is quite technical but allows treating clipped and shadow record uniformly in the OEA clustering step described in Section 5.3.3 below. Further, it ensures that the same local alignment

algorithm is performed for the shadow reads (that the read mapper could not align) and the clipped reads.

Attempt Local Alignment of Shadow Reads. This step attempts to perform local alignments of the shadow reads against a window of the reference sequence. The local alignment of the reads gives rise to clipping positions that are used in the clipping clustering step described in Section 5.3.4 below.

The local alignment computation is similar to the way that BWA performs its local alignment of anchored reads as described in Section 5.3.1.

For each anchor, a window in the reference is computed, depending on the alignment direction of the anchor and the estimated template size distribution. The alignment position information is subsequently updated in the case that such a local alignment was found. As described above, read alignments where both ends of a read are clipped away or when there are too few bases in the local alignment are ignored and these reads are kept as shadows.

Reorder Alignment Records. In this step, the records are reordered again, such that those with clipped alignments from the previous step are moved to the correct position in the list of records.

Processing of the Filter Pipeline Result. The output of the filter pipeline is a coordinate-sorted list of BAM records. These records are passed into concurrently running pipeline modules that perform the OEA signature and clipping signature clustering described in Sections 5.3.3 and 5.3.4.

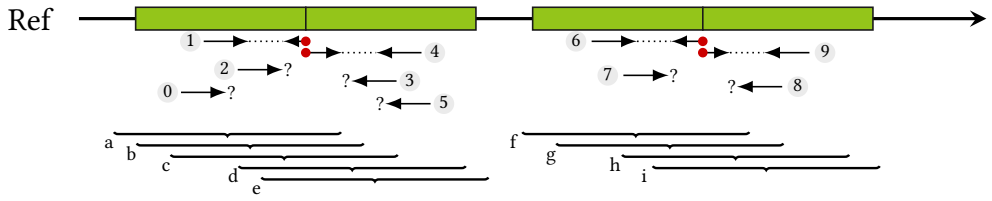
5.3.3 OEA Clustering Algorithm

The OEA signal detection is performed after the record preprocessing. This step first enumerates positions with sufficient OEA forward and reverse support. An example for such a signal is shown in Figure 5.6 (p. 100). When a range of possible breakpoint positions is identified, the center of this range is reported.

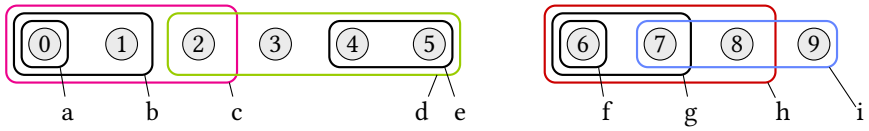
For efficiency, the enumeration is done using the linear scanning approach (see Section 2.2.3). This way, the records can be processed on the fly and only few records have to be kept in memory.

Besides returning a range of possible insertion positions, there can be another kind of ambiguities. Figure 5.9a shows an example of this. This figure shows two breakpoints with corresponding OEA and clipping signatures. In the OEA cluster step, reads with a clipped alignment are treated as shadow reads. The braces at the bottom show windows that have a non-zero total OEA support. Thus, a OEA cluster could be reported for each of these windows.

Figure 5.9b shows a set model of this situation. Each anchor is represented by a circle, in the same order as their alignment against the reference. Further, each OEA cluster window is represented by a rectangle containing the circles whose corresponding reads are spanned by this window.



(a) example for ambiguity in OEA signatures



(b) corresponding set model of the OEA clusters

Figure 5.9: Example for the ambiguity of OEA signals. (a) shows OEA and clipping alignments for two breakpoints. Reads with clipped alignments are counted as shadow reads. The braces at the bottom show windows for the OEA clustering, centered at the tip of a OEA alignment on the forward strand. The OEA reads are numbered 0–9 and the windows are labeled a–i. (b) shows this situation as a set model. Each anchor read is shown as a circle and the rectangles represent the windows and contain the circles for the reads spanned by each window. Each set contains the forward alignments in the left half of its corresponding window and the reverse alignments in the right half of its window. According to Definition 5.2, the forward alignments in the left half of the window support the center of the window as the insertion position and vice versa for the reverse alignments in the right half of the window. The labels in (b) are the same as in (a). The main text explains how the two different OEA cluster selection strategies yield different sets of explaining windows.

I considered two strategies for solving this. (i) Select a minimal cardinality subset of the enumerated positions such that all OEA alignments are explained as mrCAR from NovelSeq (Hajirasouliha *et al.*, 2010) does, for example. (ii) Select a subset of the enumerated positions such that no two positions are supported by the same OEA alignment.

For strategy (i), a solution can be found by modeling it as a Set-Cover problem and solving this using a greedy heuristic that runs in polynomial time (Cormen *et al.*, 2001, p. 1034). While the heuristic only gives a logarithmic guarantee on the result quality, it works well in practice (Hajirasouliha *et al.*, 2010). Strategy (ii) is equivalent to one-dimensional chaining (Gusfield, 1997, p. 325) and can be solved in time $\mathcal{O}(n \log n)$ since the coverage has an upper bound because of the first filter step. For the example of Figure 5.9, strategy (i) yields all four colored sets. Strategy (ii) yields the green and the blue set.

Since it is not obvious which selection strategy is best, Basil implements both, defaulting to one-dimensional chaining. Section 5.6.2 contains a comparison of both strategies using simulated data, justifying the choice for the default method.

5.3.4 Clipping Clustering Algorithm

In parallel to the OEA cluster detection, Basil clusters the clipping signals. This step is also implemented using linear scanning.

Here, Basil searches for windows of a length ℓ that contain at least c clipping positions. Redundant windows are removed, *i.e.*, those where another window exists with a superset of positions. In the case of overlapping windows, chaining is used for selecting a set of non-overlapping ones. The values of ℓ and c are configurable, the defaults are $\ell = 20$ and $c = 5$.

The center of each window is reported as a possible breakpoint position. Since the positions in a clipping position cluster are typically very close, the reported position has base pair resolution.

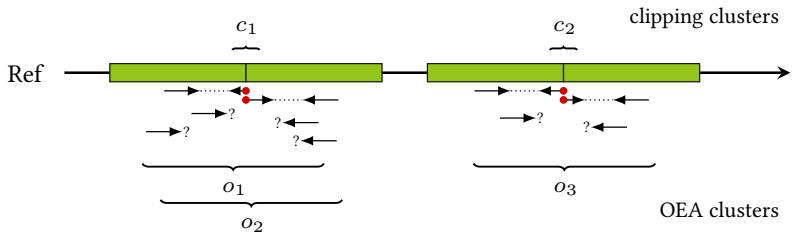
5.3.5 Combining OEA and Clipping Signals

In a final step, Basil attempts to refine the positions of the OEA clusters with the clipping cluster positions using maximum weight matching in bipartite graphs. Towards this end, a bipartite graph is created.

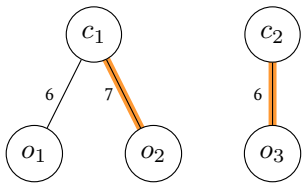
The algorithm inserts one vertex for each generated OEA and clipping cluster. An edge is inserted for each clipping cluster that lies within the interval of possible breakpoint positions of an OEA cluster. The algorithm weighs each with the sum of the OEA forward and reverse support of the OEA cluster and the clipping support of the clipping cluster. Basil computes a maximum weight matching using the Lemon library (Dezső *et al.*, 2011). For each selected edge, the OEA cluster's breakpoint position is set to the breakpoint position of the clipping cluster.

Figure 5.10 gives an example. The selected OEA and clipping cluster signatures are shown in Figure 5.10a. Here, the OEA cluster step yielded the three OEA clusters o_1, o_2, and o_3 and the clipping cluster step yielded c_1 and c_2. Figure 5.10b shows the bipartite graph that is built in the matching step. The maximum weight matching is highlighted, thus the position of c_1 is assigned to o_2 and the position of c_2 is assigned to o_3.

Basil then writes out a list of candidate breakpoint positions for each OEA cluster. If the position of the OEA cluster could be refined with a clipping cluster in the mapping step, this position is used. Otherwise, the center of the position interval associated with the OEA cluster is written out and the position is flagged as being inexact in the resulting VCF record.



(a) OEA and clipping clusters



(b) cluster combination

Figure 5.10: Example for the combination of OEA and clipping signals. Figure (a) shows read alignments for three OEA signals o_i and two clipping signals c_j. Reads with a clipped mate count as OEA anchors. o_1 contains all but the two rightmost OEA anchors it overlaps with, o_2 contains all but the leftmost OEA anchor, and o_3 contains all OEA anchors it overlaps with. From this, the bibartite graph in Figure (b) is built as described in the main text.

5.4 Anise — Insert Sequence Assembly

Anise is a method for the targeted assembly of medium and large novel insertions. That is, given a reference sequence, the alignments of a read set against the reference, and a set of (predicted) insert breakpoint positions, Anise attempts to assemble the inserted sequence at each such position. Practically, the reference sequence is given as a FASTA file, the alignments are given as a BAM file (sorted by coordinate and indexed, such that alignments from specific regions of the genome can be retrieved efficiently), and the list of candidate breakpoint positions is given as a VCF (Danecek *et al.*, 2011) file (*e.g.*, as written out by Basil).

There are two main causes for medium and large insertions in the donor with respect to the reference. First, the inserted sequence might be truly novel in the donor, *i.e.*, the sequence is not included in the reference since it was not observed with sufficient prevalence in the sequenced genomes that were used for building the reference genome. The sequence was not observed in the majority of the considered individuals or not observed at all and thus is not included in the reference. Second, the inserted sequence might be missing in the reference because the used assembler was not able to assemble this sequence correctly. One major cause for problems during the assembly is *repeated sequence*, *i.e.*, sequence that has multiple highly similar copies in the genome.

To be able to deal with such insertions that contain repeated sequence, Anise includes features for *repeat separation*. That is, if parts or all of the inserted sequence have multiple copies in the genomes, then Anise attempts to determine the correct copy to assemble. The result of Anise is a FASTA file with the assembled sequence, annotated with the position of the candidate breakpoint that was used for starting the assembly.

In this section, I describe Anise and its components. In Section 5.4.1 I give an overview of the method. In Section 5.4.2 I describe the used *overlap-layout consensus (OLC)* approach for the assembly. In Section 5.4.3 I describe the copy separation step used for resolving assembly issues in the presence of repeats, and in Section 5.4.4 I give a description of the *scaffolding* step that is used for resolving issues in the case of ambiguities during the assembly.

5.4.1 Overview

The following overview of the program Anise is also illustrated in Figures 5.11 and 5.12. Figure 5.11 shows a high-level flow diagram of the individual steps performed and Figure 5.12 gives an illustration of these steps with the reads in the assembly and the contiguously assembled sequence *(contigs)* resulting from the assembly.

Anise considers each (candidate) insertion site from the input VCF file independently. In the initial step, Anise extracts the supporting OEA alignment pairs (anchors and shadows) for the current site.

It then performs the initial *assembly step*. The extracted read sequences are subjected to assembly using the *overlap-layout-consensus (OLC)* approach (see Section 5.4.2). To deal with reads from multiple copies, a *repeat separation substep* is performed (see Section 5.4.3). This is then followed

by a *scaffolding substep* (see Section 5.4.4) where the contigs are ordered, overlaps are resolved, and a *scaffold sequence* is generated.

The assembly step is followed by the *read mapping step*. In this step, the pairs for which no alignment could be found (the orphans) are aligned to the scaffold sequence in an all-best-mapper fashion using the same algorithm as in RazerS 3 (see Section 4.4). The sites for which no new read alignments could be found are marked as inactive and they are not considered further.

At this point, each site has a set of reads associated with it either from the input BAM file or from a previous read mapping step. The sites for which new read alignments could be found are subjected to another iteration of this loop and each iteration attempts to assemble the inserted sequence further from the outer borders towards the center. This is repeated until all sites are marked as inactive. To prevent a too long running time, all sites are marked as inactive after reaching a maximal number of iterations (defaulting to 50).

Assembly Failure. There are many cases in which there are problems in the assembly around predicted insertion sites. One possible reason is that there are spurious OEA alignments around an actual insertion site or that the predicted insertion site is not an insertion breakpoint. I observed that such cases lead to assembly artifacts, *e.g.*, short contigs consisting of very few reads.

For dealing with such assembly problems, Anise considers contigs that are shorter than 150% of the average read length as artifacts and removes them from the assembly result. If all contigs created by an assembly step are discarded in the initial step then this site is disabled and not considered in the subsequent steps. I call this *failure* of the insert assembly at this site. Anise does not write out any sequence for the sites at which the insert assembly failed.

Usage of External Memory. One of my aims for Anise was that it could be run for full human HTS data sets, *e.g.*, Illumina HiSeq 2000 runs with a coverage of 30x. For such settings, the resulting BAM file with the aligned reads usually has a size of approximately 150 GB and contains a total of approximately 450 million read pairs. Running Anise should be possible on common compute servers without the need for dedicated high-memory machines as are often required for *de novo* assembly.

This implies that main memory usage should be kept relatively low and that only the active

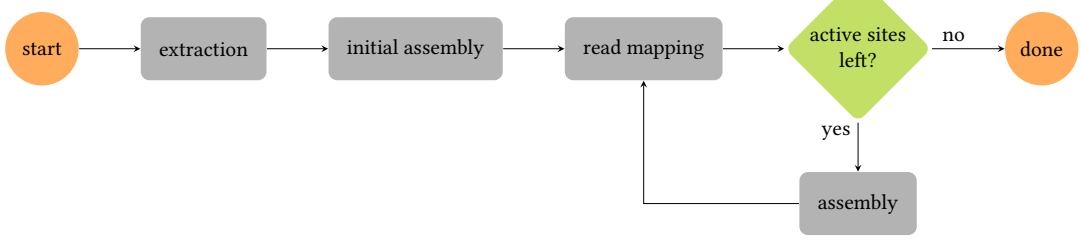
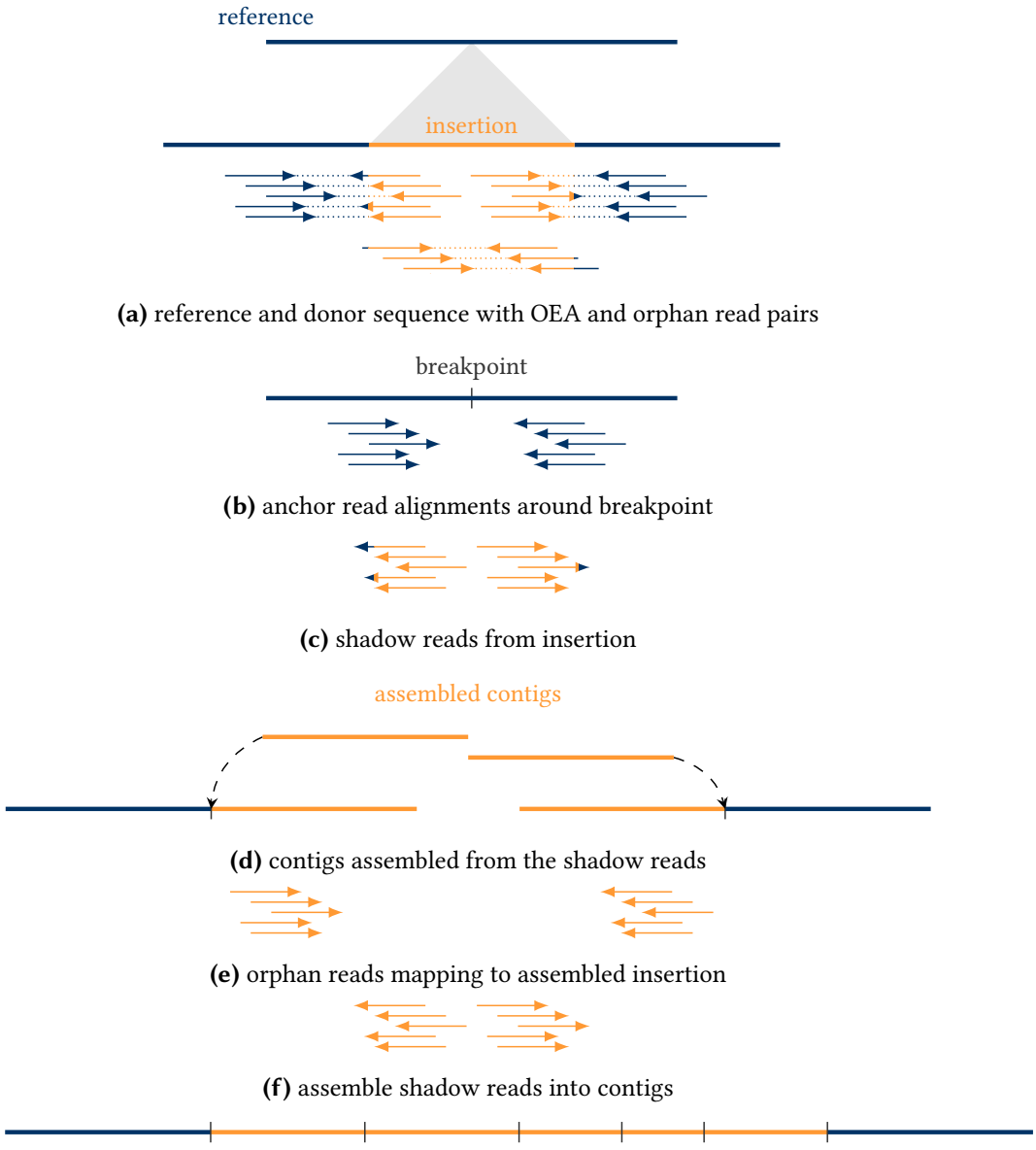


Figure 5.11: A high-level diagram of the program steps in Anise.



(a) reference and donor sequence with OEA and orphan read pairs

(b) anchor read alignments around breakpoint

(c) shadow reads from insertion

(d) contigs assembled from the shadow reads

(e) orphan reads mapping to assembled insertion

(f) assemble shadow reads into contigs

(g) the steps above are repeated until the insertion is closed

Figure 5.12: An overview of the work flow of Anise. Anise independently considers each tentative insertion site from the input. First, it extracts the OEA pairs from the input BAM file. This is followed by the assembly step for obtaining contigs, repeat separation, and scaffolding of the resulting contigs and contig copies. Then, it performs the read mapping step. The assembly step is repeated using the reads for an input.

working set of the input data should be kept in memory. This in turn means that the data has to be stored on the computer's hard drive, *i.e.*, on *external memory*. I decided on the following design that allows for practical usage of main memory while needing no complicated external libraries for data storage.

The orphan reads are extracted from the input BAM file and written to a temporary compressed FASTQ file that is kept during the whole program run. In the extraction step, the program loads the OEA read pairs from the BAM file for each candidate insertion site in the VCF file. The initial assembly step then performs an assembly of these reads for each site. The result of the assembly step is are multiple sequence alignment (MSA) of the reads for which the consensus sequence is computed and used as the contig sequence. The contig sequence and read MSA are written to a FASTA/SAM file for each site.

The read mapping step first reads the previously assembled contigs for all sites into main memory. It then sequentially reads through the orphans FASTQ file, aligns the reads to the contigs using the read mapping algorithm of RazerS 3, and appends the read alignments to the SAM file for the corresponding sites. The read information for each site is stored on the disk in the uncompressed SAM format, such that the read mapping step can simply append new lines. This is not easily possible with the compressed BAM files.

For each site, the assembly step reads in the aligned read pairs, ignoring any duplicate read alignments. Then, the program performs the sequence assembly as described in the sections below. This is followed by writing out the assembled sequence as a FASTA file and the read MSA as a SAM file to the disk. These files are then used as the input of the iteration of the read mapping/assembly loop.

Parallelization. The fact that each site is considered independently in the assembly step is used for performing the assembly in parallel. Anise starts a configurable number of threads and each atomically obtains a site to perform assembly for from a shared work queue.

Read mapping is parallelized in a similar fashion. The reads are read in chunks of several thousand reads and each chunk is then processed in parallel. The mapping results are distributed to the sites using a global lock for writing. In my experiments, I found that this global lock does not create a bottleneck for the thread/core counts (up to 64) that I could test. In the future, this locking strategy can easily be changed in the case that it creates scalability problems for servers with more cores.

5.4.2 Overlap-Layout-Consensus Contig Computation

Anise uses a variant of the classic OLC approach for the assembly of contigs.

Overlap Computation. Similar to the Celera assembler (Myers *et al.*, 2000), Anise uses a k-mer filter for identifying candidates for overlapping reads. For this, the pigeonhole string filter algorithm from the SeqAn library is used. The filter yields pairs of contigs that are candidates for

overlaps and are verified using a banded DP algorithm. For each pair of contigs, only the longest overlap is stored.

The k-mer index is also used to filter out k-mers with too high abundance (by default, k-mers with an abundance of more than 200 are ignored). Further, the overlaps are filtered such that too short overlaps are removed (by default, the overlap has to be at least 40% of the longer read, which is appropriate for 100 bp reads). Overlaps with more than 5% of errors are filtered out as well.

Layout Computation. The next step is the computation of an approximate read layout. For this, Anise combines ideas previously used in the MIRA assembler (Chevreux, 2005) and the CABOG (Celera Assembler with Best Overlap Graph) (Miller *et al.*, 2008) and builds a *best overlap graph (BOG)*.

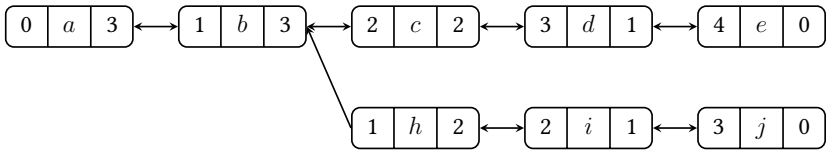
In the initial step, all contained reads (reads for which a containing overlap exists, see Section 2.2.2) are removed. That is, the reads that overlap completely with another read or are an (approximate) substring of another read are removed. These reads will be considered again in the consensus step (see below).

In a BOG, each read is represented by a vertex. Each vertex can have two outgoing arcs: one for the best overlap to the left and one for the best overlap to the right. In CABOG, the length of the overlaps is used for scoring them. This works well for the 454 and Sanger reads that CABOG is designed for but in my experience, this strategy is less suited for the shorter read length and higher coverage in Illumina data. Instead, I dediced to use the same strategy for scoring in Anise that is used in MIRA.

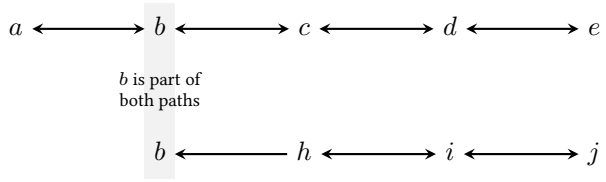
In this strategy, all overlaps to the left and right for each read are collected and sorted descendingly by the overlap length. For each read r_0, Anise considers the n reads R_1 to the left side with the longest overlaps and stores the overlap length as the score. For each of these reads r_1 from R_1, it computes the n best neighbors to the left and adds the overlap length to the score of r_1. This recursion is repeated up to a depth of m. In each recursion step, the best score for all successors is returned. The same recursion is also performed to the right. The best-scoring read for the left and the right is then selected. The values $n = m = 4$ yield good results in practice.

After computing the best overlaps to the left and right for each read, Anise then computes the lengths of the longest paths to the left and the right for each vertex. As is done in MIRA, Anise considers the vertices in random order and starts a breadth-first search (BFS) traversal to the left and right from each vertex. While performing the BFS, the program computes the length of the longest path reachable towards the left and right outgoing arc. The resulting path length towards the left and right is stored for each vertex. Once a vertex with known path length (for the currently computed side) is reached, the stored count is used instead of continuing the recursion.

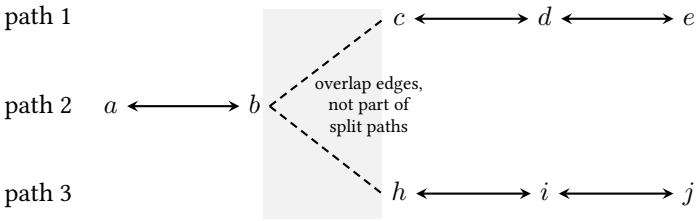
After computing the path lengths, Anise selects long paths as in CABOG, marking all vertices on the selected paths as *covered*. In the beginning, no path is selected and all vertices are marked as *uncovered*. The program greedily selects vertices by the previously determined path length in descending order and traverses the longest path up to the first covered vertex, if any. Such intersections can arise in the presence of repeats or from spurious alignments. The path extension



(a) best overlap graph



(b) path selection result



(c) paths after splitting

Figure 5.13: Example of a best overlap graph, path selection, and path splitting. (a) shows a best overlap graph. The label of each node shows the length of the longest path towards the left, the name of the node, and the length of the longest path towards the right. The arc towards the left points to the node with the best overlap towards the left and vice versa for the arc pointing towards the right. (b) shows the result of the path selection and (c) shows the result of the path splitting.

stops at such intersections and they are stored for later.

In the example of Figure 5.13 the selection first starts at node b and the result from the selection is the upper path in Figure 5.13b. Then, the algorithm starts the selection at node h. This path is first extended towards the left and immediately stops at b since it has been previously marked as covered already. Note that b is included in this path and the information of the two paths sharing b is stored for the subsequent splitting step. The path is then extended towards the right from h to j and the result is the lower path in Figure 5.13b.

After the extension, the previously selected paths are split at these intersection positions by a criteria similar to the one in CABOG. If the shorter path describes a sequence of $\geq 200\,bp$ and contains ≥ 3 reads then the longer path is split and the program stores the two overlaps at the splitting site as contig overlaps for the scaffolding step described below.

The result of this splitting step is shown in Figure 5.13c. Here, three paths are generated and the upper path from Figure 5.13b is split into two new paths (paths 1 and 2).

Each of the path segments yields an approximate layout for one contig. The overlap alignments indicate the offset of the right read with respect to the start position of the left read. From this, the approximate position of each read in the contig can be computed. The approximate layout is a list of positioned reads, where each positioned read is described by a numeric ID and the position in the approximate read layout.

Consensus Computation. Finally, the consensus sequence of the approximate read layout is computed. For this, the same method as in SeqCons (Rausch *et al.*, 2009) is used. For each pair of overlapping reads in the approximate layout, a banded overlap alignment is computed. These overlap alignments are then used to compute a MSA of the reads. Then, the MSA is refined using the realignment algorithm of Anson and Myers (1997) that I reimplemented and the consensus sequence is computed from the refined MSA.

The MSA algorithm that I used is based on the segment-wise computation of consensus-based multiple sequence alignment algorithm by Rausch *et al.* (2008). The number of segments tends to increase with the number of sequences to be aligned and with the number of gaps in the pairwise alignments. Further, the running time of the consensus step is cubic in the number of depth in the MSA. Thus, the running time of the consensus step rises with the number of gaps in the reads and it rises sharply with the highest coverage in the read MSA. When the assembled sequence is repetitive (*e.g.*, contains the sequence of a mobile element), the read coverage tends to be high and the small differences in the distinct copies can lead to indels in the pairwise alignments. This leads to high running times in the consensus computation.

To resolve this issue, a simple trick is employed. The main idea is to select a subset of the reads such that the consensus computation is performed on a MSA with a limited coverage.

Towards this end, Anise infers the approximate length of each contig from the read begin positions and the read lengths. An integer array with an entry for each position is created and initialized with zeroes. Then, the reads in the layout are considered in random order and the coverage at the spanned position with already picked reads is computed. If this is below a given threshold (the default is 50), then the read is used in the MSA. Otherwise, it is marked as *deferred*.

After computing the MSA and before refining it, the deferred reads are aligned against the consensus sequence and this alignment is used for integrating it in the MSA. Thus, the deferred reads are also considered in the final MSA refinement. The overlap alignments removed earlier are integrated in the same fashion.

This heuristic approximates the read alignment by ignoring gaps in the multi-read alignment. For most genomic sequence that can be assembled well, the read MSA contains few gaps, such that the heuristic works surprisingly well in practice. In the case of read MSAs with a large number of gaps, the reads are not very similar and the resulting assembly would be of low quality anyway.

5.4.3 Copy Separation

Contigs computed by the previous step might contain reads from multiple locations on the sequence that have high similarity. Despite these similarities, there will usually be columns in

the corresponding read MSA with non-spurious deviations from the consensus character that look similar to SNVs in the variant analysis step on resequencing. These deviations are small enough such that the OLC step assembles the reads into the same contigs but large enough such that the assembled contigs contain errors. I implemented routines in Anise, based on the work by Kuchenbecker (2011), to use such signals for resolving problems. In turn, the methods by Kuchenbecker build on prior work by Tammi *et al.* (2002).

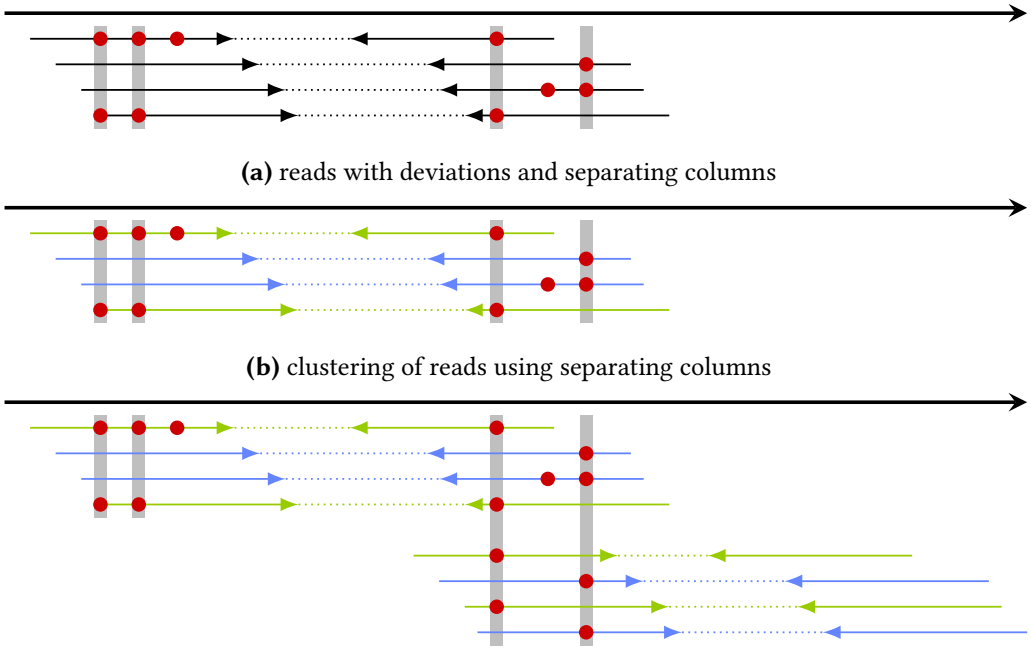
The copy separation step in Anise has three aims. The first aim is to identify the aforementioned differences of the reads from the consensus character in the sequence assembly. This allows to perform local predictions of which reads were sequenced from the same and which from different regions in the genome. The second aim is to combine multiple local predictions to more global ones, using multiple deviations in the same read sequence and read pairing information. The third and final aim is to use the larger clusters to infer the actual sequence of the inserted sequence.

Identifying Separating Columns. The method by Kuchenbecker (2011) first uses a method for identifying statistical significant pairs of columns with *co-occuring differences* that was developed by Tammi *et al.* (2002). For this, Kuchenbecker first identifies columns with deviations in the multi-read layout, then enumerates pairs of columns that have deviations *(deviating columns)* from the consensus character, and then uses the statistical test by Tammi *et al.* to check for statistical significance. For Anise, I reimplemented this method (Kuchenbecker, 2011, Algorithm 3) for identifying such *separating columns* that can be used for predicting which reads stem from different genomic locations.

Kuchenbecker then combines this with a method for clustering multiple columns by Kececioglu and Ju (2001) but I found this to be problematic with the relatively short reads from NGS sequencing. For this clustering, one has to identify a set of reads that all cover the same set of deviating columns and either the number of covering reads is too small or only few (mostly one) deviating columns are covered.

In Anise, each separating column creates a local partition, or *local clustering*, where reads are partitioned based on their character in the column (one of the five characters `C`, `G`, `A`, `T` or `-`; occurrences of `N` are ignored). Thus, each column can only be used to separate at most five copies. Figure 5.14a shows the situation after identifying a total of four deviating columns in a read MSA from two pairs of deviating columns.

Combining Local Separation Information. The second aim in copy separation is to use the previously identified features to cluster the reads such that there is one cluster for each original sample location. For this, I reimplemented parts of Kuchenbecker's approach and extended this method for the separation of copies in the context of insert assembly. An overview of the general idea is given in Figure 5.14 and I also shortly describe this *cluster linking* in this section. I explain the overall method and the technical details in Appendix I. In summary, I used Algorithms 5–8 from Kuchenbecker (2011) for combining local separation information into more global ones. Everything from the *copy reconstruction* step on is my extension of Kuchenbecker's method to the case of copy separation for insert assembly.



(a) reads with deviations and separating columns

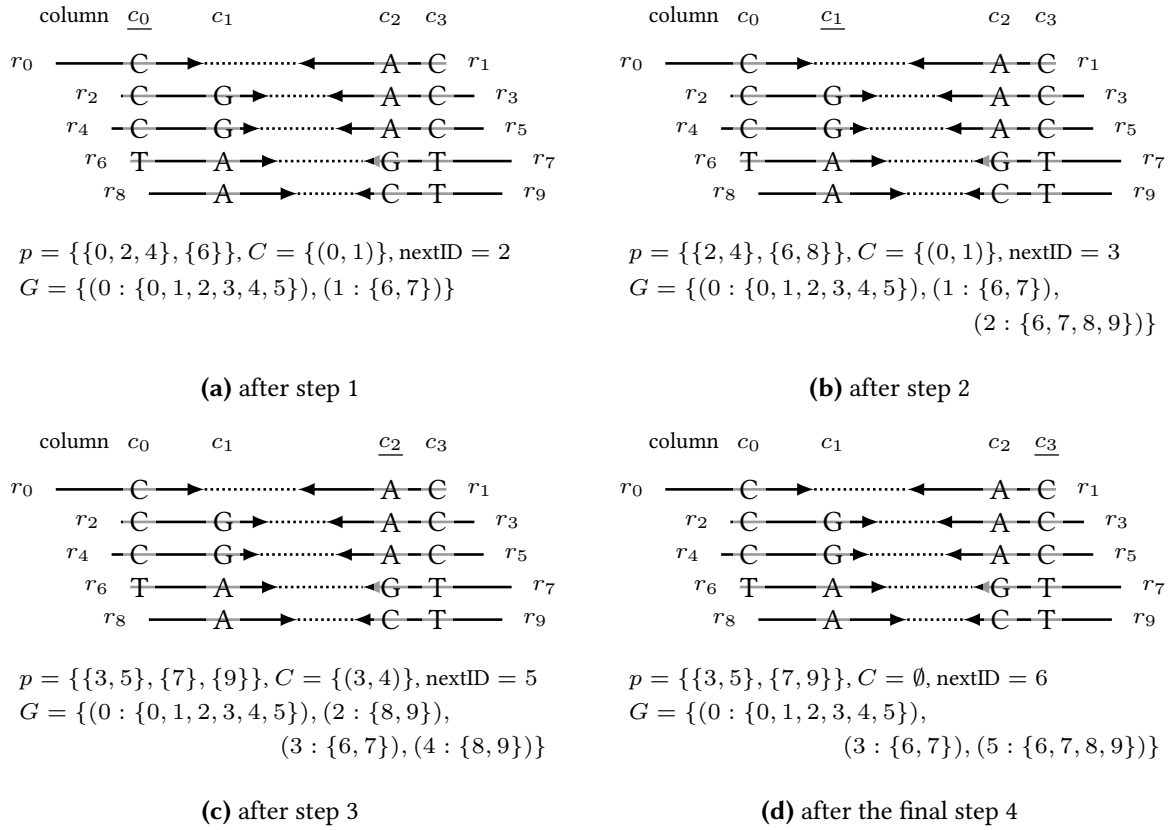
(b) clustering of reads using separating columns

(c) extension of clusters

Figure 5.14: The general idea behind the copy separation step in an ideal case. The reference is given in black at the top, reads are shown as black arrows, and read pairs are connected by dotted lines. Differences to the consensus are shown as red dots and separating columns are shown as gray rectangles. (a) First, co-occurring differences to the consensus are used to identify separating columns (differences not passing the statistical test are subsequently ignored). (b) Reads with co-occuring differences are clustered together and clusters are propagated by mate-pair information. (c) Scanning from left to right, it is attempted to separate the copies globally.

For the local read separation, ANISE creates a set of clusters of the overlapping reads for each separating column in the read MSA. For each such column, one cluster is created for each character in the alignment column. The first fundamental idea for combining the local separation information is that when multiple reads show the same characters in more than one column then this is evidence for the reads potentially stemming from the same genomic location. On the other hand, differences in these columns are evidence for the reads coming from different locations. Of course, such predictions are potentially incorrect due to sequencing errors. The second fundamental idea is that the two reads from one pair were sampled from the same genomic location.

Using these two ideas, information about local clustering can be propagated along contigs and scaffolds in the *cluster linking* algorithm. The algorithm manages a set of global clusters G and of conflicts C. The set of global clusters G contain sets of reads that are predicted to belong together by having the same character in a separating column. The set of conflicts C stores information about pairwise conflicts between elements of G. The conflicts are initially derived from reads in the clusters of G having different characters in a separating column. The algorithm iterates over all local clusterings from the separating columns (sorted by coordinate) and updates



(a) after step 1 **(b)** after step 2

(c) after step 3 **(d)** after the final step 4

Figure 5.15: Example for the cluster linking algorithm. Each sub figure shows reads $r_0, \ldots, r_9$ and four separating columns $c_0, \ldots, c_3$ and the characters of the current partition p, global cluster set G, conflict set C, and value of nextID. The currently processed column label is underlined. Each entry of G shows the cluster ID, followed by a colon and the set of the contained reads. The variable nextID stores the numeric identifier of the next global cluster that is added to G.

G and C using the local clusterings. The cluster linking algorithm is described in more detail in Appendix I.

Figure 5.15 shows an example of the cluster linking algorithm by Kuchenbecker (2011) with ten reads $r_0, \ldots, r_9$ in five read pairs having four separating columns. In each step, the current local partition p of reads is used for updating G and C. Note that the local partition only includes one read of a pair but that the other mate is included in the entries of G. The algorithm is heuristic and provides no guarantees on the final result. This is also shown in the given example where conflicts are erroneously removed from C in the global class merging (Algorithm I.3). Also note that reads can be shared by multiple entries in G.

