Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis Bioinformatics

TOPAS - TOolkit for Processing and Annotating Sequence Data

Simon Heumos

April 30, 2014

Reviewer

Dr. Kay Nieselt
(Integrative Transcriptomics)
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen

Supervisor

Günter Jäger
(Integrative Transcriptomics)
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen

Heumos, Simon:
TOPAS - TOolkit for Processing and Annotating Sequence Data
Bachelor Thesis Bioinformatics
Eberhard Karls Universität Tübingen
Period: January 01, 2014 - April 30, 2014

Abstract

Working with sequence data and genomic annotations is an everyday challenge for a bioinformatician: Sequence data and their corresponding annotations often have to be filtered, formatted, validated or processed for further, individual use. There already exist a lot of different applications trying to accomplish that task, but most of them were designed to tackle specific problems only. Hence the output formats of these tools are often-times not compatible with each other. Furthermore they lack of useful functions for validation and correction of corrupted data files. Some existing methods are not well-engineered, so alternatives are needed. Also, because their structure can be confusing and their code is unapproachable. This nuisances demand a platform independent toolbox for the efficient formatting, filtering, processing and validating of different file formats.

In this thesis, **TOPAS** (**TO**olkit for **P**rocessing and **A**nnotating **S**equence Data) a command-line toolkit written in `Java` allowing the user to efficiently filter, format, validate and process sequence and annotation data is presented. In difference to other NGS data processing programs, `TOPAS` offers not only several functions for one particular area of application, but provides a various number of modules which can be categorized into four different application fields: FASTA processing modules, FASTQ processing modules, GFF3 processing modules and VCF processing modules. This comprehensive functionality allows `TOPAS` to work as a data (format) interface between existing toolkits. Furthermore, `TOPAS` combines functions for the validation of FASTA, FASTQ and GFF3 files. `TOPAS` introduces several new features like the correction of validated FASTA files. Another one represents the newly developed way of how VCF files can be indexed. Utilizing a hierarchical, module-based approach, TOPAS is an easy to use and easy to extend set of tools. The well documented code supports the transfer of parts of `TOPAS'` code into other pipelines. In addition, `TOPAS'` code library allows the user to design his own, individual modules. To summarize, my platform independent implementation offers approved and nouveau methods to process sequence and annotation data efficiently embodying an useful collection of tools for the daily work of bioinformaticians.

Zusammenfassung

Das Arbeiten mit Sequenzdaten und genomischen Annotationen ist täglich eine neue Herausforderung für Bioinformatiker: Oft müssen Sequenzdaten und zugehörige Annotationen für den individuellen Bedarf angepasst werden. Dies geschieht durch Filterung, Formatierung, Validierung oder anderweitige Anpassung der Daten. Es gibt schon verschiedene Anwendungen, die versuchen dies zu bewältigen, jedoch sind die meisten davon nicht dafür ausgelegt, da sie sich nur auf bestimmte Problemgebiete anwenden lassen. Deswegen sind die Ausgabeformate dieser Programme oft inkompatibel miteinander. Des Weiteren fehlen ihnen nützliche Funktionen zum Validieren und Ausbessern von fehlerhaften Dateien. Wenige schon existierende Methoden sind nicht ausgereift, deswegen werden Alternativen benötigt. Auch, weil deren Aufbau verwirrend sein kann und der Code unzugänglich ist. Diese Misstände verlangen nach einem plattformunabhängigen Programm, welches verschiedene Dateiformate effizient formatieren, filtern, prozessieren und validieren kann.

Innerhalb dieser Abschlussarbeit wurde **TOPAS** (**TO**olkit for **P**rocessing and **A**nnotating **S**equence Data), ein in `Java` geschriebenes Kommandozeilenprogramm, entwickelt. Es ermöglicht dem Benutzer die effiziente Filterung, Formatierung, Validierung und anderweitige Bearbeitung von Sequenz- und Annotationsdaten. Im Unterschied zu anderen NGS Daten prozessierenden Programmen bietet `TOPAS` nicht nur einige Funktionen für ein bestimmtes Anwendungsgebiet an, sondern stellt mehrere Module, welche in vier verschiedene Anwendungsgebiete eingeteilt werden können zur Verfügung: die FASTA prozessierenden Module, die FASTQ prozessierenden Module, die GFF3 prozessierenden Module und die VCF prozessierenden Module. Aufgrund dieser vielfältigen Funktionalität kann `TOPAS` als Schnittstelle zwischen diesen Programmen eingesetzt werden. Des Weiteren verknüpft `TOPAS` Funktionen für die Validierung von FASTA, FASTQ und GFF3 Dateien. `TOPAS` stellt einige neue Funktionen wie z.B. das Korrigieren validierter Dateien bereit. Auch wird eine neue Methode zum Indizieren einer VCF Datei bereitgestellt. Aufgrund der hierarchischen, modularen Struktur von `TOPAS` ist dieses leicht zu benutzen und einfach zu erweitern. Dies wird durch den gut dokumentierte Code unterstützt. Auch ist das Übertragen von Codeteilen in andere Programme und/oder Pipelines problemlos möglich. Die von `TOPAS` bereitgestellte Codebibliothek erleichtert die Entwicklung von neuen Modulen. Zusammenfassend lässt sich sagen: Meine plattformunabhängige Implementierung stellt bewährte und neue, effiziente Methoden zum Prozessieren von Sequenz- und Annotationsdaten zur Verfügung und verkörpert damit einen nützlichen Werkzeugkasten für den täglichen Gebrauch eines Bioinformatikers.

Acknowledgements

First of all, my special thanks go to Dr. Kay Nieselt for giving me the opportunity to work on this thesis and for her helpful suggestions.

I also want to thank my supervisor Günter Jäger for his incomparable optimism, excellent know-how and virtuous patience.

In addition, I owe gratitude to all the members of the working group and my friends who supported me with critical advise, productive discussions and corrections of my thesis: Alexander Peltzer, Jakob Matthes, André Hennig, Konstantin Grupp and Linda Wack.

Finally I thank my parents, Anita and Markus Heumos, who encouraged me through my prior study period including the time of this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ASCII	American Standard Code For Information Interchange
BAM	Binary Alignment/Map
BWA	Burrows Wheeler Aligner
CHROM	Chromosome
DNA	Deoxyribonucleic Acid
EAGER	Efficient Algorithms For Ancient Human Genome Reconstruction
GATK	Genome Analysis Toolkit
GC	Guanine Cytosine
GFF	Generic Feature Format
GTF	Gene Transfer Format
GUI	General User Interface
GVF	Genome Variant Format
IGV	Integrative Genome Viewer
INDEL	Insertion/Deletion
IUPAC	International Union Of Pure And Applied Chemistry
NGS	Next Generation Sequencing
PHP	Hypertext Preprocessor
POS	Position
RAM	Random Access Memory
RNA	Ribonucleic Acid
SAM	Sequence Alignment/Map
SEQID	Sequence Identifier
SNP	Single Nucleotide Polymorphism
TOPAS	TOolkit For Processing And Annotating Sequence Data
VCF	Variant Call Format

Chapter 1

Introduction

Working with sequence data and genomic annotations is an everyday challenge for a bioinformatician: The work involves filtering, formatting, validating and processing of sequence and annotation data. Sequence data and their corresponding annotations often have to be filtered, formatted, validated or processed for further, individual use. There already exist a lot of different applications trying to accomplish these tasks, but most of them are designed to tackle specific problems only.

Molecular sequence data, and particularly nucleotide data are stored in specific formats to hold the binary sequence data itself and other information about the sequence. The simplest and most commonly known format is the FASTA [PL88] format. DNA sequencers of the second generation provide sequence data in the FASTQ[1] format. Next generation sequence alignment data is stored in the Sequence Alignment/Map Format (SAM) [LHW+09]. The common format for genome annotations is the General Feature Format (GFF2[2], GTF[3], GFF3[4]) and recently the Genome Variation Format [RMB+10] (GVF) has been published to address extend the GFF3 file format. Genomic features may even be represented in the BED format[5]. As part of the 1000 Genomes Project the Variant Call Format [DAA+11] (VCF) was developed encoding structural genetic variants. A commonly used file format for association studies is the MAP[6] format complementing the PED[6] format. Although the specifications for all those file formats are clear, there are still a number of corrupted files in online databases. Few programs exist to identify and repair or eliminate those files.

Functions for filtering and processing of sequence and annotation

[1] `http://maq.sourceforge.net/fastq.shtml` (last accessed: 04/02/2014)
[2] `http://gmod.org/wiki/GFF2` (last accessed: 04/03/2014)
[3] `http://mblab.wustl.edu/GTF2.html` (last accessed: 04/04/2014)
[4] `http://www.sequenceontology.org/gff3.shtml` (last accessed: 04/04/2014)
[5] `http://www.ensembl.org/info/website/upload/bed.html` (last accessed: 04/07/2014)
[6] `http://www.gwaspi.org/?page_id=145` (last accessed: 04/06/2014)

data exist, but they are often deeply integrated into complex software like online bioinformatic frameworks (e.g. Galaxy[GNTT10;BKC+10;GRH+05], IGV[TRM12] or FaBox[Vil07]) or they are a part of programming libraries (e.g. FASTX-Toolkit[7], GATK[MHB+10;DBP+11;VdACC+13], HOMER[HBSB10], NGSUtils[BY13], NGS QC Toolkit[PJ12], VCFtools[DAA+11], SAMtools[LHW+09], BEDTools[QH], GenomeTools[GSK13], PLINK[PNTB+07], seqtk[8], gff3-pltools[9], SnpSift[CPC+12]). The individuality and the lack of extensibility of such tools makes it difficult for the user to maintain an overview. For each particular job he might have to install a different tool. During the installation process of a tool the user might run into other problems: The tool to be installed requires additional libraries or other dependencies in order to be compilable, leading to a high installation effort. Additionally, some of these tools are badly documented, so users might have difficulties using particular functions of these tools. On top of that, these program suites have further disadvantages: Although such tools possess a lot of functions, in most cases these toolkits do not check or validate the data they are working with, which can result in ugly errors and laborious debugging work. Also, the data such programs produce might not compatible with each other, the data may be corrupted or even in the wrong format, and so forth. These considerations revealed, that a platform independent toolbox which formats, filters, processes and especially validates sequence and annotation date efficiently is missing.

In this thesis TOPAS, (**TO**olkit for **P**rocessing and **A**nnotating **S**equence Data) has been developed. It serves as an interface between these tools as well as it formats data to the required tool's data format and takes over tasks of existing programs and extends or improves existing functions. In addition, it supplies the validation of several data file formats. With the development of TOPAS, a program suite was designed providing a set of user-friendly and extendible command-line tools for the efficient manipulation and validation of sequence and annotation data. TOPAS is easy to extend (quick implementation of new modules possible, due to large existing code library), well documented and because of its implementation in Java platform independent (no special requirements are needed to run TOPAS, except a JVM).

1.1 Structure of the Thesis

The thesis consists of five chapters. Chapter two describes the biological background of this thesis. In chapter three the code architecture of TOPAS will be laid out in detail. Moreover, each of TOPAS' modules will be presented. Their general motivation and workflow will be illustrated. Chapter four contains the

[7] http://hannonlab.cshl.edu/fastx_toolkit/index.html (last accessed: 04/23/2014)
[8] https://github.com/lh3/seqtk (last accessed 04/03/2014)
[9] https://github.com/mamarjan/gff3-pltools (last accessed: 04/03/2014)

runtime evaluation of each of TOPAS' tools. In addition, these performance results are compared with the runtime measurements of comparable tools. This chapter will be closed by a presentation of how TOPAS could be integrated into an existing pipeline. In the final chapter, TOPAS' results will be critically discussed. Furthermore, a short outlook of what additional modules could still be implemented into TOPAS and where TOPAS might be heading in the future, will be given.

Chapter 2

Background

In this chapter the biological background of the general application field of this thesis is laid out. An introduction to the field of next generation sequencing (NGS) is given. A focus is set on the most commonly used next generation sequencing procedures, Illumina dye sequencing. After that, file formats related to NGS are explained in detail. Finally, a small synopsis about the most commonly used tools for processing and annotating sequence data will complete this chapter.

2.1 Next Generation Sequencing

A first big step towards DNA sequencing was the discovery of the DNA double helix in 1953. In 1977, Frederick Sanger laid the foundation of future sequencing technologies with the invention of the Sanger Sequencing Method [SNC77]. Until now, most automated sequencing machines make use of enhanced variants of Sanger's method in order to identify the order of nucleotides in a DNA sequence of a given individual [Met10]. Next generation sequencing machines find a wide application in today's genetic research.

There existed and still exist different NGS companies like Illumina, Roche, Life Technologies and Pacific Biosciences. Because the most often used technology to produce NGS data descends from Illumina, this chapter will only focus on Illumina's NGS procedure.

Illumina dye sequencing, one of the first aspiring NGS techniques, uses the concept of sequencing by synthesis. This sequencing approach is divided into four major steps, the first one being the sample preparation followed by the cluster generation after which the actual sequencing takes place. Finally the resultant image data is transformed to sequence data, which then can be analysed.

In the sample preparation step two corresponding single strand DNA fragments are created from double-stranded DNA samples and attached to the surface of an Illumina flow cell. Then up to 1000 copies of the DNA fragments are generated for the following sequence by synthesis step in the flow cell. In each sequencing cycle the flow cell is filled with fluorescence labelled bases, primers and DNA polymerase. After the polymerization of one base, it is stimulated with a laser and the radiated light is quantified [III].

Base calling takes place in each cycle, when the signal intensities are measured. Each synthesized nucleotide now has a corresponding quality value. This so called Phred quality score S is logarithmically related to the base-calling error probability E (which is defined as the probability of a wrong base call, e.g. $E = 0.1$ means a 90% base call accuracy), by the following equation 2.1:

$$S = -10 \cdot log_{10} \cdot E \tag{2.1}$$

All the measured signals of a DNA template typically are combined to a read. Such reads and their corresponding quality scores are stored in the FASTQ file format. A detailed description of that file format can be found in Chapter 2.2.3.

Mapping of Reads

NGS methods typically produce many reads (with length 30-150 100nt each). An important bioinformatic task is mapping the reads to a reference genome in order to align the sequenced reads to the best fitting region of the genome. Software tools for genome mapping require the nucleotide sequence of the reference genome (stored in the FASTA file format, see Chapter 2.2.1) to map to, and in some cases also a file of the positions of the genes of the genome (GFF3 file format, see Chapter 2.3.1) to map to. Finally, the mapped reads can be used for further analysis, e.g. variant calling. Resulting variants like SNPs or INDELs are stored in the Variant Call Format (VCF) (more precise details about that file format can be found in Chapter 2.4). If an RNA-seq experiment was performed (i.e. NGS with RNA samples instead of DNA samples), the transcriptome of the given organism can be determined.

2.2 Sequence Data Formats

Molecular sequence data is stored in specific formats to hold the sequence data itself and other information about the sequence. In the following sections the most commonly used sequence data formats FASTA format, FASTA index and FASTQ format will be explained in detail.

2.2.1 FASTA

The simplest and most commonly used sequence file format is the FASTA[1] format. It arose from the software package FASTA [PL88]. The primary structure of either nucleotide sequences as well as protein sequences are stored in a FASTA file.

The first line of a FASTA entry is called the header or description line. A '>' character indicates the beginning of such a line. An unique identifier (exactly one word) is directly followed after the greater-than symbol and may be followed by an additional description (both are optional). After the header line the sequence itself follows, splitted into usually not exceeding 80 characters long sequence lines. Sequence characters are presented in the standard IUB/IUPAC amino acid and/or nucleic acid code formats[2].

FASTA files with multiple FASTA entries form a Multi-FASTA file.
An example of a FASTA entry can be seen in Figure 2.1.

```
>gi|86553285|gb|ABC98243.1|
MALKQLGHVAVRVEDISKAVEFYEKLGMVNVWKDPDWAYMKAGDDGLALLGPGYRAAGPHFGFVFSSREE
LEEQHRRLQAAGIPVGAIHSHRDGTASFYGKDPDGNLFEFLYEPPGTFDQAKAKTAEASA
```

Figure 2.1: Example of a single FASTA file entry of a protein sequence. A single protein sequence can be seen, with identifier and the protein sequence itself.

2.2.2 FASTA Index

A bioinformatician often has to deal with huge sequence files. In order to do this efficiently, such files are indexed. For each sequence in a FASTA file a namely index is created. A FASTA index consists of five tab-delimited fields: The sequence name, the sequence length, the first base offset in bytes, the number of chars in each line and the number of bytes in each FASTA line. In Figure 2.2 the FASTA index of the sequence shown in Figure 2.1 is shown.

```
>gi|86553285|gb|ABC98243.1|        130        83        70        71
```

Figure 2.2: FASTA index of the protein sequence shown in Figure 2.1.

2.2.3 FASTQ

DNA sequencers provide sequence data (the reads) in the FASTQ[3] format. For each read, the FASTQ file contains the sequence information of the read

[1] `https://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml` (last accessed: 04/03/2014)
[2] `http://www.dna.affrc.go.jp/misc/MPsrch/InfoIUPAC.html` (last accessed: 04/03/2014)
[3] `http://maq.sourceforge.net/fastq.shtml` (last accessed: 04/03/2014)

(similar to the FASTA format) and for each base in this read a corresponding quality score.

As illustrated in Figure 2.3 a FASTQ entry consists of four lines: The first line, the sequence header, begins with an '@' letter which is followed by an unique identifier of the sequence and an optional description.
The second line contains the sequence characters (multi-lines are possible).
The third line starts with a '+' letter and is optionally followed by the same sequence header as for the sequence line. The final sequence quality line must be of the same length as the sequence line. Its character sequence encodes the quality values (Phred scores) for the sequence in the second line. The encoding of such values determines from which type of sequencing platform the reads originates. Currently the Illumina 1.8 encoding format rates raw reads with values of 0 to 41 (in ASCII encoding).

```
@F_HISEQ-155:1:1101:1042:2882/1 ~ RGR:C23;
CTAAAAACCAACGTATTATTTTCTCTATCCATTGTAGTGGGTGTGTGTGTGT
+
@@7;+=DDFFBFFAAFF4CF<HFHIIC4CF9?EG9??1?)????FGF?B?B.
```

Figure 2.3: Snapshot of a FASTQ file from Illumina 1.8 platform displaying sequence header, sequence characters, corresponding scoring values and an empty sequence quality header.

2.3 Annotation Data Formats

Annotation data files and in particular genome annotation files contain information of the locations of a genome's loci. Such files can be used to annotate NGS sequence data originating from the mapping step in order to highlight, which regions of the genome the reads originate, are covered by these reads. Ordinarily annotation data is stored in a tab-delimited file format. Annotations for reference genomes are most commonly held by GFF files. A detailed introduction of this file format is given in the following chapter.

2.3.1 Generic Feature Format

The typical format for Genome Annotation is the Generic Feature Format (GFF)[4] available in its latest release version 3. The GFF3 format describes genomic structures in a plain text file in which genomic features present a hierarchical model of the corresponding genome. A genomic feature may consist of several GFF3 entries which are represented, as one can see in Figure 2.4,

[4] `http://www.sequenceontology.org/gff3.shtml` (last accessed: 04/03/2014)

by nine tab-delimited fields. A detailed description of the fields is given in Table 2.1. The 'part of' relationship of the entries forming a feature are resolved by their 'Parent' and 'ID' tags in the last field attributes.

```
##gff-version 3
##sequence-region CP000239.1 1 2932766
CP000239.1  Genbank  region  1  2932766  .  +  .  ID=id0;Name=ANONYMOUS
CP000239.1  Genbank  gene   71     1474  .  +  .  ID=gene0;Name=dnaA
```

Figure 2.4: Excerpt of a GFF3 file. In the first line the GFF version can be seen, followed by the sequence region this GFF3 file covers. Below two GFF3 entries are situated, constructing a GFF3 feature.

2.4 Variant Call Format

A generic file format for storing variant information is the Variant Call Format[DAA+11] (VCF). It was developed as a part of the 1000 Genomes Project.

A VCF file holds variant information such as single nucleotide polymorphisms (SNPs), insertions/deletions (INDELs) and other structural variants. The first lines of a VCF file (beginning with '##') contain meta-information about the general structure and format of all the VCF entries. These are followed by a header line (starting with a single '#') determining the fields of each entry. The mandatory fields of a VCF entry can be seen in Table 2.2 while a header of a typical VCF file is illustrated in Figure 2.5.

2.5 Short Overview of Tools for Manipulating Sequence Data

There exist a number of tools which provide functions for processing and annotating sequence data. However, in this section, the focus will be laid only on the most commonly known tools, which will be introduced further in this section.

2.5.1 FASTX-Toolkit

The FASTX-Toolkit[5] is a software suite of command line tools for manipulating FASTA/FASTQ files. NGS data often has to be preprocessed before the mapping step (e.g. filter by sequence quality of short reads) in order to obtain better mapping results. FASTX-Toolkit provides functions for that task. It

[5] `http://hannonlab.cshl.edu/fastx_toolkit/index.html` (last accessed: 04/04/2014)

Table 2.1: Overview over all fields of a GFF3 entry. The first row is equivalent to the first field, and so forth.

ID	***Explanation***
seqid	the id of the reference point (usually chrom or contig) the coordinate system was generated from
source	the origin, database or method of this entry
type	the entry type, normally derived from the Sequence Ontology
start	the start coordinate of the entry in relation of the corresponding seqid, using a positive 1-based integer coordinate system
end	end coordinate of the entry
score	the score of the entry
strand	the entry's strand, either '+' for positive, '-' for negative, or '.' for undefined
phase	necessary for entries of type 'CDS', declares where the entry, begins (in relation to the reading frame)
attributes	a list of attributes providing additional information about the entry, attributes are in 'tag=value' format separated by a semicolon

```
##fileformat=VCFv4.2
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS   ID        REF ALT QUAL FILTER INFO                    FORMAT      NA00001
20     14370 rs6054257 G   A   29   PASS   NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ 0|0:48:1:51,51
20     17330 .         T   A   3    q10    NS=3;DP=11;AF=0.017     GT:GQ:DP:HQ 0|0:49:3:58,50
```

Figure 2.5: Excerpt of a VCF file in format version 4.2. The first line is a mandatory header line specifying the version of the VCF file. The first line starting with a single '#' is also a required header line, which determines the fields of each entry. Between these two additional header lines, containing information about the annotations in the VCF body, are located. The first line of the body is a classical SNP. The second line was filtered out because of quality reasons and is therefore not regarded as a SNP.

Table 2.2: Required fields of a VCF entry. In a VCF file all fields are separated by tabs, the first row is equivalent to the first field, and so forth.

ID	***Explanation***
CHROM	the chromosome
POS	the reference position, 1 based
ID	if disposable, a list of unique semicolon-delimited identifiers
REF	the base(s) of the reference
ALT	a list of comma-separated alternate bases related to at least one of the samples
QUAL	a quality score in Phred-scaled format rating the proposition in ALT
FILTER	'PASS' if the call filter was passed by this position, else a code explaining the failure is written
INFO	ancillary information

can be used to filter out low quality reads or to cut off single bases of each read in a FASTQ file in order to improve mapping results. Also, FASTX-Toolkit can generate statistical information of a given FASTQ file. These functions and many more make the FASTX-Toolkit a crucial NGS program for manipulating FASTA/FASTQ files.