Reconstruction of Explaining Copies. The aim remains to perform the *copy reconstruction*, *i.e.*, constructing a set of explaining copies from the global clusters G. In most cases, each cluster in G provides a partial explanation of a haplotype by the characters the set's reads show in the

Algorithm 5.1: Copy Reconstruction Heuristic

input : set of global read clusters G
returns : list of merged read clusters L
$G' \leftarrow$ select-read-cover(G);
$L \leftarrow$ merge-read-sets(G');
return L;

`/* Select and return a covering subset of` G`.` `*/`
Procedure select-read-cover*(G)*
  $R \leftarrow \emptyset$;
  $Q \leftarrow$ priority queue of G, the $g \in G$ with the highest number of reads not yet in R is on top;
  while *pq.size()* > 0 **do**
    $g \leftarrow$ pq.popTop();
    if g contains at least 5% of the reads for a separating column that are not yet in R **then**
      update Q according to reads in g that are not in R;
      $R \leftarrow R \cup \{g\}$;
    end
  end
  return R;

`/* Merge read sets in` G `that have few conflicts and return a list of read sets, sorted by number of reads from the initial read alignment.` `*/`
Procedure merge-read-sets*(G)*
  $L \leftarrow G$ sorted by number of reads from initial read alignment;
  for $i \leftarrow 1$ to $|L| - 1$ **do**
    for $j \leftarrow 0$ to $i - 1$ **do**
      if $L[j]$ not marked for removal and conflict count for $L[i]$ and $L[j] \leq 2$ **then**
        $G[j] \leftarrow G[j] \cup G[i]$;
        mark $G[j]$ for removal;
      end
    end
  end
  remove previously marked entries from L;
  sort L by number of reads from initial read alignment;
  return L;

separating column. The approach done by Anise for constructing explaing haplotypes is shown in Algorithm 5.1.

In the first step (procedure select-read-cover), Anise selects a subset G' of G that covers most of the reads but has a lower redundancy. For this, the entries of G are considered ordered by the number of contained reads that are not yet in the result R of this procedure. An entry g from G is selected only if there is a separating column for which it contains 5% of the reads that are not yet in a set in R. The rationale behind this is to select read sets from G that contain a sufficient

additional signal in a separating column.

In the second step (procedure merge-read-sets), Anise merges clusters of G' that have few pairwise conflicts, starting with the clusters that have the most reads that were already aligned in the input BAM file. The resulting list of merged clusters L is then sorted descendingly by the number of contained reads that had an alignment in the input BAM file.

This second step performs the actual copy reconstruction. Here, Anise considers the read sets from the G' (that has a lower redundancy than G) and puts together local haplotype explanations to global ones. Note that, by their construction, the locally explaining read sets combined here are not linked by the clustering linking algorithm. Thus, the algorithm generates *a* set of explaining copies but in general it will not be able to generate *the* correct one. Although the algorithms for the copy reconstruction are only heuristics, they work well in practice as I will show in Sections 5.5–5.7.

Read Correction. The copy reconstruction step is followed by a read correction step similar to the one described by Eriksson *et al.* (2008). For each read in the MSA, the positions differing from the consensus that are not part of a separating column are updated to the consensus character.

5.4.4 Scaffolding

The copy separation step created explaining copies for each contig (one *contig copy* for each cluster). The aim is now to select and order a correct subset of these into an assembly. This is similar to the scaffolding step in general purpose *de novo* assemblers. In Anise, the *scaffolding step* is performed using a graph-based approach.

For the scaffolding, Anise builds a directed graph G that contains one vertex for each contig copy. Two vertices are connected if the corresponding contig copies have an overlap indicated by the OLC step or are linked by mate pair information. Two vertices are not connected if they correspond to two copies of the same contig. Also, the algorithm adds a special source vertex s and a target vertex t to G.

For the scaffolding step, I call those reads that have an alignment in the input BAM file *anchors*. Of these, I call the ones aligning on the forward strand *left anchors* and the ones aligning on the reverse strand *right anchors*. For each vertex u whose corresponding contig contains a left anchor, Anise adds an arc (s, u) and it adds an arc (u, t) if the corresponding contig contains a right anchor. Each path from s to t (s-t-*path*) corresponds to an assembly of the insert sequence.

Anise labels each arc with the number of supporting mate pairs. In the case of multiple assignments from read to contig, each mate pair is weighted by $1/c$ where c is the number of copies. Each arc is also labeled with an estimation of the distance (mean and standard deviation) from overlap or mate information. The information of multiple mate pairs between the same contig is combined using the approach described by Huson *et al.* (2002). Also, the scaffolding step in Anise uses the transitive edge reduction step by Huson *et al.* (2002). Links are verified using banded DP alignment containment and overlap alignments are removed.

The graph resulting from these steps can still contain directed cycles that have to be broken. A common approach for breaking undirected cycles when computing an initial layout in general purpose *de novo* assemblers is as follows. A maximum spanning tree (MST) is computed and the non-MST edges are removed from G. Originally, I attempted to use this approach in Anise by converting the arcs to undirected edges.

In practice, I found that this approach removed too many arcs (even a DAG can contain many undirected cycles when interpreting the arcs as undirected edges). Instead, my algorithm uses a variant of the topological sorting algorithm (Cormen *et al.*, 2001, Section 22.4), which removes arcs that would introduce directed cycles *(violating arcs)*. The order of vertices used for starting the topological sorting are chosen randomly, leading to the removal of violating arcs in an arbitrary order.

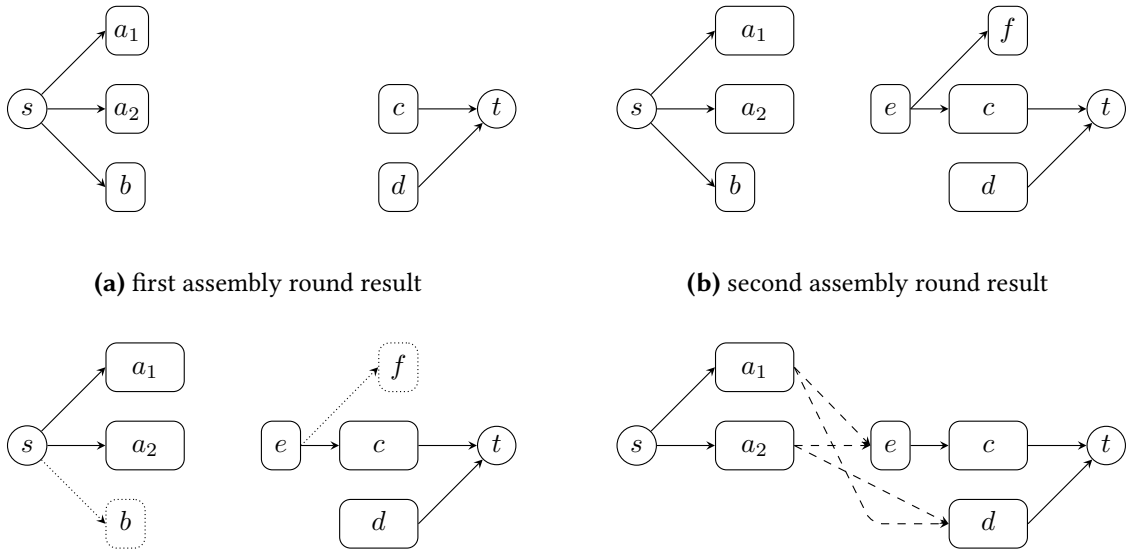
Artifact Removal. If all pairs recruited for the assembly of a site were correctly assigned and the insert sequence was unique then G would either consist of one or two contig vertices. In the first case, the assembly of the insert would have been successfully completed and G would consist of one vertex u as well as the arcs (s, u) and (u, t). In the second case, the assembly would have not been completed yet, or stopped since no new read pairs could be recruited in the read mapping step. In this case, only the two ends of the insert sequence have been assembled and the center would still be missing. Here, G would consist of two contig vertices u and v and the arcs (s, u) and (v, t). However, more complex sequence and incorrectly assigned reads complicate the procedure in general.

Figure 5.16 shows an example of scaffold graphs, certain artifacts, and their removal. Each node corresponds to a contig or contig copy and the length of each node corresponds to the length of the represented contig or contig copy. Figures 5.16a and 5.16b show the scaffold graph of the example after the first and second round of assembly. After the first step, there are contigs a, b, c, and d and a has two copies a_1 and a_2. The second step adds contigs e and f. Thus, the first step creates four contigs (with five contig copies in total) and contigs a, c, and d are extended in the second step. In this example, no new reads map after the second assembly step.

This example shows two kinds of artifacts: (1) No new reads map on the corresponding contigs for vertices on a path while they do for vertices on alternative paths. (2) A vertex u is not reachable from s although u is in the same connected component as s; or t is not reachable from u although t and u are in the same connected component.

Contig b is an example for artifact (1). No new reads mapped to it in the second alignment round and thus its length does not increase from Figures 5.16a to Figures 5.16b. However, the paths $s - a_1$ and $s - a_2$ are parallel to $s - b$ in that they do not use b. I consider this as an indication that b is likely to be an artifact and is to be removed. Contig f is an example for artifact (2). There is no path from f to t using the arcs. However, when considering the arcs as undirected edges, f and t lie in the same connected component.

I propose two simple heuristics to remove such artifacs. The first heuristic, called *dead-branch removal*, attempts to remove the first kind of artifact. While running the insert assembly algorithm, each read is labeled with the step number in which it was recruited into the assembly. Each



(a) first assembly round result

(b) second assembly round result

(c) artifact removal result

(d) completing arcs insertion result

Figure 5.16: Example of the scaffolding DAG G in the Anise scaffolding step. (a) shows the DAG after the first and (b) shows the DAG after the second assembly round that is the final one in this example. The contig a has two copies a_1 and a_2. (c) shows the result after the artifact removal step on the graph in (b). Removed arcs and vertices are shown using dotted lines. Finally, (d) shows G after adding completing arcs (drawn in dashed lines). The length of a vertex corresponds to the length of the corresponding contig copy.

contig is labeled with the highest step number n of all reads that were used to build the contig. Then, a depth-first traversal is performed, starting at s and using arcs in the forward direction. After traversing all outgoing arcs of a vertex v, the label of v is updated to $\max\{x, y\}$ where x is the initial label of v and y is the maximum of the labels of the vertices reachable over all outgoing arcs *(forward graph traversal)*. The same is done starting at t, but the arcs are traversed in their reverse direction *(reverse graph traversal)*. The step labels can now be used to remove all vertices whose corresponding contig copy only contains reads that were aligned $s = 2$ or more steps before the current.

The second heuristic, called *directional tree-growing*, attempts to remove the second kind of artifacts. Again, a forward graph traversal is performed from s and a reverse graph traversal is performed from t. The vertices that could not be reached are removed.

Figure 5.16c shows the situation after the artifact removal. In this example, s was set to 0, meaning that only contigs with reads from the current round remain after the dead branch removal heuristic. Contig b only consists of reads from the first round while a_1 and a_2 also contain reads from the second round. Contig f is not reachable by a reverse traversal from t. Thus, the dead-branch removal deletes b and the directional tree growing removes f.

Adding Completing Arcs. After removing vertices, the scaffolding step adds *completing arcs* to G for creating one or more s-t-paths, without creating cycles, as follows. First, connected components are computed and a topological sorting of all vertices is performed. Then, all vertices in the component of s with an out-degree of zero are connected to the vertices in t with an in-degree of zero. In the case of connecting two vertices u and v in the same connected component with the arc (u, v), this is only done if u is topologically smaller than v. Figure 5.16d shows G after vertex removal and adding completing arcs.

Selecting the Heaviest Path. Each s-t-path corresponds to an assembly of the insert. The final aim is to reconstruct the actual insert sequence for the current site. When the sequence left and right of the insertion breakpoint is sufficiently unique, the scaffolding step attempts to resolve this using anchoring reads from the initial mapping.

Towards this end, the scaffolding step labels each contig with the number of reads from the initial mapping step. Each arc is labeled with the number of supporting paired links. Reads assigned to more than one contig copy are weighted with $1/c$ where c is the number of copies. I define the weight of a path as the sum of all vertex and contig weights. Anise then obtains a heaviest path using dynamic programming.

This assembly corresponding to this heaviest path is returned as the result of the assembly step. The scaffolding step builds the assembly sequence by iterating over the contigs along the path and placing them accordingly to the overlap length or inferred distance from mate-pair information. Completing arcs are represented by stretches of `N`s.

Enumerating Explaining Paths. Another approach for obtaining paths is to enumerate all explaining paths using Dilworth's theorem (Dilworth, 1950), previously applied for viral quasi-species haplotyping by Eriksson *et al.* (2008). Anise also includes an option to perform such a computation but by default only writes out the heaviest path.

5.5 Evaluation Using Synthetic Sequence

I split up my description of the evaluation of the insertion site prediction and the insert assembly into three sections. In this section, I describe the results of evaluation of the insert sequence assembly for synthetic sequences. In Section 5.6 I present the evaluation and results for predicting and assembling simulated insertions using real biological sequence. Finally, in Section 5.7 I show the evaluation and results for insert site prediction and assembly using real-world data. For each of these benchmarks I describe the experimental setting, the methods and metrics used for the evaluation, and the results. These evaluations are an extension of the ones previously published in Holtgrewe *et al.* (2015).

For the *synthetic sequence benchmark* that I consider in this Section, I describe and shortly discuss the insertion site prediction and assembly methods in Section 5.5.1. I then describe the metrics *recovery and error rates* in Section 5.5.2 and the program LBA for computing these metrics in

Section 5.5.3. In Section 5.5.4 I describe the generation of the synthetic sequence benchmark. Finally, I report the results of this benchmark and discuss them in Sections 5.5.5 and 5.5.6.

5.5.1 Evaluated Pipelines

Below, I describe how I used the methods Anise/Basil and MindTheGap (Rizk *et al.*, 2014, *MTG*) that are directly aimed at the detection and assembly of medium and long insertions. As outlined below, there are no published working methods for directly solving this problem at the time of writing. Thus, I also briefly discuss how I evaluated the performance of the *de novo* assemblers ABYSS and SGA when the actually inserted sequence is known and can serve as the gold standard.

My attempts in finding and using other methods for the targeted assembly of long insertions failed. I was unable to assemble longer insertions using Cortex_var (Iqbal *et al.*, 2012), even after contacting its authors for help. From its approach, this tool could in theory be able to assemble long insertions but is not advertised for this task and has not been evaluated for it yet either. SOAPindel (Li *et al.*, 2013) is only able to assemble insertions of up to 600 bp reliably and is thus not included in this evaluation. Results for SOAPindel can be found in the supplemental material of Holtgrewe *et al.* (2015). There is no code or executable available for the method by Parrish *et al.* (2011) and I thus exclude it from the evaluation. I could not get the pioneering insert site detection and assembly method NovelSeq (Hajirasouliha *et al.*, 2010) to run, even after contacting the authors. Problems with NovelSeq were also previously reported by Rizk *et al.* (2014).

For all my evaluations, I used a server with Intel Xeon E5-2667 3.30 GHz CPUs having a total of 16 physical cores with 2x hyperthreading (thus featuring 32 virtual cores in total). Further, the RAM size is 340 GB and the server has a fast storage disk array with 2 TB in total.

Basil and Anise. I ran Basil and Anise (both in version 1.0) as in a standard NGS variant calling pipeline. The read pairs were aligned against the reference using BWA Li and Durbin (2009) with default parameters.

I passed this file to Basil as the input for insert site prediction using up to eight threads for the local alignment step. Subsequently, I filtered the resulting VCF file with raw insert site predictions to the subset of sites having at least ten supporting OEA alignments on each side. I then used this filtered VCF file as the input to Anise, which I ran using default parameters and up to 32 threads.

Anise has the practical advantage of yielding one best contig for each site, which makes the evaluation straightforward.

MindTheGap. I ran MindTheGap (version 6417) as explained in its manual and using the parameters used by Rizk *et al.* (2014). For the simulated data, I ran the MTG index module with $k = 51$ (as also done by its authors on their simulated data sets) and otherwise used default

parameters. For the real-world data set, I ran the MTG INDEX module with $k = 63$ and $t = 5$. Then, I used $mrep = 15$ in the MTG FIND module.

MTG returns multiple candidate sequences for each predicted insertion site; some of these sequences are highly similar, or even identical. Thus, the evaluation of the MTG results is not straightforward.

When using simulated data, the actual inserted sequence is known and I compute the evaluation metrics for each of them. I then use the results of the best assembled sequence for an optimistic evaluation of MTG's performance. I describe the chosen approach for evaluating the result of MTG on real-world data in Section 5.7.

ABYSS and SGA. I ran ABYSS (version 1.5.1) with $k = 64$. Otherwise, I used default parameters for ABYSS-PE with the UNITIGS command. I used SGA in version 0.10.13 with the same parameters as for the *C. elegans* configuration example that ships with the tool[1].

The general purpose *de novo* assemblers ABYSS and SGA are included in the evaluation as a proxy for evaluating the NovelSeq method (Hajirasouliha *et al.*, 2010) for which no working implementation is available. In the following, I describe the evaluation process for ABYSS when the inserted sequence is known (in simulations); it is the same for SGA.

First, I extracted the OEA and shadow reads from the read alignment BAM file and performed a *de novo* assembly of these read pairs. I then passed the OEA and orphan pairs (after mapping with BWA, extracted using Samtools) as pairs to ABYSS yielding *ABYSS contigs*. The NovelSeq approach then matches the ABYSS contigs to contigs assembled from the anchor/shadow pairs (*border contigs*) using overlap alignments. Since this step is not available, I emulate it by computing approximate local matches (using NCBI Nucleotide BLAST (Altschul *et al.*, 1990)) of the ABYSS contigs to the actual simulated insertion sequence. I used BLAST for generating local matches because of its high sensitivity.

For each simulated inserted sequence, I picked the best ABYSS contig by its BLAST bit score. This contig was then compared to the actual simulated insert sequence using a custom tool called LBA (*Local Block Aligner*, explained further in Sections 5.5.2 and 5.5.3). Note that the same orphan contig might be assigned to multiple insertion sequences. Matches with less than 95% identity were filtered out. I expect that this is an optimistic approximation of the NovelSeq matching approach since the orphan contig does not have to overlap with both or any NovelSeq *border contig* (cmp. (Hajirasouliha *et al.*, 2010)).

More details on the command lines are given in Appendix D.

5.5.2 Recovery and Error Rate

Since the actual inserted sequence S is known in this benchmark, the assembled sequence A can be compared to it directly. Because of repeated sequence and assembly errors, parts of S can be missing in A and A might also contain other assembly errors.

[1] `https://github.com/jts/sga/blob/master/src/examples/sga-celegans.sh`

I compute a pairwise alignment of S and A with affine gap scores and free end gaps in both sequences using LBA. I call longer segments (> 5 bp) in S that align against gaps *uncovered* while I call the remainder of S *recovered*. The ratio of the length of the recovered part of S with the total length of S yields the *recovery rate*. For the recovered sequence, the error rate of the alignment (*recovery error rate*) is also computed.

I describe LBA in detail in Section 5.5.3. An example alignment is given in Figure 5.17. LBA is distributed together with Anise.

5.5.3 Evaluation Program LBA

The program LBA takes two sequences (subsequently called *(actual) insert* and *assembly*), performs an alignment, computes statistics of the alignment, and then writes out the resulting statistics and the alignment to a text file. I used the sequences S and A from Section 5.5.2 as the input to LBA.

LBA computes a pairwise alignment of the sequences using the standard global alignment algorithm in SeqAn (Döring *et al.*, 2008) with affine gap costs (match = 8, mismatch = -8, gap open = -20, gap extension = -1). The program then analyzes the alignment as shown in Figure 5.17.

Leading and trailing gaps in the inserted sequence (the *insert*) are ignored. Gaps longer than 5 bp in the assembly are counted as *uncovered*, regardless of their position. Mismatches and shorter gaps are counted as *errors*, non-trailing gaps in the insert are always counted as errors. Since assemblers often insert `N`s for gaps in the assembly, whose size is not known, `N`s at the border of longer gaps are not counted as errors.

The program reports the length of both sequences, the alignment length, the number of gaps, matches, and mismatches (their sum is the number of errors), as well as the number of covered characters in the assembly. The error rate is then computed from the alignment length and the number of errors while the covered sequence is computed from the length of the insert and the number of covered insert bases.

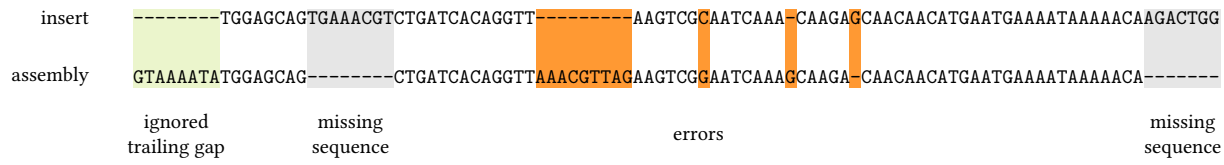


Figure 5.17: Example for scores in an LBA alignment (see Sections 5.5.2 and 5.5.3). The known insert sequence is shown at the top and the assembled sequence at the bottom.

5.5.4 Synthetic Sequence Benchmark Setting

The aim of this benchmark is to evaluate the performance of assembling inserted sequence in the case of these sequences being highly similar. This case occurs when an assembler fails to assemble regions of the genome because of such high similarities or repeats.

For each benchmark repetition I insert multiple simulated copies of the same sequence into the reference as explained below. The task is the complete and correct assembly of the inserted sequences. I did not evaluate the detection of the insert location since it is trivial when using uniformly random sequence as done here.

Figure 5.18 shows the process of generating the benchmark settings. First, I simulated a reference sequence by picking each nucleotide uniformly at random. Then, I simulated an insert sequence R of length n, also uniformly at random. For a given polymorphism rate p, I generated k copies $R_1, \ldots, R_k$ of R, each with Hamming distance $d = \lfloor p \cdot n \rfloor$ to R (*i.e.*, a pairwise difference of $\approx 2d$). This was done by changing d characters of R, picked uniformly at random.

I simulated the reference of sufficient length such that the repeats can be inserted with at least 5000 bp unique reference sequence adjacent to each copy location at both ends. I used the reference with inserted repeat copies as the donor sequence. I simulated Illumina-like reads of length 100 bp from the donor sequence with a coverage of 30x using the read simulator Mason 2 (Holtgrewe, 2010). For the simulation, I used a typical Illumina error profile, each read having an average error rate of 1%. I simulated the template length following the normal distribution $\mathcal{N}(400, 40)$. For each combination of p, k, and n, I performed ten repetitions of the simulation (with different random seeds) and the assembly.

5.5.5 Insert Assembly Results

The performance results for the insert site assembly are shown in Table 5.2. This table shows the percentage of recovered sequence and the error rate of the assembly for the four considered

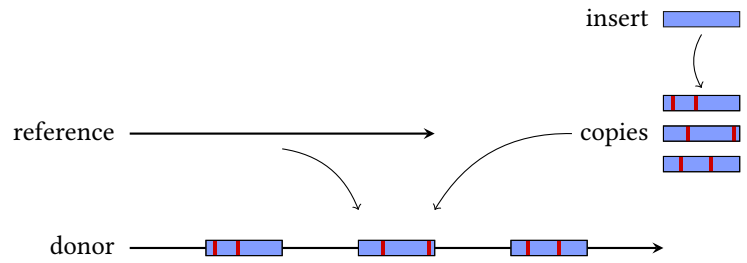


Figure 5.18: Overview of the synthetic copy benchmark generation. Reference and insert sequence are both simulated by picking nucleotides uniformly at random nucleotides. The to-be inserted sequences are then generated by placing errors on each copy, independently of each other. The errors are shown as red bars in the insert copies.

			ANISE		MTG		ABYSS		SGA	
p	k	n	r-rate	e-rate	r-rate	e-rate	r-rate	e-rate	r-rate	e-rate
0.5	2	1000	100.00	0.04	69.67	1.50	93.78	0.44	86.98	0.42
0.5	2	5000	100.00	0.28	55.35	1.35	93.54	0.50	99.25	0.50
0.5	4	1000	100.00	0.06	85.50	1.42	35.48	0.29	44.47	0.50
0.5	4	5000	100.00	0.45	99.99	0.85	18.49	0.26	23.44	0.44
1.0	2	1000	100.00	0.01	79.13	0.78	83.03	0.88	62.59	0.69
1.0	2	5000	100.00	0.03	99.99	0.96	86.90	0.96	89.17	0.97
1.0	4	1000	100.00	0.01	99.97	1.40	25.43	1.64	43.92	1.76
1.0	4	5000	100.00	0.21	-	-	7.13	1.28	11.09	1.66

Table 5.2: Synthetic copy benchmark results. The recovery rate *(r-rate)* and error rate *(e-rate)* are reported in percent for different synthetic copy benchmark data sets with varying polymorphism rate p (also given in percent), number of copies k, and insert length n. The best entries in terms of recovery and error rate are underlined. The table shows the mean results of ten repetitions.

methods and different values of p, k, and n. An extend version of this table can be found in the supplemental material of (Holtgrewe *et al.*, 2015).

Anise achieves the best recovery rate overall while keeping the error small. For a polymorphism rate of 1%, the copy separation works better than for a polymorphism rate of 0.5%.

MTG is only competitive in terms of recovery rate for few configurations and achieves overall worse results in terms of error rate. MTG is unable to assemble any insertions in the case of $p = 1\%, k = 4, n = 5000$. ABYSS and SGA achieve worse results than Anise in terms of the evaluated metrics.

5.5.6 Discussion

The assembly of similar copies is hard for general purpose *de novo* assemblers and Anise consistently achieves results that are superior to general purpose assemblers. Higher error rates in ABYSS and SGA are most probably caused by assembling reads from different locations. On the other hand, the lower recovery rates are most probably caused by contigs stopping at regions with higher number of differences (*e.g.*, ≥ 3 errors withing 10 bp). This leads to shorter assembled contigs with a high identity, since the assembler stops at problematic sequence parts.

For example, for $p = 0.5\%$, $n = 5000$, $k = 4$, SGA yielded an average of 29 contigs with an average N50 score of 345 (data not shown in table). ABYSS yielded 77 contigs on average with an average N50 score of 145 and MTG failed in assembling the contigs. My method Anise consistently yielded 4 contigs and an average N50 score of 5800 — optimal results in terms of contig counts and lengths.

Anise is consistently better than MTG for the assembly of these synthetic data sets since MTG

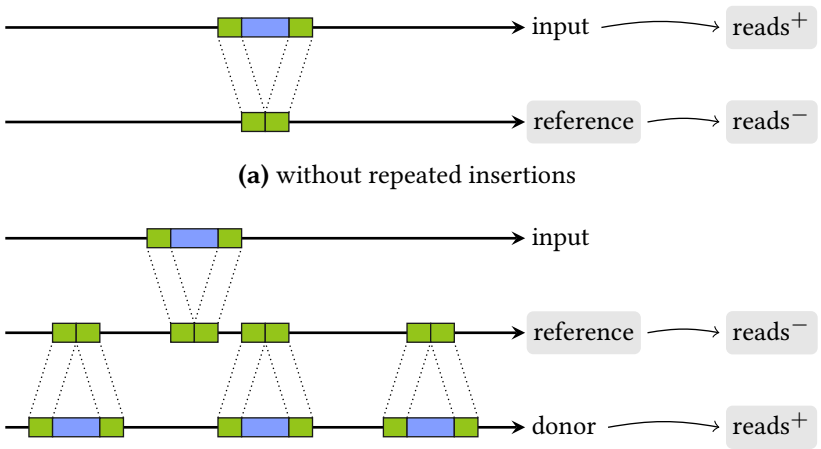
does not implement any measures for handling repeated sequence. The performance of Anise improves with higher difference between copies (*i.e.*, larger p) since the differences are used for separating the copies. Of course, there will be a point from which on the regions will differ greatly enough for the general purpose assemblers to assemble the sequence properly. MTG profits from a higher polymorphism rate p in terms of recovery and error rate. However, it does not succeed in assembling any sequence for long insertions for the higher error rate of $p = 1.0\%$.

5.6 Evaluation Using Simulated Insertions in Real Biological Sequence

In this section, I present my evaluation in the *simulated insertions benchmark*. In Section 5.6.1 I describe the benchmark setup. I then compare the two OEA cluster selection options in Basil in Section 5.6.2 and compare the results of Basil for insert site prediction to the performance of MindTheGap in Section 5.6.3. In Section 5.6.4 I describe the results of the insert sequence assembly. Finally, I discuss all these results in Section 5.6.5.

5.6.1 Simulated Insertions Benchmark Setting

The aim of this benchmark is to evaluate the performance of the considered methods on real biological sequence. Using this real sequence, I used simulated insertions and reads to have a ground truth in the evaluation. I simulated one data set with and one without repeated insert sequence



(a) without repeated insertions

(b) with repeated insertions

Figure 5.19: Visualization of the data generation for the simulated insertions benchmark (a) without and (b) with repeats. The figures also indicate which sequence is used as the donor for simulating the reads with *(reads$^+$)* and without insertions *(reads$^-$)*. The labels for the reads and reference used for the evaluation are shaded in light gray.

as shown in Figure 5.19. I generated input data for the benchmark from the sequences of the human chr. 22 (from the hg18/GRCh36 assembly), the genome of *D. melanogaster* (flybase r5.29), and the genome of *C. elegans* (ENSEMBL release 60). For each setting, I obtained a heterozygous and a homozygous data set as described below.

For obtaining a data set without repeated insertions from an input sequence, I simulated 1000 deletions into the input sequence, yielding the *reference* sequence for the subsequently variant detection pipelines (see Figure 5.19a). I picked the size of the deletions uniformly at random between 50 and 5000 bp and their positions are picked uniformly at random in the whole genome. I simulated reads from the input sequence that contains the insertions with respect to the reference sequence, yielding the read set $reads^+$. I simulated reads from the reference that contains no insertions, yielding the read set $reads^-$. For obtaining a heterozygous data set I generated reads$^+$ and reads$^-$ to a coverage of 30x each and merged them, and for obtaining a homozygous data set I generated reads$^+$ to a coverage of 60x.

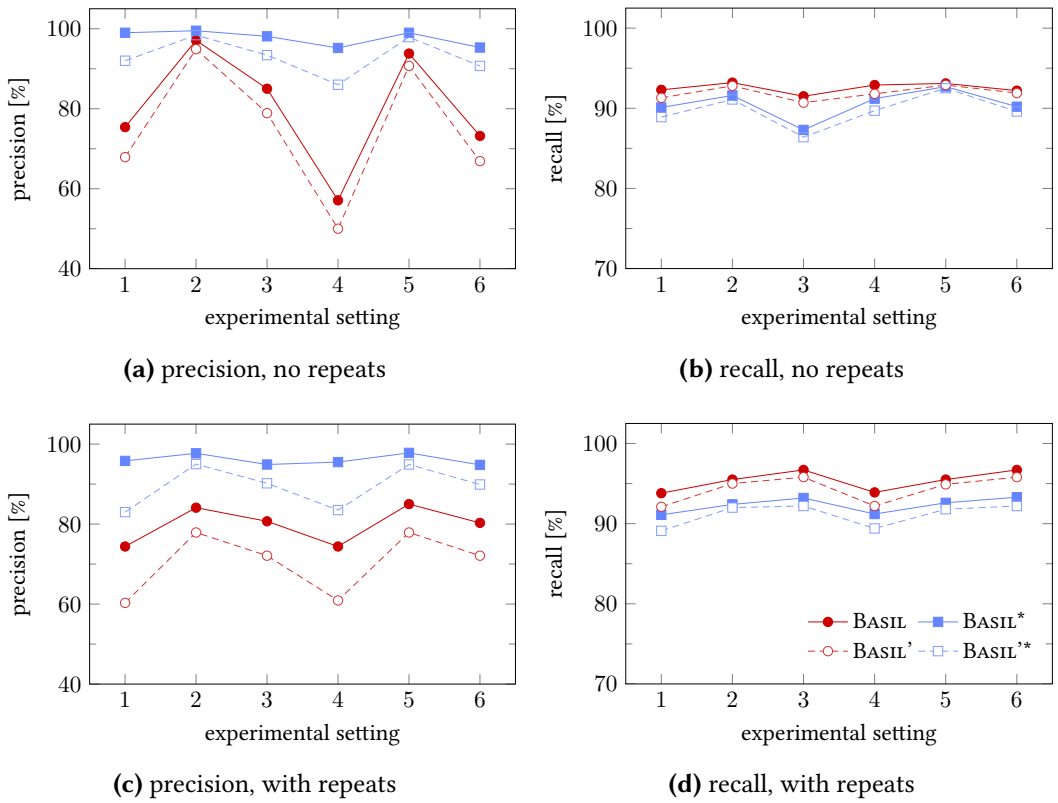
The generation of the homozygous data set is more involved (see Figure 5.19b). I first simulated the *reference* for the later variant analysis by again deleting 1000 segments in the same fashion as for the data set without repeated insertions. Again, the reads simulated from this sequence comprise the set $reads^-$. I then simulated the *donor* sequence by inserting three copies of each deleted segment at random positions in the genome. I introduced small variants into these copies as I did previously for the synthetic benchmark using $p = 0.5\%$. Simulating reads from the donor yields $reads^+$. Again, I obtained a heterozygous and a homozygous data set from the reference and the donor as for the case without repeated insertions. I used Mason 2 (Holtgrewe, 2010) for the simulation of the variants and the repeats using the same settings as described in Section 5.5.4.

The inserted sequence can contain low complexity regions and insertion sites might fall into non-unique parts of the reference. In such cases, assemblers make large-scale assembly errors. The assembly results are subjected to an analysis using LBA. Large-scale assembly errors lead to a high number of errors that skew the insert assembly evaluation results. To accommodate for this, I trim outliners. I differentiate between cases where the recovery error rate is above and below 1% and only report the results for the cases where the error rate is $\leq 1\%$.

5.6.2 Basil Cluster Selection Evaluation

In this section, I present a comparison of the two OEA cluster selection strategies in Basil described in Section 5.3.3. The first one is solving the problem through the chaining algorithm and the second one is using the set cover heuristic. Without any experimental evaluation it is unclear which method yields the better results in terms of insert site prediction.

Results. Figure 5.20 shows the results of Basil using the two OEA cluster selection strategies. The results are shown in terms of insert site prediction sensitivity and specificity on the benchmark with simulated insertions into real biological sequence on the heterozygous data sets. I include the results of Basil before and after filtration such that a comparison of the recall values



(a) precision, no repeats

(b) recall, no repeats

(c) precision, with repeats

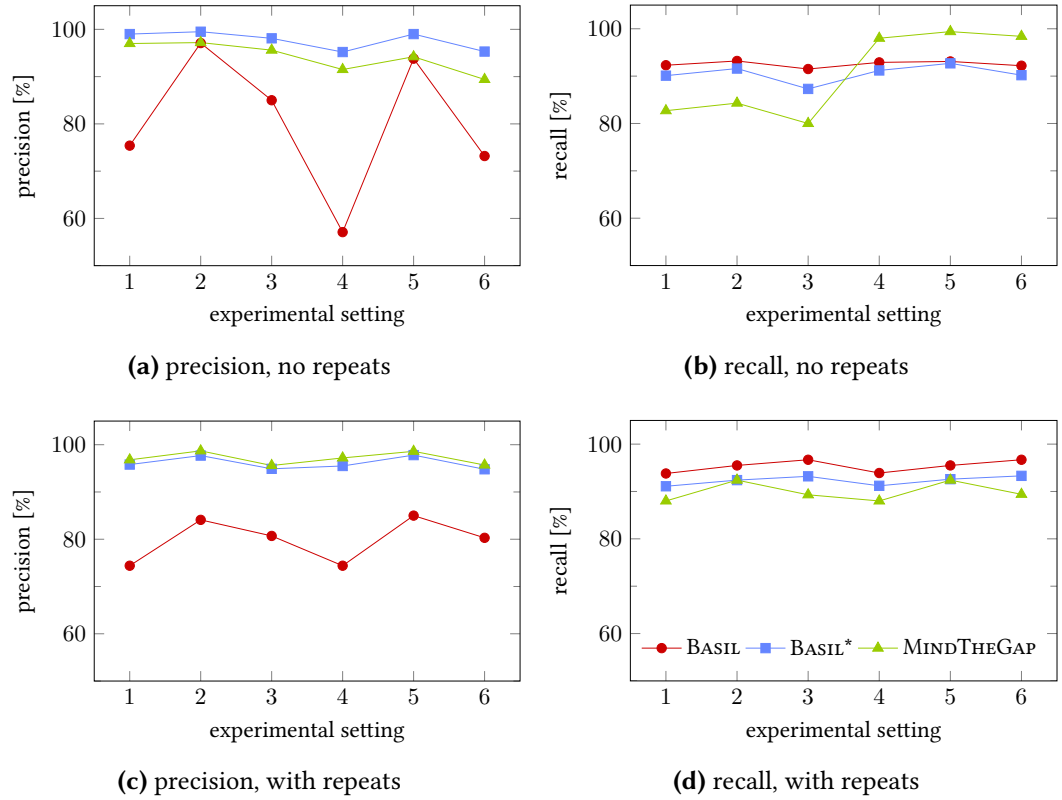
(d) recall, with repeats

Figure 5.20: Comparison of the two OEA cluster selection strategies in BASIL in terms of insert site prediction sensitivity and specificity. The experimental settings are denoted on the horizontal axes. For the heterozygous variants, they are numbered (1) human chr. 22, (2) *D. melanogaster*, (3) *C. elegans* and (4-6) for the homozygous variants using the same reference sequences. Note that the vertical axes have different lower bounds. The legend for all plots is shown in (d), the meaning of the variant names is given in the main text.

is possible without any bias of the post-filtration step. The lines labeled BASIL are for the unfiltered results of BASIL using chaining for OEA cluster selection. BASIL* is the label for the filtered results using the same selection strategy. The lines labeled BASIL' are for the unfiltered results of BASIL using the set cover heuristic for OEA cluster selection. BASIL'* is the label for the filtered results using this selection strategy.

The chaining algorithm is better than the set cover heuristic in all considered cases with respect to both sensitivity and specificity. In particular, the results are better in terms of precision.

Discussion. The set chaining algorithm consistently achieves better results than using the simple set cover heuristic. The most likely explanation is that the simple heuristic for solving the set cover model makes a notable number of suboptimal decisions by selecting too many false positives and too few true positives. Also, the chaining algorithm is much easier to implement. This makes it clearly the better choice for selecting OEA clusters and I used it for the OEA cluster



(a) precision, no repeats

(b) recall, no repeats

(c) precision, with repeats

(d) recall, with repeats

Figure 5.21: Comparison of the insert site prediction performance of Basil with and without filtration and MindTheGap find. The experimental settings are the same as in Figure 5.20. Note that the y axes have a lower bound of 50%. The legend for all plots is shown in (d).

selection in all subsequent experiments. Further, it can be seen that the filtration gives a boost in precision (up to ≈ 15 percent points) with a small trade off in terms of recall (below ≈ 5 percent points).

Using chaining for the selection of OEA clusters is a novel idea in Basil over the previous NovelSeq method that uses the set-cover model that improves the paired read driven results for insert site prediction.

5.6.3 Insert Site Prediction Results

Results. Figure 5.21 shows the performance of insert site prediction for Basil and MindTheGap in terms of precision and recall on the heterozygous data set. Again, I include both Basil and Basil*.