2.5.2 SAMtools

SAMtools[LHW+09] is a program package, that is capable of processing Sequence/Alignment Map[LHW+09] (SAM) files. The SAM format is able to store large nucleotide sequence alignments produced by NGS mappers. SAMtools converts SAM files into their binary format, the BAM format. By sorting and indexing such BAM files with SAMtools, it is possible to efficiently parse them. In addition, SAMtools provides other functions to manipulate SAM/BAM files. An example is the generation of statistical data about a given BAM file. Another feature of the SAMtools package is the indexing of FASTA files.

2.5.3 VCFtools

The VCFtools[DAA+11] are a collection of tools created for working with VCF files. This set of tools is able to validate and compare VCF files. It provides functions for filtering variants, comparing and optionally merging files, validating VCF files and applying set operations on variants. The VCFtools offer a general application programming interface (API) for the user.

Chapter 3

Material and Methods

In this chapter the architecture of TOPAS is introduced. More precisely, the general code structure of TOPAS is described and a closer look is taken at some programming methods that are implemented in TOPAS. Additionally, an overview of all the modules ordered by their general use case is given. Finally, every implemented module is presented.

The first part of this chapter focuses on the code model of TOPAS, its user interface (front-end design) and its programming features, where amongst other things a newly developed VCF index is introduced. The second part concentrates on the modules that are available in TOPAS.

3.1 Architecture of TOPAS

TOPAS is a command-line toolkit developed in Java 7. This allows its execution on a wide range of platforms. With the extensive use of Java documentation in all of the implemented classes, a good basis for the future enhancement of the implementation was created.

3.1.1 General Code Structure

The general idea of how to hierarchically structure the code in a module-based way was taken over from the PASSAGE-Toolkit[BKHN10], developed by Florian Battke. TOPAS architecture can be seen in Figure 3.1 together with its code library, displayed as a list of packages. The essence of TOPAS can be found in package *core*, where classes for processing FASTA/FASTQ relevant data are located. Within the package *io* are all the classes with methods for reading and writing the different data formats (which were introduced in Chapter 2) organized. Functions for filtering, creating, and validating GFF3 files are hold in the *gff* package. Package *comparison* contains classes that are used for

comparisons of *FastaIndex* and *GffThreeEntry* objects. The *vcf* package holds classes which mainly contain information about how to filter and index VCF files. The package *utils* includes classes that have useful commonly used methods (e.g. formatting a string, verifying the type of a class, etc.) integrated. The package *lib* consists mainly of classes that provide general utilities for parsing and constructing command-line arguments. For each implemented module there exists a class and its corresponding parameters class. These are all stored in the *topas* package.

`TOPAS` interacts with the command-line arguments entered by the user as follows: Every command-line input passes the main class, which optionally displays all the currently implemented modules available using Java Retention. If a module was chosen, `TOPAS` invokes this module with the entered parameters (the classes *Parameter* and *Parameters* for parsing the command-line were borrowed from the `PASSAGE`-Toolkit as well). Every module class possesses a *Parameters* object holding the information necessary for the construction and parsing of module-parameters.

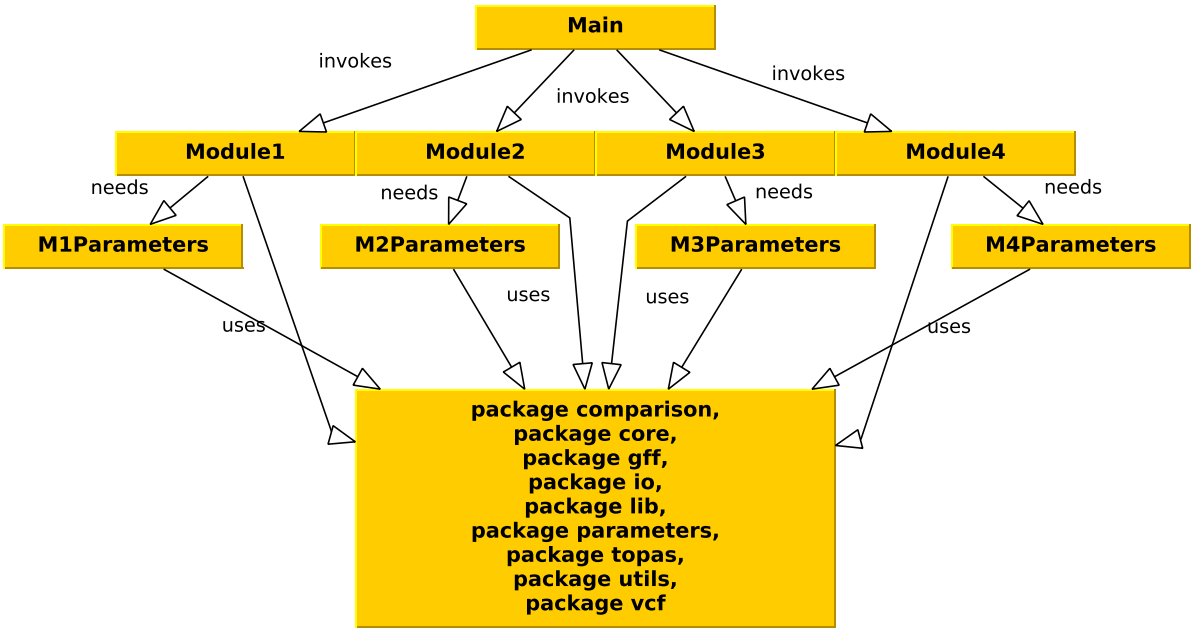

Figure 3.1: An UML diagram showing the general code structure of `TOPAS`. On top the *Main* class is shown, which is able to invoke all the modules implemented in `TOPAS`. Furthermore, it can be seen, that every module relies on its specific *Parameters* class in order to create and parse possible command line arguments correctly. Below, the code library of `TOPAS` is displayed, which is divided into several packages. Every module and corresponding *Parameters* class make use of these packages.

.

3.1.2 Software Engineering Principles

One programming feature that is applied in TOPAS is the template method pattern[1]. The template method pattern provides the main functionality of a module in an abstract class while individual methods of this functionality are passed to subclasses or submodules. This results in a modular and abstract design of the program.

To manage memory efficiently, large data is indexed (e.g. building a FASTA index for a FASTA file or a VCF index for a VCF file) or, if possible, only the data necessary for running a module is read in and then directly written out again.

VCF Index

VCF files can become very large (possibly more than 100GB for a human genome data set for example). A possible way for the efficient location of specific VCF entries in a VCF file is to build a VCF index of the VCF file. There exists a tool that is capable of performing this task (Tabix[Li11]), but it requires several libraries such as Perl and Python. Another disadvantage of this tool is that the file to index first has to be compressed with BGZIP[Li11] which results in additional use of disk space (about 25% of the input file). Thus a new, simple way of how to index a VCF file is needed: The VCF file is parsed and at the first occurrence of a VCF entry, a VCF index is generated. After that, the next VCF index is created when a fixed number of VCF entries were read (commonly 10000). This procedure is applied until the end of the file is reached.

A VCF index in TOPAS consists of four fields, being the chromosome, the position and the offset of the first VCF line belonging to that index and the length of the offset in bytes determined by this index. A VCF index of a sample VCF file is displayed in Figure 3.2. Note that a VCF index is not only generated after the fixed number of lines have been passed, but also when the field *CHROM* in the VCF file changes.

3.2 Modules of TOPAS

The modules of TOPAS are organized in four different application fields (see in Table 3.1). In the following, each of TOPAS modules are presented individually.

[1] http://sourcemaking.com/design_patterns/template_method (last accessed: 05/04/2014)

```
##fileformat=VCFv4.2
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS    ID         REF ALT QUAL FILTER INFO                              FORMAT      NA00001
20    14370  rs6054257  G   A   29   PASS   NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ 0|0:48:1:51,51
20    17330  .          T   A   3    q10    NS=3;DP=11;AF=0.017      GT:GQ:DP:HQ 0|0:49:3:58,50
```

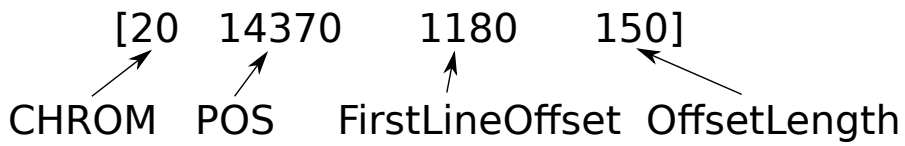

Figure 3.2: On the top a header of a VCF file can be seen, while on the bottom the corresponding VCF index is displayed.

Table 3.1: Overview over all modules of `TOPAS`, sorted by their type of data processor.

FASTA processing modules	***FASTQ processing modules***	***GFF3 processing modules***	***VCF processing modules***
ValidateFasta	ValidateFastq	ValidateGFF3	IndexVCF
CorrectFasta	FormatFastq	SortGFF3	FilterVCF
IndexFasta		FilterGFF3	AnnotateVCF
ExtractFasta			
TabulateFasta			

3.2.1 ValidateFasta

General Motivation

First of all there exist a lot of corrupted FASTA files in genome databases on the internet. Most FASTA file processing programs do not validate if a FASTA file is consistent with the required specifications for example they do not check if the the sequence headers are unique or if there are any empty and/or comment lines within a sequence content. Thus a validation tool for FASTA files is important to prevent programs from crashing or even producing wrong results.

Furthermore, validation tools are rare. Implementations like `FastaValidator`[2], a `Java` library providing functions for the validation and parsing of FASTA files, or `Validate Fasta File`[3], a tool capable of validating FASTA files, either have insufficient functions or are further not platform independent (e.g. `Validate Fasta File` is only running on windows). In addition they are badly documented. Thus the user is in need of a tool that validates a FASTA file efficiently and gives him information about the file structure and the sequence content of the FASTA file.

Workflow

`ValidateFasta` takes an arbitrary number of input files (in FASTA format), one output directory and the sequence type of the input file(s). The sequence type determines if the available file(s) contain either nucleotide or protein sequences. Then, the actual validation process starts.
The file is parsed line by line, and is checked for the following:

1. Is an empty line between a header and an end of a sequence,
2. are empty lines or comment lines within the sequence itself,
3. does a comment line after a header occur,
4. is a sequence line longer than the preceding one, and
5. are all sequence headers unique?

If one of these errors appear, an appropriate error message is generated including the line, where the error occurred. Furthermore the newline type of the FASTA file is detected. Also, statistical data like

1. the number of total lines,

[2] `https://github.com/jwaldman/FastaValidator` (last accessed: 04/04/2014)
[3] `http://www.mybiosoftware.com/protein-sequence-analysis/9270` (last accessed: 04/04/2014)

2. the number of empty lines,

3. the count of comment lines,

4. the tally of sequences, and

5. the total number of sequence characters

is calculated and reported. Additionally, the IUPAC-format compliant formatting of each sequence character is checked. For DNA sequences, the GC content both in total and percent is calculated. Finally, a short listing of the file's sequence type, sequence identifiers and corresponding sequence lengths is given.

The result of all the validation steps are written out to a validation file in the output directory. The validation file's path is composed by the output directory and the name of the input file plus the ending '.valid'.

3.2.2 CorrectFasta

General Motivation

If the user has discovered that his FASTA file does not match the FASTA specifications, he usually has to adjust the corrupted FASTA file by hand. Thus until now, a FASTA corrector does not exist, the user requires a method solving that issue. `CorrectFasta` presents a newly automated way to produce accurate FASTA files.

Workflow

This tool takes the input file(s) (in FASTA format), the output directory, the sequence type, the width of each sequence line and optionally the end of line character as input parameters. If no end of line character was passed, the one of the entered file will be used. Each input file is read in line by line, and a corrected version of this line is directly written out again, facultatively the end of line character is also replaced. If the line was a non-unique header line, the current line number is added to the header in order to make it unique. All characters of a sequence line are parsed individually for the purpose of their IUPAC-format validation. Each character that is not conform is changed to either 'N' (when correcting DNA sequences) or 'X' (when correcting protein sequences). Also, it is made sure, that sequence lines have the specified length. Finally, a short report is printed out, displaying all the corrected errors.

3.2.3 IndexFasta

General Motivation

For a lot of bioinformatic programs, the user needs a reliable FASTA index, so that tools working with that index process the corresponding FASTA files correctly. Furthermore, a FASTA index represents an efficient way of navigating through a FASTA file. Although there exists a tool capable of indexing a FASTA file, namely `SAMtools' faidx`[LHW+09], it has the huge disadvantage, that does not validate the input file. Therefore `faidx`, when working on a corrupted FASTA file, will likely create a wrong FASTA index which would lead to false results. `IndexFasta` does not only create a FASTA index, but validates the FASTA file on the fly (compare methods of ValidateFasta, see Chapter 3.2.1) and stops if an error emerges so that no incorrect FASTA index is generated allowing the user to first correct the FASTA file before continuing with the index generation.

Workflow

The command-line arguments of `IndexFasta` are an input file (in FASTA format) and an output file. For each FASTA entry this module creates a FASTA index (see Chapter 2.2.2, `IndexFasta` produces the same FASTA index as `faidx`, except that from each sequence header not only the sequence identifier but the whole sequence header is taken for the first column of the resulting FASTA index) and writes them out to 'output_file.fai', indicating that the created file is a FASTA index. Note that IndexFasta does not validate each single sequence character to save running time.

3.2.4 ExtractFasta

General Motivation

Often scientists want to take a closer look at only particular sequences in order to analyse them in more detail. Sorting the requested sequences could ease that task. Until now, there are very few tools which have integrated such functionality. `FaBox`[Vil07], an online FASTA sequence toolbox capable of manipulating and processing FASTA files is one of them. But `FaBox` is only able to extract sequences by a list of headers or fuzzy matching in a non-ordered way. It is written in *PHP* and needs an online webserver in order to be executable. `ExtractFasta` is platform independent and outputs (optionally sorted) sequences matched by a regular expression pattern.

Workflow

The arguments required for running this tool are the path to the FASTA file, the path to the FASTA index and the name of output file. One must specify the regular expression pattern for the sequence headers one wants to look for and/or the sorting function (sorting by sequence length or lexicographically by header and ascending or descending). ExtractFasta reads in the FASTA indices into a list. FASTA indices not matching the given pattern are removed from the list. If the sort option was chosen, the list is then rearranged. The sorted FASTA indices allow a fast location of their corresponding sequences in a FASTA file. These sequences are then written to a new FASTA file 'output_file.extract.fasta'.

3.2.5 TabulateFasta

General Motivation

Some toolkits require a tab-delimited FASTA file. Very few tools are able to perform that task (one tool is `FASTX-Toolkit's fasta_formatter`). `TabulateFasta` was implemented in `TOPAS` because it completes the attempt of `TOPAS` to provide functions which format sequence data. This also means, that the user does not need to install some additional program in order to tabulate a FASTA file.

Workflow

`TabulateFasta` reads in a given FASTA file line by line and directly writes these lines into the specified output file. The formatting for each sequence in the file is done as follows:

```
sequence_header \t sequence_content
```

The line consists first of all of the sequence header, then a tabulator, which is followed by the sequence content. Note, that the sequence content is built out of all the sequence lines of the corresponding sequence. What was divided into several lines before is now relocated into one single line. The output file contains all the sequences from the input file, which are now in a tab-separated format. Comment lines and empty lines are omitted.

3.2.6 ValidateFastq

General Motivation

FASTQ files stand at the beginning of almost each NGS pipeline. Therefore it is crucial that these files are not corrupted or broken, or contain invalid information. Thus a validation program for FASTQ files is needed. To my knowledge, until now, the only tool able to validate a FASTQ file is `fastQValidator`[4]. Because its validation functions are not sufficient in some cases (e.g. it can't distinguish multi-line files from single-line ones), it is written in C++ and requires additional C++ libraries, a platform independent alternative is needed. Here `ValidateFastq` steps in.

Workflow

`ValidateFastq` requires two input parameters: An input file (in FASTQ format) and the path to where the resulting report should be written. Optionally the user can decide if the uniqueness of the sequence identifiers and the quality identifiers should be verified. If this option is activated, the memory consumption of `ValidateFastq` will be relatively high. Therefore this option is only recommended on machines with efficient RAM. `ValidateFastq` reads in the given FASTQ file line by line and validates the following:

1. Is it a multi-line FASTQ file,

2. is the length of the quality sequence line the same as the length of the sequence string, and

3. are the sequence identifiers and the quality identifiers unique?

If an error occurs, an error message is generated, containing a short description of the error and the line number where the error was found. Additionally, the total number of lines, the newline type, the total reads, the different read lengths, the overall mean read quality, the highest and lowest read quality, and the highest and lowest base quality are calculated. `ValidateFastq` can also detect the encoding of the reads. It is able to distinguish between Sanger, Solexa/Illumina 1.0, Illumina 1.3, Illumina 1.5 and Illumina 1.8.

[4] `http://genome.sph.umich.edu/wiki/FastQValidator` (last accessed: 04/04/2014)

3.2.7 FormatFastq

General Motivation

Some tools produce FASTQ files with multi-lines (e.g. `SAMtools.pl`[5], a helper script for `SAMtools`, that filters SNPs and short INDELS, which can also optionally output FASTQ files). However, only few programs support this, whereas other require FASTQ files in the single-line format. Hence `FormatFastq` was implemented, being nearly the only tool able to transform a multi-line FASTQ file into a single-line form and vice versa (`seqtk`[6], a set of tools for processing sequences files in the FASTA or FASTQ format, is also able to perform that task).

Workflow

This module takes the path of the FASTQ file, the path to where the formatted file should be written and optionally the length of the resulting sequence line(s) (how long it/they should be after formatting) as input arguments. If no length is specified, the sequence/quality lines will be in single-line form. `FormatFastq` then reads the input file read for read. As soon as one read was read in, the sequence line(s) and quality line(s) are formatted as specified and directly written out to the output file.

3.2.8 ValidateGFF3

General Motivation

The number of inconsistent GFF3 files on the internet is large and growing. Typical problems reach from simple field formatting errors to non unique feature *IDs* in the file. These and other errors lead to incorrect or non existing feature relationships which then might result in the wrong annotation of NGS data or malfunctioning programs. Therefore a GFF3 validator is needed. Currently there exist only two online-tools that are able to validate a GFF3 file: `GFF3 Validator`[7] and `GenomeTools'` [GSK13] `GFF3 online validator`[8]. So a platform independent implementation that validates GFF3 files efficiently is needed.

[5] `https://github.com/lh3/samtools-legacy/blob/master/misc/samtools.pl` (last accessed: 04/03/2014)

[6] `https://github.com/lh3/seqtk` (last accessed 04/03/2014)

[7] `http://modencode.oicr.on.ca/cgi-bin/validate_gff3_online` (last accessed: 04/03/2014)

[8] `http://genometools.org/cgi-bin/gff3validator.cgi` (last accessed: 04/03/2014)

Workflow

The command line arguments of `ValidateGFF3` are the input file (in GFF3 format) and the path to the output file. Also a parameter can be set determining if all the calculated multi-features in the GFF3 file should be reported. In the first validation step, `ValidateGFF3` reads in the given file line by line and checks if it has nine tab-delimited fields. If this is the case, then each field is validated for its correct format as specified in the section "Parsing and Format Validation" of the GFF3 validation documentation[9]. Only the positively validated GFF3 entries are loaded into a GFF3 entry list. In the next step the uniqueness of the attribute tag *ID* of all GFF3 entries is verified even taking multi-features into account (though they might not be reported as chosen). Only the positively validated entries are left for the part of relationship validation. `ValidateGFF3` is able to verify the relationships of GFF3 entries of the types *region, gene, transcript, mRNA, tRNA, rRNA, ncRNA, exon, CDS, five_prime_UTR* and *three_prime_UTR*. Finally, `ValidateGFF3` produces a short validation report about the GFF3 file containing the lines and error messages of the negatively validated GFF3 entries. This report is written as the output file.

3.2.9 SortGFF3

General Motivation

The user wants a sorted GFF3 file because, *a*) a sorted GFF3 is much better human readable than an unsorted one, and *b*) several programs (e.g `Integrative Genomics Viewer`[TRM12], (`IGV`), a visualizer of integrative genomic datasets like NGS data and genomic annotations) require a sorted GFF3 in order to function efficiently and/or correctly. Often functions for the sorting of a GFF3 file are deeply integrated into existing software modules. Thus a simple, efficient and open-source sorting tool is needed. Here `SortGFF3` steps in.

Workflow

`SortGFF3` takes two input parameters: The GFF3 file to sort and the output path, where the sorted file should be written to. The module reads all the GFF3 entries into a list which is sorted by *seqid* and *start/end*. Then, the header of the given GFF3 file and the sorted list is written to the path of the output file.

[9] `http://modencode.oicr.on.ca/validate_gff3_online/validate_gff3.html` (last accessed: 04/03/2014)

3.2.10 FilterGFF3

General Motivation

Often one is in need of only specific GFF3 entries and/or their sequences but programs able to perform that task are very rare. There does exist a toolkit (`pltools`[10], a command line toolkit capable of processing GFF3 and GTF[11] files) which has integrated a function to filter a GFF3 file, but `pltools' gff3-filter utility` is not able to also write out the sequences of the found GFF3 entries. However, `FilterGFF3` represents a new method to filter GFF3 files and additionally extract the resulting sequences from the corresponding FASTA file.

Workflow

Required arguments are a GFF3 file and the path to the output file. Furthermore, at least one gene locus range has to be specified in the following format: *SEQID:START-END*. Either a file, containing gene loci ranges line by line, or manually entered gene loci ranges have to be entered. For refinement of the search the GFF3 file can be additionally filtered by *source, type, score, strand, phase* and *attributes*. `FilterGFF3` reads in the GFF3 file (and optionally the file containing gene loci ranges) in a list of GFF3 entries, then applies the chosen filter options to each entry. The resulting list of GFF3 entries is written out to 'output_file' (in GFF3 format), complemented by the header of the given GFF3 file.

If a FASTA file and its corresponding FASTA index were entered as command-line arguments, of every found GFF3 entrie the corresponding sequence is efficiently located in the FASTA file with the FASTA index. Then, the sequences are written out to the path of the output file in FASTA format. The sequence headers of the written sequences are built out from the GFF3 entries: *seqid*|*source*|*type*|*start*|*end*|*score*|*phase*|*strand*|*attributes*. Only the tags *ID, Parent* and *locus_tag* are included in *attributes* to preserve a reasonable length for the header.

3.2.11 IndexVCF

General Motivation

VCF files can grow incredibly big (over 100 GB). Often they have to be filtered and/or annotated. Therefore, an index structure is needed for the efficient parsing of a VCF file. There are tools performing that task (`SAMtools' tabix`)

[10] `https://github.com/mamarjan/gff3-pltools` (last accessed: 04/03/2014)
[11] `http://mblab.wustl.edu/GTF22.html`

but they are badly documented, have a lot of library dependencies or are deeply integrated into existing software packages. So, an alternative version of indexing a VCF file is needed. This task is fulfilled by `IndexVCF`.

Workflow

`IndexVCF` takes the path of the VCF file to be indexed as an input parameter. The user can also decide, after which number of lines a new index should be created. It is important that the VCF file to index is sorted by *CHROM* and *POS*. Note that `IndexVCF` does not check that. `IndexVCF` reads in the given VCF file line by line. At the first occuring VCF line, the first VCF index is created. After the specified number of lines are passed, another index is built and so forth. If the *CHROM* in the VCF file changes, a new index is generated, too. When the end of the file is reached, all generated VCF indices are written to one output file naming 'input_file.vai'.

3.2.12 FilterVCF

General Motivation

Tools that are able to filter VCF entries are rare, the most important ones are `VCFtools' vcf-filter`, `GATK's VariantFiltration`[MHB+10;DBP+11;VdACC+13] and `SnpSifts`[CPC+12]`, Filter`. `GATK` is a software suit, that provides a wide variety of tools for the analysis of NGS data. `SnpSift` is a suite of command line tools for manipulating VCF files. `FilterVCF` represents a light and efficient implementation of dealing with VCF files.