Basil* shows recall results of around 90%. MTG shows worse results when simulating no repeated copies in the case of heterozygous simulated insertions but better results in the case of homozygous simulated insertions. In the case of simulating repeated copies, both variants of

		Anise				MindTheGap			
repeats	data set	ins. Mbp	rec. Mbp	r.-rate	e.-rate	ins. Mbp	rec. Mbp	r.-rate	e.-rate
no	chr. 22	2.1	2.0	92.6	0.03	0.9	0.8	96.3	0.03
	C. elegans	2.0	1.9	93.5	0.02	1.5	1.5	97.5	0.03
	D. melanogaster	2.4	2.3	98.8	0.01	2.0	2.0	99.2	0.02
yes	chr. 22	6.1	5.8	95.7	0.24	2.1	1.8	86.3	0.65
	C. elegans	6.3	5.8	92.5	0.25	4.1	3.8	90.7	0.66
	D. melanogaster	6.9	6.7	97.4	0.24	5.8	5.3	91.1	0.66

		ABYSS				SGA			
repeats	data set	ins. Mbp	rec. Mbp	r.-rate	e.-rate	ins. Mbp	rec. Mbp	r.-rate	e.-rate
no	chr. 22	2.5	2.1	84.7	0.01	2.4	2.3	93.6	0.01
	C. elegans	2.4	2.1	86.8	0.01	2.4	2.2	92.1	0.01
	D. melanogaster	2.5	2.3	93.8	0.01	2.5	2.4	97.4	0.01
yes	chr. 22	7.3	2.7	37.0	0.38	7.1	4.1	57.9	0.49
	C. elegans	7.3	2.7	37.3	0.38	7.2	4.1	57.0	0.50
	D. melanogaster	7.5	2.9	38.8	0.38	7.4	4.4	59.2	0.50

Table 5.3: Insert assembly results on simulated insertions into real biological sequence for heterozygous insertions. The table shows the number of inserted Mbp (*ins. Mbp*), recovered Mbp (*rec. Mbp*), recovery rate (*r.-rate*, in %), and error rate (*e.-rate*, in %) of the assembled sequence. The values for *ins. Mbp* and *r.-rate* are given for the sites that the methods attempted to assemble. The table shows results for the homozygous data sets without (first three rows) and with (last three rows) repeated insertions. The best values for recovered sequence and error rate in each row are underlined.

Basil achieve the consistently best results in terms of recall. Both Basil and MTG achieve precision rates between 90 and 100%. Basil is slightly better when simulating no repeated copies and slightly worse when simulating such repeated copies.

Discussion. In terms of recall, Basil* outperformed MTG in three quarters of all cases and in terms of precision, both tools are competitive. Thus, both Basil* and MTG are competitive state of the art methods for accurate insert site prediction.

Remarkably, MTG shows better recall results when simulating repeated copies. This could be caused by the inserted sequence having a higher coverage than the rest of the reference. This fact could give a stronger signal in the read k-mer analysis of the MTG find module.

5.6.4 Insert Assembly Results

Sequence Assembly Results. Table 5.3 shows the results of the insert assembly in the heterozygous case. Table G.1 (p. 213) shows the results for the homozygous case. The difference between the results on homozygous and heterozyous insertions is small for all but MindTheGap that recovers up to 20 percent points more sequence of the homozygous insertions. For all

methods, the results for *D. melanogaster* are better than for the other reference genomes.

It is important to keep the difference between recovered bases *(rec. Mbp)* and the recovery rate (*rec.-rate*) in mind. The first is the absolute number of bases that were recovered (in the case of Anise and MTG) or contigs for which a BLAST match was found (in the case of ABYSS and SGA). The second is the number of recovered bases, relative to the number of bases of insertions for which assembly was attempted (in the case of Anise and MTG) or for insertions for which at least one BLAST match was found (in the case of the ABYSS and SGA).

Overall, the general purpose assemblers achieve better results in terms of recovered bases than the targeted assemblers in terms of recovered bases and error rate on the data sets without repeats. Here, SGA is slightly better than ABYSS. When only considering the targeted assemblers, Anise consistently yields better results than MTG.

On the data sets with repeats, the relative order of the tools in terms of the considered metrics changes. While Anise continues to yield good results in terms of recovery rate and error rate, the performance of the other methods decreases. Anise achieves the highest number of recovered bases and the lowest error rate. MTG continues to have a high recovery rate but has a relatively low number of recovered bases and a relatively high error rate. Overall, MTG is better than ABYSS but worse than SGA.

Sequence Assembly Results, Depending on Insert Length. The numbers presented in the paragraph before already give a good evaluation of the success in insert assembly. However, they do not allow to estimate success in assembling inserted sequence depending on the length of the inserted sequence. Figure 5.22 shows the percentage of recovered sequence for each considered assembly method on the heterozygous data set, broken down by the length of the inserted sequence. Figure G.1 (p. 214) shows the results on the homozygous data sets.

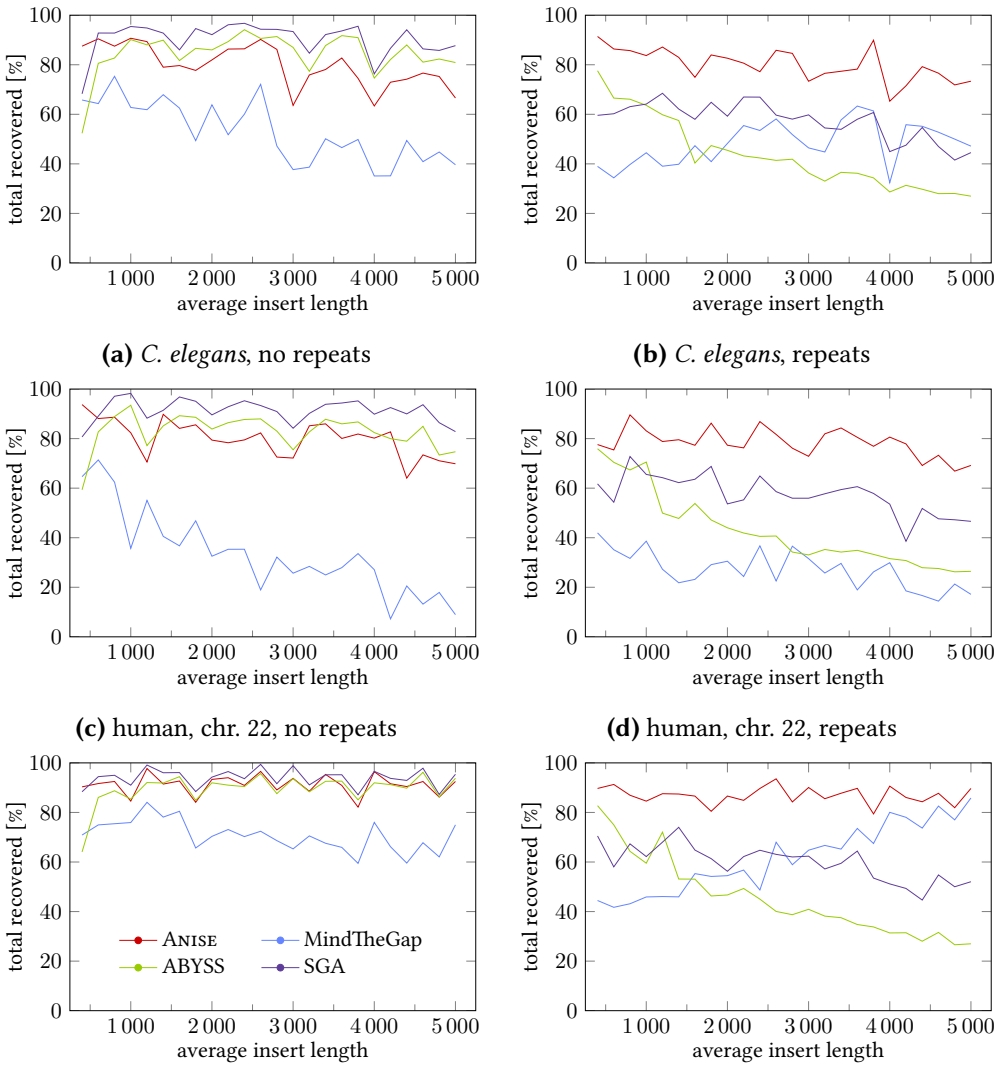
To obtain these figures, I computed the percent of recovered bases for each inserted sequence for each method. This yielded a list of pairs (ℓ, r) where ℓ is the length of the insert and r is the percentage of recovered sequence. For plotting, I partitioned the insertion lengths into window of lengths 200 and computed the average recovery rate.

Generally, there is a downward trend of recovered sequence percentage with growing insert size, except for MTG on the sequence of *D. melanogaster*. The general purpose assemblers show the best results on the non-repeat data sets. Anise shows robust results on all data sets. Of the two targeted assembly methods, Anise is consistently better than MTG in assembling the inserted sequence.

5.6.5 Discussion

Anise robustly achieves competitive results in comparison to all other methods or is superior to them in terms of recovered sequence, both on the data sets with and without repeated insertions. Also, Anise shows a consistently low error rate.

The general-purpose *de novo* assemblers SGA and ABYSS are quite successful in the considered



(a) *C. elegans*, no repeats

(b) *C. elegans*, repeats

(c) human, chr. 22, no repeats

(d) human, chr. 22, repeats

(e) *D. melanogaster*, no repeats

(f) *D. melanogaster*, repeats

Figure 5.22: Percentage of recovered sequence data for the heterozygous data sets with and without repeats. The windows are shown on the x axis and a point (x, y) describes the average y of all insertions with lengths $(x - 200, x]$. The legend for all figures is given in (e).

metrics when no repeats are present. However, they have problems achieving good results in the presence of repeats.

This is further evidence that the good results come from "jigsawing together" the results with BLAST matches, which however fails in the presence of repeats (also compare Section 5.6.5). In the case of having no repeats, the local alignments generate segments that align with few errors against the true inserted sequence. When repeated copies are present, these assemblers fail in correctly separating these copies and a large portion of the inserted sequences remains uncovered.

MTG is more successful in assembling the homozygous cases than the heterozygous cases. This can be explained by the higher coverage of the inserted sequence (60x instead of 30x) which is an easier input for the memory-optimized (instead of quality-optimized) assembly module. MTG is more robust in terms of recovery rate than the general purpose assemblers but not as robust as Anise.

When considering the success in assembly dependent on the insertion length, MTG shows problems with growing insertion length. It is more successful on the simpler *D. melanogaster* data set than on the human and the *C. elegans* data sets. Remarkably, MTG is more robust in assembling long sequences in the presence of repeats, presumably because the insertions have three times higher coverage than in the case of having no repeats. However, the resulting error rate is much higher in the case of simulating repeats and it is not competitive to Anise.

5.7 Evaluation using Real-World Data

In this section, I describe the evaluation of insert site prediction and sequence assembly using real-world data. After a short description of the data set, I give a list of experiments and the results of Anise/Basil and MindTheGap in Sections 5.7.1–5.7.10. For each experiment, I describe the performed computations, report the results, and shortly discuss them. Finally, I give an overall discussion of these results in Section 5.7.11.

This evaluation is a greatly extended and refined version of the previously published one from (Holtgrewe *et al.*, 2015). The main differences are filtering the Anise contigs to having at least 10 supporting OEA pairs on each side instead of requiring a count of 30 as in the original publication. Further, I made the definition of *novel bases* (given below) more clear and stringent.

Data Set. In this third benchmark, I use an Illumina HiSeq 2000 paired-end whole-genome sequencing data set. The reads have a length of 101 bp and cover the genome to 64x. From the read alignments I estimated the template size to be approximately normally distributed with $\mu = 422$ and $\sigma = 56$. The reads were sequenced from the individual NA12878 that has been used extensively in previous studies, *e.g.*, the original GATK publication by DePristo *et al.* (2011). For short, I call this benchmark the *real-world benchmark*. The aim of this benchmark is to evaluate the performance of the insert site detection and the targeted assembly of inserted sequence on real-world data.

I use the BAM file[2] provided on the NCBI servers as the input for Basil and Anise and convert it into a FASTA file to use it with MindTheGap. The creators of the BAM file used the human reference sequence hg18/GRCh36 for their analysis.

Since the ground truth is not known for my data set and this evaluation aims at targeted assembly, I do not attempt to run any general purpose *de novo* assemblers on this data set. The lack of

[2] `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/working/20101201_cg_NA12878/NA12878.hiseq.wgs.bwa.raw.bam`

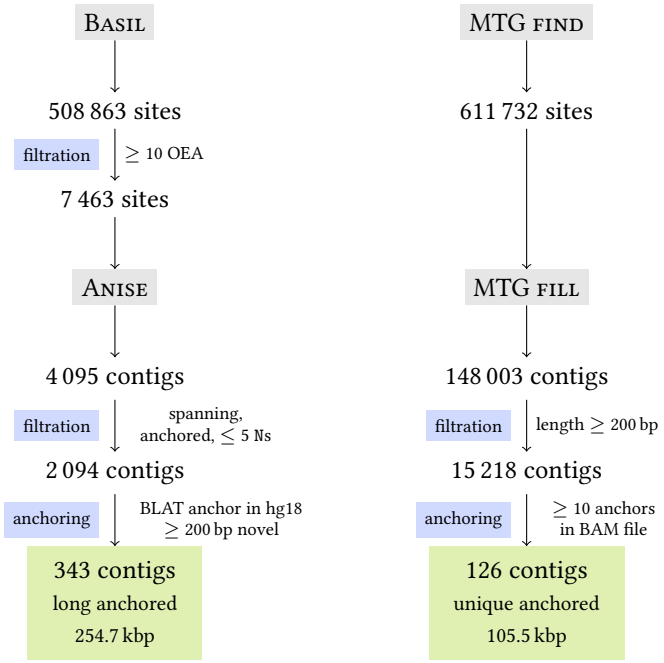


Figure 5.23: Overview of the filtration of predicted sites and the filtration and anchoring of assembled contigs. This overview corresponds to Sections 5.7.1– 5.7.3.

ground truth makes it hard to gauge the quality of the assembly results. Thus, I use several complementing approaches for obtaining quality metrics.

Resource Consumption. I ran Basil allowing up to 8 threads, it took 2h 22min to finish and used up to 8.4 GB of RAM. Anise ran with 32 threads, took 30h 34min to complete, and had a peak memory consumption of 130 GB of RAM (4 GB per core). MTG offers no parallelization and was run on a single core. It took 40h 15min for the k-mer indexing step, 5h 7min for the breakpoint detection step, and 16h 28min for the assembly. The three MTG steps had a peak memory consumption of 6, 14, and 6 GB of RAM.

5.7.1 Filtration of Predicted Sites

Methods. For Basil, I filtered the resulting sites to those having at least 10 supporting OEA pairs on each site. The result of the MindTheGap find module does not have to be filtered before being passed to the fill module.

Results. Basil yields a total of 508 863 tentative insert sites and 7 463 remain after filtration. The MTG find module detects 611 732 tentative insert sites. Figure 5.23 gives an overview of

the structure of this and the following two steps and their results. I will discuss these results together with the results of Section 5.7.2.

5.7.2 Filtration of Assembled Contigs

Methods. With default settings, Anise yields one sequence for each Basil site when the assembly succeeds. Each such *Anise contig* is annotated with the following information: (1) the genomic location of the site as predicted by Basil, (2) whether it was anchored on the left and/or right side, and (3) whether Anise was able to span the insertion using overlaps and read pairing information. I call contigs fulfilling these conditions *spanning* contigs. I call an Anise contig *anchored* on the left side if its assembly incorporated a read on the forward strand of the genome in the input BAM file and vice versa for right anchoring of contigs.

I filter the Anise contigs to those marked as spanning and anchored on both sides and have no stretch of 5 `N`s or more. This yields *prefiltered* Anise contigs.

Since MindTheGap is based on the analysis of k-mers and does not use read pairing information, there is no obvious way of filtering the result of its find module. As described in its manual, I directly pass the find result to the fill module. For filtration, I limit the resulting MTG contigs to those with a length of at least 200 bp.

Results. Figure 5.23 gives an overview of the structure of this, the previous, and the next section.

Anise yielded 4 059 contigs in total, thus the assembly fails to produce contigs for 3 404 insertion site candidates. Of these contigs, 2 094 prefiltered Anise contigs remained.

MTG The fill module generated candidate insert sequences for a lot of the candidate insert sites, including a large number of very short contigs and up to 62 candidate insert sequences for an insertion site. The generated inserted sequences for one site were often very similar. A quick inspection revealed that a lot of the longer assembled insert sequence contains sequence from low-complexity repeat regions. MTG fill generated 148 003 contigs in total, of these 15 518 remain after filtering to a minimal size of 200 bp.

Discussion. Of the approximately 7.5 k predicted breakpoints, Anise only yielded approximately 4 k contigs. For the remaining approximately 3.5 k contigs, the assembly failed. The most likely reason is that the majority of these predicted locations are false positives generated by Basil.

The two main reasons for false positives in Basil are spurious alignments and ambiguities in the paired-end signals. For spurious alignments, the shadow reads of the OEA pairs cannot be assembled into contigs of sufficient length. In this case, the assembly process stops after failing in the first step. In the case of ambiguous paired-end signals, the shadow reads can be assembled into contigs but the assembly cannot continue in the next step (also see Section 5.3.2).

Another reason for failures to create spanning contigs is that Anise fails to assemble the whole insertion because of repeats or a lack of coverage. However, judging by the results of the success in sequence assembly depending on insert length in Section 5.6.4, I expect this to be the reason only for a small percentage.

The high numbers of contigs per site yielded by MTG makes the results of this method cumbersome to interpret. Also, about nine tenth of the contigs yielded by MTG are shorter than 200 bp and are not of further interest in this evaluation.

5.7.3 Re-Anchoring of the Assembled Contigs

Methods.

Anise For Anise, the generated contigs are generated from the anchors of the OEA read pairs as well. Thus, the assembled contigs contain parts of the reference sequence at their ends. I used the local aligner BLAT (Kent, 2002) for aligning the Anise contigs to hg18.

I chose BLAT for this task since it is much faster than BLAST, allows large gaps in its generated alignments, and creates alignments similar to those generated by LBA. This allows the alignment of the borders of the Anise contigs while keeping the insertion itself unaligned. In this experiment and in all other experiments below where I use BLAT, I use the terms *identity*, *error rate* and *score* to mean *BLAT identity*, *BLAT error rate* and *BLAT score* (also see Appendix H).

In more detail, I discarded all BLAT matches that align less than 300 bp of the Anise contigs and have an identity of less than 90%. I chose to allow a BLAT error rate of 10% since BLAT identity includes a (scaled-down) penalty for gaps in the case of insertions and there might be small variants (so-called *micro-indels*) around the insertion site (Yalcin *et al.*, 2012). Also, the BLAT matches were filtered to those that align within 400 bp of the insertion site prediction for which assembly was attempted.

I call all such Anise contigs *anchored*. Of these contigs, I call the ones that have at least 200 bp that are not covered by the anchor alignments *long anchored*. I do not consider contigs that are not long anchored further in this evaluation. For the Anise contigs, I estimated the number of *novel bases* in an assembled contig by counting the bases between the anchoring alignments at the border.

MTG The MindTheGap contigs only consist of the actual insert and the genomic context itself is not available. Thus, I counted all bases in the assembled contigs as *novel*. Further, there are still multiple MTG contigs for many insertion sites. I attempted to anchor the MTG contigs using paired-end information in the following approach.

For each insert site call with at least one filtered MTG contig, I extracted the OEA pairs from the original BAM file where the anchors mapped within 800 bp of the predicted site. I then attempted to anchor the MTG contigs to the reference using these read pairs by mapping the shadow reads to the contig with BWA (Li and Durbin, 2009).

For each aligning shadow read, I counted the OEA read pair as *anchoring* for the contig that it aligns to. If the anchor aligned to the forward strand in the input BAM file, I counted the OEA pair as *left anchoring* and I counted it as *right anchoring* if the anchor aligned to the reverse strand. I discarded all contigs with less than 10 anchors on either side. If there were multiple contigs for a predicted site, I picked the one with the highest anchor count for the further analysis (ties were broken randomly).

In summary, an Anise contig is anchored if its borders align to the reference with sufficient quality. A MTG contig is anchored if there are at least 10 OEA pairs on each side of the insertion site where one read (the anchor from the BAM file) aligns on the reference and the other read (the shadow from the BAM file) aligns to the contig. Both kinds of anchoring are evidence that the contig is part of the donor's genome at the given position.

Results. Figure 5.23 (p. 136) shows the structure of the evaluation process and the results up to this point.

Anise For Anise, of the 2 094 prefiltered contigs, 2 093 could be anchored in hg18 using BLAT. Of the anchored contigs, 343 are long anchored contigs, containing a total of 254.7 kbp of estimated inserted sequence. I manually inspected the single unanchorable contig and the predicted corresponding site was directly next to an ALU repeat.

MTG For MTG, of the 15 218 filtered contigs, 127 can be anchored using the aligned reads in the BAM file (one is a duplicate of another contig). These 126 unique contigs have a total of 105.5 kbp of sequence.

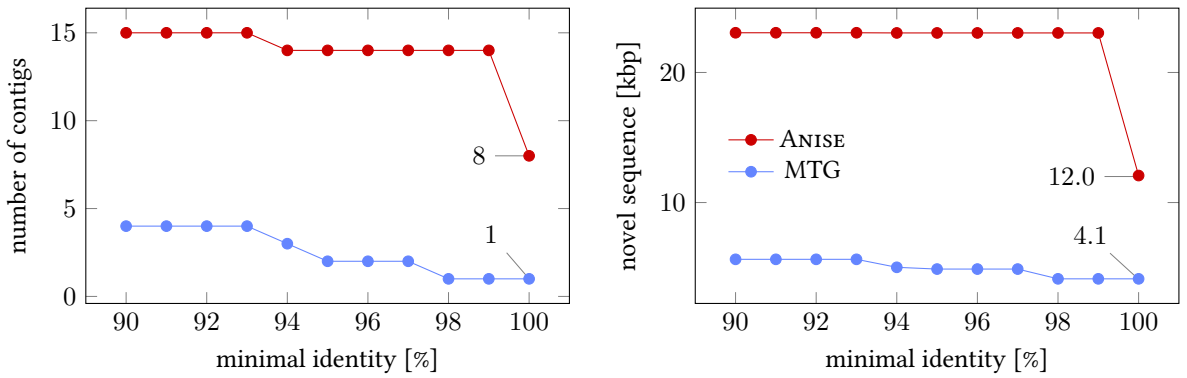
Discussion. All but one prefiltered contig created by Anise can be anchored to hg18, thus passing this sanity check. Of the filtered MTG contigs, only 126 unique contigs can be anchored to the reference sequence using the aligned reads.

Thus, Anise creates 2.7 times as many contigs that pass the simple sanity check of anchoring contigs and have at least 200 bp. The Anise contigs consist of 2.4 times more novel sequence than the MTG contigs.

5.7.4 Validation Using Fosmid Sequence

In this experiment, I attempt to validate the assembled contigs with the known sequence of previously published fosmids. Figure 5.25a (p. 141) gives an overview of the situation in this evaluation.

Methods. In a previous study of structural variants, Kidd *et al.* (2008) published the full sequence of 454 fosmids from the same individual NA12878 *(fosmid contigs).* The sequences were assembled from traditional capillary sequencing, have lengths of approximately 40 kbp, and are annotated to be of high quality.



(a) contig count

(b) novel base pairs

Figure 5.24: Results for the validation of the insert sequence assembly using fosmid sequences in terms of (a) aligned contigs for which BLAT matches with the given or better identity exist. (b) shows the number of novel base pairs for these contigs. The legend for both figures is shown in (b).

These fosmid contigs were generated for the validation of SV breakpoints in general and do not exclusively contain insert sequence. Thus, it cannot be expected that all Anise and MindTheGap contigs can be found within the 17.5 Mbp of these fosmids. Nevertheless, they can be used as ground truth for parts of the genome sequence of the considered individual.

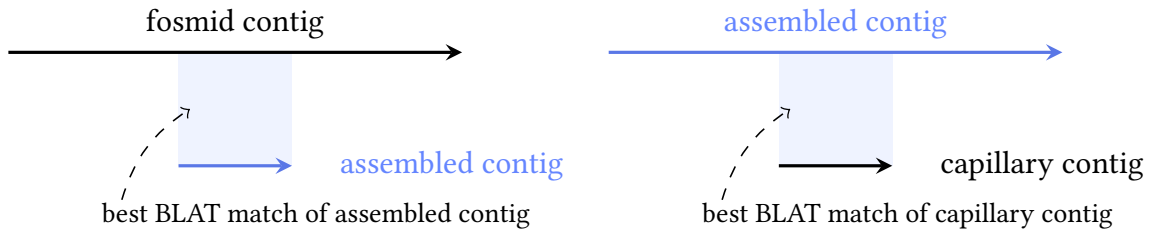
High-quality alignments of Anise and MTG contigs to the fosmid sequence are evidence that these contigs are of high quality. I compute BLAT alignments using the Anise and MTG contigs as the query and using the fosmid contigs as the reference. I filtered these BLAT matches to those covering at least 90% of the query and the BLAT alignment with the highest score (cmp. Appendix H) was picked for each query for the further evaluation.

Results. Figure 5.24 shows the results of the verification using fosmid sequence in terms of contig and novel base pair count for Anise and MTG.

Anise In total, I could align 15 Anise contigs to the fosmids using the approach described above. Of these, 8 have an identity of 100%, 14 have an identity of 99% or better, and all have an identity of 93% or better. The contigs in these classes correspond to approximately 12 kbp, 23 kbp, and 23 kbp of novel sequence.

MTG Two of the aligned MTG contigs align to the same fosmid at the same location with the same identity and I ignore the second contig in this evaluation. After this filtration, I could align 4 MTG contigs using the approach described above. Of these, 1 has an identity of 100%, 2 have an identity of 97% or better, and all 4 have an identity of 93% or better. The contigs in these classes consist of 4 kbp, 5 kbp, 5 kbp, and 6 kbp of novel sequence.

I also compared the matches for both methods and found that half of the MTG matches are contained in Anise matches.



(a) validation using fosmid sequence **(b)** validation using capillary sequence

Figure 5.25: The situation for validating assembled contigs using (a) fosmid and (b) capillary sequence. In (a), the fosmid contigs are used as the reference and the assembled contigs are used as the query in a BLAT search. In (b), the assembled contigs are used as the reference and the capillary contigs are used as the query. The best BLAT match (passing the quality filters described in Sections 5.7.4 and 5.7.5) is picked as the location to compare with, if any.

Discussion. Anise yielded almost four times as many contigs that can be aligned with high identity against the fosmid contigs than MTG. Also, the Anise contigs can be aligned with higher identity to the fosmid contigs (14 of 15 align with an identity of $\geq$ 99%). Only 2 of the 4 alignable contigs generated by MTG are exclusive to MTG.

5.7.5 Validation Using Capillary Sequenced Contigs

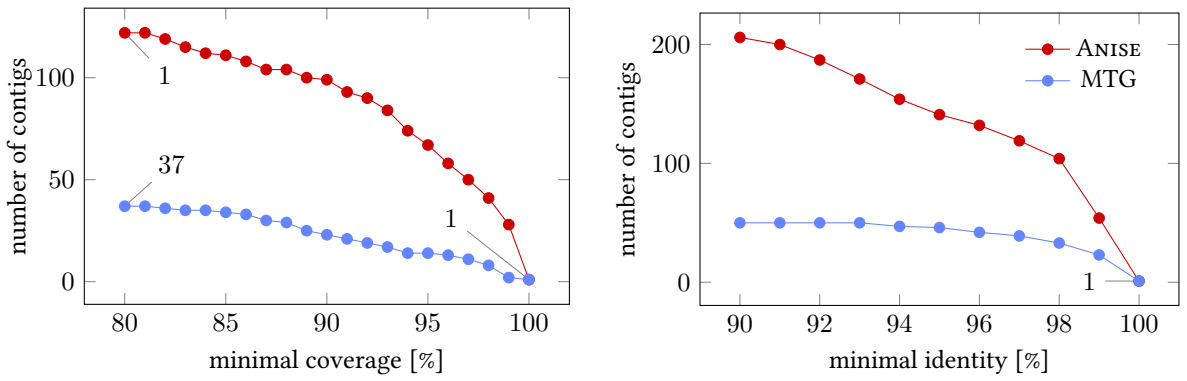
In this experiment, I attempt to validate the assembled contigs with contigs obtained from previously published shorter sequence obtained from capillary sequencing. Figure 5.25b gives an overview of the situation in this evaluation.

Methods. Kidd *et al.* (2010) published the sequence of 1 736 fosmid ends assembled from capillary sequenced reads *(capillary contigs)*. This data set contains sequence from NA12878 and other human individuals. The contigs have lengths between 310 bp and 2 683 bp and they are annotated as having a potentially lower quality towards the ends. Although they have a shorter length and lower quality than the fosmid contigs and are from different individuals, these contigs can be used for the validation of the results of Anise and MTG.

I computed BLAT alignments using the Anise and MTG contigs as the reference and the capillary contigs as the query. High-quality local alignments between the capillary contigs and the assembled contigs support that the assembled sequence is correct. I filtered the BLAT matches to those having at least 100 matching base pairs in the local alignments.

Results. Figure 5.26 shows the results of the verification using capillary sequence in terms of contig count and novel base pairs for Anise and MTG.

Anise 230 capillary contigs can be aligned to the long anchored Anise contigs with at least 100 matching base pairs. Of these, 122 align with a minimal query coverage of 80%, 99 align with



(a) minimal coverage

(b) minimal identity

Figure 5.26: Results for the validation of the insert sequence assembly using capillary contigs. (a) shows the number of capillary contigs that could be aligned to an assembled contig with a certain minimal coverage. (b) shows the counts for matches with a given minimal identity. For both ANISE and MTG, there is one capillary contig that aligns with 100% coverage and minimal identity. The legend for both figures is shown in (b).

90%, 41 with 98%, and 1 with a query coverage of at least 100%. Of the 53 original contigs, 37 align with a minimal identity of 90%, 104 with 98% and 1 with a minimal identity of 100%.

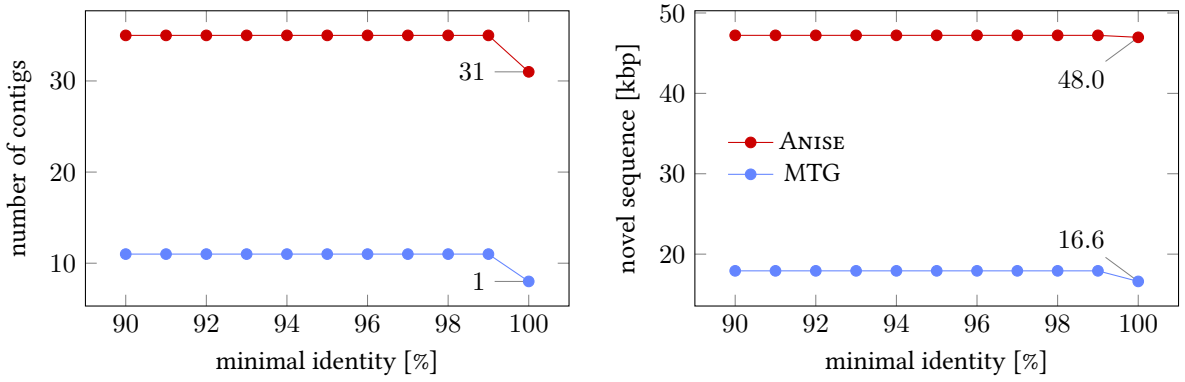
MTG 53 capillary contigs can be aligned to the anchored MTG contigs with at least 100 matching base pairs. Of these, 37 align with a minimal query coverage of 80%, 23 align with 90%, 8 with 98%, and 1 with a query coverage of at least 100%. Of the 53 original contigs, 50 align with a minimal identity of 90%, 33 with 98% and 1 with a minimal identity of 100%.

Discussion. The results of this experiment also show a higher support for ANISE, both when considering the capillary sequences aligning with minimal coverage and when considering them aligning with minimal identity.

5.7.6 Validation Using the hg38/GRCh38 Reference Sequence

Methods. At the time of writing, the reference in release hg18/GRCh36 that was used for the alignment and insert site prediction is two releases behind the current release hg38/GRCh38. In the case that sequence present in NA12878 and missing in hg18 and is now included in hg38, I can use these parts of hg38 for the validation of the ANISE and MINDTHEGAP contigs. Since hg38 does not contain the exact sequence of NA12878, it is no full ground truth but can still serve as a good sanity check.

I used BLAT to align the ANISE and MTG contigs to hg38. I call the ANISE and MTG contigs that align against hg38 with less than 10 uncovered reference bases *reanchored contigs*. Assembled contigs with high sequence recovery and low error rate, with respect to the segments of hg38, are evidence for a high quality of the assembled sequence. The overall situation is the same as



(a) number of contigs

(b) novel base count

Figure 5.27: Results for the validation of the insert sequence assembly using the sequence of hg38. (a) shows the number of capillary contigs that could be aligned to hg38 with the given minimal identity and (b) shows the amount of novel sequence for this. The legend is the same for both figures but only shown in (b).

the one shown in Figure 5.25a, with the hg38 sequence taking the role of the fosmid sequence. Possible sources for missing sequence and errors are assembly errors and differences between NA12878 and the reference sequence.

Results. Figure 5.27 shows the results of the validation of the insert assembly result using the sequence of hg38.

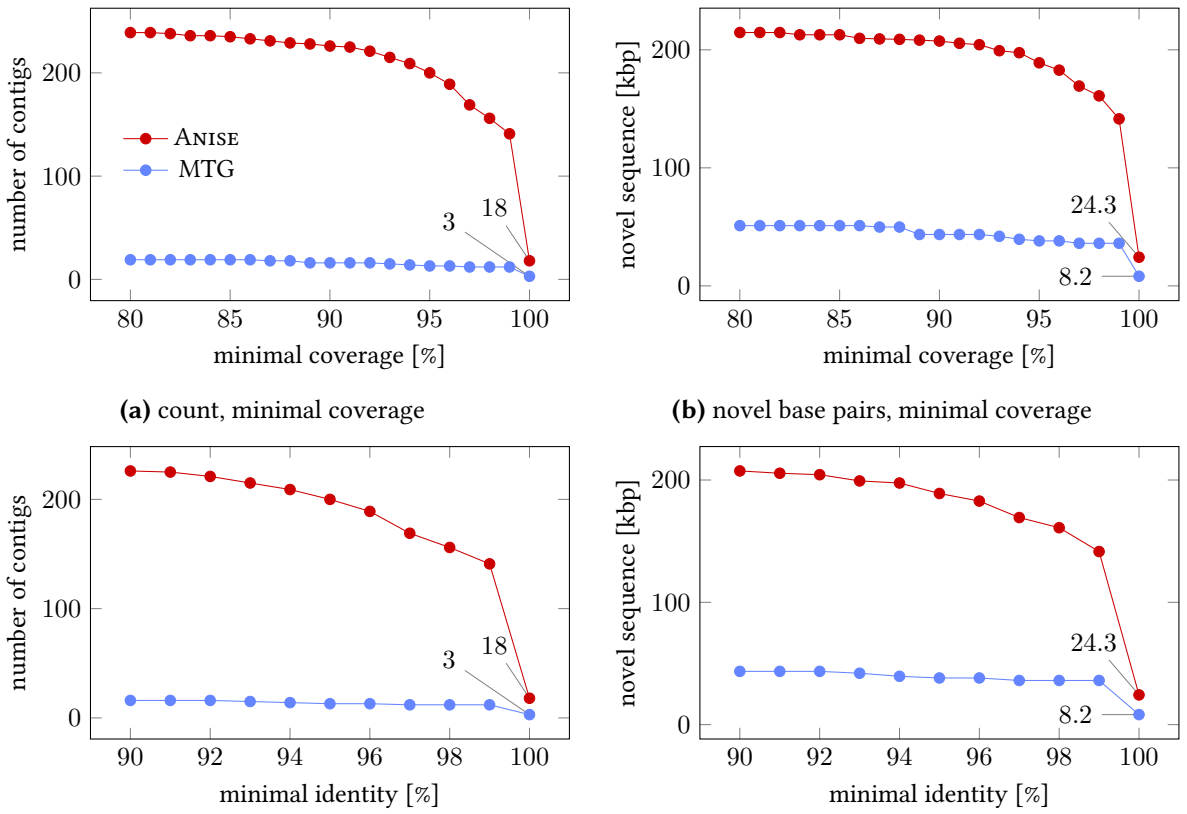
Anise I could reanchor 34 Anise contigs in hg38, each with an identity of 99.7% and a coverage of 99.8% or better. These contigs correspond to a total of 47.2 kb novel sequence.

MTG I could reanchor 10 of the anchored MTG contigs in hg38. Of the anchoring BLAT matches, 4 had an identity of 99% or better and 8 had an identity of 95% or better. The 10 contigs had a total of 13.7 kbp of sequence.

Discussion. By the results of this experiment, Anise achieves better results than MTG. The results can be interpreted as Anise predicting the sequence of 3.4 more updates from hg18 to hg38 than MTG. Further, these updates are predicted with a higher accuracy in terms of sequence identity.

5.7.7 Validation Using the NCBI BLAST *nt* Database

The *nt* database of NCBI Nucleotide BLAST is another valuable resource for validating Anise and MTG contigs. This database contains many high-quality fosmid and BAC sequences which can serve for the validation of the assembled insert sequence, in the case that the *nt* sequences overlap with the Anise and MTG contigs. As is also the case for hg38, the sequences from this



(a) count, minimal coverage

(b) novel base pairs, minimal coverage

(c) count, minimal identity

(d) novel base pairs, minimal identity

Figure 5.28: Validation results using the NCBI BLAST *nt* Database. (a) and (b) show the number of ANISE and MTG contigs for which BLAST matches could be found with a given minimal coverage and the corresponding novel base pairs for (c) and (d) show these metrics for contigs for which matches with a given minimal identity exist. The legend for all figures is shown in (a).

database were not obtained from the individual NA12878 and there might be variants between the donor genome and the assembled sequence. However, finding long high identity matches between the assembled contigs and the *nt* database is a good sanity check.

Methods. I uploaded the anchored ANISE and MTG contigs to the NCBI BLAST website and aligned them to the *nt* database with default settings. I filtered the resulting matches to those aligning to reference sequences of human origin. For each pair of query and subject sequence, I greedily selected the best bit-scoring, non-overlapping alignments similar to the way the BLAST website does for graphic display. I assigned each ANISE and MTG contig to the subject with the highest sum of bit-scores for the selected matches.

Results. Figure 5.28 shows the results of this experiment.

Anise The BLAST search of the 343 anchored Anise contigs yielded 239 matches with minimal length 1000 bp. Of these, 18 had a minimal identity of 100% in the BLAST match, 141 had a minimal identity of 99%, and 200 had a minimal identity of 95%. For these contigs, the number of novel base estimate is 24.3 kbp, 141.5 kbp, and 189.0 kbp, respectively. For 18, 141, and 226 Anise contigs, I found best matches covering at least 100%, 99%, and 95% of the Anise contig. This corresponds to 24.3 kbp, 141.5 kbp, and 207.4 kbp of novel sequence.

MTG Of the 126 unique long anchored contigs, the BLAST search yields 20 matches with sizes of at least 1000 bp, of these 3, 12, and 134 had an identity of 100%, 99%, and 95% or better, respectively, which corresponds to 8.2 kbp, 36.1 kbp, and 83.1 kbp of novel sequence. For 3, 12, and 13 contigs, I found best matches covering at least 100%, 99%, and 95% of the MTG contig. This corresponds to 8.2 kpb, 36.1 kbp, and 43.5 kbp of novel sequence.

Discussion. The results of the validation using the BLAST *nt* database are consistent with the other evaluation results. Anise creates more than 10 times more contigs with high-quality BLAST matches, containing more than 4 times more novel sequence than MTG. Overall, BLAST matches could be found for 70% of the long anchored Anise contigs but only for 16% of the long anchored MTG contigs, although there are 2.7 times fewer of the latter.

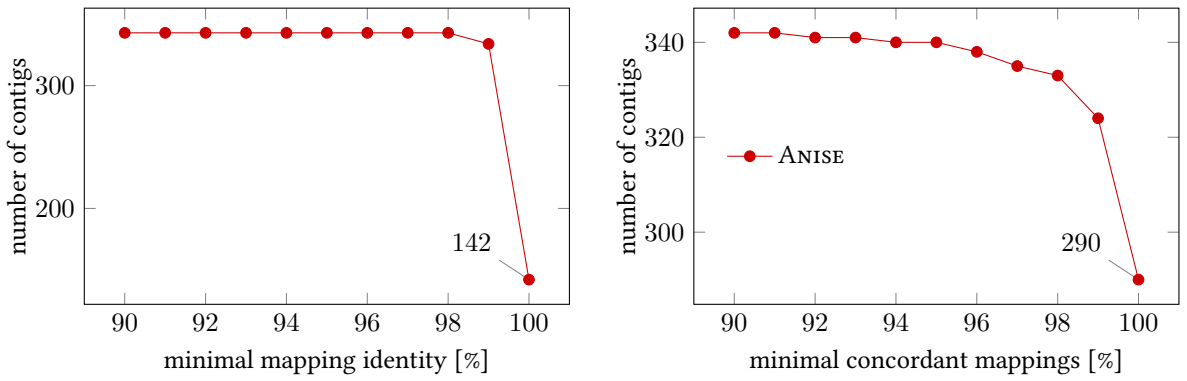
5.7.8 Validation Using Paired-Read Data

A common method for the validation of assembly results is to map the original input read pairs against the assembled sequence. The resulting read alignment can be analyzed similar to calling variations of a donor to the reference. In this case, the assembled sequence takes the role of the reference and there should be few, if any, variants. When analyzed for SNVs and small indels, such variants indicate possible small-scale assembly errors. Discordantly aligning reads are signals for possible larger-scale assembly errors.

Methods. Anise allows to write out the reads used for the assembly of each contig. For each long anchored contig, I mapped the reads back to their contig using BWA and used the mpileup command of Samtools for calling SNVs and small indels. Further, I counted pairs showing more than six standard deviations from the mean template size as discordant.

Since MindTheGap is not capable of writing out the reads used for the assembly, I only performed this experiment for Anise.

Results. Figure 5.29 shows the results of the evaluation using paired-end mapping of the original reads to the assembled contigs. I aligned the reads of the 343 long anchored contigs to their corresponding assembled contigs. For almost all (334 of 343) contigs, the minimal mapping identity of the aligned read is $\geq$ 99%. When considering the percentage of concordantly aligning read pairs, 324 of 343 contigs $\geq$ 99% concordantly aligning reads.



(a) by mapping identity

(b) by concordant mappings

Figure 5.29: Results for the validation of the insert sequence assembly by mapping the original paired reads to the assembled sequences. (a) shows the number of contigs for which the average identity rate of the aligned read had a minimal given value. (b) shows the same but requires a minimal reate of concordantly aligning read pairs.

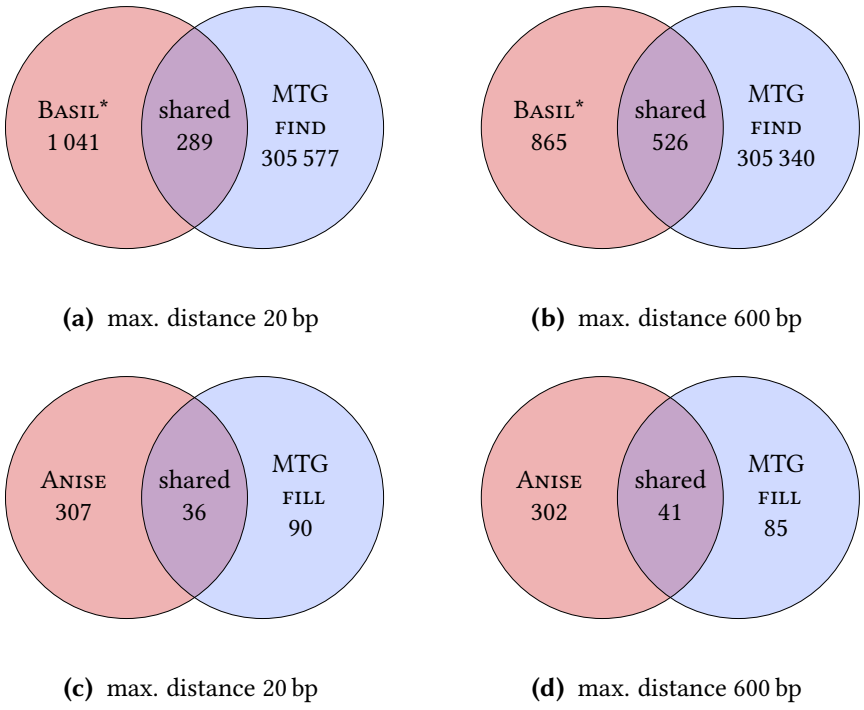
Discussion. The high number of contigs for which the reads align with high identity show that the assembled contigs strongly agree with the reads on the sequence level. An error rate of 1% is approximately what one expects from Illumina HiSeq 2000 reads. Considering the minimal percentage of concordantly aligning reads shows agreement with the read pairing information. The discordantly aligning read pairs can be explained by outliers in terms of pair template size or artifacts caused by the read pair belonging to a different genomic location.

5.7.9 Overlap of Predicted Breakpoints

I computed the concordance of the insert site breakpoint detection of Basil and MindTheGap. Using concordance is a good sanity check since a low concordance means that at least one method is missing many insert sites. However, a high concordance alone is only weak evidence that a method works correctly since all methods could show the same bias.

Nevertheless, when complemented with results from simulated data, concordance is a meaningful validation method. I presented results for simulated data in Section 5.6.3, thus enabling a meaningful evaluation using concordance of breakpoint prediction.

Methods. For this evaluation, I compared the breakpoints from Basil* and the MTG find module using BedTools (Quinlan *et al.*, 2010). I computed the breakpoint overlap when allowing a maximal distance of 20 bp and 600 bp. The shorter allowed distance of 20 bp allows to account for ambiguities in the base-pair resolution insert site detection. For some predicted insertion sites, Basil* was not able to refine the position to base-pair resolution. For this case, I also consider the larger allowed distance of 600 bp.



(a) max. distance 20 bp **(b)** max. distance 600 bp

(c) max. distance 20 bp **(d)** max. distance 600 bp

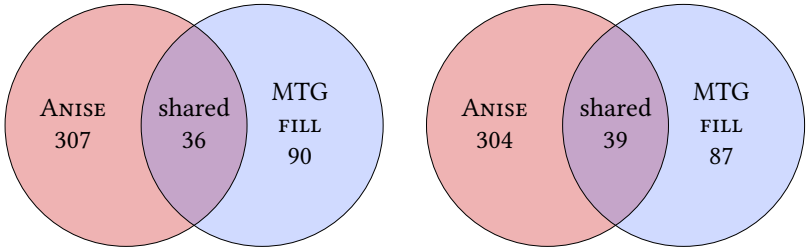
Figure 5.30: Venn diagrams of the breakpoint-wise overlap, allowing for a maximal distance of 20 bp and 600 bp. (a) and (b) show the overlap between Basil* and MTG find whereas (c) and (d) show the overlap between breakpoints of anchored Anise and MTG fill contigs.

Results. The breakpoint candidate overlap size of Basil* and the MTG find module is 289/526 when allowing a maximal distance of 20/600 bp. 1 041/865 breakpoints are exclusive to Basil* and 305 577/305 340 are exclusive to MTG. The lack of read pair information usage in MTG find does not allow for a more specific filtering of the breakpoints.

When comparing the breakpoints of the anchored Anise and MTG contigs, 36/41 bp are shared (with a maximal distance of 20/600 bp). 307/302 are exclusive to Anise and 90/85 are exclusive to MTG.

Discussion. The large number of breakpoints yielded by MTG find stems from the fact that it uses a k-mer–based approach and is not particularly focused on long insertions. The overlap of anchored contigs is thus more interesting, since it includes the filtration steps in terms of contig sequence length and evidence of anchoring in the reference sequence.

Here, only 10% of the anchor breakpoints of Anise contigs overlap with the ones of MTG and for MTG this is true for 25–33%. Increasing the allowed distance only slightly improves this overlap. I will discuss these results further in Section 5.7.10.



(a) min. overlap 100% **(b)** min. overlap 95%

Figure 5.31: Venn diagrams of the sequence-wise overlap requiring a minimal overlap of (a) 100% and (b) 95% of overlap (in the MTG contig).

5.7.10 Sequence-Based Overlap of Assembled Contigs

Methods. I computed the overlaps of contigs using local alignments between Anise and MindTheGap. For this, I computed the sequence-wise overlap of the contigs assembled by Anise and MTG using BLAT. Since Anise yields inserts with neighboring reference sequence and MTG yields the predicted insertions only, I proceeded as follows.

I performed a BLAT search using the MTG contigs as the query and the Anise contigs as the reference. I then counted the number of MTG contigs that are covered to at least 95 and 100% and have a BLAT identity of over 95%. This yields the sequence-wise overlap of the anchored contigs.

Results. The overlap by sequence yields similar results to the overlap by breakpoint position. When requiring a minimal overlap of 100%, 36 assembled contigs are shared, 307 are exclusive to Anise, and 90 are exclusive to MTG. The number of shared contigs increases by 3 when only requiring a minimal overlap of 95%.

Additionally, I considered the contigs exlusive to Anise and MTG (*i.e.*, those without an overlap). For each set, I computed how many are supported by fosmid matches, capillary matches, hg38 matches, and BLAST *nt* matches (requiring a minimal identity of 95%).

Of the 342 exclusive Anise contigs, I could find fosmid matches for 14, capillary matches for 91, hg38 matches for 35, and BLAST *nt* matches for 200. This corresponds to 4%, 27%, 10,%, and 59% of the contigs exclusive to Anise.

For the 87 MTG matches, I found fosmid matches for 1, capillary matches for 16, hg38 matches for 7, and BLAST *nt* matches for 8. This corresponds to 1%, 18%, 8,%, and 9% of the contigs exclusive to MTG.

Table 5.4 shows these numbers in tabular format for easier comparison.

method	total exclusive	supported by			
		fosmids	capillary	hg38	BLAST *nt*
Anise	342	14 (4%)	91 (27%)	35 (10%)	200 (59%)
MTG	87	1 (1%)	16 (18%)	7 (8%)	8 (9%)

Table 5.4: Statistics about verification of contigs exclusive to one method. This table shows the number of total exclusive contigs, and the number (and percentage) of exclusive contigs that are supported by matches to fosmid contigs, capillary contigs, the hg38 reference sequence, and the BLAST *nt* database.

Discussion. The number of shared contigs between Anise and MTG is relatively low. More contigs are exclusive to Anise and also a higher percentage of these contigs are supported using the validation experiments. This evidence shows that Anise is not only able to assemble more long insertions than MTG but also that a higher percentage of these contigs can be verified given existing sequence databases.

Nevertheless, MTG yielded a number of contigs that was almost one third of the number of contigs exclusive to Anise. The support (from the available databases) for this contigs is not as good as the support for the exclusive Anise contigs. However, there is still a (small) number of contigs assembled by MTG that are well-supported but that were not found using Anise.

5.7.11 Discussion

In this section, I presented a comprehensive evaluation of the insert assembly (and to a lesser extend) of the insert site prediction using a real-world data set. The filtration of the contigs to the long anchoring ones allows a good comparison of the MindTheGap and Basil/Anise results. For both methods, the anchored sequences show evidence using sequence information (BLAT alignments for Anise and the k-mer analysis in MTG find) and paired read information (directly in Basil and through the anchoring in the evaluation for MTG). Anise yields 2.7 times as many long anchored contigs as does MTG and 2.4 times as many novel bases.

In the evaluation using the fosmid sequences from the same donor, I found evindence for 3 times as many Anise contigs than for MTG contigs and for almost 4 times as many base pairs. The evaluation using capillary contigs (that included sequence from the same individual), the updated reference sequence hg38, and the BLAST *nt* database also support that Anise yields more sequence of higher quality than MTG.

The evaluation of the predicted breakpoints and assembled sequence is relatively low. The number of contigs exclusive to Anise is more than three times higher than those exclusive to MTG. Also, the rate of contigs supported by my experiments is higher in the exclusive Anise contigs than in the exclusive MTG contigs.

Given that the wall-clock running time of Basil and Anise is lower than the one of MTG, my

recommendation for future studies of large insertion is to start the analysis with Basil and Anise. The results of my methods are more comprehensive and reliable than the ones of MTG. Also, for the practitioner, the interpretation of one best contig for each site is easier than the interpretation of many (up to 60) candidates as MTG yields.

Nevertheless, MTG can be used for complementing the results of Anise in a second step. A parallelization of the MTG method would greatly improve the practicability of this method. In particular, the MTG index step could benefit from a parallelization similar to the one in Jellyfish (Marcais and Kingsford, 2011). While the authors of MTG propose to exchange Jellyfish for their own k-mer counting method in low-memory situations (Rizk *et al.*, 2013), my recommendation is to prefer a parallelizable method over a particularly memory-saving one.

5.8 Chapter Conclusion

In this chapter, I described

- the tasks of variant calling and genotyping of small and structural variants,
- the general approach to small variant calling by considering the read alignments and methods for performing small variant calling and genotyping,
- the basic types of structural variants (SVs) and
- the approaches for SV calling,
- a list of SV calling methods and the approaches they use,
- the method Basil for the detection and location of breakpoints, including insert breakpoints,
- the method Anise for the targeted assembly around breakpoints that is able to assemble large insertions, and
- an extensive experimental evaluation of insert site prediction and insert sequence assembly using synthetic, simulated, and real-world data.

I observed that

- using chaining for the OEA cluster selection works better than the previously considered method of an approximate solution of a Set-Cover model,
- Basil shows competitive performance for detecting insertion breakpoints on simulated insertions in real biological sequence,
- filtering of the Basil results is required such that spurious read alignments are ignored and do not greatly lower specificity,
- Anise is superior to the popular general purpose assemblers ABYSS and SGA and the targeted insert assembler MTG on synthetic data with repeats,
- Anise is competitive with ABYSS and SGA for the assembly of simulated insertions into real biological sequence and superior to MTG without the presence of repeats,
- Anise robustly assembles inserted sequence, also in the presence of repeats, and that
- Anise proved to be superior to MTG for the assembly of insertions in real-world sequence.

I conclude that

- Basil is a competitive method for the detection of breakpoints from short read pairs in base resolution,
- Anise is the best and most robust available method (at the time of writing) for the assembly of novel sequence in resequencing projects for personal genomes.

My main contributions in the area of variant analysis are

- the development of the method Basil for the effective and efficient location of breakpoints using the paired read and split read approach,
- the development of the method Anise for the targeted assembly around breakpoints, capable of assembling long insertions,
- the implementation and integration of repeat separation into the insert assembly method,
- the development of a comprehensive benchmark for insert site detection and insert sequence assembly and an evaluation of state-of-the-art tools, and
- using the chaining algorithm for the OEA cluster selection in Basil and showing that this method's performance is superior to the previously used approximate solution of a Set-Cover model.

Chapter 6

Discussion and Conclusion

In my thesis, I described my contributions to the field of variant analysis for resequencing. First, I gave a brief introduction to the background in biology, mathematics, and computer science. In the rest of the thesis, I presented the different steps in a typical resequencing pipeline and my contributions to these steps: preprocessing, read mapping, and variant analysis.

In this section, I give a short summary of my contributions, discuss recent developments in the different subfields, and give an outlook on future work and upcoming challenges.

Contribution Summary

For the quality control and preprocessing step (Chapter 3), I contributed to the Fiona method for HTS read correction. I discussed different approaches for evaluating read error correction tools and the trade-offs of each. Further, I presented an extensive benchmark of state-of-the art read correction tools as well as an efficient tool for supporting such benchmarks.

For the read mapping step (Chapter 4), my first contribution is a formal definition for edit distance–based read mapping. I derived this definition from practical observations of read alignments and ambiguities therein. In this framework, matches are defined as classes of a well-defined equivalence relation. To support benchmarks using my formal framework, I implemented the program suite Rabema that allows for the practical evaluation of read mapping software, both on simulated and on real-world data.

My second contribution in the read mapping step is to the parallel, fully sensitive read mapper RazerS 3. Besides others improvements, this program uses multi-core parallelism for improving the performance over the previous version. Finally, I presented the results of a benchmark comparing different read mappers using Rabema, both on simulated and real-world data.

For the variant analysis step (Chapter 5), my main contributions are the methods Basil and

Anise for the prediction of insertion sites and the targeted assembly of insertions. Overall, these methods are my largest contribution to the area of resequencing.

Basil combines the previous ideas of clustering paired-read and clipping signals for improving the precision of insert site prediction. Further, using the chaining algorithm for the paired-read cluster selection is a novel idea that improves both the recall and sensitivity in Basil. Moreover, I engineered the used approaches to yield better performance than a previous implementation.

My tool Anise combines approaches from read mapping, assembly, and read separation into a novel method for the targeted assembly of inserted sequence. For this, I reused the algorithmic core of our read mapper RazerS 3: pigeonhole-based filtering and bit-parallel verification. During my work, I engineered and reimplemented methods for the separation and repeats and the refining of multi-sequence alignments.

The evaluation of bioinformatics methods is not trivial in general because of the lack of ground truth for large-scale, real-world data and biases therein that are hard to predict. Thus, the chapter closed with an extensive evaluation of Basil and Anise. Here, I showed that Basil is competitive with a recent method for insert site prediction. Further, I found that the repeat separation features of Anise make it superior to widely-used state of the art general purpose assemblers on data with simulated repeats. Finally, Anise was shown to be superior to the (at the time of writing) only other published method for insert assembly. In this benchmark, I performed comprehensive tests using different, orthogonal methods and many different data sources.

An important, recurring topic of this thesis is the benchmarking of methods. Here, a last (albeit secondary) general contribution is the development of the Mason software suite for the simulation of HTS reads and sequence variants (Appendix A).

Recent Developments

The area of quality control and preprocessing is intrinsically tied to sequencing technology used for generating the input and the read mapping methods in the subsequent step. With their most recent sequencing machines, Illumina improved on the read length and quality, making reads with lengths of 150–300 bp of high qualities readily available. This makes quality-based trimming less important, but there continues to be work in the area of efficient and precise adapter removal, *e.g.*, by O'Connell *et al.* (2015) and Jiang *et al.* (2014).

In the area of read error correction, current work focuses on lowering resource consumption and processing time for scaling up to rising throughput rates, using the spectral alignment approach. One promising idea is to replace counting by Bloom filters which leads to using only a few bits per k-mer, thus lower memory usage and running time, *e.g.*, as implemented by Li (2015) and Song *et al.* (2014).

The area of read mapping continues to be very active, *e.g.*, the work by Kerpedjiev *et al.* (2014) or Butterfield *et al.* (2014), and today, most users use best-mapping when aligning reads. Notably, methods such as the one by Siragusa *et al.* allow the controlled, separate enumeration of different match strata, enabling a good estimate on a read alignment's ambiguity. However, there are also

some advances in the field of all-mapping, *e.g.*, by Köster and Rahmann (2014) and Hieu Tran and Chen (2015).

The field of variant analysis also remains active, mostly in the area of structural variant calling. For example, Bickhart *et al.* (2015) recently published a new method for the detection of structural variants using aligned reads and Weisenfeld *et al.* (2014) published a method for SV detection using *de novo* assembly. Also recently, Kehr *et al.* (2015) have published a manuscript on their population-scale insert novel insert detection method that internally uses a general purpose assembler.

Future Work and Challenges

In read preprocessing and quality control the main future work is the adaption to the properties of upcoming sequencing machines and improving the scalability of the method while keeping the resource consumption low.

Future read mapping methods will continue to show performance improvements, adapting to new hardware developments, such as wider SIMD registers and many-core (co-)processors, such as AVX and the Xeon Phi Processor. Also, being able to get a good estimate on the ambiguity by a read by counting or enumerating suboptimal matches can serve as important information for downstream analysis methods. General purpose read mappers that have decent split read alignment support are already available, *e.g.*, by Li (2013).

The integration of such methods into variant detection tools could further improve the latter. For example, a read mapper could pass non-concordant and split aligned reads on the fly to a SV or CNV analysis tool for processing, bypassing the need to store all of them in the alignment file. These specialized tools would then have access to the full information and also perform additional queries, as necessary. All of this should then be integrated with repeat copy separation for further improving the result quality.

Another future question concerns the reference sequence. While having one common reference coordinate system enabled huge advances in the life sciences, certain regions show a strong variability between populations. In their recent release GRCh38, the Genome Reference Consortium *(GRC)* have accounted for this by including the sequence of 58 alternative loci.

Another approach, that was, *e.g.*, performed by Besenbacher *et al.* (2015), is to build a pan-genome for the considered population. This way, most common variants of a population to the reference sequences are not considered during the resequencing. This decreasing ambiguity in the read mapping and variant analysis step and Besenbacher *et al.* used this for decreasing the false discovery rate of variant discovery.

While having some advantages, using different reference sequences also has several drawbacks. These include problems with comparing two analysis results using different reference genomes and also the challenge of building a new reference assembly. The GRC reference sequence was built with a huge amount of work and resources. Achieving similar quality for the whole genome sequence (*e.g.*, using *de novo* assembly) in individual projects is virtually impossible. However,

for certain applications and projects with sufficient resources, it will be possible to achieve a sufficient quality. This leaves another problem, that of obtaining a rich and high-quality of annotation. Projecting such annotation from one reference sequence to another is hard, as made evident by the only slowly growing number of tracks in the UCSC genome browser for the GRCh38 when compared to the previous release GRCh37 (at the time of writing).

Finally, one must not forget that a hard set of challenges remain, even if a resequencing pipeline yields good results: the interpretation of the variants. While good methods for the priotization of small variants in coding regions exist, *e.g.*, by Robinson *et al.* (2014), the problem of interpretating structural variants and variants in non-coding regions on a large scale remains largely unsolved.

Thus, HTS has not led to an "end of history" situation in bioinformatics by far. Rather, each solved problem opens up the opportunity to solve several new ones, as usual in the sciences. New aims either come up as new questions initiated by the just answered one or they get into one's reach by using recent advances in the field. Further, bioinformatics is also highly influenced by new and improved wet-lab protocols and devices. This leads to dynamics that are reached by few computer science–related fields. A curious mind can only look forward to working in bioinformatics in the future.

Bibliography

Abecasis, G. R., Altshuler, D., Auton, A., Brooks, L. D., Durbin, R. M., Gibbs, R. A., Hurles, M. E., McVean, G. A., Bentley, D. R., Chakravarti, A., Clark, A. G., Collins, F. S., De La Vega, F. M., Donnelly, P., Egholm, M., Flicek, P., Gabriel, S. B., Knoppers, B. M., Lander, E. S., Lehrach, H., Mardis, E. R., Nickerson, D. A., Peltonen, L., Schafer, A. J., Sherry, S. T., Wang, J., Wilson, R., Deiros, D., Metzker, M., Muzny, D., Reid, J., Wheeler, D., Li, J., Jian, M., Li, G., Li, R., Liang, H., Tian, G., Wang, B., Wang, W., Yang, H., Zhang, X., Zheng, H., Altshuler, D. L., Ambrogio, L., Bloom, T., Cibulskis, K., Fennell, T. J., Jaffe, D. B., Shefler, E., Sougnez, C. L., Gormley, N., Humphray, S., Kingsbury, Z., Kokko-Gonzales, P., Stone, J., McKernan, K. J., Costa, G. L., Ichikawa, J. K., Lee, C. C., Sudbrak, R., Borodina, T. A., Dahl, A., Davydov, A. N., Marquardt, P., Mertes, F., Nietfeld, W., Rosenstiel, P., Schreiber, S., Soldatov, A. V., Timmermann, B., Tolzmann, M., Affourtit, J., Ashworth, D., Attiya, S., Bachorski, M., Buglione, E., Burke, A., Caprio, A., Celone, C., Clark, S., Conners, D., Desany, B., Gu, L., Guccione, L., Kao, K., Kebbler, J., Knowlton, J., Labrecque, M., McDade, L., Mealmaker, C., Minderman, M., Nawrocki, A., Niazi, F., Pareja, K., Ramenani, R., Riches, D., Song, W., Turcotte, C., Wang, S., Wilson, R. K., Dooling, D., Fulton, L., Fulton, R., Weinstock, G., Burton, J., Carter, D. M., Churcher, C., Coffey, A., Cox, A., Palotie, A., Quail, M., Skelly, T., Stalker, J., Swerdlow, H. P., Turner, D., De Witte, A., Giles, S., Bainbridge, M., Challis, D., Sabo, A., Yu, F., Yu, J., Fang, X., Guo, X., Li, Y., Luo, R., Tai, S., Wu, H., Zheng, X., Zhou, Y., Marth, G. T., Garrison, E. P., Huang, W., Indap, A., Kural, D., Lee, W. P., Leong, W. F., Quinlan, A. R., Stewart, C., Stromberg, M. P., Ward, A. N., Wu, J., Lee, C., Mills, R. E., Shi, X., Daly, M. J., DePristo, M. A., Ball, A. D., Banks, E., Browning, B. L., Garimella, K. V., Grossman, S. R., Handsaker, R. E., Hanna, M., Hartl, C., Kernytsky, A. M., Korn, J. M., Li, H., Maguire, J. R., McCarroll, S. A., McKenna, A., Nemesh, J. C., Philippakis, A. A., Poplin, R. E., Price, A., Rivas, M. A., Sabeti, P. C., Schaffner, S. F., Shlyakhter, I. A., Cooper, D. N., Ball, E. V., Mort, M., Phillips, A. D., Stenson, P. D., Sebat, J., Makarov, V., Ye, K., Yoon, S. C., Bustamante, C. D., Boyko, A., Degenhardt, J., Gravel, S., Gutenkunst, R. N., Kaganovich, M., Keinan, A., Lacroute, P., Ma, X., Reynolds, A., Clarke, L., Cunningham, F., Herrero, J., Keenen, S., Kulesha, E., Leinonen, R., McLaren, W. M., Radhakrishnan, R., Smith, R. E., Zalunin, V., Zheng-Bradley, X., Korbel, J. O., Stutz, A. M., Bauer, M., Cheetham, R. K., Cox, T., Eberle, M., James, T., Kahn, S., Murray, L., Fu, Y., Hyland, F. C., Manning, J. M., McLaughlin, S. F., Peckham, H. E., Sakarya,

O., Sun, Y. A., Tsung, E. F., Batzer, M. A., Konkel, M. K., Walker, J. A., Albrecht, M. W., Amstislavskiy, V. S., Herwig, R., Parkhomchuk, D. V., Agarwala, R., Khouri, H. M., Morgulis, A. O., Paschall, J. E., Phan, L. D., Rotmistrovsky, K. E., Sanders, R. D., Shumway, M. F., Xiao, C., Iqbal, Z., Lunter, G., Marchini, J. L., Moutsianas, L., Myers, S., Tumian, A., Knight, J., Winer, R., Craig, D. W., Beckstrom-Sternberg, S. M., Christoforides, A., Kurdoglu, A. A., Pearson, J. V., Sinari, S. A., Tembe, W. D., Haussler, D., Hinrichs, A. S., Katzman, S. J., Kern, A., Kuhn, R. M., Przeworski, M., Hernandez, R. D., Howie, B., Kelley, J. L., Melton, S. C., Anderson, P., Blackwell, T., Chen, W., Cookson, W. O., Ding, J., Kang, H. M., Lathrop, M., Liang, L., Moffatt, M. F., Scheet, P., Sidore, C., Snyder, M., Zhan, X., Zollner, S., Awadalla, P., Casals, F., Idaghdour, Y., Keebler, J., Stone, E. A., Zilversmit, M., Jorde, L., Xing, J., Eichler, E. E., Aksay, G., Alkan, C., Hajirasouliha, I., Hormozdiari, F., Kidd, J. M., Sahinalp, S. C., Sudmant, P. H., Chen, K., Chinwalla, A., Ding, L., Koboldt, D. C., McLellan, M. D., Wallis, J. W., Wendl, M. C., Zhang, Q., Albers, C. A., Ayub, Q., Balasubramaniam, S., Barrett, J. C., Chen, Y., Conrad, D. F., Danecek, P., Dermitzakis, E. T., Hu, M., Huang, N., Jin, H., Jostins, L., Keane, T. M., Le, S. Q., Lindsay, S., Long, Q., MacArthur, D. G., Montgomery, S. B., Parts, L., Tyler-Smith, C., Walter, K., Zhang, Y., Gerstein, M. B., Abyzov, A., Balasubramanian, S., Bjornson, R., Du, J., Grubert, F., Habegger, L., Haraksingh, R., Jee, J., Khurana, E., Lam, H. Y., Leng, J., Mu, X. J., Urban, A. E., Zhang, Z., Coafra, C., Dinh, H., Kovar, C., Lee, S., Nazareth, L., Wilkinson, J., Scott, C., Gharani, N., Kaye, J. S., Kent, A., Li, T., McGuire, A. L., Ossorio, P. N., Rotimi, C. N., Su, Y., Toji, L. H., Felsenfeld, A. L., McEwen, J. E., Abdallah, A., Juenger, C. R., Clemm, N. C., Duncanson, A., Green, E. D., Guyer, M. S., Peterson, J. L., Xue, Y., and Cartwright, R. A. (2010). A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073.

Abecasis, G. R., Auton, A., Brooks, L. D., DePristo, M. A., Durbin, R. M., Handsaker, R. E., Kang, H. M., Marth, G. T., McVean, G. A., Altshuler, D. M., Bentley, D. R., Chakravarti, A., Clark, A. G., Donnelly, P., Eichler, E. E., Flicek, P., Gabriel, S. B., Gibbs, R. A., Green, E. D., Hurles, M. E., Knoppers, B. M., Korbel, J. O., Lander, E. S., Lee, C., Lehrach, H., Mardis, E. R., Nickerson, D. A., Schmidt, J. P., Sherry, S. T., Wang, J., Wilson, R. K., Dinh, H., Kovar, C., Lee, S., Lewis, L., Muzny, D., Reid, J., Wang, M., Fang, X., Guo, X., Jian, M., Jiang, H., Jin, X., Li, G., Li, J., Li, Y., Li, Z., Liu, X., Lu, Y., Ma, X., Su, Z., Tai, S., Tang, M., Wang, B., Wang, G., Wu, H., Wu, R., Yin, Y., Zhang, W., Zhao, J., Zhao, M., Zheng, X., Zhou, Y., Gupta, N., Clarke, L., Leinonen, R., Smith, R. E., Zheng-Bradley, X., Grocock, R., Humphray, S., James, T., Kingsbury, Z., Sudbrak, R., Albrecht, M. W., Amstislavskiy, V. S., Borodina, T. A., Lienhard, M., Mertes, F., Sultan, M., Timmermann, B., Yaspo, M. L., Fulton, L., Fulton, R., Weinstock, G. M., Balasubramaniam, S., Burton, J., Danecek, P., Keane, T. M., Kolb-Kokocinski, A., McCarthy, S., Stalker, J., Quail, M., Davies, C. J., Gollub, J., Webster, T., Wong, B., Zhan, Y., Yu, F., Bainbridge, M., Challis, D., Evani, U. S., Lu, J., Nagaswamy, U., Sabo, A., Wang, Y., Yu, J., Coin, L. J., Fang, L., Li, Q., Lin, H., Liu, B., Luo, R., Qin, N., Shao, H., Xie, Y., Ye, C., Yu, C., Zhang, F., Zheng, H., Zhu, H., Garrison, E. P., Kural, D., Lee, W. P., Leong, W. F., Ward, A. N., Wu, J., Zhang, M., Griffin, L., Hsieh, C. H., Mills, R. E., Shi, X., von Grotthuss, M., Zhang, C., Daly, M. J., Banks, E., Bhatia, G., Carneiro, M. O., del Angel, G., Genovese, G., Hartl, C., McCarroll, S. A., Nemesh, J. C., Poplin, R. E., Schaffner, S. F., Shakir, K., Yoon, S. C., Lihm, J., Makarov, V., Jin, H., Kim, W., Kim, K. C., Rausch, T., Beal, K., Cunningham, F., Herrero, J., McLaren, W. M., Ritchie, G. R., Gottipati, S., Keinan, A., Rodriguez-Flores, J. L., Sabeti, P. C., Grossman, S. R., Tabrizi, S., Tariyal, R., Cooper, D. N., Ball, E. V., Stenson, P. D., Barnes, B., Bauer, M., Cheetham, R., Cox, T., Eberle, M., Kahn, S., Murray, L., Peden, J., Shaw,

R., Ye, K., Batzer, M. A., Konkel, M. K., Walker, J. A., MacArthur, D. G., Lek, M., Herwig, R., Shriver, M. D., Bustamante, C. D., Byrnes, J. K., De La Vega, F. M., Gravel, S., Kenny, E. E., Kidd, J. M., Lacroute, P., Maples, B. K., Moreno-Estrada, A., Zakharia, F., Halperin, E., Baran, Y., Craig, D. W., Christoforides, A., Homer, N., Izatt, T., Kurdoglu, A. A., Sinari, S. A., Squire, K., Xiao, C., Sebat, J., Bafna, V., Burchard, E. G., Hernandez, R. D., Gignoux, C. R., Haussler, D., Katzman, S. J., Kent, W. J., Howie, B., Ruiz-Linares, A., Dermitzakis, E. T., Lappalainen, T., Devine, S. E., Maroo, A., Tallon, L. J., Rosenfeld, J. A., Michelson, L. P., Anderson, P., Angius, A., Bigham, A., Blackwell, T., Busonero, F., Cucca, F., Fuchsberger, C., Jones, C., Jun, G., Lyons, R., Maschio, A., Porcu, E., Reinier, F., Sanna, S., Schlessinger, D., Sidore, C., Tan, A., Trost, M. K., Awadalla, P., Hodgkinson, A., Lunter, G., Marchini, J. L., Myers, S., Churchhouse, C., Delaneau, O., Gupta-Hinch, A., Iqbal, Z., Mathieson, I., Rimmer, A., Xifara, D. K., Oleksyk, T. K., Fu, Y., Xiong, M., Jorde, L., Witherspoon, D., Xing, J., Browning, B. L., Alkan, C., Hajirasouliha, I., Hormozdiari, F., Ko, A., Sudmant, P. H., Chen, K., Chinwalla, A., Ding, L., Dooling, D., Koboldt, D. C., McLellan, M. D., Wallis, J. W., Wendl, M. C., Zhang, Q., Tyler-Smith, C., Albers, C. A., Ayub, Q., Chen, Y., Coffey, A. J., Colonna, V., Huang, N., Jostins, L., Li, H., Scally, A., Walter, K., Xue, Y., Zhang, Y., Gerstein, M. B., Abyzov, A., Balasubramanian, S., Chen, J., Clarke, D., Habegger, L., Harmanci, A. O., Jin, M., Khurana, E., Mu, X. J., Sisu, C., Degenhardt, J., Stutz, A. M., Church, D., Michaelson, J. J., Blackburne, B., Lindsay, S. J., Ning, Z., Frankish, A., Harrow, J., Fowler, G., Hale, W., Kalra, D., Barker, J., Kelman, G., Kulesha, E., Radhakrishnan, R., Roa, A., Smirnov, D., Streeter, I., Toneva, I., Vaughan, B., Ananiev, V., Belaia, Z., Beloslyudtsev, D., Bouk, N., Chen, C., Cohen, R., Cook, C., Garner, J., Hefferon, T., Kimelman, M., Liu, C., Lopez, J., Meric, P., O'Sullivan, C., Ostapchuk, Y., Phan, L., Ponomarov, S., Schneider, V., Shekhtman, E., Sirotkin, K., Slotta, D., Zhang, H., Barnes, K. C., Beiswanger, C., Cai, H., Cao, H., Gharani, N., Henn, B., Jones, D., Kaye, J. S., Kent, A., Kerasidou, A., Mathias, R., Ossorio, P. N., Parker, M., Reich, D., Rotimi, C. N., Royal, C. D., Sandoval, K., Su, Y., Tian, Z., Tishkoff, S., Toji, L. H., Via, M., Yang, H., Yang, L., Zhu, J., Bodmer, W., Bedoya, G., Ming, C. Z., Yang, G., You, C. J., Peltonen, L., Garcia-Montero, A., Orfao, A., Dutil, J., Martinez-Cruzado, J. C., Felsenfeld, A. L., McEwen, J. E., Clemm, N. C., Duncanson, A., Dunn, M., Guyer, M. S., and Peterson, J. L. (2012). An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65.

Abyzov, A., Urban, A. E., Snyder, M., and Gerstein, M. (2011). CNVnator: an approach to discover, genotype, and characterize typical and atypical CNVs from family and population genome sequencing. *Genome Research*, 21(6):974–984.

Aguiar, D. and Istrail, S. (2012). HapCompass: a fast cycle basis algorithm for accurate haplotype assembly of sequence data. *Journal of Computational Biology*, 19(6):577–590.

Ahmadi, A., Behm, A., Honnalli, N., Li, C., Weng, L., and Xie, X. (2012). Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic acids research*, 40(6):e41.

Ajay, S. S., Parker, S. C., Abaan, H. O., Fajardo, K. V., and Margulies, E. H. (2011). Accurate and comprehensive sequencing of personal genomes. *Genome Res.*, 21(9):1498–1505.

Alkan, C., Coe, B. P., and Eichler, E. E. (2011). Genome structural variation discovery and genotyping. *Nature Reviews Genetics*, 12(5):363–376.

Alkan, C., Kidd, J. M., Marques-Bonet, T., Aksay, G., Antonacci, F., Hormozdiari, F., Kitzman,

J. O., Baker, C., Malig, M., Mutlu, O., *et al.* (2009). Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 41(10):1061–1067.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Andrews, S. (2010). FastQC: A quality control tool for high throughput sequence data. `http://www.bioinformatics.babraham.ac.uk/projects/fastqc/`.

Anson, E. L. and Myers, E. W. (1997). ReAligner: a program for refining DNA sequence multi-alignments. *Journal of Computational Biology*, 4(3):369–383.

Batzoglou, S., Jaffe, D. B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J. P., and Lander, E. S. (2002). ARACHNE: a whole-genome shotgun assembler. *Genome Research*, 12(1):177–189.

Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., Rapp, B. A., and Wheeler, D. L. (2000). GenBank. *Nucleic acids research*, 28(1):15–18.

Bentley, D. R. (2006). Whole-genome re-sequencing. *Current Opinion in Genetics Development*, 16(6):545–552.