Workflow

The command-line parameters required are the input file (in sorted VCF format), a corresponding VCF index, the path of the output file and either the path of a file with a list of chromosome ranges (in *CHROM:START-END* format) or a single chromosome range directly entered in the command-line. Optionally ids which should be found in the VCF file can be chosen by either a file containing a list of ids or a single id. Additionally, the extraction of either SNPs or INDELs can be specified. `FilterVCF` reads in the VCF indices, then the indices which do not match in position and chromosome as specified with the chromosome ranges are filtered out. After that, for each index left the corresponding VCF entries are read in and filtered optionally by ids and/or SNPs/INDELs. Next, the resulting entries are written to a new VCF file (with file path 'output_path'), which has the same header as the input file.

3.2.13 AnnotateVCF

General Motivation

In order to annotate VCF files, as far as I know, only two tools exist, none of them written in `Java`. These are (`VCFtools' vcf-annotate` and `SnpSifts' Annotate&AnnMem`). AnnotateVCF was created to offer an alternative. It also completes the set of tools to manipulate VCF files.

Workflow

`AnnotateVCF` takes a VCF file, the path to the output file and the GFF3 file from which the annotations should be extracted as input parameters. A possible other argument is a map file which contains a mapping from VCF *CHROM* to GFF3 *seqid*. This map file is necessary if the *CHROM* of the entered VCF and the *seqid* of the entered GFF3 do not match together. This module first reads in the GFF3 file and loads the entries into a GFF3 entry list. After that, the optionally specified map file is read in. Then, each single VCF entry is read in, annotated and directly written out again. The annotation step is done ss follows: The GFF3 file is checked for the chromosomal position of the VCF entry. Only the attribute tags *ID, Name, Parent* and *locus_tag* and their corresponding values are extracted from the found GFF3 entries. For the annotation, a new field *ANNOTATION* in the VCF file, directly after the *INFO* field is created. In this new field the resulting annotation is put. It has the following format: The annotation starts with a annotation entry number, representing the number of annotations for this VCF entry. This is followed by an equals sign. Behind, the attribute tags and their correlated values are situated, all separated by a ','. The tags and values of these attributes again are separated by a ':'. An example of such an annotation would be:

Entry_1=ID:mrna1,Name:sonic,Parent:gene1,locus_tag:sonic

Chapter 4

Results

The first part of this chapter presents the runtime performance of each module of `TOPAS`. These results are compared with the runtime performance of alternative tools. In the second part of this chapter, it is shown, how `TOPAS` could be integrated into an existing pipeline, in this case being `EAGER`[1]. The `EAGER` pipeline was designed to efficiently reconstruct ancient human genomes.

4.1 Performance

All programs ran on a machine with four Quad-Core AMD Opteron(tm) 8354@2,2Ghz processors and 128GB memory installed. The operation system was Ubuntu 12.04.4 LTS with kernel version 3.8.0-35-generic x86_64. The Java(TM) SE Runtime Environment had version 1.7.0_51-b13. The modules of `TOPAS` normally only make use of one core, because multi-threading is not implemented for any module, yet.

The runtime of every tool was evaluated with GNU's `time`[2] command. The execution time of every module was monitored for 10 independent runs. For the final runtime evaluation the mean of these runs was taken. For each task four data sets of different sizes were chosen, in order to get a detailed view of the scalability of each module.

4.1.1 FASTA Processing Modules

All FASTA processing modules were applied to four different FASTA files, originating from the human reference genome version *GRCh_37.10*. The used FASTA files were *chromosome MT*, *chromosome 21*, *chromosome 1* and all the chromosomes of *GRCh_37.10* (which will now be referred to as '*complete*') (see

[1] A. Peltzer. EAGER. Unpublished, personal communication. 2013

[2] `http://man7.org/linux/man-pages/man1/time.1.html` (last accessed: 04/15/2014)

Table 4.1). The size of the sequences in the FASTA files range from $1.6 \cdot 10^4$ nucleotides to $3.1 \cdot 10^9$ nucleotides. `ValidateFasta` and `CorrectFasta` were executed with 2GB of RAM. For `CorrectFasta` the parameter *width*, specifying the length of each sequences, was set to 80. `IndexFasta` and `TabulateFasta` were executed with 1GB and `ExtractFasta` with 4 GB of RAM. Table 4.2 gives an overview of the performance results of `TOPAS'` FASTA processing modules of `TOPAS`. The runtime results are compared to those of similar tools (if available).

Table 4.1: Table listing the FASTA files, that have been used to evaluate the performance of the FASTA processing modules of `TOPAS` and their corresponding alternative tools. All FASTA files originate from the human genome version *GRCh_37.10*.

File Content	***Size in MB***	***# Lines in Fasta***	***Sequence Size***
chrMT	0.02	238	16,569
chr21	48.80	687,571	48,129,895
chr1	252.80	3,560,725	249,250,621
all chromosomes	2994.50	44,224,234	3,095,693,981

Table 4.2: Results of the runtime performance (in seconds) of `TOPAS` FASTA processing modules and similarly working tools.

Tool/Runtime in seconds	***chrMT***	***chr21***	***chr1***	***complete***
`TOPAS' ValidateFasta`	0.3	3.0	11.7	134.6
`FastaValidator`	0.3	15.7	69.6	1097.1
`TOPAS' CorrectFasta`	0.3	10.6	48.3	627.5
`TOPAS' IndexFasta`	0.2	1.9	5.4	54.3
`faidx`	0.1	0.6	3.1	37.3
`TOPAS' ExtractFasta`	0.2	5.1	23.3	303.7
`FaBox`[3]	-	-	-	-
`TOPAS' TabulateFasta`	0.2	4.7	21.7	261.0
`fasta_formatter`	0.1	3.8	20.0	236.0

[3]Theoretically `FaBox` is able to extract sequences from a FASTA file but due to its sole online availability, an objective runtime measurement was not feasible.

ValidateFasta

`ValidateFasta` is five to eight times faster than `FastaValidator`, as can be seen in Figure 4.1, which displays the validated lines per second. One possible explanation can be attributed to an essential functional difference of these tools: `TOPAS' ValidateFasta` generates a report that is written to an output file, whereas `FastaValidator` writes out every validated FASTA line resulting in a huge disk consumption and dissipation. What can also be obtained is, that `ValidateFasta's` velocity increases with the size of the file. Possibly, due to the overhead of the `JVM`. In contrast, `FastaValidator's` validation speed does not vary. Both tools also seem to perform badly when processing a very small FASTA file. This is due to the possible measuring inaccuracies of very short timings. A sample output of *chromosome MT's* validation with `ValidateFasta` is attached in the appendix (see Figure A.1).

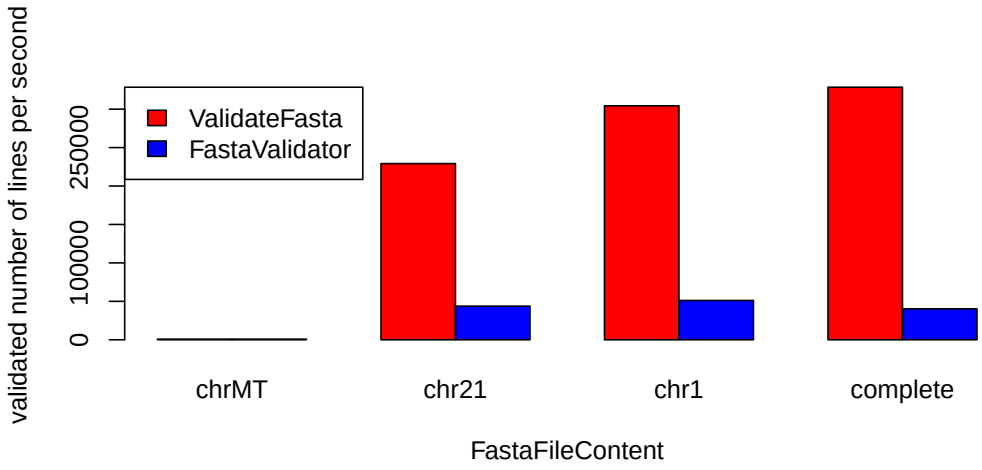

Figure 4.1: A bar plot of the runtime results of `ValidateFasta` and `Fasta Validator`. The x-axis shows the four different FASTA files, the tools were applied to. The y-axis displays the validated number of lines per second. The height of the bars presents the value for each program and FASTA file, respectively.

CorrectFasta

The runtime performance can be seen in Table 4.2. A sample console printout of `CorrectFasta` after the correction of the *chromosome MT* FASTA file is shown in Figure A.2.

IndexFasta

As can be seen in Figure 4.2, `IndexFasta` is slower than `faidx`. A reason stating that proposition could be, that the methods of `Java's` code library, reading a file line by line, are not fast enough to keep up with the C-implementation of `faidx`. One might suspect, that the validation function of `IndexFasta` slows the module down, but this was not the case. Though, `IndexFasta` performs obviously slower with small FASTA files than `faidx`, its scalability performance significantly increases with the size of the FASTA file, while `faidx` maintains the scalability performance over the three biggest FASTA files. A possible explanation for this behaviour of `IndexFasta` is the overhead of the `JVM`. The time for processing the *chromosome MT* file is extremely low. One explanation is, that measuring inaccuracies might occur with quick timings.

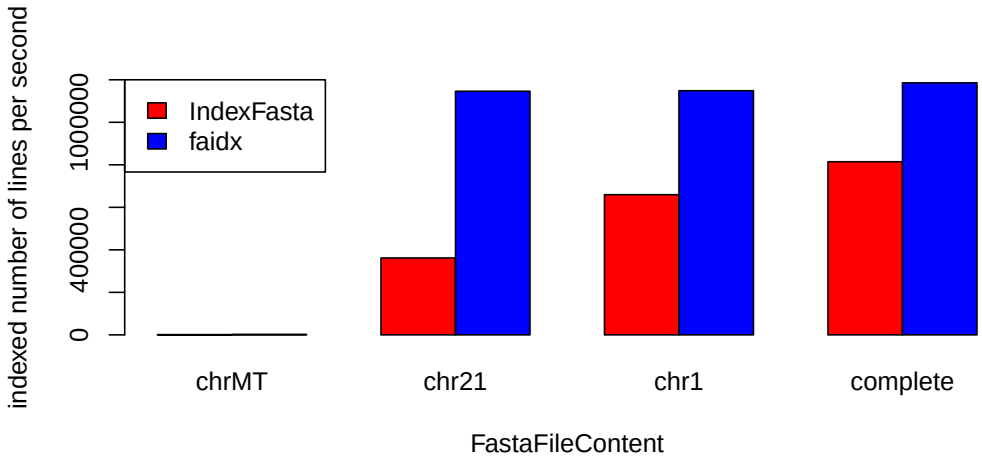

Figure 4.2: A bar plot of the runtime results of `IndexFasta` and `faidx`. The x-axis shows the four different FASTA files, the tools were applied to. The y-axis displays the indexed number of lines per second. The height of the bars presents the value for each program and FASTA file, respectively.

ExtractFasta

In each run, `ExtractFasta` had to extract all sequences contained in the given FASTA file, sort them in ascending order, and write them out to the output file. The runtime measurement of these tasks can be seen in Table 4.2. `FaBox` should also extract sequences from the given four FASTA files, but, as shown in Table 4.2, the runtimes were not leviable, because `FaBox` is a solely online working toolkit. This hindered objective runtime measurement on a local machine.

TabulateFasta

The performance results for `TabulateFasta` and `FASTX-Toolkit's fasta_formatter` are practically indifferent, what can be seen in Figure 4.3. Additionally, the two tools show a good scalability. When processing the smallest FASTA file, both tools seem to perform badly, but this can be attributed to potential measure errors for short time measurements.