Bentley, D. R., Balasubramanian, S., Swerdlow, H. P., Smith, G. P., Milton, J., Brown, C. G., Hall, K. P., Evers, D. J., Barnes, C. L., Bignell, H. R., Boutell, J. M., Bryant, J., Carter, R. J., Keira Cheetham, R., Cox, A. J., Ellis, D. J., Flatbush, M. R., Gormley, N. A., Humphray, S. J., Irving, L. J., Karbelashvili, M. S., Kirk, S. M., Li, H., Liu, X., Maisinger, K. S., Murray, L. J., Obradovic, B., Ost, T., Parkinson, M. L., Pratt, M. R., Rasolonjatovo, I. M., Reed, M. T., Rigatti, R., Rodighiero, C., Ross, M. T., Sabot, A., Sankar, S. V., Scally, A., Schroth, G. P., Smith, M. E., Smith, V. P., Spiridou, A., Torrance, P. E., Tzonev, S. S., Vermaas, E. H., Walter, K., Wu, X., Zhang, L., Alam, M. D., Anastasi, C., Aniebo, I. C., Bailey, D. M., Bancarz, I. R., Banerjee, S., Barbour, S. G., Baybayan, P. A., Benoit, V. A., Benson, K. F., Bevis, C., Black, P. J., Boodhun, A., Brennan, J. S., Bridgham, J. A., Brown, R. C., Brown, A. A., Buermann, D. H., Bundu, A. A., Burrows, J. C., Carter, N. P., Castillo, N., Chiara E Catenazzi, M., Chang, S., Neil Cooley, R., Crake, N. R., Dada, O. O., Diakoumakos, K. D., Dominguez-Fernandez, B., Earnshaw, D. J., Egbujor, U. C., Elmore, D. W., Etchin, S. S., Ewan, M. R., Fedurco, M., Fraser, L. J., Fuentes Fajardo, K. V., Scott Furey, W., George, D., Gietzen, K. J., Goddard, C. P., Golda, G. S., Granieri, P. A., Green, D. E., Gustafson, D. L., Hansen, N. F., Harnish, K., Haudenschild, C. D., Heyer, N. I., Hims, M. M., Ho, J. T., Horgan, A. M., Hoschler, K., Hurwitz, S., Ivanov, D. V., Johnson, M. Q., James, T., Huw Jones, T. A., Kang, G. D., Kerelska, T. H., Kersey, A. D., Khrebtukova, I., Kindwall, A. P., Kingsbury, Z., Kokko-Gonzales, P. I., Kumar, A., Laurent, M. A., Lawley, C. T., Lee, S. E., Lee, X., Liao, A. K., Loch, J. A., Lok, M., Luo, S., Mammen, R. M., Martin, J. W., McCauley, P. G., McNitt, P., Mehta, P., Moon, K. W., Mullens, J. W., Newington, T., Ning, Z., Ling Ng, B., Novo, S. M., O'Neill, M. J., Osborne, M. A., Osnowski, A., Ostadan, O., Paraschos, L. L., Pickering, L., Pike, A. C., Chris Pinkard, D., Pliskin, D. P., Podhasky, J., Quijano, V. J., Raczy, C., Rae, V. H., Rawlings, S. R., Chiva Rodriguez, A., Roe, P. M., Rogers, J., Rogert Bacigalupo, M. C., Romanov, N., Romieu, A., Roth, R. K., Rourke, N. J., Ruediger, S. T., Rusman, E., Sanches-Kuiper, R. M., Schenker, M. R., Seoane, J. M., Shaw, R. J., Shiver, M. K., Short, S. W., Sizto, N. L., Sluis, J. P., Smith, M. A., Ernest Sohna Sohna, J., Spence, E. J., Stevens, K., Sutton, N., Szajkowski,

L., Tregidgo, C. L., Turcatti, G., Vandevondele, S., Verhovsky, Y., Virk, S. M., Wakelin, S., Walcott, G. C., Wang, J., Worsley, G. J., Yan, J., Yau, L., Zuerlein, M., Mullikin, J. C., Hurles, M. E., McCooke, N. J., West, J. S., Oaks, F. L., Lundberg, P. L., Klenerman, D., Durbin, R., and Smith, A. J. (2008). Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59.

Berg, J. M., Tymoczko, J. L., and Stryer, L. (2002). *Biochemistry*. Freeman and Company: New York, 5th edition.

Besenbacher, S., Liu, S., Izarzugaza, J. M., Grove, J., Belling, K., Bork-Jensen, J., Huang, S., Als, T. D., Li, S., Yadav, R., Rubio-Garcia, A., Lescai, F., Demontis, D., Rao, J., Ye, W., Mailund, T., Friborg, R. M., Pedersen, C. N., Xu, R., Sun, J., Liu, H., Wang, O., Cheng, X., Flores, D., Rydza, E., Rapacki, K., Damm S?rensen, J., Chmura, P., Westergaard, D., Dworzynski, P., S?rensen, T. I., Lund, O., Hansen, T., Xu, X., Li, N., Bolund, L., Pedersen, O., Eiberg, H., Krogh, A., B?rglum, A. D., Brunak, S., Kristiansen, K., Schierup, M. H., Wang, J., Gupta, R., Villesen, P., and Rasmussen, S. (2015). Novel variation and de novo mutation rates in population-wide de novo assembled Danish trios. *Nature Communications*, 6:5969.

Bickhart, D. M., Hutchison, J. L., Xu, L., Schnabel, R. D., Taylor, J. F., Reecy, J. M., Schroeder, S., Van Tassell, C. P., Sonstegard, T. S., and Liu, G. E. (2015). RAPTR-SV: a hybrid method for the detection of structural variants. *Bioinformatics*.

Bolger, A. M., Lohse, M., and Usadel, B. (2014). Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics*, 30(15):2114–2120.

Bonizzoni, P., Della Vedova, G., Dondi, R., and Li, J. (2003). The haplotyping problem: an overview of computational models and solutions. *Journal of Computer Science and Technology*, 18(6):675–688.

Bragg, L. M., Stone, G., Butler, M. K., Hugenholtz, P., and Tyson, G. W. (2013). Shining a light on dark sequencing: characterising errors in Ion Torrent PGM data. *PLoS Computational Biolology*, 9(4):e1003031.

Burkhardt, S. and Kärkkäinen, J. (2001). Better filtering with gapped q-grams. In *Combinatorial Pattern Matching*, pages 73–85.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

Butterfield, Y. S., Kreitzman, M., Thiessen, N., Corbett, R. D., Li, Y., Pang, J., Ma, Y. P., Jones, S. J., and Birol, . (2014). JAGuaR: junction alignments to genome for RNA-seq reads. *PLoS ONE*, 9(7):e102398.

Campbell, P. J., Stephens, P. J., Pleasance, E. D., O'Meara, S., Li, H., Santarius, T., Stebbings, L. A., Leroy, C., Edkins, S., Hardy, C., Teague, J. W., Menzies, A., Goodhead, I., Turner, D. J., Clee, C. M., Quail, M. A., Cox, A., Brown, C., Durbin, R., Hurles, M. E., Edwards, P. A., Bignell, G. R., Stratton, M. R., and Futreal, P. A. (2008). Identification of somatically acquired rearrangements in cancer using genome-wide massively parallel paired-end sequencing. *Nature Genetics*, 40(6):722–729.

Chaisson, M., Pevzner, P., and Tang, H. (2004). Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074.

Chen, K., Chen, L., Fan, X., Wallis, J., Ding, L., and Weinstock, G. (2014). TIGRA: A targeted iterative graph routing assembler for breakpoint assembly. *Genome Research*, 24(2):310–317.

Chen, K., Wallis, J. W., McLellan, M. D., Larson, D. E., Kalicki, J. M., Pohl, C. S., McGrath, S. D., Wendl, M. C., Zhang, Q., Locke, D. P., *et al.* (2009). BreakDancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature Methods*, 6(9):677–681.

Chevreux, B. (2005). *MIRA: An Automated Genome and EST Assembler*. PhD thesis, Ruprechts-Karls-University Heidelberg.

Cock, P. J., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. (2010). The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771.

Collins, F. S. and Hamburg, M. A. (2013). First FDA authorization for next-generation sequencer. *New England Journal of Medicine*, 369(25):2369–2371.

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Danecek, P., Auton, A., Abecasis, G., Albers, C. A., Banks, E., DePristo, M. A., Handsaker, R. E., Lunter, G., Marth, G. T., and Sherry, S. T. (2011). The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158.

David, M., Dzamba, M., Lister, D., Ilie, L., and Brudno, M. (2011). SHRiMP2: sensitive yet practical short read mapping. *Bioinformatics*, 27(7):1011–1012.

de Koning, A. J., Gu, W., Castoe, T. A., Batzer, M. A., and Pollock, D. D. (2011). Repetitive elements may comprise over two-thirds of the human genome. *PLoS Genetics*, 7(12):e1002384.

Dementiev, R., Kettner, L., and Sanders, P. (2008). STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637.

Deng, J., Shoemaker, R., Xie, B., Gore, A., LeProust, E. M., Antosiewicz-Bourget, J., Egli, D., Maherali, N., Park, I. H., Yu, J., Daley, G. Q., Eggan, K., Hochedlinger, K., Thomson, J., Wang, W., Gao, Y., and Zhang, K. (2009). Targeted bisulfite sequencing reveals changes in DNA methylation associated with nuclear reprogramming. *Nature Biotechnology*, 27(4):353–360.

DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., del Angel, G., Rivas, M. A., and Hanna, M. (2011). A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature Genetics*, 43(5):491–498.

Dezső, B., Jüttner, A., and Kovács, P. (2011). LEMON–an open source C++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45.

Dilworth, R. P. (1950). A decomposition theorem for partially ordered sets. *Annals of Mathematics*, pages 161–166.

Dodt, M., Roehr, J. T., Ahmed, R., and Dieterich, C. (2012). FLEXBAR—flexible barcode and adapter processing for next-generation sequencing platforms. *Biology*, 1(3):895–905.

Dohm, J. C., Lottaz, C., Borodina, T., and Himmelbauer, H. (2008). Substantial biases in ultra-short read data sets from high-throughput DNA sequencing. *Nucleic acids research*, 36(16):e105.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9(1):11.

Eid, J., Fehr, A., Gray, J., Luong, K., Lyle, J., Otto, G., Peluso, P., Rank, D., Baybayan, P., Bettman, B., Bibillo, A., Bjornson, K., Chaudhuri, B., Christians, F., Cicero, R., Clark, S., Dalal, R., Dewinter, A., Dixon, J., Foquet, M., Gaertner, A., Hardenbol, P., Heiner, C., Hester, K., Holden, D., Kearns, G., Kong, X., Kuse, R., Lacroix, Y., Lin, S., Lundquist, P., Ma, C., Marks, P., Maxham, M., Murphy, D., Park, I., Pham, T., Phillips, M., Roy, J., Sebra, R., Shen, G., Sorenson, J., Tomaney, A., Travers, K., Trulson, M., Vieceli, J., Wegener, J., Wu, D., Yang, A., Zaccarin, D., Zhao, P., Zhong, F., Korlach, J., and Turner, S. (2009). Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138.

Emde, A.-K., Schulz, M. H., Weese, D., Sun, R., Vingron, M., Kalscheuer, V. M., Haas, S. A., and Reinert, K. (2012). Detecting genomic indel variants with exact breakpoints in single-and paired-end sequencing data using SplazerS. *Bioinformatics*, 28(5):619–627.

Enattah, N. S., Sahi, T., Savilahti, E., Terwilliger, J. D., Peltonen, L., and Jarvela, I. (2002). Identification of a variant associated with adult-type hypolactasia. *Nature Genetics*, 30(2):233–237.

Eriksson, N., Pachter, L., Mitsuya, Y., Rhee, S.-Y., Wang, C., Gharizadeh, B., Ronaghi, M., Shafer, R. W., and Beerenwinkel, N. (2008). Viral population estimation using pyrosequencing. *PLoS Computational Biology*, 4(5):e1000074.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Research*, 8(3):186–194.

Ferragina, P. and Manzini, G. (2001). An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278.

Fonseca, N. A., Rung, J., Brazma, A., and Marioni, J. C. (2012). Tools for mapping high-throughput sequencing data. *Bioinformatics*, 28(24):3169–3177.

Gnerre, S., Maccallum, I., Przybylski, D., Ribeiro, F. J., Burton, J. N., Walker, B. J., Sharpe, T., Hall, G., Shea, T. P., Sykes, S., Berlin, A. M., Aird, D., Costello, M., Daza, R., Williams, L., Nicol, R., Gnirke, A., Nusbaum, C., Lander, E. S., and Jaffe, D. B. (2011). High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518.

Gomez-Alvarez, V., Teal, T. K., and Schmidt, T. M. (2009). Systematic artifacts in metagenomes from complex microbial communities. *The ISME journal*, 3(11):1314–1317.

Gordon, A. and Hannon, G. J. (2010). Fastx Toolkit. `http://hannonlab.cshl.edu/fastx_toolkit`.

Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708.

Green, P. (1997). Against a whole-genome shotgun. *Genome Research*, 7(5):410–417.

Greenfield, P., Duesing, K., Papanicolaou, A., and Bauer, D. C. (2014). Blue: correcting sequencing errors using consensus and context. *Bioinformatics*, 30(19):2723–2732.

Gusfield, D. (1997). *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press.

Hach, F., Hormozdiari, F., Alkan, C., Hormozdiari, F., Birol, I., Eichler, E. E., and Sahinalp, S. C. (2010). mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7(8):576–577.

Hajirasouliha, I., Hormozdiari, F., Alkan, C., Kidd, J. M., Birol, I., Eichler, E. E., and Sahinalp, S. C. (2010). Detection and characterization of novel sequence insertions using paired-end next-generation sequencing. *Bioinformatics*, 26(10):1277–1283.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160.

Handsaker, R. E., Korn, J. M., Nemesh, J., and McCarroll, S. A. (2011). Discovery and genotyping of genome structural polymorphism by sequencing on a population scale. *Nature Genetics*, 43(3):269–276.

Hatem, A., Bozda, D., Toland, A. E., and Catalyurek, U. V. (2013). Benchmarking short sequence mapping tools. *BMC Bioinformatics*, 14:184.

Hayden, E. C. (2014). Is the $1,000 genome for real? *Nature News*.

Hieu Tran, N. and Chen, X. (2015). AMAS: optimizing the partition and filtration of adaptive seeds to speed up read mapping. *arXiv preprint*.

Holtgrewe, M. (2010). Mason–a read simulator for second generation sequencing data. Technical report, Freie Universität Berlin.

Holtgrewe, M., Emde, A.-K., Weese, D., and Reinert, K. (2011). A novel and well-defined benchmarking method for second generation read mapping. *BMC Bioinformatics*, 12(1):210.

Holtgrewe, M., Kuchenbecker, L., and Reinert, K. (2015). Methods for the Detection and Assembly of Novel Sequence in High-Throughput Sequencing Data. *Bioinformatics*, page btv051.

Homer, N. and Nelson, S. F. (2010). Improved variant discovery through local re-alignment of short-read next-generation sequencing data using SRMA. *Genome Biology*, 11(10):R99.

Hormozdiari, F., Alkan, C., Eichler, E. E., and Sahinalp, S. C. (2009). Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes. *Genome Research*, 19(7):1270–1278.

Hormozdiari, F., Hajirasouliha, I., Dao, P., Hach, F., Yorukoglu, D., Alkan, C., Eichler, E. E., and Sahinalp, S. C. (2010). Next-generation VariationHunter: combinatorial algorithms for transposon insertion discovery. *Bioinformatics*, 26(12):i350–357.

Huson, D. H., Reinert, K., and Myers, E. W. (2002). The greedy path-merging algorithm for contig scaffolding. *Journal of the ACM*, 49(5):603–615.

Hyyrö, H. (2003). A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Computing*, 10(1):29–39.

Ilie, L., Fazayeli, F., and Ilie, S. (2011). HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302.

Ilie, L. and Molnar, M. (2013). RACER: Rapid and accurate correction of errors in reads. *Bioinformatics*, 29(19):2490–2493.

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. (2012). De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232.

Izutsu, M., Zhou, J., Sugiyama, Y., Nishimura, O., Aizu, T., Toyoda, A., Fujiyama, A., Agata, K., and Fuse, N. (2012). Genome features of "Dark-fly", a Drosophila line reared long-term in a dark environment. *PLoS ONE*, 7(3):e33288.

Jiang, H., Lei, R., Ding, S. W., and Zhu, S. (2014). Skewer: a fast and accurate adapter trimmer for next-generation sequencing paired-end reads. *BMC Bioinformatics*, 15:182.

Jokinen, P. and Ukkonen, E. (1991). Two algorithms for approxmate string matching in static texts. In *Mathematical Foundations of Computer Science 1991*, pages 240–248. Springer.

Kao, W.-C., Chan, A. H., and Song, Y. S. (2011). ECHO: a reference-free short-read error correction algorithm. *Genome Research*, 21(7):1181–1192.

Kececioglu, J. and Ju, J. (2001). Separating repeats in DNA sequence assembly. In *Proceedings of the fifth annual international conference on Computational biology*, pages 176–183.

Kehr, B., Melsted, P., and Halldórsson, B. V. (2015). PopIns: population-scale detection of novel sequence insertions. *arXiv preprint.* submitted.

Kehr, B., Trappe, K., Holtgrewe, M., and Reinert, K. (2014). Genome alignment with graph data structures: a comparison. *BMC Bioinformatics*, 15:99.

Kehr, B., Weese, D., and Reinert, K. (2011). STELLAR: fast and exact local alignments. *BMC Bioinformatics*, 12 Suppl 9:S15.

Kelley, D. R., Schatz, M. C., and Salzberg, S. L. (2010). Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):R116.

Kent, W. J. (2002). BLAT – the BLAST-like alignment tool. *Genome Research*, 12(4):656–664.

Kerpedjiev, P., Frellsen, J., Lindgreen, S., and Krogh, A. (2014). Adaptable probabilistic mapping of short reads using position specific scoring matrices. *BMC Bioinformatics*, 15:100.

Keupp, K., Beleggia, F., Kayserili, H., Barnes, A. M., Steiner, M., Semler, O., Fischer, B., Yigit, G., Janda, C. Y., Becker, J., Breer, S., Altunoglu, U., Grunhagen, J., Krawitz, P., Hecht, J., Schinke, T., Makareeva, E., Lausch, E., Cankaya, T., Caparros-Martin, J. A., Lapunzina, P., Temtamy, S., Aglan, M., Zabel, B., Eysel, P., Koerber, F., Leikin, S., Garcia, K. C., Netzer, C., Schonau, E., Ruiz-Perez, V. L., Mundlos, S., Amling, M., Kornak, U., Marini, J., and Wollnik, B. (2013). Mutations in WNT1 cause different forms of bone fragility. *Am. J. Hum. Genet.*, 92(4):565–574.

Kidd, J. M., Cooper, G. M., Donahue, W. F., Hayden, H. S., Sampas, N., Graves, T., Hansen, N., Teague, B., Alkan, C., and Antonacci, F. (2008). Mapping and sequencing of structural variation from eight human genomes. *Nature*, 453(7191):56–64.

Kidd, J. M., Sampas, N., Antonacci, F., Graves, T., Fulton, R., Hayden, H. S., Alkan, C., Malig, M., Ventura, M., Giannuzzi, G., Kallicki, J., Anderson, P., Tsalenko, A., Yamada, N. A., Tsang, P., Kaul, R., Wilson, R. K., Bruhn, L., and Eichler, E. E. (2010). Characterization of missing human genome sequences and copy-number polymorphic insertions. *Nature Methods*, 7(5):365–U47.

Kim, S., Medvedev, P., Paton, T. A., and Bafna, V. (2013). Reprever: resolving low-copy duplicated sequences using template driven assembly. *Nucleic acids research*, 41(12):e128.

Kitts, P., Madden, T., Sicotte, H., Black, L., and Ostell, J. The UniVec Database. `http://www.ncbi.nlm.nih.gov/VecScreen/UniVec.html`.

Koboldt, D. C., Chen, K., Wylie, T., Larson, D. E., McLellan, M. D., Mardis, E. R., Weinstock, G. M., Wilson, R. K., and Ding, L. (2009). VarScan: variant detection in massively parallel sequencing of individual and pooled samples. *Bioinformatics*, 25(17):2283–2285.

Koboldt, D. C., Zhang, Q., Larson, D. E., Shen, D., McLellan, M. D., Lin, L., Miller, C. A., Mardis, E. R., Ding, L., and Wilson, R. K. (2012). VarScan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome Research*, 22(3):568–576.

Korbel, J. O., Abyzov, A., Mu, X. J., Carriero, N., Cayting, P., Zhang, Z., Snyder, M., and Gerstein, M. B. (2009). PEMer: a computational framework with simulation-based error models for inferring genomic structural variants from massive paired-end sequencing data. *Genome Biology*, 10(2):R23.

Korbel, J. O., Urban, A. E., Affourtit, J. P., Godwin, B., Grubert, F., Simons, J. F., Kim, P. M., Palejev, D., Carriero, N. J., Du, L., Taillon, B. E., Chen, Z., Tanzer, A., Saunders, A. C., Chi, J., Yang, F., Carter, N. P., Hurles, M. E., Weissman, S. M., Harkins, T. T., Gerstein, M. B., Egholm, M., and Snyder, M. (2007). Paired-end mapping reveals extensive structural variation in the human genome. *Science*, 318(5849):420–426.

Köster, J. and Rahmann, S. (2014). Massively parallel read mapping on GPUs with the q-group index and PEANUT. *PeerJ*, 2:e606.

Kuchenbecker, S.-L. (2011). Handling ambiguity in read mapping applications. Master's thesis, Freie Universität Berlin.

Kuleshov, V. (2014). Probabilistic single-individual haplotyping. *Bioinformatics*, 30(17):i379–38.

Lam, H. Y., Mu, X. J., Stütz, A. M., Tanzer, A., Cayting, P. D., Snyder, M., Kim, P. M., Korbel, J. O., and Gerstein, M. B. (2010). Nucleotide-resolution analysis of structural variants using BreakSeq and a breakpoint library. *Nature Biotechnology*, 28(1):47–55.

Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., Funke, R., Gage, D., Harris, K., Heaford, A., Howland, J., Kann, L., Lehoczky, J., LeVine, R., McEwan, P., McKernan, K., Meldrim, J., Mesirov, J. P., Miranda, C., Morris, W., Naylor, J., Raymond, C., Rosetti, M., Santos, R., Sheridan, A., Sougnez, C., Stange-Thomann, N., Stojanovic, N., Subramanian, A., Wyman, D., Rogers, J., Sulston, J., Ainscough,

R., Beck, S., Bentley, D., Burton, J., Clee, C., Carter, N., Coulson, A., Deadman, R., Deloukas, P., Dunham, A., Dunham, I., Durbin, R., French, L., Grafham, D., Gregory, S., Hubbard, T., Humphray, S., Hunt, A., Jones, M., Lloyd, C., McMurray, A., Matthews, L., Mercer, S., Milne, S., Mullikin, J. C., Mungall, A., Plumb, R., Ross, M., Shownkeen, R., Sims, S., Waterston, R. H., Wilson, R. K., Hillier, L. W., McPherson, J. D., Marra, M. A., Mardis, E. R., Fulton, L. A., Chinwalla, A. T., Pepin, K. H., Gish, W. R., Chissoe, S. L., Wendl, M. C., Delehaunty, K. D., Miner, T. L., Delehaunty, A., Kramer, J. B., Cook, L. L., Fulton, R. S., Johnson, D. L., Minx, P. J., Clifton, S. W., Hawkins, T., Branscomb, E., Predki, P., Richardson, P., Wenning, S., Slezak, T., Doggett, N., Cheng, J. F., Olsen, A., Lucas, S., Elkin, C., Uberbacher, E., Frazier, M., Gibbs, R. A., Muzny, D. M., Scherer, S. E., Bouck, J. B., Sodergren, E. J., Worley, K. C., Rives, C. M., Gorrell, J. H., Metzker, M. L., Naylor, S. L., Kucherlapati, R. S., Nelson, D. L., Weinstock, G. M., Sakaki, Y., Fujiyama, A., Hattori, M., Yada, T., Toyoda, A., Itoh, T., Kawagoe, C., Watanabe, H., Totoki, Y., Taylor, T., Weissenbach, J., Heilig, R., Saurin, W., Artiguenave, F., Brottier, P., Bruls, T., Pelletier, E., Robert, C., Wincker, P., Smith, D. R., Doucette-Stamm, L., Rubenfield, M., Weinstock, K., Lee, H. M., Dubois, J., Rosenthal, A., Platzer, M., Nyakatura, G., Taudien, S., Rump, A., Yang, H., Yu, J., Wang, J., Huang, G., Gu, J., Hood, L., Rowen, L., Madan, A., Qin, S., Davis, R. W., Federspiel, N. A., Abola, A. P., Proctor, M. J., Myers, R. M., Schmutz, J., Dickson, M., Grimwood, J., Cox, D. R., Olson, M. V., Kaul, R., Shimizu, N., Kawasaki, K., Minoshima, S., Evans, G. A., Athanasiou, M., Schultz, R., Roe, B. A., Chen, F., Pan, H., Ramser, J., Lehrach, H., Reinhardt, R., McCombie, W. R., de la Bastide, M., Dedhia, N., Blocker, H., Hornischer, K., Nordsiek, G., Agarwala, R., Aravind, L., Bailey, J. A., Bateman, A., Batzoglou, S., Birney, E., Bork, P., Brown, D. G., Burge, C. B., Cerutti, L., Chen, H. C., Church, D., Clamp, M., Copley, R. R., Doerks, T., Eddy, S. R., Eichler, E. E., Furey, T. S., Galagan, J., Gilbert, J. G., Harmon, C., Hayashizaki, Y., Haussler, D., Hermjakob, H., Hokamp, K., Jang, W., Johnson, L. S., Jones, T. A., Kasif, S., Kaspryzk, A., Kennedy, S., Kent, W. J., Kitts, P., Koonin, E. V., Korf, I., Kulp, D., Lancet, D., Lowe, T. M., McLysaght, A., Mikkelsen, T., Moran, J. V., Mulder, N., Pollara, V. J., Ponting, C. P., Schuler, G., Schultz, J., Slater, G., Smit, A. F., Stupka, E., Szustakowski, J., Thierry-Mieg, D., Thierry-Mieg, J., Wagner, L., Wallis, J., Wheeler, R., Williams, A., Wolf, Y. I., Wolfe, K. H., Yang, S. P., Yeh, R. F., Collins, F., Guyer, M. S., Peterson, J., Felsenfeld, A., Wetterstrand, K. A., Patrinos, A., Morgan, M. J., de Jong, P., Catanese, J. J., Osoegawa, K., Shizuya, H., Choi, S., Chen, Y. J., and Szustakowki, J. (2001). Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921.

Lander, E. S. and Waterman, M. S. (1988). Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 2(3):231–239.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25.

Laurence, M., Hatzis, C., and Brash, D. E. (2014). Common Contaminants in Next-Generation Sequencing That Hinder Discovery of Low-Abundance Microbes. *PLoS ONE*, 9(5).

Leinonen, R., Akhtar, R., Birney, E., Bower, L., Cerdeno-Tarraga, A., Cheng, Y., Cleland, I., Faruque, N., Goodgame, N., Gibson, R., Hoad, G., Jang, M., Pakseresht, N., Plaister, S., Radhakr-

ishnan, R., Reddy, K., Sobhany, S., Ten Hoopen, P., Vaughan, R., Zalunin, V., and Cochrane, G. (2011a). The European Nucleotide Archive. *Nucleic Acids Research*, 39(Database issue):28–31.

Leinonen, R., Sugawara, H., and Shumway, M. (2011b). The sequence read archive. *Nucleic Acids Research*, 39(Database issue):19–21.

Lemaitre, C., Ciortuz, L., and Peterlongo, P. (2014). Mapping-Free and Assembly-Free Discovery of Inversion Breakpoints from Raw NGS Reads. In Dediu, A.-H., Martín-Vide, C., and Truthe, B., editors, *Algorithms for Computational Biology*, volume 8542, chapter Lecture Notes in Computer Science, pages 119–130. Springer International Publishing.

Levenshtein, V. (1965). Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17.

Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint*.

Li, H. (2015). Correcting Illumina sequencing errors for human data. *arXiv preprint*.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760.

Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., and Durbin, R. (2009a). The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079.

Li, H., Ruan, J., and Durbin, R. (2008a). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858.

Li, R., Li, Y., Fang, X., Yang, H., Wang, J., Kristiansen, K., and Wang, J. (2009b). SNP detection for massively parallel whole-genome resequencing. *Genome Research*, 19(6):1124–1132.

Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008b). SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714.

Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. (2009c). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967.

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., *et al.* (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272.

Li, S., Li, R., Li, H., Lu, J., Li, Y., Bolund, L., Schierup, M. H., and Wang, J. (2013). SOAPindel: Efficient identification of indels from short paired reads. *Genome Research*, 23(1):195–200.

Li, Y., Zheng, H., Luo, R., Wu, H., Zhu, H., Li, R., Cao, H., Wu, B., Huang, S., Shao, H., *et al.* (2011). Structural variation in two human genomes mapped at single-nucleotide resolution by whole genome de novo assembly. *Nature Biotechnology*, 29(8):723–730.

Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., Lin, D., Lu, L., and Law, M. (2012). Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012:251364.

Marcais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770.

Marco-Sola, S., Sammeth, M., Guigó, R., and Ribeca, P. (2012). The GEM mapper: fast, accurate and versatile alignment by filtration. *Nature Methods*, 9(12):1185–1188.

Mardis, E. R. (2006). Anticipating the $1,000 genome. *Genome Biology*, 7(7):12.

Mardis, E. R. (2008). Next-generation DNA sequencing methods. *Annual Review of Genomics and Human Genetics*, 9:387–402.

Mardis, E. R. (2013). Next-generation sequencing platforms. *Annual Review of Analytical Chemistry*, 6:287–303.

Margulies, M., Egholm, M., Altman, W. E., Attiya, S., Bader, J. S., Bemben, L. A., Berka, J., Braverman, M. S., Chen, Y. J., Chen, Z., Dewell, S. B., Du, L., Fierro, J. M., Gomes, X. V., Godwin, B. C., He, W., Helgesen, S., Ho, C. H., Irzyk, G. P., Jando, S. C., Alenquer, M. L., Jarvie, T. P., Jirage, K. B., Kim, J. B., Knight, J. R., Lanza, J. R., Leamon, J. H., Lefkowitz, S. M., Lei, M., Li, J., Lohman, K. L., Lu, H., Makhijani, V. B., McDade, K. E., McKenna, M. P., Myers, E. W., Nickerson, E., Nobile, J. R., Plant, R., Puc, B. P., Ronan, M. T., Roth, G. T., Sarkis, G. J., Simons, J. F., Simpson, J. W., Srinivasan, M., Tartaro, K. R., Tomasz, A., Vogt, K. A., Volkmer, G. A., Wang, S. H., Wang, Y., Weiner, M. P., Yu, P., Begley, R. F., and Rothberg, J. M. (2005). Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380.

Marschall, T., Costa, I. G., Canzar, S., Bauer, M., Klau, G. W., Schliep, A., and Schönhuth, A. (2012). CLEVER: clique-enumerating variant finder. *Bioinformatics*, 28(22):2875–2882.

Marschall, T., Hajirasouliha, I., and Schonhuth, A. (2013). MATE-CLEVER: Mendelian-inheritance-aware discovery and genotyping of midsize and long indels. *Bioinformatics*, 29(24):3143–3150.

Mazrouee, S. and Wang, W. (2014). FastHap: fast and accurate single individual haplotype reconstruction using fuzzy conflict graphs. *Bioinformatics*, 30(17):i371–378.

Medvedev, P., Stanciu, M., and Brudno, M. (2009). Computational methods for discovering structural variation with next-generation sequencing. *Nature Methods*, 6(11 Suppl):13–20.

Michaelson, J. J. and Sebat, J. (2012). forestSV: structural variant discovery through statistical learning. *Nature Methods*, 9(8):819–821.

Miller, J. R., Delcher, A. L., Koren, S., Venter, E., Walenz, B. P., Brownley, A., Johnson, J., Li, K., Mobarry, C., and Sutton, G. (2008). Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824.

Mills, R. E., Walter, K., Stewart, C., Handsaker, R. E., Chen, K., Alkan, C., Abyzov, A., Yoon, S. C., Ye, K., Cheetham, R. K., Chinwalla, A., Conrad, D. F., Fu, Y., Grubert, F., Hajirasouliha, I., Hormozdiari, F., Iakoucheva, L. M., Iqbal, Z., Kang, S., Kidd, J. M., Konkel, M. K., Korn, J., Khurana, E., Kural, D., Lam, H. Y., Leng, J., Li, R., Li, Y., Lin, C. Y., Luo, R., Mu, X. J., Nemesh,

J., Peckham, H. E., Rausch, T., Scally, A., Shi, X., Stromberg, M. P., Stutz, A. M., Urban, A. E., Walker, J. A., Wu, J., Zhang, Y., Zhang, Z. D., Batzer, M. A., Ding, L., Marth, G. T., McVean, G., Sebat, J., Snyder, M., Wang, J., Eichler, E. E., Gerstein, M. B., Hurles, M. E., Lee, C., McCarroll, S. A., Korbel, J. O., Altshuler, D. L., Durbin, R. M., Abecasis, G. R., Bentley, D. R., Chakravarti, A., Clark, A. G., Collins, F. S., De La Vega, F. M., Donnelly, P., Egholm, M., Flicek, P., Gabriel, S. B., Gibbs, R. A., Knoppers, B. M., Lander, E. S., Lehrach, H., Mardis, E. R., McVean, G. A., Nickerson, D. A., Peltonen, L., Schafer, A. J., Sherry, S. T., Wilson, R. K., Deiros, D., Metzker, M., Muzny, D., Reid, J., Wheeler, D., Li, J., Jian, M., Li, G., Liang, H., Tian, G., Wang, B., Wang, W., Yang, H., Zhang, X., Zheng, H., Ambrogio, L., Bloom, T., Cibulskis, K., Fennell, T. J., Jaffe, D. B., Shefler, E., Sougnez, C. L., Gormley, N., Humphray, S., Kingsbury, Z., Koko-Gonzales, P., Stone, J., McKernan, K. J., Costa, G. L., Ichikawa, J. K., Lee, C. C., Sudbrak, R., Borodina, T. A., Dahl, A., Davydov, A. N., Marquardt, P., Mertes, F., Nietfeld, W., Rosenstiel, P., Schreiber, S., Soldatov, A. V., Timmermann, B., Tolzmann, M., Affourtit, J., Ashworth, D., Attiya, S., Bachorski, M., Buglione, E., Burke, A., Caprio, A., Celone, C., Clark, S., Conners, D., Desany, B., Gu, L., Guccione, L., Kao, K., Kebbel, A., Knowlton, J., Labrecque, M., McDade, L., Mealmaker, C., Minderman, M., Nawrocki, A., Niazi, F., Pareja, K., Ramenani, R., Riches, D., Song, W., Turcotte, C., Wang, S., Dooling, D., Fulton, L., Fulton, R., Weinstock, G., Burton, J., Carter, D. M., Churcher, C., Coffey, A., Cox, A., Palotie, A., Quail, M., Skelly, T., Stalker, J., Swerdlow, H. P., Turner, D., De Witte, A., Giles, S., Bainbridge, M., Challis, D., Sabo, A., Yu, F., Yu, J., Fang, X., Guo, X., Tai, S., Wu, H., Zheng, X., Zhou, Y., Garrison, E. P., Huang, W., Indap, A., Lee, W. P., Leong, W. F., Quinlan, A. R., Ward, A. N., Daly, M. J., DePristo, M. A., Ball, A. D., Banks, E., Browning, B. L., Garimella, K. V., Grossman, S. R., Hanna, M., Hartl, C., Kernytsky, A. M., Korn, J. M., Li, H., Maguire, J. R., McKenna, A., Nemesh, J. C., Philippakis, A. A., Poplin, R. E., Price, A., Rivas, M. A., Sabeti, P. C., Schaffner, S. F., Shlyakhter, I. A., Cooper, D. N., Ball, E. V., Mort, M., Phillips, A. D., Stenson, P. D., Makarov, V., Bustamante, C. D., Boyko, A., Degenhardt, J., Gravel, S., Gutenkunst, R. N., Kaganovich, M., Keinan, A., Lacroute, P., Ma, X., Reynolds, A., Clarke, L., Cunningham, F., Herrero, J., Keenen, S., Kulesha, E., Leinonen, R., McLaren, W. M., Radhakrishnan, R., Smith, R. E., Zalunin, V., Zheng-Bradley, X., Bauer, M., Cox, T., Eberle, M., James, T., Kahn, S., Murray, L., Hyland, F. C., Manning, J. M., McLaughlin, S. F., Sakarya, O., Sun, Y. A., Tsung, E. F., Albrecht, M. W., Amstislavskiy, V. S., Herwig, R., Parkhomchuk, D. V., Agarwala, R., Khouri, H., Morgulis, A. O., Paschall, J. E., Phan, L. D., Rotmistrovsky, K. E., Sanders, R. D., Shumway, M. F., Xiao, C., Auton, A., Lunter, G., Marchini, J. L., Moutsianas, L., Myers, S., Tumian, A., Knight, J., Winer, R., Craig, D. W., Beckstrom-Sternberg, S. M., Christoforides, A., Kurdoglu, A. A., Pearson, J. V., Sinari, S. A., Tembe, W. D., Haussler, D., Hinrichs, A. S., Katzman, S. J., Kern, A., Kuhn, R. M., Przeworski, M., Hernandez, R. D., Howie, B., Kelley, J. L., Melton, S. C., Anderson, P., Blackwell, T., Chen, W., Cookson, W. O., Ding, J., Kang, H. M., Lathrop, M., Liang, L., Moffatt, M. F., Scheet, P., Sidore, C., Zhan, X., Zollner, S., Awadalla, P., Casals, F., Idaghdour, Y., Keebler, J., Stone, E. A., Zilversmit, M., Jorde, L., Xing, J., Aksay, G., Sahinalp, S. C., Sudmant, P. H., Koboldt, D. C., McLellan, M. D., Wallis, J. W., Wendl, M. C., Zhang, Q., Albers, C. A., Ayub, Q., Balasubramaniam, S., Barrett, J. C., Chen, Y., Danecek, P., Dermitzakis, E. T., Hu, M., Huang, N., Jin, H., Jostins, L., Keane, T. M., Le, S. Q., Lindsay, S., Long, Q., MacArthur, D. G., Montgomery, S. B., Parts, L., Tyler-Smith, C., Balasubramanian, S., Bjornson, R., Du, J., Habegger, L., Haraksingh, R., Jee, J., Jeng, J., Zhang, Z., Bank, E., Yoon, S., Kidd, J., Coafra, C., Dinh, H., Kovar, C., Lee, S., Nazareth, L., Wilkinson, J., Khouri, H. M.,

Scott, C., Gharani, N., Kaye, J. S., Kent, A., Li, T., McGuire, A. L., Ossorio, P. N., Rotimi, C. N., Su, Y., Toji, L. H., Brooks, L. D., Felsenfeld, A. L., McEwen, J. E., Abdallah, A., Juenger, C. R., Clemm, N. C., Duncanson, A., Green, E. D., Guyer, M. S., and Peterson, J. L. (2011). Mapping copy number variation by population-scale genome sequencing. *Nature*, 470(7332):59–65.

Myers, E. W. (1994). Algorithmic advances for searching biosequence databases. In *Computational Methods in Genome Research*, pages 121–135. Springer.

Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H., and Remington, K. A. (2000). A whole-genome assembly of Drosophila. *Science*, 287(5461):2196–2204.

Myers, G. (1999a). A Dataset Generator for Whole Genome Shotgun Sequencing. *ISMB 1999*, pages 202–210.

Myers, G. (1999b). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415.

Navarro, G. and Raffinot, M. (2002). *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453.

Ng, S. B., Turner, E. H., Robertson, P. D., Flygare, S. D., Bigham, A. W., Lee, C., Shaffer, T., Wong, M., Bhattacharjee, A., Eichler, E. E., Bamshad, M., Nickerson, D. A., and Shendure, J. (2009). Targeted capture and massively parallel sequencing of 12 human exomes. *Nature*, 461(7261):272–276.

Nielsen, R., Paul, J. S., Albrechtsen, A., and Song, Y. S. (2011). Genotype and SNP calling from next-generation sequencing data. *Nature Reviews Genetics*, 12(6):443–451.

O'Connell, J., Schulz-Trieglaff, O., Carlson, E., Hims, M. M., Gormley, N. A., and Cox, A. J. (2015). NxTrim: optimized trimming of Illumina mate pair reads. *Bioinformatics*.

Parrish, N., Hormozdiari, F., and Eskin, E. (2011). Assembly of non-unique insertion content using next-generation sequencing. *BMC Bioinformatics*, 12(Suppl 6):–3.

Pevzner, P. A., Tang, H., and Waterman, M. S. (2001). An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753.

Popper, K. (1934). *Karl Popper: Logik der Forschung*. Springer.

Pyon, Y. S., Hayes, M., and Li, J. (2011). Model based clustering approach for identifying structural variation using next generation sequencing data. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2011 IEEE 1st International Conference on*, pages 153–158.

Qin, J., Li, R., Raes, J., Arumugam, M., Burgdorf, K. S., Manichanh, C., Nielsen, T., Pons, N., Levenez, F., Yamada, T., Mende, D. R., Li, J., Xu, J., Li, S., Li, D., Cao, J., Wang, B., Liang, H., Zheng, H., Xie, Y., Tap, J., Lepage, P., Bertalan, M., Batto, J. M., Hansen, T., Le Paslier, D., Linneberg, A., Nielsen, H. B., Pelletier, E., Renault, P., Sicheritz-Ponten, T., Turner, K., Zhu, H., Yu, C., Jian, M., Zhou, Y., Li, Y., Zhang, X., Qin, N., Yang, H., Wang, J., Brunak, S., Dore, J.,

Guarner, F., Kristiansen, K., Pedersen, O., Parkhill, J., Weissenbach, J., Bork, P., Ehrlich, S. D., Antolin, M., Artiguenave, F., Blottiere, H., Borruel, N., Bruls, T., Casellas, F., Chervaux, C., Cultrone, A., Delorme, C., Denariaz, G., Dervyn, R., Forte, M., Friss, C., van de Guchte, M., Guedon, E., Haimet, F., Jamet, A., Juste, C., Kaci, G., Kleerebezem, M., Knol, J., Kristensen, M., Layec, S., Le Roux, K., Leclerc, M., Maguin, E., Minardi, R. M., Oozeer, R., Rescigno, M., Sanchez, N., Tims, S., Torrejon, T., Varela, E., de Vos, W., Winogradsky, Y., and Zoetendal, E. (2010). A human gut microbial gene catalogue established by metagenomic sequencing. *Nature*, 464(7285):59–65.

Quail, M. A., Smith, M., Coupland, P., Otto, T. D., Harris, S. R., Connor, T. R., Bertoni, A., Swerdlow, H. P., and Gu, Y. (2012). A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC genomics*, 13(1):341.

Quinlan, A. R., Clark, R. A., Sokolova, S., Leibowitz, M. L., Zhang, Y., Hurles, M. E., Mell, J. C., and Hall, I. M. (2010). Genome-wide mapping and assembly of structural variant breakpoints in the mouse genome. *Genome Research*, 20(5):623–635.

Rahn, R., Weese, D., and Reinert, K. (2014). Journaled string tree-a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24):3499–3505.

Rasmussen, K. R., Stoye, J., and Myers, E. W. (2006). Efficient q-gram filters for finding all ε-matches over a given length. *Journal of Computational Biology*, 13(2):296–308.

Rausch, T., Emde, A. K., Weese, D., Doring, A., Notredame, C., and Reinert, K. (2008). Segment-based multiple sequence alignment. *Bioinformatics*, 24(16):–187.

Rausch, T., Koren, S., Denisov, G., Weese, D., Emde, A.-K., Döring, A., and Reinert, K. (2009). A consistency-based consensus algorithm for de novo and reference-guided sequence assembly of short reads. *Bioinformatics*, 25(9):1118–1124.

Rausch, T., Zichner, T., Schlattl, A., Stütz, A. M., Benes, V., and Korbel, J. O. (2012). DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–i339.

Reinert, K., Langmead, B., Weese, D., and Evers, D. J. (2015). Alignment of Next-Generation Sequencing Reads. *Annual Review of Genomics and Human Genetics*.

Rice, P., Longden, I., and Bleasby, A. (2000). EMBOSS: the European molecular biology open software suite. *Trends in Genetics*, 16(6):276–277.

Richter, D. C., Ott, F., Auch, A. F., Schmid, R., and Huson, D. H. (2008). MetaSim—A sequencing simulator for genomics and metagenomics. *PLoS ONE*, 3(10):e3373.

Rizk, G., Gouin, A., Chikhi, R., and Lemaitre, C. (2014). MindTheGap: integrated detection and assembly of short and long insertions. *Bioinformatics*, 30(24):3451–3457.

Rizk, G., Lavenier, D., and Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653.

Roach, J. C., Boysen, C., Wang, K., and Hood, L. (1995). Pairwise end sequencing: a unified approach to genomic mapping and sequencing. *Genomics*, 26(2):345–353.

Robinson, J. T., Thorvaldsdóttir, H., Winckler, W., Guttman, M., Lander, E. S., Getz, G., and Mesirov, J. P. (2011). Integrative genomics viewer. *Nature Biotechnology*, 29(1):24–26.

Robinson, P. N., Kohler, S., Oellrich, A., Wang, K., Mungall, C. J., Lewis, S. E., Washington, N., Bauer, S., Seelow, D., Krawitz, P., Gilissen, C., Haendel, M., and Smedley, D. (2014). Improved exome prioritization of disease genes through cross-species phenotype comparison. *Genome Research*, 24(2):340–348.

Ross, M. G., Russ, C., Costello, M., Hollinger, A., Lennon, N. J., Hegarty, R., Nusbaum, C., and Jaffe, D. B. (2013). Characterizing and measuring bias in sequence data. *Genome Biology*, 14(5):R51.

Rothberg, J. M., Hinz, W., Rearick, T. M., Schultz, J., Mileski, W., Davey, M., Leamon, J. H., Johnson, K., Milgrew, M. J., Edwards, M., Hoon, J., Simons, J. F., Marran, D., Myers, J. W., Davidson, J. F., Branting, A., Nobile, J. R., Puc, B. P., Light, D., Clark, T. A., Huber, M., Branciforte, J. T., Stoner, I. B., Cawley, S. E., Lyons, M., Fu, Y., Homer, N., Sedova, M., Miao, X., Reed, B., Sabina, J., Feierstein, E., Schorn, M., Alanjary, M., Dimalanta, E., Dressman, D., Kasinskas, R., Sokolsky, T., Fidanza, J. A., Namsaraev, E., McKernan, K. J., Williams, A., Roth, G. T., and Bustillo, J. (2011). An integrated semiconductor device enabling non-optical genome sequencing. *Nature*, 475(7356):348–352.

Salmela, L. (2010). Correction of sequencing errors in a mixed set of reads. *Bioinformatics*, 26(10):1284–1290.

Salmela, L. and Schröder, J. (2011). Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461.

Salzberg, S. L., Phillippy, A. M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T. J., Schatz, M. C., Delcher, A. L., Roberts, M., Marcais, G., Pop, M., and Yorke, J. A. (2012). GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 22(3):557–567.

Sanders, P. (2009). Algorithm engineering–an attempt at a definition. In *Efficient Algorithms*, pages 321–340. Springer.

Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467.

SanMiguel, P., Tikhonov, A., Jin, Y. K., Motchoulskaia, N., Zakharov, D., Melake-Berhan, A., Springer, P. S., Edwards, K. J., Lee, M., Avramova, Z., and Bennetzen, J. L. (1996). Nested retrotransposons in the intergenic regions of the maize genome. *Science*, 274(5288):765–768.

Schröder, J., Schröder, H., Puglisi, S. J., Sinha, R., and Schmidt, B. (2009). SHREC: a short-read error correction method. *Bioinformatics*, 25(17):2157–2163.

Schulz, M. H., Weese, D., Holtgrewe, M., Dimitrova, V., Niu, S., Reinert, K., and Richard, H. (2014). Fiona: a parallel and automatic strategy for read error correction. *Bioinformatics*, 30(17):i356–363.

Shamos, M. I. and Hoey, D. (1976). Geometric intersection problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 208–215.

Shen, S., Lin, L., Cai, J. J., Jiang, P., Kenkel, E. J., Stroik, M. R., Sato, S., Davidson, B. L., and Xing,

Y. (2011). Widespread establishment and regulatory impact of Alu exons in human genes. *Proceedings of the National Academy of Sciences*, 108(7):2837–2842.

Sherry, S. T., Ward, M. H., Kholodov, M., Baker, J., Phan, L., Smigielski, E. M., and Sirotkin, K. (2001). dbSNP: the NCBI database of genetic variation. *Nucleic Acids Research*, 29(1):308–311.

Simola, D. F. and Kim, J. (2011). Sniper: improved SNP discovery by multiply mapping deep sequenced reads. *Genome Biology*, 12(6):R55.

Sindi, S., Helman, E., Bashir, A., and Raphael, B. J. (2009). A geometric approach for classification and comparison of structural variants. *Bioinformatics*, 25(12):i222–23.

Sindi, S. S., Onal, S., Peng, L., Wu, H.-T., and Raphael, B. J. (2012). An integrative probabilistic model for identification of structural variation in sequencing data. *Genome Biology*, 13(3):i222–230.

Singler, J., Sanders, P., and Putze, F. (2007). MCSTL: The multi-core standard template library. In *Euro-Par 2007 Parallel Processing*, pages 682–694. Springer.

Siragusa, E., Weese, D., and Reinert, K. Yara: well-defined alignment of high-throughput sequencing reads. to appear.

Siragusa, E., Weese, D., and Reinert, K. (2013). Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic acids research*, 41(7):e78.

Sokol, D. and Atagun, F. (2010). TRedD–a database for tandem repeats over the edit distance. *Database (Oxford)*, 2010:baq003.

Song, L., Florea, L., and Langmead, B. (2014). Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*, 15(11):509.

Sun, R., Love, M. I., Zemojtel, T., Emde, A.-K., Chung, H.-R., Vingron, M., and Haas, S. A. (2012). Breakpointer: using local mapping artifacts to support sequence breakpoint discovery from single-end reads. *Bioinformatics*, 28(7):1024–1025.

Suzuki, S., Yasuda, T., Shiraishi, Y., Miyano, S., and Nagasaki, M. (2011). ClipCrop: a tool for detecting structural variations with single-base resolution using soft-clipping information. *BMC Bioinformatics*, 12(Suppl 14):S7.

Tammi, M. T., Arner, E., Britton, T., and Andersson, B. (2002). Separation of nearly identical repeats in shotgun assemblies using defined nucleotide positions, DNPs. *Bioinformatics*, 18(3):379–388.

Thompson, J. F. and Steinmann, K. E. (2010). Single molecule sequencing with a HeliScope genetic analysis system. *Current Protocols in Molecular Biology*, pages 7–10.

Tishkoff, S. A. and Kidd, K. K. (2004). Implications of biogeography of human populations for 'race' and medicine. *Nature Genetics*, 36(11 Suppl):S21–27.

Trappe, K., Emde, A.-K., Ehrlich, H.-C., and Reinert, K. (2014). Gustaf: Detecting and correctly classifying SVs in the NGS twilight zone. *Bioinformatics*, 30(24):3484–3490.

Valouev, A., Johnson, D. S., Sundquist, A., Medina, C., Anton, E., Batzoglou, S., Myers, R. M., and Sidow, A. (2008). Genome-wide analysis of transcription factor binding sites based on ChIP-Seq data. *Nature Methods*, 5(9):829–834.

Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A., Gocayne, J. D., Amanatides, P., Ballew, R. M., Huson, D. H., Wortman, J. R., Zhang, Q., Kodira, C. D., Zheng, X. H., Chen, L., Skupski, M., Subramanian, G., Thomas, P. D., Zhang, J., Gabor Miklos, G. L., Nelson, C., Broder, S., Clark, A. G., Nadeau, J., McKusick, V. A., Zinder, N., Levine, A. J., Roberts, R. J., Simon, M., Slayman, C., Hunkapiller, M., Bolanos, R., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Florea, L., Halpern, A., Hannenhalli, S., Kravitz, S., Levy, S., Mobarry, C., Reinert, K., Remington, K., Abu-Threideh, J., Beasley, E., Biddick, K., Bonazzi, V., Brandon, R., Cargill, M., Chandramouliswaran, I., Charlab, R., Chaturvedi, K., Deng, Z., Di Francesco, V., Dunn, P., Eilbeck, K., Evangelista, C., Gabrielian, A. E., Gan, W., Ge, W., Gong, F., Gu, Z., Guan, P., Heiman, T. J., Higgins, M. E., Ji, R. R., Ke, Z., Ketchum, K. A., Lai, Z., Lei, Y., Li, Z., Li, J., Liang, Y., Lin, X., Lu, F., Merkulov, G. V., Milshina, N., Moore, H. M., Naik, A. K., Narayan, V. A., Neelam, B., Nusskern, D., Rusch, D. B., Salzberg, S., Shao, W., Shue, B., Sun, J., Wang, Z., Wang, A., Wang, X., Wang, J., Wei, M., Wides, R., Xiao, C., Yan, C., Yao, A., Ye, J., Zhan, M., Zhang, W., Zhang, H., Zhao, Q., Zheng, L., Zhong, F., Zhong, W., Zhu, S., Zhao, S., Gilbert, D., Baumhueter, S., Spier, G., Carter, C., Cravchik, A., Woodage, T., Ali, F., An, H., Awe, A., Baldwin, D., Baden, H., Barnstead, M., Barrow, I., Beeson, K., Busam, D., Carver, A., Center, A., Cheng, M. L., Curry, L., Danaher, S., Davenport, L., Desilets, R., Dietz, S., Dodson, K., Doup, L., Ferriera, S., Garg, N., Gluecksmann, A., Hart, B., Haynes, J., Haynes, C., Heiner, C., Hladun, S., Hostin, D., Houck, J., Howland, T., Ibegwam, C., Johnson, J., Kalush, F., Kline, L., Koduru, S., Love, A., Mann, F., May, D., McCawley, S., McIntosh, T., McMullen, I., Moy, M., Moy, L., Murphy, B., Nelson, K., Pfannkoch, C., Pratts, E., Puri, V., Qureshi, H., Reardon, M., Rodriguez, R., Rogers, Y. H., Romblad, D., Ruhfel, B., Scott, R., Sitter, C., Smallwood, M., Stewart, E., Strong, R., Suh, E., Thomas, R., Tint, N. N., Tse, S., Vech, C., Wang, G., Wetter, J., Williams, S., Williams, M., Windsor, S., Winn-Deen, E., Wolfe, K., Zaveri, J., Zaveri, K., Abril, J. F., Guigo, R., Campbell, M. J., Sjolander, K. V., Karlak, B., Kejariwal, A., Mi, H., Lazareva, B., Hatton, T., Narechania, A., Diemer, K., Muruganujan, A., Guo, N., Sato, S., Bafna, V., Istrail, S., Lippert, R., Schwartz, R., Walenz, B., Yooseph, S., Allen, D., Basu, A., Baxendale, J., Blick, L., Caminha, M., Carnes-Stine, J., Caulk, P., Chiang, Y. H., Coyne, M., Dahlke, C., Mays, A., Dombroski, M., Donnelly, M., Ely, D., Esparham, S., Fosler, C., Gire, H., Glanowski, S., Glasser, K., Glodek, A., Gorokhov, M., Graham, K., Gropman, B., Harris, M., Heil, J., Henderson, S., Hoover, J., Jennings, D., Jordan, C., Jordan, J., Kasha, J., Kagan, L., Kraft, C., Levitsky, A., Lewis, M., Liu, X., Lopez, J., Ma, D., Majoros, W., McDaniel, J., Murphy, S., Newman, M., Nguyen, T., Nguyen, N., Nodell, M., Pan, S., Peck, J., Peterson, M., Rowe, W., Sanders, R., Scott, J., Simpson, M., Smith, T., Sprague, A., Stockwell, T., Turner, R., Venter, E., Wang, M., Wen, M., Wu, D., Wu, M., Xia, A., Zandieh, A., and Zhu, X. (2001). The sequence of the human genome. *Science*, 291(5507):1304–1351.

Wang, J., Mullighan, C. G., Easton, J., Roberts, S., Heatley, S. L., Ma, J., Rusch, M. C., Chen, K., Harris, C. C., Ding, L., *et al.* (2011). CREST maps somatic structural variation in cancer genomes with base-pair resolution. *Nature Methods*, 8(8):652–654.

Watson, J. D. and Crick, F. H. C. (1953). A structure for deoxyribose nucleic acid. *Nature*,

421(6921):397–3988.

Weber, J. L. and Myers, E. W. (1997). Human whole-genome shotgun sequencing. *Genome Research*, 7(5):401–409.

Weese, D., Emde, A.-K., Rausch, T., Döring, A., and Reinert, K. (2009). RazerS—fast read mapping with sensitivity control. *Genome Research*, 19(9):1646–1654.

Weese, D., Holtgrewe, M., and Reinert, K. (2012). RazerS 3: faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599.

Weisenfeld, N. I., Yin, S., Sharpe, T., Lau, B., Hegarty, R., Holmes, L., Sogoloff, B., Tabbaa, D., Williams, L., Russ, C., Nusbaum, C., Lander, E. S., MacCallum, I., and Jaffe, D. B. (2014). Comprehensive variation discovery in single human genomes. *Nature Genetics*, 46(12):1350–1355.

Wong, K., Keane, T. M., Stalker, J., and Adams, D. J. (2010). Enhanced structural variant and breakpoint detection using SVMerge by integration of multiple detection methods and local assembly. *Genome Biology*, 11(12):R128.

Xie, C. and Tammi, M. T. (2009). CNV-seq, a new method to detect copy number variation using high-throughput sequencing. *BMC Bioinformatics*, 10(1):80.

Yalcin, B., Wong, K., Bhomra, A., Goodson, M., Keane, T. M., Adams, D. J., and Flint, J. (2012). The fine-scale architecture of structural variants in 17 mouse genomes. *Genome Biology*, 13(3):R18.

Yang, X., Chockalingam, S. P., and Aluru, S. (2013). A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66.

Yang, X., Dorman, K. S., and Aluru, S. (2010). Reptile: representative tiling for short read error correction. *Bioinformatics*, 26(20):2526–2533.

Ye, K., Schulz, M. H., Long, Q., Apweiler, R., and Ning, Z. (2009). Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871.

Yoon, S., Xuan, Z., Makarov, V., Ye, K., and Sebat, J. (2009). Sensitive and accurate detection of copy number variants using read depth of coverage. *Genome Research*, 19(9):1586–1592.

Zeitouni, B., Boeva, V., Janoueix-Lerosey, I., Loeillet, S., Legoix-Né, P., Nicolas, A., Delattre, O., and Barillot, E. (2010). SVDetect: a tool to identify genomic structural variations from paired-end and mate-pair sequencing data. *Bioinformatics*, 26(15):1895–1896.

Zhang, J. and Wu, Y. (2011). SVseq: an approach for detecting exact breakpoints of deletions with low-coverage sequence data. *Bioinformatics*, 27(23):3228–3234.

Zhu, M., Need, A. C., Han, Y., Ge, D., Maia, J. M., Zhu, Q., Heinzen, E. L., Cirulli, E. T., Pelak, K., He, M., *et al.* (2012). Using ERDS to infer copy-number variants in high-coverage genomes. *The American Journal of Human Genetics*, 91(3):408–421.

Appendix A

Mason - Simulation of Nucleic Sequence

> All models are wrong but some are useful.
>
> *(George E. P. Box)*

Mason is a versatile software package for the simulation of nucleic sequence with a focus on read simulation of whole genome sequencing. Such simulated data is very useful as test inputs since one has a tight grip on the properties of such data through simulation. Simulated data also allows to observe the behaviour of algorithms (and their implementation) when certain properties vary. When simulating NGS reads, for example, it is possible to increase the error rate in a controlled fashion or to simulate reads that are longer than the current technology permits. Further, benchmarks with simulated reads can complement the evaluation with real-world data since the ground truth is known. A simulator can write out the sample location of the read and, for example, how many SNV bases are covered by the read or information about sequencing errors.

Originally, Mason was conceived and implemented as a simulator for the read mapping benchmark in (Holtgrewe *et al.*, 2011). However, in the current second version, it is a software package for the simulation of small and structural variants as well as the simulation of reads. Also, the package allows to simulate methylation levels and bisulphite-treated reads. Mason 2 allows to integrate the tools into a coherent pipeline for the simulation of genomic variants and reads. For example, this allows to simulate variants and store the results in a VCF file and then simulate paired-end reads generating the donor genome from the VCF file on the fly. For a second example, this allows to simulate methylation levels for the genome and store the resulting levels as a FASTA file, and then load the methylation levels and a VCF file with variants and simulate bisulphite-treated reads generating the donor genome on the fly.

Genomic Variants

The program Mason-Variator allows the simulation of small and large variants. The input is a reference FASTA file and the output is a VCF file with the simulated variants. Optionally, Mason-Variator can also apply the variants from all simulated haplotypes to the input FASTA file and write out a FASTA file with all variation applied to it.

Mason-Variator processes each contig individually. First, it simulates non-overlapping structural variants. These are either also simulated with per-base probabilities or a TSV file with structural variant types and sizes can be read and according SVs are randomly distributed into the genome. This file can also contain explicit sequence for SV insertions. Second, it simulates small variants (SNVs and small indels) into the contig using per-base probabilities.

The simulated variants are subject to some constraints:

- small variants have a distance of at least one base to SV breakpoints,
- SVs are non-overlapping and intra-chromosomal, and
- SV breakpoints do not span gaps (`N`s in the reference) and have a distance of at least one base to the gap.

The program Mason-Variator is implemented in a memory-efficient manner; treating each contig separately allows to only load one sequence (*e.g.*, chromosome) from the reference at a time.

Mason-Materializer can read in a FASTA file and a VCF file with variants (*e.g.*, generated by Mason-Variator) and generate the haplotypes of the individual encoded in the VCF file. The resulting FASTA file contains one sequence for each sequence in the input FASTA file and each haplotype in the VCF file. Both, Mason-Materializer and Mason-Variator can also read in a methylation level FASTA file and also write out the methylation level FASTA file for the materialized variants.

The program Mason-Genome allows to simulate DNA sequences from random nucleotides and store them as a FASTA file.

Read Simulation

The program Mason-Simulator allows the simulation of reads. In the current version, it provides the functionality to simulate Sanger reads (using the model described by Myers (1999a)), 454 pyrosequencing reads (using the model described by Richter *et al.* (2008)), and Illumina reads (using a simple custom error model).

Mason-Simulator gets as the input a genome FASTA file and optionally a variant VCF file. For each read (or pair) that is to be simulated, the program picks a chromsome (sequence in the input FASTA file) and a haplotype from the VCF file and stores this information in a files on the disk. There is a file for each chromosome haplotype, containing the integer IDs of the reads that are

to be simulated. The program then generates the haplotypes from the VCF file chromosome by chromosome and samples segments (*templates*) from these haplotype sequences. These template are not allowed to span gaps (`N`s) in the haplotypes and their length is chosen randomly either following a normal distribution or uniformly selected from an interval of length values, given in the program's configuration. Then, sequencing is simulated from one or both ends of the template, using the selected read simulation model. The resulting reads and base pair qualities are then written to a SAM file together with their sample position and some meta information (*e.g.*, how many SNV bases are spanned by the read). There is one such SAM file for each haplotype.

After the simulation of all reads, the read alignments from all files are read mimicking multi-way merging and a FASTQ file for the left reads (and one for the right reads, if applicable) is generated. Optionally, a SAM or BAM file with the alignments of the reads against the reference sequence (without variants) can be generated. This alignment file also contains meta information about the read's alignment (*e.g.*, the number of sequencing errors, spanned SNV and small indel bases, original sample position in the haplotype, etc.).

Mason-Simulator can also simulate bisulphite-treated reads. The per-base methylation levels can be read from a methylation FASTA file or be simulated on the fly.

Internally, the same code for the materialization is used as in Mason-Variator, Mason-Materializer, and Mason-Methylation. This means that using the same seeds, the same haplotypes and methylation levels are generated.

The program Mason-Frag-Sequencing provides access to the read simulation code. Instead of genomic sequence, the program expects a FASTA file with generated templates. It then simulates single-end or paired-end sequencing of these templates. The user can generate this template FASTA file from arbitrary data, *e.g.*, simulate the ligation of barcodes or adapters or any other protocol.

Nucleic Simulation Utilies

Besides the program described above, the Mason 2 software package also contains the following programs.

Methylation Rate Simulation. Mason-Methylation allows the simulation of methylation level based on the cysteine context. In a cell's DNA macromolecules, the cysteine might be methylated and the methylation of either strand (called *top* and *bottom* strand here) might be different.

A methylation FASTA file describes the methylation levels of the top and bottom strand with one sequence for each sequence in the reference. The methylation level for each base is given and set to 0% for non-cysteine bases. The levels are encoded in printable ASCII characters in steps of 1.25% from "`'!'`" (0%) to "`'p'`" (100%). The character "`'>'`" is skipped to circumvent any ambiguity problems with the FASTA format.

As mentioned above, the methylation levels are simulated in a context-dependent fashion following a beta distribution. For each of the contexts *CpG* (C followed by a G), *CHG* (C followed by non-G, followed by a G), and *CHH* (C followed by two non-G characters), the beta distribution can be selected by specifying a mean and standard deviation value.

The methylation simulation has its own seed for random number generation in the materialization and read simulation programs, such that different levels can be generated for each individual and also multiple read sets can be generated for the same methylation levels.

Transcriptome Computation. The program MASON-SPLICING allows the generation of transcriptomes from a reference FASTA file and a GFF or GTF file describing the transcripts.

Appendix B

Sequencing Technology Overview

The aim of *sequencing* is to determine the genomic sequence in a sample. This appendix describes a selection of relevant sequencing technologies as outlined by Mardis (2008, 2013).

Sanger Sequencing

Sanger *et al.* (1977) developed a method for sequencing based on *chain-termination*. The general idea is to force the DNA polymerase to include modified nucleotides (ddNTPs) into the DNA molecule synthesis that can be later used to read out DNA sequence.

As a first step, the part *(template)* of the DNA that is to be sequenced is amplified using *polymerase chain reaction (PCR)*. Then, another PCR step is performed for the actual sequencing process. The amplified sample from the first step is divided into four separate parts and the second PCR step is prepared for each part with normal nucleotides. Additionally, a different kind of ddNTP is added to each part (one for each `A`, `C`, `G`, and `T`). The incorporation of ddNTPs into the elongated molecule occurs randomly and stops the elongation of this molecule.

After the elgongation, the fragments are heat-denatured and then separated by size, *e.g.*, using gel electrophoresis. The ddNTPs include a radioactive label for the subsequent detection step. This way, an X-ray photograph of the gel can be taken from which the sequence of the DNA molecule can be read.

This general idea was used for more than 25 years, and also for the first whole human genome sequences. During this time, the method was refined, *e.g.*, by using flouroscenctly labeled ddNTPs and automatizing the process in complex sequencing machines, *e.g.*, *capillary sequencing machines*. At the time of writing, a current sequencing machine based on Sanger sequencing is able to yield reads with a length of 400–900 base pairs (*bp*s) at very high accuracy.

Sequencing methods based on the Sanger approach with chain-terminating ddNTPs are commonly called *first-generation sequencing* methods.

454 Pyrosequencing

The first commercially available *second-generation sequencing* (2GS) method, called *pyrosequencing* was developed by 454 Life Sciences (Margulies *et al.*, 2005). The main innovation in 2GS is to reuse the sequencing templates during sequencing.

In the 454 method, the single-stranded DNA molecule templates are captured by beads of chemicals in which they are enriched by *emulsion PCR*. These beads are then placed in small (44 μm) wells on a plate.

During sequencing, the plate is flooded with solutions in cycles of the same modified nucleotide. In each cycle, the missing strand is elongated and the incorporated modified nucleotide lead to light being emitted that can be recorded by a camera. The intensity of the light depends on the number of incorporated nucleotides. This intensity is then converted into a nucleotide count and together with the information of which nucleotides are currently used for flooding, the next stretch of homopolymers can be determined for each read.

Illumina (Solexa) Sequencing

The second commercially available 2GS method was developed by Solexa (Bentley *et al.*, 2008). The main difference between Illumina and 454 sequencing is that Illumina uses so-called *reversible terminator chemistry*: Similar to 454 sequencing, in Illumina sequencing, the single-ended DNA molecules are attached to a plate (flow cell). This attachment occurs directly and not using wells and beads and the templates are amplified using *bridge amplification* (another type of PCR). This creates clusters on the flow cell of the same sequence.

Then, solutions with modified nucleotides are flooded over the plate and elongate the templates. However, with Illumina chemistry, only one additional nucleotide can be incorporated by the polymerase for the elongation. A laser is then used to stimulate a light flash of the modified nucleotides in each cluster and this is recorded by a camera. Then, another solution is flooded over the plate such that the elongation by one nucleotide can continue in the next cycle.

IonTorrent Sequencing

Another recent 2GS method, called IonTorrent sequencing, was developed by Life Technology (Rothberg *et al.*, 2011). Similar to pyrosequencing, the templates are captured in beads which are placed in wells on a CMOS chip. Instead of measuring light, however, the chemistry used in IonTorrent sequencing generates changes in the pH level which are directly detected by the CMOS chip with the wells.

Single Molecule Sequencing

Single-molecule sequencing (or *third-generation sequencing (3GS)* is another recent innovation in sequencing technology. All 2GS methods described above use PCR for amplifying the signal of individual reads. In contrast, 3GS methods do not need such a PCR step and thus circumvent certain problems associated with its usage.

Examples are the Single Molecule Real-Time Sequencing by Pacific Biosciences (PacBio) (Eid *et al.*, 2009), which is a currently available commercial system and Helicos Single Molecule Fluorescent Sequencing (Thompson and Steinmann, 2010) which was a commercially available system.

The main advantage of 2GS and 3GS methods is the high throughput when compared to Sanger sequencing. Thus, both 2GS and 3GS methods are often called *high-throughput sequencing (HTS)* methods.

Appendix C

Contributions to Software Engineering and Usability

As outlined in Section 1.3, the design, analysis, implementation, and experiments of algorithms are at the core of algorithm engineering. The aim of algorithm engineering is to improve applications with algorithm implementations yielded by this cycle. This means that at least a driver program should be made available such that the algorithm can be run by users or it should be integrated as a *component* in larger applications. Ideally, the developed algorithmic components are also made available to a broader public by including them in a library, such that they can be *easily reused*. Of course, the algorithmic components (as well as the programs) should be well-tested and well-documented such that they can be used by a larger public.

The result of my work towards this thesis includes practical, tested, and documented programs as well as contributions of components as SeqAn (Döring *et al.*, 2008) library modules. Further, I performed work on the tools and frameworks for writing and building documentation and writing and running tests. In this appendix, I describe these software engineering contributions. I describe the implemented programs and algorithmic SeqAn modules, my contributions to SeqAn that facilitate writing self-documenting programs, my contributions to the library's documentation. Finally, I describe contributions to the testing framework in SeqAn.

Implemented Programs

All programs described below use standard bioinformatics file formats where possible. Thus, they allow for the easy integration into standard software pipelines. Automated tests are available for all programs.

RazerS 3 is a parallel, string-filtration based read mapper that achieves high sensitivity in all-mapping mode. Section 4.4.

Rabema is a software package for the evaluation of read mappers. It is based on the formal definition of read mapping described Section 4.1. The software package consists of the programs Rabema-Build-Gold-Standard and Rabema-Evaluate and is described in detail in Section 4.2.

Mason is a software package for the simulation of nucleic sequence. I give a detailed description in Appendix A The software package contains the programs Mason-Genome, Mason-Variator, Mason-Materializer, Mason-Methylation, and Mason-Splicing for the simulation and processing of genomic sequence and the programs Mason-Simulator and Mason-Frag-Sequencing for the simulation of sequencing data.

Basil and Anise are programs for the detection of insertion breakpoints and the subsequent assembly of inserted data. I describe them in detail in Chapter 5. Both programs feature an efficient implementation and employ multi-core parallelism. Further, the consensus module from Anise is used in the second version of the SeqAn tool SeqCons. The previous version of SeqCons did not work reliably any more since there were no tests for this program and the library changed greatly after the program's initial development.

Fiona is a parallel tool for the correction of NGS read and I developed and benchmarked in collaboration with my coauthors in (Schulz *et al.*, 2014). My main contribution lie in the experimental evaluation and testing.

SeqAn Module Contributions

For all modules, API documentation is provided together with demo programs. For most modules, tutorials are available on the SeqAn website.

realign is a module implementing the realignment algorithm by Anson and Myers (1997). It is a reimplementation of the algorithm; a previous implementation in the SeqAn library showed various bugs, presumably since the behaviour of the used library functionality changed subtly. The original implementation was not tested and thus these bugs were not detected.

align is the SeqAn alignment module. I restructured the module and improved the public data structures such as *Gaps* and *Align* as well as the algorithm interfaces.

consensus is a module for consensus computation. I made the consensus module from Anise available in this module which now supersedes the previous version of the module.

SEQUENCE_JOURNALED is a module providing a `String` class specialization that allows to store changes to a string in a lightweight implementation. Originally implemented for the first version of MASON, the module has now been greatly extended by René Rahn for his journaled string tree project (Rahn *et al.*, 2014).

STREAM is a module providing low-level file I/O functionality that I implemented an earlier version of. Later, David Weese and Enrico Siragusa further improved the I/O functionality in SeqAn and rewrote the module.

BAM_IO, BED_IO, GFF_IO, SEQ_IO, VCF_IO are modules providing the data structures and routines for read and write access to the SAM/BAM, BED, GFF/GTF, FASTA/FASTQ, and VCF file formats. The modules have subsequently been updated to the Weese's and Siragusa's new I/O layer. Thus, most important bioinformatics file formats are now available to SeqAn users and in SeqAn applications.

Further work includes the modules ALIGN_SPLIT for the split alignment with the updated ALIGN module by René Rahn, refactorization and documentation to the modules BASIC, improvements and test to the GRAPH_ALGORITHMS module, and together with David Weese and Enrico Siragusa the development of a basic framework for parallel algorithms in PARALLEL.

Self-Documenting Programs

It can be considered best practice that command line programs are self-documenting when being passed a `--help` (or similarly named) flag. The program should then print a textual help and perform no other action. In SeqAn, the `ArgumentParser` class in the ARG_PARSE module provides the functionality for describing a program's parameters programatically and then parsing the command line. This class allows the annotation of positional arguments and named options with a description and to provide additional information.

During the work for this thesis, the basic functionality was greatly extended by the SeqAn team and I contributed code for the documentation of the programs. The representation of a program's documentation is built in an object of class `ToolDoc` when the `ArgumentParser` class' interface is used to describe the command line interface of a program. Figure C.1 shows an example usage of the `ArgumentParser` class.

The documentation of the program can then be generated in several formats. Textual output is meant for the console and the output is wrapped to the user's terminal dimensions. The documentation can also be generated in HTML format for publishing on the web or in Unix man format such that a program's man page can be generated automatically. Figure C.2 shows the automatically generated man page by RABEMA-BUILD-GOLD-STANDARD.

```
#include <seqan/arg_parse.h>

// ...

seqan::ArgumentParser parser("rabema_build_gold_standard");
setShortDescription(parser, "RABEMA Gold Standard Builder");
setVersion(parser, "1.2.0");
setDate(parser, "March 14, 2013");
setCategory(parser, "Benchmarking");

addUsageLine(parser,
             "[\\fIOPTIONS\\fP] \\fB--out-gsi\\fP \\fIOUT.gsi\\fP \\fB--reference\\fP "
             "\\fIREF.fa\\fP \\fB--in-sam\\fP \\fIPERFECT.sam\\fP");
// ...
```

Figure C.1: Example usage of the `seqan::ArgumentParser` class for defining the program documentation in the program.

```
RABEMA_BUILD_GOLD_STANDARD(1)                    RABEMA_BUILD_GOLD_STANDARD(1)

NAME
       rabema_build_gold_standard - RABEMA Gold Standard Builder

SYNOPSIS
       rabema_build_gold_standard   [OPTIONS]  --out-gsi  OUT.gsi  --reference
       REF.fa --in-sam PERFECT.sam
       rabema_build_gold_standard  [OPTIONS]  --out-gsi  OUT.gsi   --reference
       REF.fa --in-bam PERFECT.bam

DESCRIPTION
       This  program  allows  to build a RABEMA gold standard.  The input is a
       reference FASTA file and a perfect SAM/BAM map (e.g. created using Raz-
       erS 3 in full-sensitivity mode).

       The  input SAM/BAM file must be sorted by coordinate.  The program will
       create a FASTA index file REF.fa.fai for fast random access to the ref-
       erence.
```

Figure C.2: Top of the output of the argument parser, displayed as Linux man page.

```
/**
.Function.globalAlignment:
..summary:Computes the best global alignment of the two sequences.
..cat:Alignments
..signature:globalAlignment(align, score [, align_config], tag)
..signature:globalAlignment(result, strings, score [, align_config], tag)
..signature:globalAlignment(result, strings, score [, align_config], diagLow, diagHigh, tag)
..param.align:An alignment data structure containing two sequences.
...type:Spec.Alignment Graph
...type:Class.Align
..param.result:A data structure that gets the result of the alignment procedure,
 e.g., a file stream, or std::cout for a textual alignment, or a FragmentString for storing
all the matches.
..param.strings:A string set with that contains two sequences.
```

Figure C.3: Beginning of C++ comment with DDDoc comment for the `globalAlignment()` function.

API Documentation

The usability of a library (how easy or hard it is to use a library) highly depends on the quality of the library's API documentation. Most of today's API documentation is written together with the source code and embedded in comments following a special format. The documentation is then extracted from the comments by tools that build a human-readable document, often a HTML website or a PDF file. Prominent examples of such systems are Doxygen for C++ and Javadoc for Java.