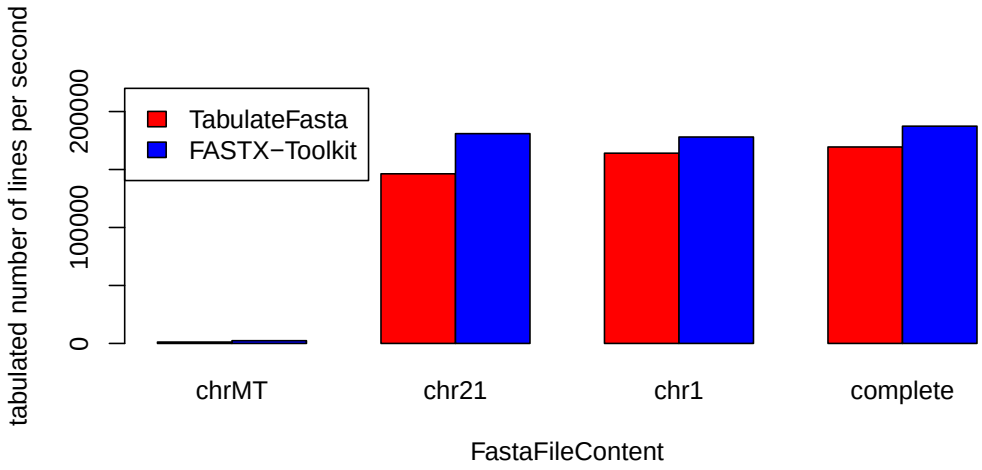

Figure 4.3: A bar plot of the runtime results of `TabulateFasta` and `FASTX-Toolkit's fasta_formatter`. The x-axis shows the four different FASTA files, the tools were applied to. The y-axis displays the tabulated number of lines per second. The height of the bars presents the value for each program and FASTA file, respectively.

4.1.2 FASTQ Processing Modules

The FASTQ processing tools were tested on four different FASTQ files. These four data files all originate from one FASTQ file, which was the result of a sequencing project of an ancient *Mycobacterium leprae*. From this ancient DNA containing FASTQ file, the first 10,000/100,000/1,000,000/10,000,000 reads were extracted and stored into four different FASTQ files, in the following referred to as '*leprae*_10^4', '*leprae*_10^5', '*leprae*_10^6' or '*leprae*_10^7'. These file sizes were picked out, in order to get a synopsis of the scalability of the executed tools (because of the logarithmic relationship of the sizes of the FASTQ files). An overview over the FASTQ data set is given in Table 4.3. All FASTQ files are in multi-line format. This implies, that each read in a FASTQ file is spread over multiple lines.

`ValidateFastq` was executed with 10GB of RAM. This big allocation of RAM was necessary, in order to make the validation of the uniqueness of the sequence and quality identifiers possible. `FormatFastq` started with 2GB of RAM. In Table 4.4 the performance results of the FASTQ processing modules of `TOPAS` are shown. For comparison reasons the runtime results of similar tools (if available) are also listed.

Table 4.3: FASTQ files that have been used to evaluate the runtime performance of the FASTQ processing modules of `TOPAS` and their corresponding alternative tools. All FASTQ files originate from the result of a sequencing project of an ancient *Mycobacterium leprae*.

File Content	***# Reads***	***# Lines in FASTQ***
*leprae*_10^4	10,000	100,000
*leprae*_10^5	100,000	1,000,000
*leprae*_10^6	1,000,000	10,000,000
*leprae*_10^7	10,000,000	100,000,000

Table 4.4: Results of the runtime performance (in seconds) of `TOPAS` FASTA processing modules and similarly working tools.

Tool/Runtime in seconds	***leprae***_10^4	***leprae***_10^5	***leprae***_10^6	***leprae***_10^7
`TOPAS' ValidateFastq`	0.7	2.3	13.9	160.0
`fastQValidator`	0.1	1.1	12.3	127.3
`TOPAS' FormatFastq`	0.8	5.3	44.4	374.9
`seqtk`	0.7	8.4	92.9	945.4

ValidateFastq

As can be seen in Figure 4.4, both `fastQValidator` and `TOPAS' ValidateFastq` validate a FASTQ file in practically the same time span. For smaller FASTQ files `ValidateFastq` seems to be slower, but this could be the case because the overhead of `Java's JVM` slows `ValidateFastq` down. Also, both tools show a good scalability. See Figure A.3 for a sample validation report of `ValidateFastq`.

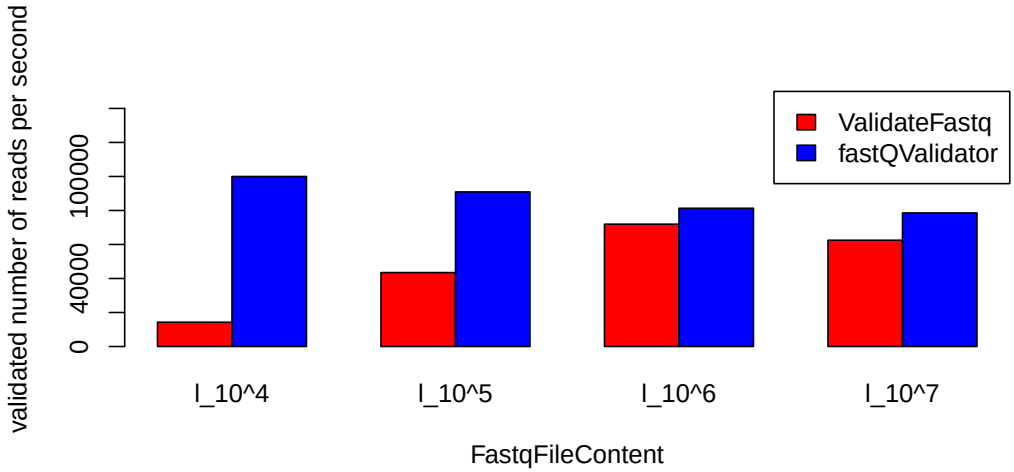

Figure 4.4: A bar plot of the runtime results of `TOPAS' ValidateFastq` and `fastQValidator`. The x-axis shows the four different FASTQ files, the tools were applied to. Their size increases exponentially (10^x declares the number of reads, being to the power of x). The y-axis displays the validated number of reads per second. The height of the bars presents the value for each program and FASTQ file, respectively.

FormatFastq

The two FASTQ formatting programs `FormatFastq` and `seqtk` had to convert each of the multi-line FASTQ files in Table 4.3 into single-line ones. As soon as `TOPAS' FormatFastq` overcomes the overhead of the `JVM`, it performs up to two times faster than `seqtk`, what Figure 4.5 shows. This can be explained with an efficient implementation of `TOPAS' FormatFastq`.

4.1.3 GFF3 Processing Modules

Four different unsorted GFF3 files were chosen as a dataset for the runtime evaluation for the GFF3 processing modules, all referring to the human genome version *GRCh_37.10*. The *GFF3_MT* file holds the whole genome annotation for the mitochondrial chromosome, *GFF3_21* the whole genome annotation for chromosome 21, *GFF3_1* the whole genome annotation for chromosome 1 and *GFF3_all* stores the genome annotation for all the chromosomes of *GRCh_37.10*. The file sizes range from 0.02 megabyte to 253.6 megabyte. A short summary of the GFF3 files can be obtained in Table 4.5. *GFF3_MT* was picked, because it contains the smallest genome annotation of a chromosome of *GRCh_37.10* and *GFF3_all* was selected, because it is the biggest GFF3 file of *GRCh_37.10*. *GFF3_21* and *GFF_1* were chosen, because they fit in between

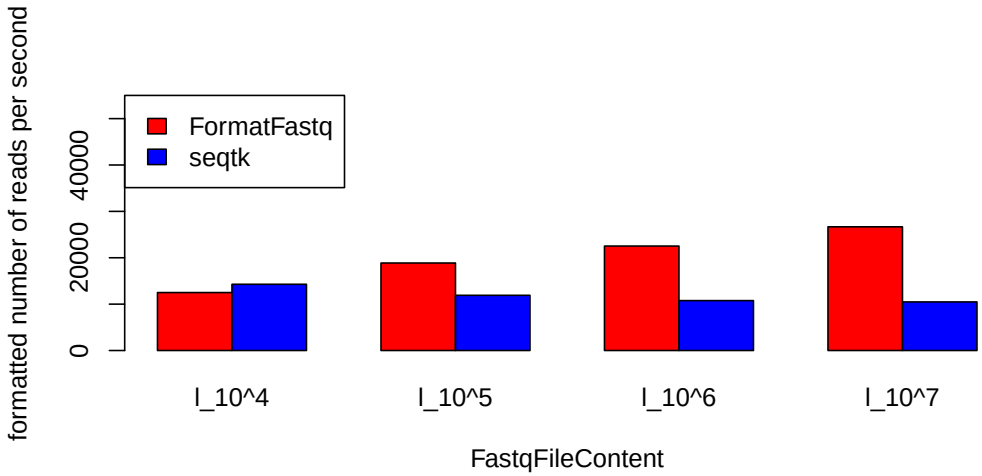

Figure 4.5: A bar plot of the runtime results of `TOPAS' FormatFastq` and `seqtk`. The x-axis shows the four different FASTQ files, the tools were applied to. Their size increases exponentially (10^x declares the number of reads, being to the power of x). The y-axis displays the formatted number of reads per second. The height of the bars presents the value for each program and FASTQ file, respectively.

the biggest and smallest GFF3 files of *GRCh_37.10*.

All three GFF3 processing modules of `TOPAS` were executed with 4GB of RAM. Their runtime performance can be obtained in Table 4.6, where the runtime results of alternative tools (if available) are shown.

Table 4.5: GFF3 files that have been used to evaluate the performance of the GFF3 processing modules of `TOPAS` and alternative tools. All unsorted GFF3 files originate from the human genome version *GRCh_37.10*.

File Content	***Size in MB***	***# Lines in GFF3***
GFF3_MT	0.02	104
GFF3_21	2.80	10,505
GFF3_1	22.60	86,254
GFF3_all	253.60	962,862

ValidateGFF3

The runtime measurement (see Table 4.6) of `GFF3 online validator` seems to indicate, that it validates GFF3 files much faster than `TOPAS'`

Table 4.6: Results of the runtime performance (in seconds) of `TOPAS'` GFF3 processing modules and similarly working tools.

Tool/Runtime in seconds	*GFF3_MT*	*GFF3_21*	*GFF3_1*	*GFF3_all*
`TOPAS' ValidateGFF3`	0.4	3.1	7.8	73.9
`GFF3 online validator`	0.02	0.2	0.03	0.2
`TOPAS' SortGFF3`	0.3	1.4	5.4	57.9
`gff3-sort utility`	0.1	0.3	2.4	55.6
`TOPAS' FilterGFF3`	0.3	0.9	2.6	10.6
`gff3-filter utility`	-	-	-	-

`ValidateGFF3`. It has to be pointed out here, that `GFF3 online validator` validates a GFF3 file line by line until an error is found. Then, it just stops, displaying this error. This actually happened with every GFF3 file `GFF3 online validator` had to process. Therefore, its runtime results are not comparable with the performance results of `TOPAS' ValidateGFF3`, which always scans the whole file for errors. A sample output of `TOPAS' ValidateGFF3` of the validated *GFF3_MT* is displayed in Figure A.4 in the appendix.

SortGFF3

`TOPAS' SortGFF3's` and `GFF3-sort utility's` runtimes are practically the same, as can be obtained in Table 4.6. Nonetheless, `Gff3-sort utility` sorts small files faster than `TOPAS' SortGFF3`. One reason stating that proposition is the overhead of `Java's JVM`. It has to be remarked, that `Gff3-sort utlitiy` does not only sort the GFF3 file by *seqid/start/end* but also by feature membership.

FilterGFF3

All the four GFF3 files had to be filtered by type (only 'gene' allowed) and strand (only '+' allowed). The runtime results of `TOPAS' FilterGFF3` performing this task is listed in Table 4.6. There are no performance measurements for `Gff3-filter utility`, because `Gff3-pltools` version 0.3.0 did not have this function yet and version 0.4.0 did not compile on the machine.

4.1.4 VCF Processing Modules

The VCF test set was built out of a 224,921,831 lines VCF file (prospectively referred to as '*vcf_large*'), containing 224,921,796 positions ranging from 10,000 to 249,240,621 of the human chromosome 1 of *GRCh_37.10*. Out of this

big VCF file three smaller ones were created as follows: '*vcf_tiny*' consists of the first 224,921 lines of *vcf_large*, '*vcf_small*' of the first 2,249,218 lines and '*vcf_big*' of the first 22,492,183 lines of *vcf_large*. This data set named '*idfil*' was designed to gain a logarithmic scale of the runtime evaluation of the VCF Processing Modules. It has to be mentioned, that this data set will only be processed by VCF index/filter tools, for the VCF annotate tools, another data set (data set '*ann*') was created: From all the four VCF files in the *idfil* data set only the SNPs were extracted. The four resulting VCF files, only containing SNPs, form the *ann* data set. The new VCF files are named 'vcf_large_ann', 'vcf_big_ann', 'vcf_small_ann' and 'vcf_tiny_ann'. Both VCF data sets are listed in Table 4.7. The runtime results of all VCF processing modules are shown in Table 4.8. `IndexVCF` was executed with 2GB of RAM. `FilterVCF` and `AnnotateVCF` were both started with 4GB of RAM. The performance results of similar tools are also listed.

Table 4.7: The VCF files, that have been used to evaluate the performance of the VCF processing modules of `TOPAS` are shown.

File Content	***Size in MB***	***Number of Lines***
vcf_tiny_idfil	23.2	224,921
vcf_small_idfil	365.8	22,49,218
vcf_big_idfil	4537.1	22,492,183
vcf_large_idfil	47135.7	224,921,831
vcf_tiny_ann	0.03	202
vcf_small_ann	1.1	3,375
vcf_big_ann	12.5	35,941
vcf_large_ann	108.0	298,913

Table 4.8: Table listing the results of the runtime performance (in seconds) of `TOPAS` VCF processing modules and similarly working tools.

Tool/Runtime in seconds	***vcf_tiny/ _ann***	***vcf_small/ _ann***	***vcf_big/ _ann***	***vcf_large/ _ann***
`TOPAS' IndexVCF`	1.50	8.0	80.3	814.9
`BGZIP/TABIX`	0.84	13.7	169.9	1741.4
`TOPAS' FilterVCF`	2.30	14.0	132.2	1325.8
`VCFTools`	3.80	38.9	414.2	3625.6
`TOPAS' AnnotateVCF`	2.10	3.2	7.5	42.2
`VCFTools' annotate`	0.04	0.4	4.3	42.2

IndexVCF

The time to index the files of dataset *idfil* of `TOPAS' IndexVCF` and `BGZIP/TABIX` can be read from Table 4.8. As illustrated in Figure 4.6, `IndexVCF` is up to two times faster than `BGZIP/TABIX` (except for the smallest VCF file, probably due to the JVM overhead). One reason for this is, that the VCF file to be indexed is first compressed by `BGZIP`. The `BGZIPPED` file then can be indexed by `TABIX`. This procedure takes additional time in comparison to `TOPAS' IndexVCF's` indexing method.

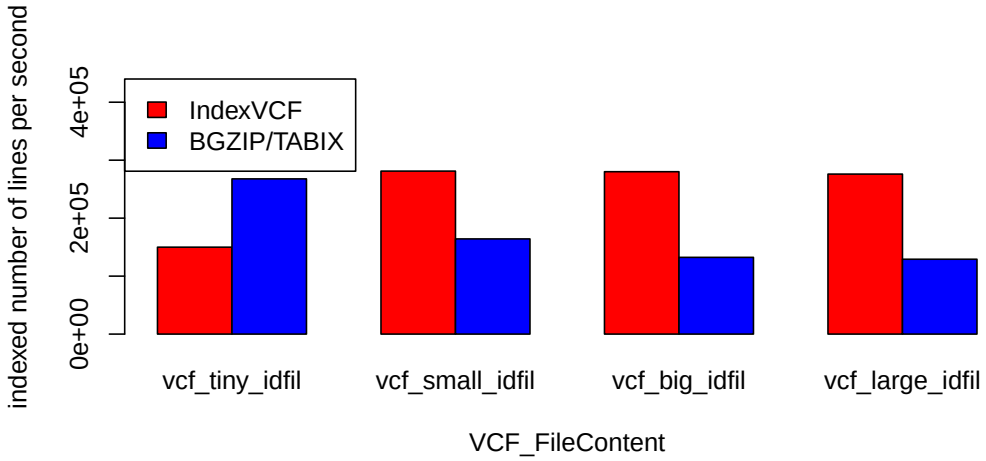

Figure 4.6: A bar plot of the runtime results of `TOPAS' IndexVCF` and `BGZIP/TABIX`. The x-axis shows the four different VCF files, the tools were applied to. Their size increases exponentially. The y-axis displays the indexed number of lines per second. The height of the bars presents the value for each program and VCF file, respectively.

FilterVCF

The time measurements of the filtering processes for both `TOPAS' FilterVCF` and `VCFTools` are listed in Table 4.8. It can be seen, that `VCFTools` performs up to three times slower than `TOPAS' FilterVCF`. This is also observable in Figure 4.7. One proposition stating that issue is `TOPAS' FilterVCF's` lightweight implementation works very efficiently.

AnnotateVCF

The annotation time of both `TOPAS' AnnotateVCF` and `VCFTools' annotate` is shown in Table 4.8. It shows, that `VCFTools' annotate` performs faster for

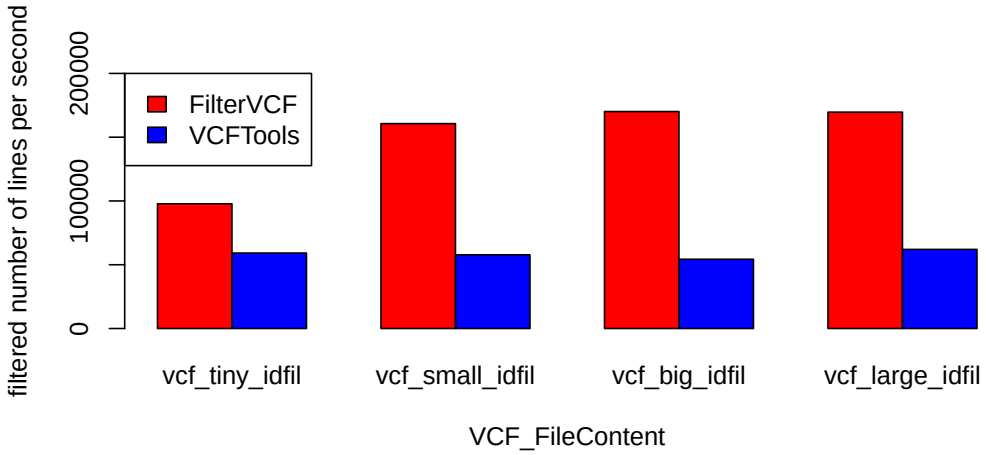

Figure 4.7: A bar plot of the runtime results of `TOPAS' FilterVCF` and `VCFTools`. The x-axis shows the four different VCF files, the tools were applied to. Their size increases exponentially. The y-axis displays the filtered number of lines per second. The height of the bars presents the value for each program and VCF file, respectively.

small files, as can be seen in Figure 4.8. One reason for this might be the `Java` overhead again.

4.2 Application of TOPAS in an existing Pipeline

In this section a number of `TOPAS'` practical usage possibilities are demonstrated. It is revealed, that `TOPAS` can interfere in the `EAGER` pipeline, both extending and improving the pipeline's features. The `EAGER` pipeline was designed in order to efficiently reconstruct ancient human genomes. It is a program written in Java, that regulates the ordered execution of a lot of individual/separate NGS tools, which were formerly loosely connected by scripts. `EAGER's` main pipeline tools are presented in short and it is shown, where `TOPAS` can step in the pipeline: The raw (ancient) sequence data, usually being output by a sequencing machine in FASTQ format, is quality controlled by `FastQC`[4]. `FastQC` produces a short statistical report and plots about the quality of such NGS data. However, `FastQC` does not validate the FASTQ files it processes. `TOPAS' FastqValidator` can undertake that task before the execution of `FastQC` preserving the pipeline from crashing or producing wrong

[4] `http://www.bioinformatics.babraham.ac.uk/projects/fastqc/` (last accessed: 04/22/2014)

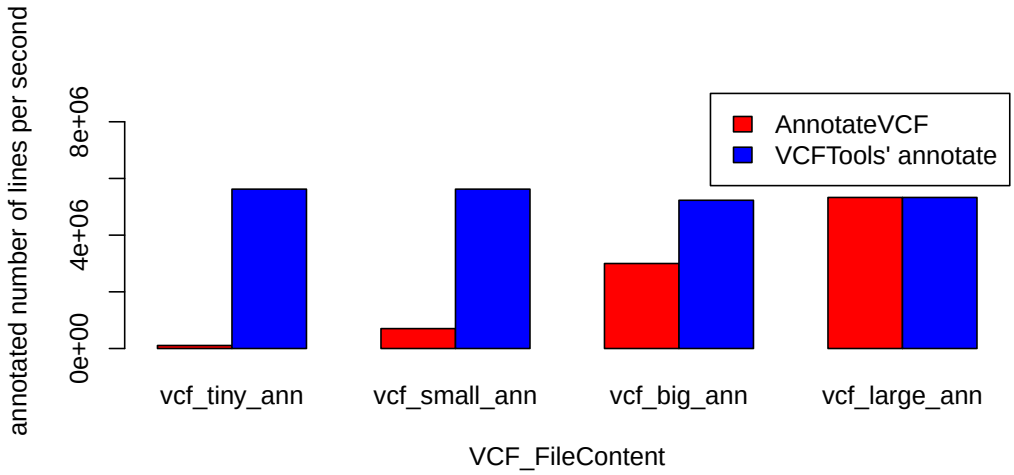

Figure 4.8: A bar plot of the runtime results of `TOPAS' AnnotateVCF` and `VCFTools' annotate`. The x-axis shows the four different VCF files, the tools were applied to. The y-axis displays the annotated number of lines per second. The height of the bars present that value for each program and VCF file, respectively.

results. This states an important extension of `EAGER`, as corrupted FASTQ files can be detected very early in the pipeline. Then, if quality flaws were detected by `FastQC`, they are corrected with the application of the `FASTX-Toolkit`. The raw read data, which should now be in good condition, is mapped to a reference genome (stored in the FASTA file format) by the `Burrows Wheeler Aligner`[LD10] (`BWA`). Here an error-free FASTA file is needed to assure, that the reads are aligned to the right positions of the reference genome. That is the task of `TOPAS' FastaValidator`. With the interference of this module before the mapping with `BWA`, the reference genome file is validated. If errors are detected, the reference genome file can be corrected with `TOPAS' CorrectFasta`. Also, `TOPAS' IndexFasta` could index the FASTA file in order to enable a further analysis exceeding this pipeline. After the mapping step, possible multi aligned reads are removed with `SAMtools' rmdup`. Then, the mapping quality is evaluated with `QualiMap`[GAOC+22], a toolkit capable of producing statistical data and plots about the alignment quality of mapping data. With `MapDMG`[GBH05] (a tool, that is able to detect nucleotide misincorporations and fragmentation patterns), the progress of the degeneration of the ancient input data is calculated. If the result of `MapDMG` allows a continuing of the pipeline, then, in the analysis step, `GATK`[MHB+10;DBP+11;VdACC+13] (a set of tools, capable of analysing NGS data, mostly focusing on data quality revision, variant discovery and genotyping) is applied to the mapping data for SNP calling. The resulting VCF files can be indexed with `TOPAS' IndexVCF`.