During the conception of SeqAn, the original library author introduced his own system DotDotDoc. The reason for this is that SeqAn uses programming entities such as concepts, template inheritance, and global interface functions. These elements are not part of the C++ language and thus the support for them does not exist in Doxygen. The emulation of these features in Doxygen was not satisfactory. DotDotDoc has explicit support for these features. Figure C.3 shows an example of DotDotDoc documentation for the `globalAlignment()` function and Figure C.4 shows the generated HTML documentation for this function.

However, this system was found to be lacking. For one, the format for defining the documentation was idiosyncratic to the DDDoc system. Also, a single page was generated for each global interface function, such as `length()` whereas functions such as `length()` should occur on the documentation for the `ContainerConcept` concept.

Thus, a new system based on the syntax from Doxygen was written from scratch: *dox*. Dox is implemented in Python and allows users to use similar documentation markup as in Javadoc and Doxygen. The resulting HTML document was also improved, allowing for a much better search. Figure C.5 shows the dox documentation of the function `globalAlignment()` and Figure C.6 shows the resulting HTML page.

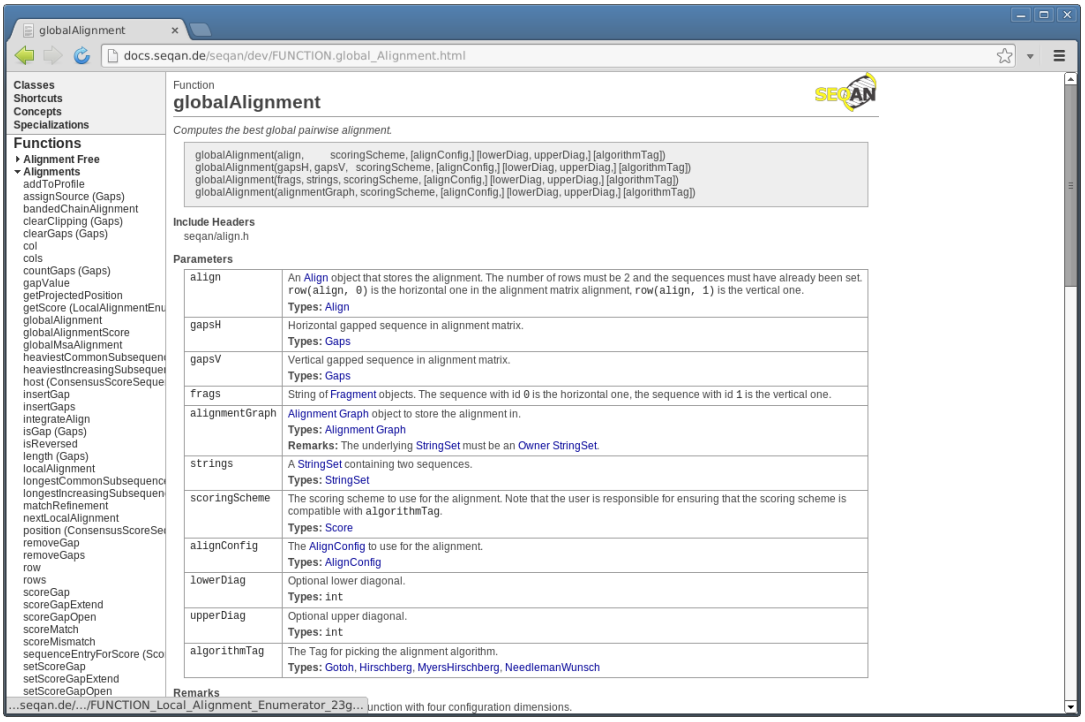


Figure C.4: HTML documentation generated for function `globalAlignment()` from DDDoc source in Figure C.3.

```
/*!
 * @fn globalAlignment
 *
 * @headerfile seqan/align.h
 *
 * @brief Computes the best global pairwise alignment.
 *
 * @signature TScoreVal globalAlignment(align,          scoringScheme, [alignConfig,]
 *                                      [lowerDiag, upperDiag,] [algorithmTag]);
 * @signature TScoreVal globalAlignment(gapsH, gapsV,   scoringScheme, [alignConfig,]
 *                                      [lowerDiag, upperDiag,] [algorithmTag]);
 * @signature TScoreVal globalAlignment(frags, strings, scoringScheme, [alignConfig,]
 *                                      [lowerDiag, upperDiag,] [algorithmTag]);
 * @signature TScoreVal globalAlignment(alignGraph,     scoringScheme, [alignConfig,]
 *                                      [lowerDiag, upperDiag,] [algorithmTag]);
 *
 * @param align        The @link Align @endlink object to use for storing the
 *                     pairwise alignment.
 * @param gapsH        The @link Gaps @endlink object for the first row (horizontal
 *                     in the DP matrix).
 * @param gapsV        The @link Gaps @endlink object for the second row (vertical in
 *                     the DP matrix).
```

Figure C.5: Beginning of the C++ comment with dox documentation for `globalAlignment()`.

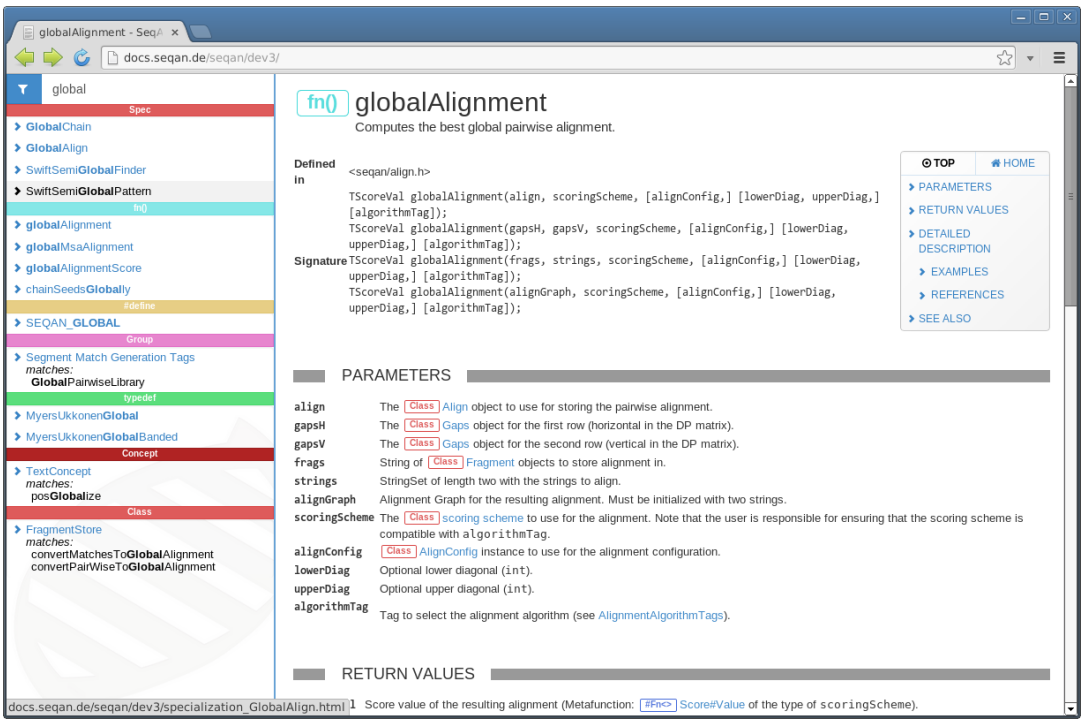


Figure C.6: HTML documentation generated for function `globalAlignment()` from dox source in Figure C.5.

Testing

Testing is a very important aspect of software engineering. At the beginning of the work of this thesis, there were few tests for the SeqAn library and no tests for the SeqAn applications. During the work for this thesis, I rewrote the test framework for the SeqAn library and also provided a framework for application tests. Together with the rest of the SeqAn team, I extended the framework and wrote many tests for the library functionality and also for the applications. At the time of writing, there are more than 2 500 individual tests for the SeqAn library and appplication tests for almost all applications. This allows to easily detect whether a change in the library breaks another part or an application.

The tests are run each night and the results are submitted to CDash dashboard, a web application for collecting information about project builds and tests contained in the CMake software package[1]. Figure C.7 shows an example of SeqAn's CDash Dashboard.

[1] `http://www.cmake.org/`

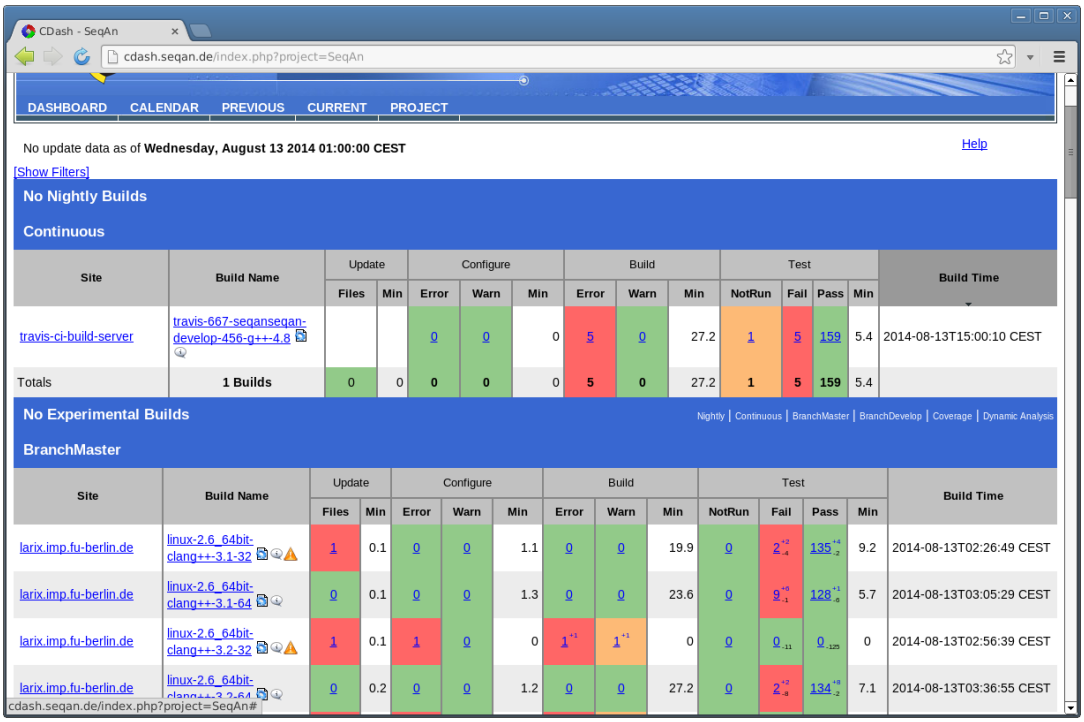


Figure C.7: Example of the CDash dashboard showing information about compiler warnings, errors, and test failures.

Appendix D

Program Versions and Calls Used in the Evaluation

Evaluation of Read Error Correction

This section gives the versions and command lines of the read error correction programs used in the evaluation in Section 3.4.

For all programs the gain analysis was done on the complete read set, even if some programs discarded reads from their output. Below, the placeholders `IN.fq` and `IN.fa` are used for the path to the input FASTQ/FASTA file, `OUT.fq` and `OUT.fa` are used for the path to the output FASTQ/FASTA file, and `GLEN` is used for the genome length.

Allpaths-LG was used in release 44994. The read correction program `ErrorCorrectReads.pl` was run with the settings shown in the example below. The output was postprocessed to add back the reads that were removed by the read correction routine although `REMOVE_DODGY_READS=0` was set. Example:

```
ErrorCorrectReads.pl THREADS=8 PHRED_ENCODING=33 REMOVE_DODGY_READS=0 \
UNPAIRED_READS_IN=IN.fq READS_OUT=OUT.fq
```

Coral was used in version 1.4. It was run with default parameters, i.e. `-fq` for the input FASTQ files, `-o` for the output file, and `-454` for both 454 and IonTorrend data. For some 454 and Ion Torrent data, Coral's parameters had to be adjusted to allow higher error rates to yield better gain (as instructed by the Coral authors).

Coral was run with default parameters (an error rate of 7% or `-e 0.07`). It was also tried to increase the error rate in Coral to 10%, 15%, 20%, and 25%. The best variant for each data set is

given as Coral*. Coral was stopped after 24 hours on the 18x *D. melanogaster* data set with `-e 0.20` and `-e 0.25`.

Coral uses 8 threads by default. Example:

```
coral -fq IN.fq -o OUT.fa -454 -e 0.10.
```

Fiona was used in version 0.2. The values chosen automatically by Fiona for $k_{\min}$ were 14 for the *E. coli* data sets, 15 for *S. cerevisae*, 16 for *RAL399-2L*, and 17 for *C. elegans* and *D. melanogaster*. The parameter `-t 8` was used for eight threads, and pass the genome length using `-g GLEN`. A k-mer of length 9 was hard-coded into Fiona for benchmarking (and can be changed after recompiling), and the default error rate of 0.01 was used. The sequencing technology was selected using `--sequencing-technology` with the values `454` and `iontorrent`. Example:

```
fiona --sequencing-technology 454 -nt 8 -g 1000 IN.fq OUT.fa.
```

HiTEC was used in version 1.0 (BaseSpaceShrec). An error rate of 1% was used by default as adviced by the HiTEC authors. Example:

```
hitec IN.fq OUT.fq GLEN 1
```

HybridShrec was used in version 1.0 (`BaseSpaceShrec`). HybridShrec was run in two versions. The variant HybridShrec was run with default parameters as instructed by the authors. For most data sets, the program did not run through but exited with the suggestion to lower the strictness parameter `-s`. Because the achieved gain did not correlate with the strictness parameter value, it was run with strictness values set from 2 to 7 and reported the best achieved gain.

The default setting for levels to use is 14 to 17. The variant HybridShrec^F used the same levels as fiona. This variant was run with strictness values 2 to 7 as well. Example:

```
java -Xmx40960m Shrec -n 5348428 -s 7 IN.fa
```

Quake was used in version 1.0 (BaseSpaceShrec). It was run with default parameters as given in the manual. The value for `K` was chosen as $\log(200 \cdot \texttt{GLEN})/\log 4$ as given by the Quake manual. Example:

```
quake.py -f IN.fq -k K -p 8
```

Racer was used in the version from the 25th of June, 2014, as sent to the author by Lucian Ilie. Here is an example for a program call:

```
RACER IN.fq OUT.fq GLEN
```

Evaluation of Read Mapping

This section describes the parameters used in the comparison of read mappers in Section 4.5. Also, this section gives the versions of the used read mapping programs.

`MIN` and `MAX` are placeholders for minimal and maximal insert size, `INS` is the mean insert size, and `IERR` the allowed deviation (`INS = (MIN + MAX) / 2`, `IERR = (MAX - MIN) / 2`). For the tools using indices, the index was built using default options.

Bowtie 2. Version 2.0.0-beta6 was used. The number of threads was selected using the parameter (`-p`). The parameter `--end-to-end` was used to enforce semi-global read alignments. The parameter `-k 100` was used for the Rabema experiment, otherwise `-k 1` was used. In paired-end mode, the parameters `--minins MIN --maxins MAX` were used.

BWA. Version 0.6.1-r104 was used. The parameter `-t` was used to select the number of reads in the `aln` step. The `sampe` and `samse` steps were performed using one thread since BWA does not offer a parallelization here. When mapping for the Rabema experiment, the parameter `-N` was passed to `aln` and `-n 100` to `samse`. Otherwise, the parameter `-n 1` was passed to `samse`. The insert size was not passed to BWA, however the insert size and allowed error from BWA's output was passed to the other read mappers.

Hobbes. Version 1.3 was used. Since the focus lies on edit distance, the 16 bit bit-vector version was used as described in (Ahmadi *et al.*, 2012). The index was built using the recommended[1] k-mer length 11. Indels were enabled using `--indels`. Maximal edit distance was set using `-v`. Multi-threading was enabled using `-p`. For resource measurement, the output without CIGAR was used. For analyzing the results, the CIGAR output was enabled using `--cigar`. In paired-end mode, the parameters `--pe --min MIN --max MAX` were used.

mrFAST. Version 2.1.0.6 was used. It was used as explained in the manual[2]. mrFAST does not support multithreading. The input was divided into blocks of 500 k reads and processed each chunk in a separate process using the program ts[3]. Long reads were split into packages of 100 k reads. This way, always 8 processes were executed in parallel. The edit distance error rate was set to 4% of the read length.

RazerS 3. Version 3.0 was used. RazerS 3 was parameterized as follows: The native or SAM output format was selected with `-of 0` or `-of 4`. Indel support was disabled with `--no-gaps` when required. The number of threads was set with the `-tc` parameter. The percent recognition rate was set using the `-rr` parameter, *e.g.*, `-rr 100` or `-rr 99`. The error rate was set through the `-i` parameter, e.g. `-i 96` to map with 4% errors[4]. The pigeonhole or SWIFT filter was se-

[1] `http://hobbes.ics.uci.edu/manual.jsp`
[2] `http://mrfast.sourceforge.net/manual.html`
[3] `http://vicerveza.homeunix.net/~viric/soft/ts/`
[4] RazerS uses the percent identity, which is 100 minus error rate in percents.

lected using `-fl pigeonhole` or `-fl swift`. As an all-mapper, the parameter `-m 1000000` was used and as a best-mapper `-m 1` was used. In paired-end mode, the parameters used were `--library-length INS --library-error IERR`.

SHRiMP 2. Version 2.2.2 was used. The number of threads was selected with `--threads`. In paired-end mode, the options used are `--pair-mode opp-in --isize MIN,MAX`.

Soap 2. Version 2.1 was used. The number of threads was selected with `-p`. In paired-end mode, the options used are `-m MIN -x MAX`.

Evaluation of Insert Assembly

This section gives the program calls for the insert assembly evaluation from Sections 5.5–5.7.

ABYSS. ABYSS was run in version 1.5.1 using $k = 64$. Otherwise, I used default parameters for `abyss-pe` with the `unitigs` command.

BWA. BWA (Li and Durbin, 2009) was used in version 0.6.1-r104. I used the commands `aln` and `sampe` for aligning the paired-end reads using default options. I used the program as described in its documentation.

BLAT. BLAT (Kent, 2002) was used in version 35x1 with default options as used in the documentation.

Basil and Anise. Basil and Anise were used in version 1.0. The used command lines were for Basil:

```
# basil --input-reference REF.fa \
    --input-mapping MAPPING.bam \
    --out-vcf OUT.vcf
```

and for Anise:

```
# anise --num-threads THREADS \
    --input-reference REF.fa \
    --input-vcf IN.vcf \
    --input-mapping IN.bam \
    --output-fasta OUT.fa \
    --output-mapping OUT.bam
```

MindTheGap was used in version 6417. On the non-real-world data, I ran MindTheGap with $k = 51$ (as is the case in (Rizk *et al.*, 2014) for the simpler data sets):

```
# mindthegap index READS.fa -p whole\_human \
    -g 2870630962 -k 51
# mindthegap find REF.fa -p whole_human
# mindthegap fill data.breakpoints -p data
```

On the real-world data, it was run the same parameters as in (Rizk *et al.*, 2014):

```
# mindthegap index READS.fa -p whole_human \
    -g 2870630962 -k 63 -t 5
# mindthegap find REF.fa \
    -p whole_human -mrep 15
# mindthegap fill whole_human.breakpoints \
    -p whole_human
```

SGA was used in version 0.10.13. I used the same parameters as in the *C. elegans* configuration example that ships with the program: https://github.com/jts/sga/blob/master/src/examples/sga-celegans.sh.

Appendix E

Full Read Mapping Tables

This appendix contains further tables for the read mapping evaluation presented in Section 4.5.

reference	read length	read set ID	original length
E. coli	100 bp	ERR022075	100 bp
E. coli	70 bp	ERR022075	100 bp
E. coli	50 bp	ERR022075	100 bp
E. coli	30 bp	ERR032371	36 bp
C. elegans	100 bp	SRR065390	100 bp
C. elegans	70 bp	SRR065390	100 bp
C. elegans	50 bp	SRR065390	100 bp
C. elegans	30 bp	SRR107574	34 bp
human chr. 2	100 bp	ERR012100	101 bp
human chr. 2	70 bp	SRR029194	88 bp
human chr. 2	50 bp	SRR029194	88 bp
human chr. 2	30 bp	ERR003244	37 bp

Table E.1: This table gives information which datasets were used when creating the experimental maps. If original length and read length m are not equal then the first m bases were used.

		(0,0)		(1,0)		(2,0)		(3,0)		(4,0)		(1,1)		(1,2)		(0,3)		(0,4)	
	method	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	97.6	97.3	95.6	94.8	94.6	92.0	93.3	88.7	92.6	82.5	95.3	93.3	93.5	92.3	96.1	95.4	97.6	97.4
	BWA	98.2	97.9	97.1	96.4	97.6	95.3	96.5	90.2	94.9	85.1	97.4	90.9	97.1	80.3	96.3	66.5	97.5	67.1
	Soap 2	98.1	82.9	97.0	63.6	97.4	31.0	0.0	0.0	0.0	0.0	90.6	6.2	0.0	0.0	0.0	0.0	0.0	0.0
	R3-100	98.4	98.4	97.7	97.7	98.2	98.2	97.5	97.5	96.3	96.3	98.1	98.1	97.9	97.9	97.6	97.6	98.4	98.4
	R3-99	98.4	98.4	97.7	97.7	98.2	98.0	97.4	96.6	96.2	95.1	98.2	98.1	97.9	97.9	97.6	97.6	98.4	98.4
	R3-95	98.4	98.3	97.7	97.5	98.2	97.3	97.5	94.9	96.1	91.7	98.2	97.6	97.9	97.6	97.5	97.5	98.4	98.4
all-mappers	Hobbes	99.9	99.9	99.9	99.9	99.9	99.9	99.9	99.9	100.0	100.0	100.0	99.8	100.0	93.6	99.6	90.5	99.6	87.6
	mrFAST	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.3
	SHRiMP 2	100.0	99.4	100.0	99.5	100.0	99.7	100.0	99.9	100.0	99.7	100.0	99.5	100.0	99.2	100.0	99.6	100.0	99.6
	R3-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R3-99	100.0	100.0	100.0	99.9	100.0	99.8	100.0	99.1	100.0	98.9	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0
	R3-95	100.0	99.9	100.0	99.7	100.0	99.0	100.0	97.3	100.0	95.4	100.0	99.4	100.0	99.6	100.0	99.9	100.0	100.0

Table E.2: Full results for the variation detection experiments from Table 4.4a with single-end reads.

	method	(0,0) prec.	(0,0) recl.	(1,0) prec.	(1,0) recl.	(2,0) prec.	(2,0) recl.	(3,0) prec.	(3,0) recl.	(4,0) prec.	(4,0) recl.	(1,1) prec.	(1,1) recl.	(1,2) prec.	(1,2) recl.	(0,3) prec.	(0,3) recl.	(0,4) prec.	(0,4) recl.
best-mappers	Bowtie 2	97.6	97.3	95.6	94.8	94.6	92.0	93.3	88.7	92.6	82.5	95.3	93.3	93.5	92.3	96.1	95.4	97.6	97.4
	BWA	98.2	97.9	97.1	96.4	97.6	95.3	96.5	90.2	94.9	85.1	97.4	90.9	97.1	80.3	96.3	66.5	97.5	67.1
	Soap 2	98.1	82.9	97.0	63.6	97.4	31.0	0.0	0.0	0.0	0.0	90.6	6.2	0.0	0.0	0.0	0.0	0.0	0.0
	R3-100	98.4	98.4	97.7	97.7	98.2	98.2	97.5	97.5	96.3	96.3	98.1	98.1	97.9	97.9	97.6	97.6	98.4	98.4
	R3-99	98.4	98.4	97.7	97.7	98.2	98.0	97.4	96.6	96.2	95.1	98.2	98.1	97.9	97.9	97.6	97.6	98.4	98.4
	R3-95	98.4	98.3	97.7	97.5	98.2	97.3	97.5	94.9	96.1	91.7	98.2	97.6	97.9	97.6	97.5	97.5	98.4	98.4
all-mappers	Hobbes	99.9	99.9	99.9	99.9	99.9	99.9	99.9	99.9	100.0	100.0	100.0	99.8	100.0	93.6	99.6	90.5	99.6	87.6
	mrFAST	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.3
	SHRiMP 2	100.0	99.4	100.0	99.5	100.0	99.7	100.0	99.9	100.0	99.7	100.0	99.5	100.0	99.2	100.0	99.6	100.0	99.6
	R3-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R3-99	100.0	100.0	100.0	99.9	100.0	99.8	100.0	99.1	100.0	98.9	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0
	R3-95	100.0	99.9	100.0	99.7	100.0	99.0	100.0	97.3	100.0	95.4	100.0	99.4	100.0	99.6	100.0	99.9	100.0	100.0

Table E.3: Full results for the variation detection experiments from Table 4.4b with paired-end reads.

	dataset	ERR022075 E. coli					SRR065390 C. elegans				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	1:19	15:21	162	99.77	99.32	2:08	25:14	277	99.38	92.58
best-mappers	BWA	3:04	13:17	184	99.78	97.98	5:21	36:50	325	99.46	89.33
best-mappers	Soap 2	1:30	3:16	729	98.00	95.68	1:32	5:26	813	96.74	85.95
best-mappers	R3-100	0:50	2:00	5705	100.00	97.59	1:42	11:42	5841	100.00	88.79
best-mappers	R3-99	0:48	1:59	5705	100.00	97.59	1:38	11:17	5841	100.00	88.78
best-mappers	R3-95	0:47	1:57	5705	100.00	97.59	1:30	9:52	5841	99.98	88.77
all-mappers	Hobbes	2:02	12:55	769	91.46	89.28	26:46	287:26	3136	93.27	82.83
all-mappers	mrFAST	0:36	4:30	8269	100.00	97.59	6:01	57:42	10497	100.00	88.79
all-mappers	SHRiMP 2	4:17	49:42	1553	99.86	99.28	52:44	608:58	3372	98.70	91.91
all-mappers	R3-100	0:50	2:10	5705	100.00	97.59	2:20	16:57	9203	100.00	88.79
all-mappers	R3-99	0:49	2:09	5705	100.00	97.59	2:16	16:07	9036	100.00	88.78
all-mappers	R3-95	0:49	2:08	5705	100.00	97.59	2:11	14:24	8598	99.98	88.77

	dataset	SRR497711 D. melanogaster					ERR012100 H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	2:00	23:43	299	99.65	85.71	5:37	66:40	3374	99.62	96.72
best-mappers	BWA	5:35	36:30	413	98.96	79.37	13:45	98:30	4475	99.66	93.53
best-mappers	Soap 2	1:55	8:37	900	91.78	72.49	2:34	15:11	5481	96.45	89.73
best-mappers	R3-100	1:28	9:45	5795	100.00	78.92	85:56	1001:18	9096	100.00	92.99
best-mappers	R3-99	1:26	9:14	5795	99.98	78.90	73:09	848:59	8679	99.99	92.98
best-mappers	R3-95	1:26	9:24	5795	99.87	78.82	43:16	493:42	7640	99.96	92.95
all-mappers	Hobbes	4:51	39:57	2141	96.49	76.16	265:48	2851:00	70683	95.97	89.24
all-mappers	mrFAST	4:01	37:32	10844	100.00	78.92	413:40	3987:49	11324	100.00	92.99
all-mappers	SHRiMP 2	23:40	255:09	3801	99.83	89.91	1312:09	14466:34	38188	99.81	99.06
all-mappers	R3-100	1:51	11:59	7329	100.00	78.92	118:26	1384:18	15298	100.00	92.99
all-mappers	R3-99	1:49	11:21	7302	99.98	78.90	100:19	1169:22	15238	99.99	92.98
all-mappers	R3-95	1:45	10:40	7270	99.87	78.82	58:13	665:47	14384	99.96	92.95

Table E.4: Extended experimental results for real-world single-end data, extending Table 4.5a. The results are shown for the first 10 M × 100 bp single-end reads of Illumina datasets.

	dataset	simulated, $m = 200$ D. melanogaster					simulated, $m = 400$ D. melanogaster					simulated, $m = 800$ D. melanogaster				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	0:29	5:23	318	99.46	99.70	1:35	18:15	355	98.68	99.94	13:48	164:45	628	96.73	99.99
	BWA	1:09	6:41	440	98.49	91.83	2:40	18:03	935	93.95	84.57	5:38	36:44	1055	74.96	68.09
	Soap 2	0:21	1:40	1074	65.88	61.20	0:38	3:40	1554	56.07	50.99	0:54	4:33	1546	41.21	38.14
	R3-100	0:46	6:32	1529	100.00	92.87	0:56	7:05	2674	100.00	89.47	1:17	8:10	4963	100.00	90.43
	R3-99	0:46	6:59	1529	100.00	92.87	0:56	6:46	2674	100.00	89.47	1:17	7:52	4963	100.00	90.43
	R3-95	0:46	6:34	1529	99.99	92.86	0:55	6:51	2674	100.00	89.47	1:15	7:52	4963	100.00	90.43
all-mappers	mrFAST	1:11	10:29	6909	99.24	92.17	2:17	20:54	7972	94.15	85.02	5:16	49:22	9115	65.25	69.32
	SHRiMP 2	7:25	64:30	3758	99.49	99.96	42:53	484:02	3814	98.44	99.85	796:06	9442:21	4023	95.70	99.31
	R3-100	0:49	6:55	1529	100.00	92.87	0:58	6:57	2674	100.00	89.47	1:20	8:48	4963	100.00	90.43
	R3-99	0:48	6:19	1529	100.00	92.87	0:58	7:24	2674	100.00	89.47	1:20	8:38	4963	100.00	90.43
	R3-95	0:48	6:43	1529	99.99	92.86	0:55	6:42	2674	100.00	89.47	1:20	8:35	4963	100.00	90.43

	dataset	simulated, $m = 200$ H. sapiens					simulated, $m = 400$ H. sapiens					simulated, $m = 800$ H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	1:19	15:13	3372	99.02	99.57	4:25	52:06	3413	98.56	99.94	63:15	756:12	3686	96.83	100.00
	BWA	4:53	44:30	6619	98.44	91.82	20:21	185:24	11043	93.64	84.33	26:00	257:32	10946	74.38	67.60
	Soap 2	0:49	5:08	5651	65.82	61.15	1:37	12:30	5938	56.22	51.09	2:05	15:04	6122	41.51	38.22
	R3-100	26:34	299:17	3341	100.00	92.88	44:21	511:05	3532	100.00	89.53	102:48	1205:31	4964	100.00	90.49
	R3-99	16:17	179:00	3341	100.00	92.88	20:31	226:47	3532	100.00	89.53	38:50	442:12	4964	100.00	90.49
	R3-95	13:55	153:13	3341	99.99	92.87	20:31	227:05	3532	100.00	89.53	37:21	424:50	4964	100.00	90.49
all-mappers	mrFAST	103:02	1024:12	7395	99.25	92.18	321:50	4143:26	7926	65.33	59.38	1116:03	13750:58	9601	44.19	47.24
	SHRiMP 2	546:29	5323:53	38570	99.32	99.95	1705:04	92111:02	45100	98.10	99.75	–	–	–	–	–
	R3-100	28:30	323:07	3341	100.00	92.88	48:52	564:57	3532	100.00	89.53	105:48	1243:35	4964	100.00	90.49
	R3-99	17:22	191:32	3341	100.00	92.88	21:46	241:16	3532	100.00	89.53	41:41	477:03	4964	100.00	90.49
	R3-95	14:40	160:48	3341	99.99	92.87	21:13	236:06	3532	100.00	89.53	39:52	455:24	4964	100.00	90.49

Table E.5: Extended experimental results for long simulated single-end data, extending Table 4.5a. The results are shown for the first 10 M × 100 bp single-end reads of Illumina datasets. Hobbes is not capable of mapping long reads and thus not shown here. Some mrFAST processes crashed for the D. melanogaster and H. sapiens $m = 400$ and $m = 800$ datasets which explains the low number of mapped reads. SHRiMP 2 was not able to map the 800 bp human dataset within 96 hours.

	dataset	ERR022075 E. coli					SRR065390 C. elegans				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	4:09	49:15	195	99.69 100.00 99.94 99.51 97.97 95.89	98.69 18.92 78.51 88.74 91.84 93.46	6:19	74:50	322	97.30 99.98 99.35 96.91 93.85 87.33	84.59 13.20 16.42 17.74 18.64 19.33
	BWA	7:35	30:58	343	99.66 100.00 99.94 99.48 97.54 95.39	95.87 18.92 78.51 88.74 91.82 93.44	12:32	75:47	479	92.01 99.97 95.50 82.94 67.12 56.05	81.20 13.56 16.88 18.21 19.00 19.56
	Soap 2	3:51	17:29	743	96.58 100.00 99.81 98.87 68.19 38.61	94.70 18.93 78.48 89.09 91.58 92.37	5:56	42:02	833	89.51 100.00 94.23 88.19 59.50 29.87	24.48 13.25 16.34 17.70 18.34 18.73
	R3-100	1:53	7:33	11113	100.00 100.00 100.00 100.00 100.00 100.00	94.66 18.92 78.55 88.79 91.91 93.54	6:29	64:44	11230	100.00 100.00 100.00 100.00 100.00 100.00	20.38 13.20 16.43 17.78 18.70 19.43
	R3-99	1:45	6:08	11113	99.99 100.00 100.00 100.00 99.98 99.79	94.66 18.92 78.55 88.79 91.91 93.54	6:20	62:39	11230	99.97 100.00 100.00 99.99 99.98 99.85	20.38 13.20 16.43 17.78 18.70 19.43
	R3-95	1:44	6:22	11113	99.96 100.00 100.00 100.00 99.84 98.96	94.62 18.92 78.55 88.79 91.90 93.52	6:07	60:22	11230	99.81 100.00 100.00 99.99 99.83 99.21	20.35 13.20 16.43 17.78 18.70 19.43
all-mappers	Hobbes	5:21	39:44	1196	90.57 90.81 90.62 90.19 89.95 90.21	86.15 17.18 71.25 80.61 83.50 85.05	10:27	103:38	2852	86.49 92.91 85.94 72.90 65.00 62.82	18.66 12.26 15.09 16.14 16.82 17.41
	mrFAST	1:21	10:22	52868	99.99 99.99 99.99 99.99 99.99 99.99	95.31 18.92 78.55 88.78 91.90 93.54	14:02	134:13	54510	99.29 99.75 99.48 98.82 98.22 97.68	79.37 13.18 16.41 17.75 18.66 19.40
	SHRiMP 2	8:18	97:16	1612	99.85 100.00 99.99 99.80 99.06 97.80	97.74 18.92 78.54 88.76 91.85 93.46	105:30	1238:32	3468	94.81 99.77 96.89 88.75 80.41 73.48	53.92 13.32 16.58 17.88 18.72 19.36
	R3-100	1:53	7:34	11113	100.00 100.00 100.00 100.00 100.00 100.00	94.66 18.92 78.55 88.79 91.91 93.54	7:49	78:35	11816	100.00 100.00 100.00 100.00 100.00 100.00	20.38 13.20 16.43 17.78 18.70 19.43
	R3-99	1:48	6:40	11113	99.99 100.00 100.00 100.00 99.98 99.79	94.66 18.92 78.55 88.79 91.91 93.54	6:37	64:39	11588	99.97 100.00 100.00 99.99 99.98 99.85	20.38 13.20 16.43 17.78 18.70 19.43
	R3-95	1:45	6:13	11113	99.96 100.00 100.00 100.00 99.84 98.96	94.62 18.92 78.55 88.79 91.90 93.52	6:33	63:35	11230	99.81 100.00 100.00 99.99 99.83 99.21	20.35 13.20 16.43 17.78 18.70 19.43

	dataset	SRR497711 D. melanogaster					ERR012100 H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	6:32	77:40	349	98.94 100.00 99.03 98.43 97.52 96.11	81.94 32.50 50.89 60.48 66.19 69.88	10:51	128:58	3555	99.51 99.97 99.90 99.46 98.34 96.40	94.19 15.04 62.97 77.57 82.66 85.16
	BWA	13:33	77:14	503	97.47 100.00 98.85 97.77 93.52 87.15	73.41 32.51 50.87 60.41 65.93 69.30	34:35	241:41	4662	98.84 99.99 99.88 98.95 95.16 90.79	88.06 15.04 62.97 77.50 82.47 84.86
	Soap 2	5:29	38:22	940	88.67 100.00 94.93 89.40 70.79 41.11	72.77 32.58 50.33 59.65 64.13 65.93	8:24	61:32	5463	91.58 99.99 98.89 93.72 51.38 26.08	87.47 15.07 62.57 77.33 80.49 81.46
	R3-100	9:01	93:39	11199	100.00 100.00 100.00 100.00 100.00 100.00	72.95 32.50 51.07 60.63 66.35 70.04	176:29	2077:35	18568	100.00 100.00 100.00 100.00 100.00 100.00	86.93 15.04 63.02 77.65 82.76 85.27
	R3-99	7:00	69:26	11199	99.97 100.00 100.00 100.00 99.97 99.80	72.93 32.50 51.07 60.63 66.34 70.04	159:03	1872:33	16568	99.98 100.00 100.00 100.00 99.98 99.75	86.91 15.04 63.02 77.65 82.76 85.26
	R3-95	6:56	68:48	11199	99.78 100.00 100.00 100.00 99.68 98.65	72.80 32.50 51.07 60.63 66.33 69.98	135:44	1599:10	13678	99.89 100.00 100.00 100.00 99.81 98.78	86.84 15.04 63.02 77.65 82.75 85.23
all-mappers	Hobbes	8:43	75:57	5870	84.78 84.27 85.76 86.51 85.66 83.24	62.48 27.39 43.41 51.81 56.81 59.99	89:35	884:05	47270	95.11 95.68 95.83 94.73 92.92 90.73	84.05 14.39 60.42 74.46 79.44 81.95
	mrFAST	8:26	81:25	48032	100.00 100.00 100.00 99.99 99.99 99.99	73.16 32.50 51.07 60.63 66.35 70.04	779:12	7649:19	57625	99.94 99.98 99.97 99.93 99.85 99.76	87.79 15.04 63.01 77.64 82.75 85.26
	SHRiMP 2	47:07	537:29	3838	99.67 100.00 99.99 99.84 99.21 97.77	87.36 32.50 51.07 60.62 66.30 69.95	2762:32	31710:26	38594	99.74 99.91 99.90 99.81 99.40 98.40	97.51 15.03 62.96 77.57 82.66 85.15
	R3-100	7:59	76:28	13558	100.00 100.00 100.00 100.00 100.00 100.00	72.95 32.50 51.07 60.63 66.35 70.04	184:27	2167:14	27623	100.00 100.00 100.00 100.00 100.00 100.00	86.93 15.04 63.02 77.65 82.76 85.27
	R3-99	7:44	73:54	13485	99.97 100.00 100.00 100.00 99.97 99.80	72.93 32.50 51.07 60.63 66.34 70.04	177:56	2100:43	25866	99.98 100.00 100.00 100.00 99.98 99.75	86.91 15.04 63.02 77.65 82.76 85.26
	R3-95	7:36	72:22	13241	99.78 100.00 100.00 100.00 99.68 98.65	72.80 32.50 51.07 60.63 66.33 69.98	166:22	1956:20	23026	99.89 100.00 100.00 100.00 99.81 98.78	86.84 15.04 63.02 77.65 82.75 85.23

Table E.6: Extended experimental results for real-world paired-end data, extending Table 4.5b. The results are shown for the first 10 M × 2 × 100 bp pairs of Illumina datasets.

	dataset	simulated, $m = 200$ D. melanogaster					simulated, $m = 400$ D. melanogaster					simulated, $m = 800$ D. melanogaster				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	1:25	16:37	390	98.91	99.24	5:45	68:17	515	97.37	99.62	39:07	468:30	1418	93.64	99.70
	BWA	2:13	12:41	674	96.98	84.30	6:13	38:07	975	88.31	71.56	11:26	73:40	1569	56.28	46.44
	Soap 2	1:55	16:35	1092	74.85	71.96	4:40	44:16	1377	50.03	49.55	12:36	124:23	1561	23.55	28.23
	R3-100	1:26	12:21	3075	100.00	74.91	1:45	13:44	5364	100.00	69.68	2:22	16:05	9942	100.00	71.16
	R3-99	1:25	12:44	3075	99.99	74.91	1:48	14:30	5364	100.00	69.68	2:19	15:24	9942	100.00	71.16
	R3-95	1:23	12:42	3075	99.98	74.89	1:41	13:31	5364	100.00	69.68	2:19	15:27	9942	100.00	71.16
all-mappers	mrFAST	2:16	20:37	11104	98.46	84.68	4:51	45:52	12493	88.68	72.13	10:47	102:53	15392	44.19	49.69
	SHRiMP 2	13:10	133:22	3746	98.99	99.97	83:57	979:35	3945	96.87	99.83	1617:26	19264:14	4211	91.64	98.62
	R3-100	1:33	14:02	3075	100.00	74.91	1:54	15:36	5364	100.00	69.68	2:30	15:55	9942	100.00	71.16
	R3-99	1:30	13:23	3075	99.99	74.91	1:53	14:53	5364	100.00	69.68	2:30	15:45	9942	100.00	71.16
	R3-95	1:26	12:32	3075	99.98	74.89	1:51	15:15	5364	100.00	69.68	2:29	15:47	9942	100.00	71.16

	dataset	simulated, $m = 200$ H. sapiens					simulated, $m = 400$ H. sapiens					simulated, $m = 800$ H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	2:05	24:14	3439	98.75	99.29	7:43	90:09	3544	97.34	99.81	71:12	850:50	4380	93.84	99.90
	BWA	10:39	90:33	6790	96.88	84.24	38:15	378:52	11137	87.72	71.10	62:08	568:37	10955	55.46	45.78
	Soap 2	2:05	16:55	5667	74.84	71.89	4:46	43:57	6385	50.27	49.64	11:16	111:05	7385	23.83	28.27
	R3-100	28:39	327:39	3759	100.00	74.91	34:07	390:25	5366	100.00	69.68	47:09	539:22	9944	100.00	71.11
	R3-99	25:03	285:08	3759	99.99	74.91	27:30	312:19	5366	100.00	69.68	38:57	443:16	9944	100.00	71.11
	R3-95	24:14	275:23	3759	99.98	74.90	28:03	319:08	5366	100.00	69.68	32:30	366:04	9944	100.00	71.11
all-mappers	mrFAST	217:29	2160:07	14319	98.44	84.77	328:45	1991:56	13630	22.23	18.10	1043:59	9427:14	18019	21.01	23.71
	SHRiMP 2	996:11	10678:56	39094	98.82	99.98	3243:49	187639:34	48973	96.64	99.86	–	–	–	–	–
	R3-100	29:59	343:42	3759	100.00	74.91	33:11	378:59	5366	100.00	69.68	43:27	495:53	9944	100.00	71.11
	R3-99	26:31	302:45	3759	99.99	74.91	26:40	299:26	5366	100.00	69.68	32:34	365:36	9944	100.00	71.11
	R3-95	25:21	288:57	3759	99.98	74.90	26:34	300:32	5366	100.00	69.68	32:27	366:30	9944	100.00	71.11

Table E.7: Extended experimental results for long simulated paired-end data, extending Table 4.5b. The results are shown for 1 M paired-end reads of the given lengths simulated with Mason using the default Illumina error model. Some mrFAST processes crashed for the D. melanogaster $m = 400$ and H. sapiens $m = 400$ and $m = 800$ datasets which explains the low number of mapped reads. SHRiMP 2 was not able to map the 800 bp human dataset within 96 hours.

Appendix F

Full Error Correction Tables

This appendix contains further tables for the read error correction evaluation presented in Section 3.4.

organism	accession	genome length
B. pertussis 18323	NC_018518.1	4 043 846
C. elegans	ENSEMBL release 60	100 286 070
D. melanogaster	flybase r5.29	120 381 546
E. coli K-21	NC_000913.2	4 639 675
E. coli O104:H4	NC_018658.1	5 273 097
H. sapiens	GRCh37	2 861 343 787
P. falciparum 3D7	ASM276v1	23 264 338
P. syringae	NC_007005.1	6 093 698
S. aureus LGA251	NC_017348, NC_017349	2 799 725
S. cerevisae	NCBI release 54	12 156 676

Table F.1: Identifiers and sources of the reference sequences used in the evaluation and their lengths.

organism	accession	avg. length	read count	coverage	Gbp
B. pertussis	ERR161541[2]	142 bp	2 464 690	85x	0.3
C. elegans	SRR443373[I]	100 bp	29 657 035	30x	3
D. melanogaster	SRR018294[I]	75 bp	9 463 720	5x	0.7
D. melanogaster	SRR492060[I]	76 bp	51 727 822	28x	3.4
D. melanogaster	SRX016210[1]	544 bp	4 692 486	18x	2.2
E. coli K-12	ERR022075[I]	100 bp	22 720 100	490x	2.3
E. coli K-12	ERR022075[I]	100 bp	1 378 122	30x	0.14
E. coli K-12	SRR000868[1]	253 bp	230 517	13x	0.06
E. coli K-12	ERR039477[2]	92 bp	390 976	8x	0.04
E. coli K-12	SRR611140[2]	162 bp	4 669 065	163x	0.8
E. coli K-12	SRR620425[2]	170 bp	4 237 734	156x	0.7
E. coli O104:H4	SRR254209[2]	178 bp	977 971	32x	0.2
H. sapiens	SRR1238539[2]	177 bp	186 132 134	11x	31.5
P. falciparum	ERR161543[2]	154 bp	1 959 564	13x	0.3
P. syringae	ERR005143[I]	36 bp	14 204 532	42x	0.26
S. aureus	ERR236069[2]	228 bp	1 338 465	109x	0.31
S. aureus	SRR070596[1]	514 bp	185 384	34x	0.1
S. cerevisae	SRR031259[I]	36 bp	7 485 708	22x	0.27
S. cerevisae	SRX039441[1]	274 bp	690 237	16x	0.19

Table F.2: Information on the read sets used in the evaluation.
[1] 454. [2] Ion Torrent,[I] Illumina.

	ALG		Blue		Coral		Echo		Fiona		Hitec		Quake		Racer	
data set	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
C. el. (30x)	50.2	13.3	35.0	**5.7**	129.3	49.3	—		218.2	21.8	—		**13.9**	9.8	1351.4	10.2
D. mel. (30x)	68.2	15.5	**20.2**	6.1	143.5	53.5	—		113.6	33.9	—		24.8	**2.7**	31.5	8.1
D. mel. (5x)	8.2	**3.2**	11.7	4.7	17.4	17.5	279.1	11.3	28.5	6.2	106.2	13.7	21.1	10.6	**6.7**	6.6
E. col. (30x)	3.0	0.7	**0.5**	**0.2**	3.5	3.5	60.9	3.3	4.1	1.7	17.6	2.6	0.7	**0.2**	0.8	**0.3**
E. col. (490x)	41.2	9.1	8.4	**0.7**	195.2	24.9	—		40.1	17.2	351.0	9.9	**7.2**	**0.7**	11.7	1.5
P. syr. (21x)	2.0	0.6	1.2	0.4	1.5	3.7	50.0	5.6	3.3	1.6	11.5	2.5	3.8	**0.2**	**0.6**	0.4
S. cer. (22x)	3.5	1.1	2.8	0.7	8.0	5.2	87.9	10.0	4.4	3.1	25.0	5.0	20.8	**0.4**	**1.1**	0.8

Table F.3: Running time (in minutes and fractions thereof) and memory consumption (in GB, rounded to the next GB) on Illumina data.
[1] The program crashed. [2] The program ran too long. The problems were: Coral and HiTEC produced a segfault, requiring more than 72 GB of memory. ECHO was killed after running more than 4 days in the case of *C. elegans* and *D. melanogaster* data sets and the subprogram `NeighborJoin` crashed on the full *E. coli* data set.

	A.-LG		Blue		Coral		Coral*		Fiona		HybridShrec		HybridShrec-F	
data set	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
D. mel. (18x)	145.0	10.7	**61.6**	**6.1**	496.1	59.1	1414.1	59.6	242.2	16.9	333.2	41.4	499.5	41.5
E. col. (13x)	2.6	**0.3**	**0.5**	**0.2**	3.4	2.9	4.4	2.9	2.7	1.0	7.1	4.8	9.2	8.1
S. aur. (34x)	3.0	**0.6**	**2.2**	0.8	5.5	4.7	112.2	4.8	10.1	1.3	12.0	14.2	13.6	14.8
S. cer. (16x)	6.5	**1.0**	**3.9**	1.3	7.1	5.1	19.6	5.1	13.2	1.7	22.5	14.6	30.5	14.9
B. pert. (85x)	**6.0**	1.7	6.3	**1.0**	13.5	9.2	81.2	9.4	26.1	3.0	58.3	16.7	54.0	20.8
E. col. (8x)	1.0	**0.2**	**0.5**	0.5	0.8	2.6	0.9	2.6	2.2	1.0	4.8	5.0	5.0	11.8
E. col. (153x)	15.0	7.2	**10.0**	**1.1**	249.1	12.0	290.1	11.9	50.5	7.8	111.4	17.5	111.0	6.0
E. col. (160x)	14.2	3.5	**12.0**	**1.1**	243.0	13.2	373.8	13.2	70.5	9.1	111.3	18.8	160.2	6.1
H. sap. (11x)[1]	**565.3**	126.9	1056.1	**94.7**	—[2]		—[2]		1118.5	349.7	—[2]		—[2]	
P. falc. (12x)	**5.6**	**1.4**	7.2	2.0	11.0	11.0	24.8	11.4	14.5	3.0	38.9	16.0	49.7	20.3
S. aur. (109x)	**4.4**	**1.4**	9.7	**1.3**	12.0	12.9	175.8	13.3	33.7	2.7	51.1	17.5	53.8	29.1

Table F.4: Running time (in minutes and fractions thereof) and memory consumption (in GB, rounded to the next GB) on 454 (top) and IonTorrent (bottom) data.

[1] The programs were run on machine with 16 physical and 32 virtual cores and 370 GB of RAM. [2] Out of memory.

data set		original e-rate	Coral -e 0.07 e-rate	gain	Coral -e0.10 e-rate	gain	Coral -e0.15 e-rate	gain	Coral -e0.20 e-rate	gain	Coral -e0.25 e-rate	gain
B. pertussis	85x	3.71	3.71	0.02	3.71	0.06	3.69	0.40	3.55	4.18	**2.57**	**30.60**
D. melanogaster	18x	1.17	0.72	38.81	0.58	50.68	**0.55**	**53.30**	–[1]	–[1]	–[1]	–[1]
E. coli K-12	13x	1.06	0.54	49.42	0.43	59.25	**0.38**	**63.79**	0.39	63.28	0.41	61.46
E. coli K-12	8x	0.62	0.33	46.86	**0.30**	**51.86**	0.32	48.07	0.36	42.47	0.40	35.72
E. coli K-12	162x	1.46	0.59	59.70	0.41	71.62	**0.38**	**73.72**	0.39	73.11	0.42	71.37
E. coli K-12	156x	1.11	0.43	61.07	0.32	71.37	**0.28**	**74.70**	0.29	73.60	0.32	70.59
E. coli O104:H4	32x	5.19	5.19	0.00	5.19	0.03	5.15	0.66	4.80	7.44	**3.44**	**33.82**
P. falciparum 3D7	13x	5.06	5.05	0.03	5.05	0.17	4.97	1.60	4.57	9.65	**3.80**	**24.94**
S. aureus	109x	3.32	3.32	0.24	3.29	1.08	3.03	8.90	1.92	42.38	**1.44**	**56.91**
S. aureus	34x	1.76	1.76	0.00	1.73	2.09	1.51	14.23	1.00	43.51	**0.44**	**74.90**
S. cerevisae	16x	0.95	0.95	0.56	0.94	1.06	0.93	2.41	**0.92**	**2.99**	1.01	-5.84

Table F.5: Complete list achieved for different parameterizations of Coral. The values selected for Coral* are highlighted in bold.
[1] Coral did not finish within 24 h.

data set		original e-rate	HS* -s 2 e-rate	gain	HS^F -s 2 e-rate	gain	HS* -s 3 e-rate	gain	HS^F -s 3 e-rate	gain	HS* -s 4 e-rate	gain	HS^F -s 4 e-rate	gain	HS* -s 5 e-rate	gain	HS^F -s 5 e-rate	gain	HS* -s 6 e-rate	gain	HS^F -s 6 e-rate	gain	HS* -s 7 e-rate	gain	HS^F -s 7 e-rate	gain
B. pertussis	85x	3.71	_[1]	_[1]	4.07	-9.68	_[1]	_[1]	4.11	-10.89	_[1]	_[1]	4.15	-11.83	_[1]	_[1]	4.17	-12.61	_[1]	_[1]	4.20	-13.22	**12.44**	**-235.48**	4.18	-12.78
D. melanogaster	18x	1.17	13.89	-1,086.21	1.15	1.89	14.17	-1,109.80	1.08	7.94	10.72	-813.56	_[1]	_[1]	**4.46**	**-279.51**	**0.73**	**38.17**	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]
E. coli K-12	8x	1.06	1.20	-12.98	0.81	23.52	**0.64**	**40.05**	**0.70**	**34.28**	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]
E. coli K-12	13x	0.62	1.20	-93.69	0.55	10.50	0.97	-56.04	0.46	25.55	**0.36**	**41.81**	**0.37**	**40.26**	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]
E. coli K-12	163x	1.46	**1.73**	**-18.90**	**1.46**	**0.00**	1.81	-23.85	1.46	0.00	1.86	-27.80	1.46	0.00	1.90	-30.45	1.46	0.00	1.94	-33.01	1.46	0.00	2.07	-41.73	1.46	0.00
E. coli K-12	156x	1.11	**1.38**	**-24.15**	**1.11**	**0.00**	1.45	-30.62	1.11	0.00	1.50	-35.16	1.11	0.00	1.53	-37.76	1.11	0.00	1.56	-40.71	1.11	0.00	1.68	-51.48	1.11	0.00
E. coli O104:H4	32x	5.19	5.40	-3.98	5.16	0.51	5.44	-4.82	5.16	0.61	5.56	-7.11	5.15	0.77	5.79	-11.56	5.12	1.35	6.08	-17.35	4.83	6.87	**4.39**	**15.36**	**4.31**	**16.76**
P. falciparum 3D7	13x	5.06	13.27	-162.26	5.14	-1.76	11.38	-124.62	4.87	3.54	**7.67**	**-51.29**	**4.63**	**8.50**	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]
S. aureus	109x	3.32	**3.77**	**-13.44**	**3.31**	**0.32**	**3.89**	-17.02	3.32	0.04	4.04	-21.57	3.33	-0.33	4.26	-28.01	3.35	-0.88	4.55	-36.89	3.38	-1.64	5.00	-50.38	3.41	-2.79
S. aureus	34x	1.76	2.26	-28.03	1.68	4.60	2.28	-29.71	1.70	3.77	2.64	-49.79	1.70	3.35	3.11	-76.57	1.72	2.51	2.50	-41.84	1.53	13.39	**1.59**	**9.62**	**1.40**	**20.50**
S. cerevisae	16x	0.95	2.57	-169.97	0.92	3.67	3.17	-233.71	0.88	7.09	2.06	-116.27	0.76	19.85	**0.90**	**5.48**	**0.73**	**23.11**	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]	_[1]

Table F.6: Complete list achieved for different parameterizations of HybridShrec. The values selected for HybridShrec* and HybridShrec^F are highlighted in bold.

[1] Coral did not finish within 24 h.

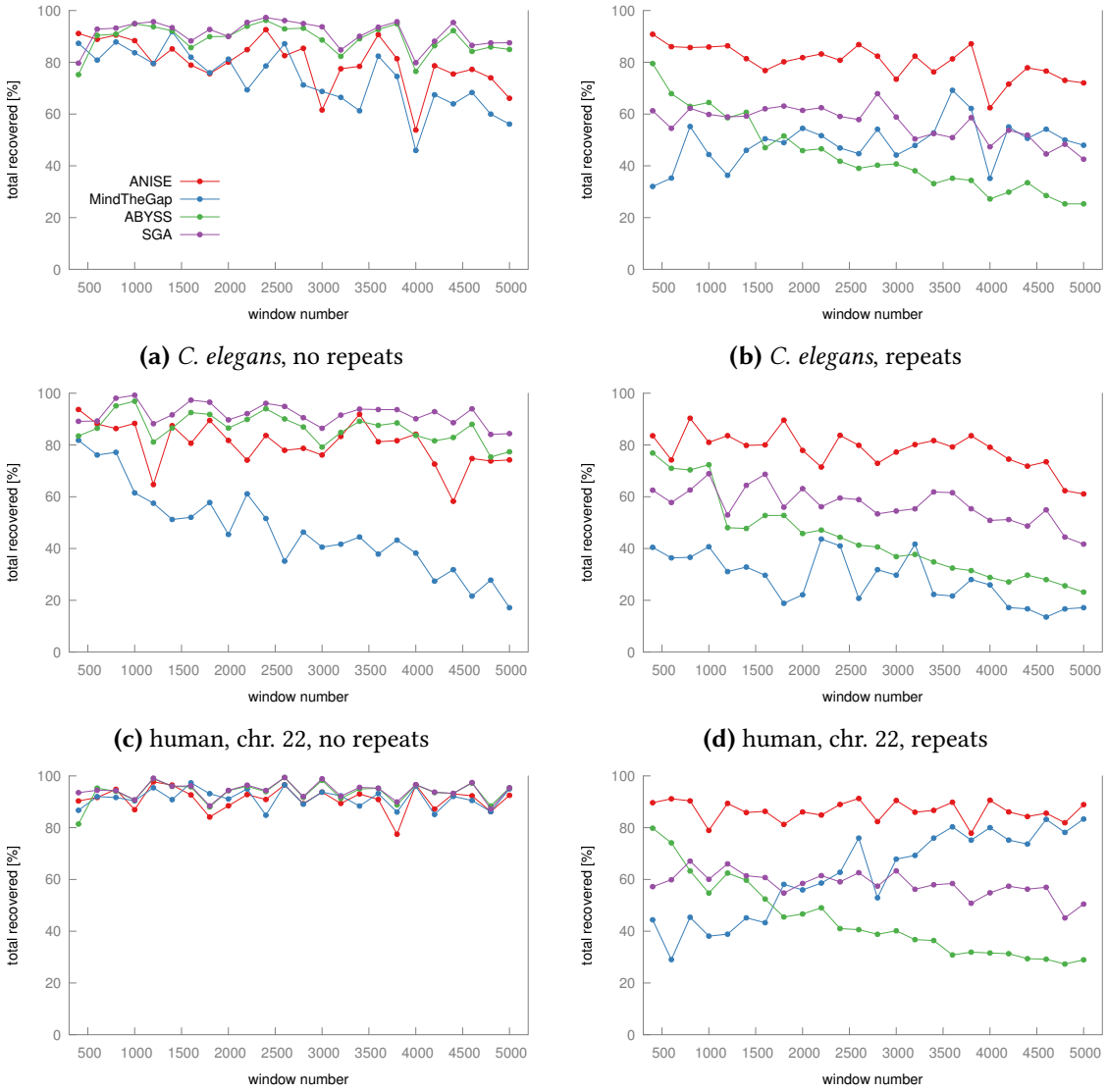
Appendix G

Extended Variant Calling Results

		ABYSS				SGA			
repeats	data set	ins. Mbp	rec. Mbp	r.-rate	e.-rate	ins. Mbp	rec. Mbp	r.-rate	e.-rate
no	*C. elegans*	2.44	2.20	90.4	0.01	2.44	2.25	92.5	0.01
	chr. 22	2.45	2.14	87.5	0.01	2.44	2.29	93.8	0.01
	D. melanogaster	2.51	2.42	96.4	0.01	2.49	2.43	97.6	0.01
yes	*C. elegans*	7.34	2.75	37.5	0.38	7.15	4.01	56.0	0.49
	chr. 22	7.28	2.72	37.3	0.38	7.12	4.11	57.8	0.49
	D. melanogaster	7.50	2.84	37.9	0.39	7.35	4.37	59.4	0.50

		Anise				MindTheGap			
repeats	data set	ins. Mbp	rec. Mbp	r.-rate	e.-rate	ins. Mbp	rec. Mbp	r.-rate	e.-rate
no	*C. elegans*	2.04	1.91	93.6	0.02	1.76	1.71	97.1	0.03
	chr. 22	2.06	1.96	95.4	0.02	1.01	0.98	96.6	0.03
	D. melanogaster	2.36	2.33	99.0	0.01	2.36	2.34	99.1	0.01
yes	*C. elegans*	6.26	5.80	92.6	0.25	4.18	3.76	89.9	0.65
	chr. 22	5.97	5.72	95.8	0.26	2.14	1.86	86.8	0.65
	D. melanogaster	6.84	6.65	97.2	0.25	5.91	5.38	91.1	0.66

Table G.1: Insert assembly results on simulated insertions into real biological sequence for homozyous insertions. The table shows the number of inserted Mbp (*ins. Mbp*), recovered Mbp (*rec. Mbp*), recovery rate *r.-rate* (in %), and error rate *e-rate* (in %) of the assembled sequence. The values for *inserted* and *r.-rate* are given for the sites that the methods attempted to assemble. The table shows results for the homozygous data sets without (first three rows) and with (last three rows) repeated insertions. The best values for recovered sequence and error rate in each row are underlined.



(a) *C. elegans*, no repeats

(b) *C. elegans*, repeats

(c) human, chr. 22, no repeats

(d) human, chr. 22, repeats

(e) *D. melanogaster*, no repeats

(f) *D. melanogaster*, repeats

Figure G.1: Percentage of recovered sequence data for the homozygous data sets with and without repeats.

Appendix H

BLAT Identities and Scores

While the BLAT (Kent, 2002) executable itself does not yield an identity or score in its output files, the web front-end does. However, the BLAT FAQ[1] explains how to obtain these identities and scores. The computation is reproduced here to make the description of the evaluation more self-contained and precise.

Algorithm H.1 shows the computation of BLAT identity, adapted from the BLAT website. For this, the algorithm computes the BLAT error rate, the number of errors per hundred nucleotides. BLAT computes the following values for each of its match. The begin and end position in the query and reference sequence, the number of matches of bases marked as non-repeats and the number of matches marked as repeats, the number of gaps in each sequence, and the number of bases in gaps in each sequence. The identity is computed from these values, giving gaps a lower

[1] http://genome.ucsc.edu/FAQ/FAQblat.html#blat4

weight than in edit distance identity or even BLAST scores.

Algorithm H.1: BLAT Identity Computation

```
/* Get length of the match and return 100% in case of the match being
   empty.                                                              */
qAliSize ← qEnd - qStart;                          /* Length of match in query.  */
tAliSize ← tEnd - tStart;                        /* Length of match in reference.  */
aliSize ← min(qAliSize, tAliSize);
if aliSize = 0 then
    return 100;
end
sizeDiff ← max(0, qAliSize - tAliSize);
insertFactor ← qNumInsert;              /* number of query bases aligning to gaps */
total = (numMatches + numRepeatMatches + numMismatches);
if total = 0 then
    return 100;
else
    errorRate ← (numMismatches + insertFactor + round(3 * log(1 + sizedDif)));
    errorRate ← 100 · errorRate;
    errorRate ← errorRate / total;
    return 100 - errorRate;
end
```

The BLAT score computation is shown in Algorithm H.2, again adapted from the BLAT website. Basically, the score is the number of matches minus half the number of mismatches and gaps in either sequence (but not number of bases aligned against gaps).

Algorithm H.2: BLAT Score Computation

```
score = (numMatches + numRepeatMatches / 2);
score = score - numMismatches - qNumInsertions - tNumInsertions;
return score;
```

Appendix I

Cluster Linking Algorithms

This appendix describes the ideas and algorithm behind the copy separation in Anise in more detail. The algorithms presented here are based on the algorithms by Kuchenbecker (2011), and the corresponding algorithm from Kuchenbecker are given as appropriate. The algorithm listings here are only a small improvement to the original work by Kuchenbecker in that I removed some small bugs. Further, they are not presented using Kuchenbecker's set-based notation but a representation that is closer to my reimplementation and I find them easier to understand in this form.

Introduction

When two or more different regions from the genome are very similar, reads sampled from these partitions will be assembled into the same contig. However, one would expect small differences in these two regions, *i.e.*, SNVs or small indels.

Such differences in the MSA of the assembled reads can be detected using Tammi's (Tammi *et al.*, 2002) test. The test considers pairs of columns that share reads and have characters deviating from the column's consensus. The number of coinciding deviations (*i.e.*, occuring in the same read) from the consensus are counted and then checked for statistical significance. A result of this approach is that singular separating columns cannot be used for separating copies.

When reads span more than one of these *local partitions*, the idea is to try to combine this information into larger clusters of the reads as outlined in Section 5.4.3.

The numbers $0, \ldots, k-1$ are assigned to the found k separating columns and also assign numeric ids to each read. Let $\text{char}(i, c)$ be the character of read i in column c.

One can now create partitions of the reads with id i overlapping a column c with the character of $\text{char}(i, c)$. Reads having a character `N` in c are ignored. In the example, in column 0, the reads

$0, 1$, and 2 have the same value as do have 3 and 4 and it is known that the value in this column for each from the first set is different from the second set. The first set *conflicts* with the second. Similarly, in column 1, the partition entry $\{0, 1, 2\}$ conflicts with $\{3, 4, 5, 6\}$. Thus, this would yield the partition $\{\{0, 1, 2\}, \{3, 4\}\}$ for column 0 and $\{\{0, 1, 2\}, \{3, 4, 5, 6\}\}$ for column 1.

Further, the fact that some reads span more than one column is used to get to a separation of the reads in larger regions. For example, given the information from the example, two read clusters $\{0, 1, 2\}$ and $\{3, 4, 5, 6\}$ can be obtained, each containing reads from one copy.

Note that the number of copies is not known in advance and might be larger than two. It is known that a similar problem, the haplotyping of biallelic (similar to two copies in this case) from paired reads is $\mathcal{NP}$-hard (Bonizzoni *et al.*, 2003). Thus, the method presented here does not attempt to optimally solve this problem but uses a rather simple heuristic. Figure 5.15 in Section 5.4.3 of the main text gives another example that includes mate pairs.

Algorithm Overview

The algorithm works in an iterative fashion. For each contig in the assembly, the algorithm processes the columns from left to right, the contigs are processed in arbitrary order.

The algorithm maintains two data structures: first, a set of *global clusters* G and second a set of conflicts C between two global clusters. Each entry of G is a pair (i, S) with a unique integer identifier i and a set of read ids S with the reads in this cluster. C simply stores unordered pairs $\{i, j\}$ of global clusters that were previously determined to be conflicting. Both are easily implemented using binary search trees which also allows access to the members of G by cluster id and membership queries of S from two read ids in logarithmic time. For a given global cluster id i, the corresponding set S read can be retrieved with $G[i]$. Note that a read might be a member of multiple global clusters.

A top level view of the algorithm is shown in Algorithm I.1. The subroutines used are explained below in this section. For the initial local partition, multi-class-assignment-removal(p) and refine-global-partition(p) do not do anything, thus global-class-assignment(p) is described first.

Global Class Assignment

Algorithm I.2 (p. 221) shows the code for the global class assignment. The input is a local read partition, *i.e.*, a partition of the ids of the reads that span a local feature such as a separating column. The partition is given as a list of sets with read ids, thus each local partition entry has an index. First, for each entry e of the local partition, the algorithm determines the ids of global clusters that share a read with e. If none such cluster can be found, a new global cluster is created. In the case that more than one such global cluster could be found, global classes are merged using the routine global-class-merging as described below. After this merging, each read from e that is not in any global cluster yet is assigned to all global clusters previously identified to overlap with e. Finally, local conflicts from p are projected to the global conflict set C.

Algorithm I.1: Cluster Linking Overview

```
G ← ∅ ;                                              /* Initialize G and C */
C ← ∅;
nextID ← 0 ;                                         /* next id to use for G */
for local partitions p do
    /* globalize local partitions                                            */
    multi-class-assignment-removal(p);
    global-partition-refinement(p);
    global-class-assignment(p);
end
Remove duplicated classes ;                                          /* Finalize */
```

Note that in the description above, the test for "sharing reads" is actually implemented as a test for a minimal overlap in read ids with a threshold of 2 (a user-configurable parameter).

The routine global-class-merging() shown in Algorithm I.3 (p. 222) takes as the input the set of global cluster ids identified in Algorithm I.2. First, the routine identifies a subset of non-conflicting clusters of the set ids and also computes its complement. If this set of non-conflicting clusters is not empty then a new cluster is created with the non-conflicting ids. The new cluster inherits all conflicts of all merged clusters and the conflicts between all merged clusters are removed.

The global class assignment algorithm and all other algorithms in this section describe the processing of single-end reads only. The extension to handling read pairs is straightforward and implemented as follows in Anise. In each case where a read id is added to or removed from a set, the id of its mate is also added to/removed from the set. The same is true for the conflict set.

Multi-Cluster Assignment Removal

The routine multi-cluster-assignment-removal() is first called in the cluster linking algorithm's main loop shown in Algorithm I.1 (p. 219). The aim of this routine is to resolve assignments of a read to multiple global clusters. Such assignments occur when a read is locally linked to two or more reads that are assigned to different global clusters from a previously processed sites. Algorithm I.4 (p. 223) shows this routine.

For each entry e in the local partition p, the algorithm determines the unique aligned reads from e and store them in u. The algorithm also builds a list of global clusters that each $r \in e$ is assigned to. Then it removes r from global clusters that it has been assigned to, for each $r \in e$ that are not uniquely assigned.

Global Cluster Refinement

A refinement of the global clusters is required if reads that have previously been assigned to the same class are separated by the local partition p for the first time. If a global cluster is linked to more than one entry in p, those reads that have been locally separated are reassigned to a new global cluster. This is shown in Algorithm I.5 (p. 224).

First, the algorithm collect the potentially refining local partition entries for each global cluster with id i in refining$[i]$. Then, each global cluster that has more than one partition entry in refining$[i]$, is split using the entries. After each splitting, the class split is recorded in C, the new class inherits all conflicts from the previous class.

Final Duplicate Class Removal

After processing all local partitions, there might be two or more classes that pairwisely contain the same reads. In the final duplicate class removal step, the method removes such duplicates.

Algorithm I.2: global-class-assignment(ids), Algorithm 7 from (Kuchenbecker, 2011)

input : local partition p
side-effects: Updates G and C to reflect the information of ids.

```
/* Collecting mappings (i, S) from entry id in p to set of ids in G. For
   each entry in p with ids i, record all entries in G with ids j where a
   read from i was assigned to j. This is used for the projection in the
   conflict projection at the bottom;                                    */
assignments ← ∅;
/* Process all partition entries.                                        */
for each partition entry e with id idx in p do
    /* Collect ids of global clusters that share a read with e or create a
       new cluster if no such read is found                              */
    ids ← {i : (i, S) ∈ G s.t. ∃ r ∈ e ∧ r ∈ S};
    if ids = ∅ then                    /* no global cluster shares a read with e */
        G ← G ∪ {(nextID, ∅)};              /* create a new empty cluster in G */
        ids ← {nextID};
        nextID ← nextID + 1;
    end
    /* Merge global clusters in G with selected ids since they occured in
       the same local partition.                                         */
    if |ids| > 1 then
        global-class-merging(ids);
    end
    /* Assign each read r (yet not seen in any global cluster) from e to all
       clusters that e shares a read with                                */
    for each read r ∈ e do
        if not r ∈ S for any (i, S) ∈ G then
            for i ∈ ids do
                G[i] ← G[i] ∪ {r};
                assignments[idx] ← assignments[idx] ∪ {i};
            end
        end
    end
end
/* Project conflicts from p to C.                                        */
for each unordered pair of {(i, S), (j, T)} from assignments do
    C ← C ∪ {i, j};
end
```

Algorithm I.3: global-class-merging(ids), Algorithm 8 from (Kuchenbecker, 2011)

input : set of global cluster ids
side-effects: Merges clusters in G and updates global conflicts S accordingly
returns : updated ids after merging

```
/* collect entries of ids that are in conflict with another entry in ids   */
```
conflicting $\leftarrow \{i, j : i \in \text{ids} \wedge j \in \text{ids} \wedge \{i, j\} \in C\}$;
non-conflicting $\leftarrow$ ids $\setminus$ conflicting;
```
/* no merging if there are no non-conflicting ids                          */
```
if non-conflicting $= \emptyset$ **then**
  return;
end
```
/* create new global cluster                                               */
```
newID $\leftarrow$ nextID;
$R \leftarrow \bigcup_{c \in \text{non-conflicting}} c$; `/* Merge non-conflicting classes. */`
$G \leftarrow G \cup \{(\text{newID}, R)\}$;
nextID $\leftarrow$ nextID $+ 1$;
```
/* build list of conflicts to remove and to add                            */
```
toRemove, toAdd $\leftarrow \emptyset, \emptyset$;
for $\{i, j\} \in C$ **do**
  if $i \in$ non-conflicting $\vee$ $j \in$ non-conflicting **then**
    toRemove $\leftarrow \{i, j\}$;
    if $i \in$ non-conflicting **then**
      toAdd $\leftarrow \{newID, j\}$;
    else
      toAdd $\leftarrow \{newID, i\}$;
    end
  end
end
```
/* update conflicts in C                                                   */
```
remove entries of toRemove from C;
add entries of toAdd to C;
```
/* update global clusters in G                                             */
```
$G \leftarrow \{(i, S) \in G : i \notin \text{non-conflicting}\}$;
```
/* build result                                                            */
```
result $\leftarrow \{i \in \text{ids} : i \notin \text{non-conflicting}\} \cup \{\text{newID}\}$;
return result;

Algorithm I.4: multi-class-assignment-removal(ids), Algorithm 5 from (Kuchenbecker, 2011)

input : local partition p

side-effects: Updates G to contain fewer multi-cluster assignments.

for each entry $e \in p$ **do**

  $u \leftarrow \emptyset$; `/* set of read ids that have an assignment to only one cluster */`

  `/*` ℓ `is a mapping` (i, S) `from read id` i `to set of all global cluster ids` S `that contain read with id` i `*/`

  $\ell \leftarrow \emptyset$;

  `/* collect uniquely assigned reads in` u `and build mapping` ℓ `*/`

  for each read $r \in e$ **do**

    $\ell[r] \leftarrow \{i : r \in G[i]\}$; `/* ids of global clusters containing` r `*/`

    if $|\ell[r]| = 1$ **then**

      $u \leftarrow u \cup g$;

    end

  end

  `/* remove multi-assigned reads that occur in` p `*/`

  for each read $r \in p$ **do**

    for each global cluster id $g \in S, g \notin u$ with $S = \ell[r]$ **do**

      $G[g] \leftarrow G[g] \setminus \{r\}$;

    end

  end

end

Algorithm I.5: global-cluster-refinement(ids), Algorithm 6 from (Kuchenbecker, 2011)

input : local partition p
side-effects: Updates G by splitting/refining global clusters by separating information from p.

```
/* collect refining entries for each global cluster                    */
refining ← ∅;
for each entry e ∈ p do
    for each read r ∈ e do
        for each i ∈ {i : r ∈ G[i]} do
            refining[i] ← refining[i] ∪ {e};
        end
    end
end
/* refine clusters                                                     */
G' ← ∅;                                          /* added global clusters */
for each global cluster (i, S) ∈ G do
    if |refining[i]| > 1 then
        for each local partion entry e ∈ refining[i] do
            newID ← nextID;                          /* create new cluster */
            G' ← G' ∪ {(newID, ∅)};
            nextID ← nextID + 1;
            for each read r ∈ e do /* Perform splitting                  */

                if r ∈ S then
                    S ← S \ {r};
                    G'[newID] ← G'[newID] ∪ {r};
                end
            end
            Record class split in C: The new class G'[newID] inherits all conflicts of G[i].
        end
    end
end
G ← G ∪ G';                                      /* add new global clusters */
```

Abbreviations

1KGP thousand genomes project

2GS second-generation sequencing

3GS third-generation sequencing

ABI Advanced Biotechnologies, a biotechnology company

ANISE Assembly of Novel Inserted Sequence

ASCII American Standard Code for Information Interchange

BAM Binary Alignment and Mapping format

BASIL Base-level Insertion Locator

BFS breadth-first search

BOG best overlap graph

BWT Burrows-Wheeler Transform

CMOS Complementary Metal Oxide Semiconductor

CNV copy number variation

CPU central processing unit

ChIP chromatin immunoprecipitation

DAG directed acyclic graph

dbSNP a database of small polymorphisms

DNA Deoxyribonucleic acid

DP Dynamic Programming

EBI EMBL European Bioinformatics Institute

EMBL European Molecular Biology Laboratory

EMBOSS European Molecular Biology Open Software Suite

ENA European Read Archive

FASTA text format for encoding biological sequences

FASTQ text format for encoding biological sequences with quality information

FDA Food and Drug Agency

FDR false discovery rate

FN false negative

FP false positive

GB gigabyte (2^{30} bytes)

GHz gigahertz

GSI gold standard intervals

GWAS genome-wide association studies

HGP Human Genome Project

HTS High-Throughput Sequencing

ID identifier

LBA local block aligner

LINE Long Interspersed

MCSTL Multi-Core Standard Template Library

$\mathcal{NP}$ Nondeterministic-Polynomial

MSA Multiple Sequencing Alignment

MST Maximum Spanning Tree

NCBI National Center for Biotechnology Information

NGS Next-Generation Sequencing

OEA One-End Anchored

OLC overlap-layout-consensus

PR paired read

QC quality control

RABEMA Read Alignment Benchmark

RAM random access memory

RD read depth

RNA Ribonucleic Acid

SAM Sequence Alignment and Mapping format

SIMD Single Instruction Multiple Data

SNP Single Nucleotide Polymorphism

SNV Single Nucleotide Variant

SOLiD Sequencing by Oligonucleotide Ligation and Detection

SRA Sequence Read Archive

SR split-read

STXXL STL variant for "XXL" datasets

STL Standard Template Library

SV Structural Variant

SeqAn a C++ library for sequence analysis

TB terabyte (2^{40} bytes)

TN true negative

TP true positive

VCF Variant Call Format

w.l.o.g. without loss of generality

Index

S

T

V

Curriculum Vitae

For reasons of privacy, the curriculum vitae is not part of the online version.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Berlin, den 10.5.2015 (Manuel Holtgrewe)