This allows TOPAS' FilterVCF a fast navigation and therefore filtering of the indexed VCF file. For deeper analysis, the filtered VCF file is annotated, usually by the use of a GFF3 file. But before this GFF3 file is employed by TOPAS' AnnotateVCF for the annotation process, it should be checked by TOPAS' ValidateGFF3. If the validation result is positive, AnnotateVCF can be run with the filtered VCF file and the validated GFF3 file.

Hence, TOPAS improves EAGER with useful extensions, presenting a feature enrichment of the pipeline. Particular tools of EAGER depend on the output of other tools, but until now, the integrity of this output is not checked. Here, TOPAS is able to interfere as a validating tool, in order to secure a even more hitch-free working of the pipeline.

Chapter 5

Discussion and Outlook

5.1 Discussion

As part of this thesis, the new `Java` program suite `TOPAS` was implemented. In this discussion, the implementation of `TOPAS` is analysed first. This is followed by a statement about how good `TOPAS` is executable/applicable in general and in comparison to other tools. Then, the features of `TOPAS` and of competing software suits are compared and analysed. Finally, the runtime results of `TOPAS` are discussed briefly.

There are many good reasons that justify `TOPAS'` implementation in the `Java` programming language, despite the fact that `Java's JVM` consumes a trifle amount of resources (as the runtime results indicate, this does not practically influence the working velocity my toolkit in respect to other NGS toolkits): For the execution of a `Java` program, none but a `JVM` is required. Additionally, all libraries needed by an executing `Java` program are always included in the `Java` binaries. These `Java` features make `TOPAS` a platform independent set of tools. On the contrary, most of the existing NGS software for instance is written in C, C++, D or other programming languages. Thereby, they are not platform independent, hence their error-free execution relies on individual compilers, libraries or other dependencies. This can lead to a high installation effort, which `TOPAS` does not have.

Furthermore, `Java` allows `TOPAS'` code structure to be organized in a hierarchical, abstract and modular way. This brings about many advantages: The abstractive code design of the toolkit facilitates the reusing of parts of the code, so that duplications are prevented. With the modular code structure programming tasks, algorithms and problems can be broken into smaller pieces. This simplifies the code, allowing the software developer to modify or extend it, exchange modules and fix bugs (good traceability with debugger) more easily. Additionally, the implementation sticks to the software development principle *information hiding*. Moreover, the hierarchical architecture

divides TOPAS' code in logical packages. All the mentioned aspects above contribute to a good human readable code. Note that TOPAS' code is likewise well documented, maintaining the porting of modules/parts/pieces of TOPAS into other tools. However, this does often not apply to the source code of existing NGS toolkits. As my experiences with these tools have revealed, the tools' source code is frequently not available, making a debugging impossible. Executing one of these tools, the user does not have access to detailed information about what exactly happens when the tool is running. In addition, some tools are available and executable solely online, again reducing the user's control over these tools, partly because he has no insight in the code and documentations are rare. Furthermore, local data has to be uploaded for processing and then the result has to be downloaded again. This takes a huge amount of time. To state in general, data should not be brought to the program, but the program should be brought to the data, so that the user can decide, when, were, how and on which machine a program is executed, preventing him from laborious working steps. Due to TOPAS high portability, none of these disadvantages arise here. But even if program suits are executed on a local machine, they give none or insufficient feedback about the current execution status and runtime issues, like the abortion of a program without any or unsatisfying (error) messages (aggravated debugging). By contrast, TOPAS always prints out the current execution status. The modular Java source code my tool enables an efficient debugging process. What also has to be mentioned here is that programs suits in general should not only be well executable but should also be easily accessible. Generally, if a toolkit is well documented, its execution and/or application might still be complicated, due to a difficult installation process, a bulky user interface, and so forth. Also, often-times, the command-line parameter settings of NGS tools are not well communicated. However, for each of TOPAS' modules a corresponding manual exists explaining the invocation of the module. Therefore, they can easily be accessed via the command-line.

Another issue that software packages have to master is a good applicability. An application example of TOPAS is its good integrability into existing pipelines, as was shown in section 4.2. It would be possible, that not only modules of TOPAS are integrated into a pipeline, but also parts of the code itself could be ported into that pipeline. This is supported by the abstract, modular and well documented code allowing a fast induction of it. Then, TOPAS can be easily adjusted for a user's individual purposes within few hours. Only a little number of NGS software suits give the user such power over the software code (single tools of large frameworks are often deeply integrated into its code, preventing its export/manipulation). This makes TOPAS an attractive user and software developer oriented piece of software. What also has been shown is that the modules needs relatively little RAM (at maximum 4GB, only ValidateFastq needs more) in order to be executable, so it could even be run

on a common notebook/desktop pc.

The application fields of a large number of software packages are limited, respectively focusing on particular application areas only. This individual orientation maintains a relatively large (and complex) tool/method library specialised for specific tasks only. But other problems might arise: It might be difficult to keep an overview over the whole library. Furthermore, the data of different tools might not be compatible with each other, for example because the data is corrupted or stored in the wrong file format. Additionally, the tools rarely validate the data they are working with. Here `TOPAS` steps in: It is able to validate, format and transform annotation and sequence data, working as an interface between these tools. In contrast to other NGS data processing programs, `TOPAS` offers not only several functions for one particular area of application, but provides a various number of modules, which can be categorized into four different application fields: FASTA processing modules, FASTQ processing modules, GFF3 processing modules and VCF processing modules. Naturally, `TOPAS` does not have the comprehensive functionality of some NGS packages, yet. They provide further useful features which do not appear in my implementation. Furthermore, just a few methods of alternative tools are more mature. Even if my toolkit does not have these features, it still supplies various functions in one software suit having no dependencies. However, with `TOPAS'` code library missing functions could be implemented with low effort.

What also has to be considered is, that existing software solutions are partially not well-engineered. For example, when validating data files, some tools check the input file for errors line by line, stopping once an error occurs. Assuming the validated file contains 1000 errors, then the user would have to run the validation tool 1000 times (and correct it 1000 times). Sometimes, these tools also write the already validated lines out to a new file. If errors where found in the file to validate, then the user is left with two corrupted and/or incomplete files. A further disadvantage of this validation method is, that the generation of the new (validated) file requires additional computing time and disk space. With `TOPAS` these inconveniences are obsolete: `TOPAS` always scans the whole file to validate first, and then generates a detailed error report. Another example of an engineering nuisance is, that NGS utilities often require a compressed counterpart of the file they should actually process. The generation of such a file consumes (unnecessary) time and disk space. `TOPAS` does not have such features integrated.

As was shown in the results, `TOPAS` does not solely provide improved methods for known problems, but it introduces several new features. Until now, negatively validated FASTA files had to be corrected by hand, because existing programs were not able to perform that task. Here my toolkit serves a new developed module, `CorrectFasta`, that is capable of correcting corrupted FASTA files stating an easement for the user, because he would have to do this

task manually. With the development of `IndexVCF` `TOPAS` introduces a new way of how to index VCF files. Current implementations require that the VCF file to index first is compressed. The new compressed VCF file allocates additional disk space with respect to the already existing one. Furthermore, the compression hinders the human readability of the VCF file. With `IndexVCF's` way of indexing VCF files the resulting index structure can be adjusted for the best possible performance. Also, the VCF index is very small, which is another plus point for `IndexVCF`. As stated above, `IndexVCF` presents an new, lightweight and fast method to index VCF files.

But all the powerful features are of no good use, if they work slowly. Although, few of `TOPAS'` modules perform slower than similar functioning tools, the implementation can keep up with competitive tools in general. Processing large data sets, the overall runtime differences are very tolerable. On the plus side, however, some modules of `TOPAS` are several times faster than alternative ones. Though, the efficient runtime is not `TOPAS` biggest achievement, but the combination of many tools united in one software solution.

5.1.1 Conclusion

It has been shown that `TOPAS` is a multi-featured and powerful tool for the formatting, validation, manipulating and processing of NGS data in FASTA, FASTQ, GFF3 and VCF format. The collection of modules was successfully applied to several NGS data. It was also observed that the toolkit could be prosperously integrated into a NGS processing pipeline. `TOPAS'` modular structured and well documented code makes it an easy to use and easy to extend set of tools. Additionally, as shown above, the NGS data processing modules of `TOPAS` already work well together, only some modifications will have to be done in order to make them a perfect manipulative pipeline. A performance comparison with other tools illustrated, that `TOPAS` can not only keep up with these tools, but sometimes exceeds their working velocity.

In summary, `TOPAS` is an openly structured set of tools delivering approved, efficient and new methods to process sequence and annotation data supporting NGS data analysts in their daily work.

5.2 Outlook

As `TOPAS` is still under development, the potential of its modules can be extended as follows:

- `ValidateFasta`, `CorrectFasta`: A more efficient way to correct a FASTA file would be the pipeline like combination of `ValidateFasta` and `CorrectFasta`: `CorrectFasta` reads in the errors produced by

ValidateFasta, jumps directly to the error's lines in the FASTA file and corrects them. This way, the needed disk space of CorrectFasta is reduced and its runtime will only depend on the number of errors to be corrected.

- IndexFasta: A useful extension for IndexFasta would be to give the user the choice of validating and optionally correcting the FASTA file to be indexed before the indexing takes place. This pipeline would secure an FASTA index generation without halt.
- ExtractFasta: What ExtractFasta still lacks, is the possibility to extract certain sequences by pattern matching of the sequence string itself.
- SortGFF3: Prospectively, SortGFF3 should not only be able to sort a GFF3 file by *seqid/start/end*, but also by feature membership.

Adding of new modules to TOPAS increases its potential of dealing with NGS data:

- AlterGFF3: Enables the user to modify and/or add GFF3 entries of/to a GFF3 file. This tool might come in handy, when the user has to fit a GFF3 file to his needs.
- TranslateFasta: Gives the user the possibilty to convert a DNA sequence FASTA file into a protein sequence FASTA file and vice versa.
- JoinVCF: Often the user has a lot of VCF files with several different samples and positions in them. JoinVCF joins these VCF files together, so that the user only has to work with one VCF file instead of several.

Additionally, TOPAS' library should be extended to allow processing of SAM/BAM files, because the SAM/BAM format is the standard file format storing mapping information. The SAM-JDK of Picard[1] already provides such a library, which could be integrated into TOPAS. Moreover, a general user interface (GUI) could be implemented, so that unexperienced command line users are also able to use TOPAS. Another possibility would be to port all modules into KNIME. There users could build their own pipelines and still adjust TOPAS' modules to their needs.

[1] http://picard.sourceforge.net/ (last accessed: 04/27/2014)

Bibliography

[BKC+10] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor. Galaxy: A Web-Based Genome Analysis Tool for Experimentalists. *Current protocols in molecular biology*, pages 19–10, 2010.

[BKHN10] F. Battke, S. Körner, S. Hüttner, and K. Nieselt. Efficient sequence clustering for RNA-seq data without a reference genome. *German Conference Bioinformatics 2010. Lecture Notes in Informatics. Proceedings of the German Conference on Bioinformatics 2010*, P-173:21–30, 2010.

[BY13] M. R. Breese and L. Yunlong. NGSUtils: a software suite for analyzing and manipulating next-generation sequencing datasets. *Bioinformatics*, 29(4):494–496, 2013.

[CPC+12] P. Cingolani, V.M. Patel, M. Coon, T. Nguyen, S.J. Land, D.M. Ruden, and X. Lu. Using Drosophila melanogaster as a model for genotoxic chemical mutational studies with a new program, SnpSift. *Frontiers in Genetics*, 3, 2012.

[DAA+11] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, G. McVean, R. Durbin, and 1000 Genomes Project Analysis Group. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

[DBP+11] M. DePristo, E. Banks, R. Poplin, K. Garimella, J. Maguire, C. Hartl, A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. Fennell, A. Kernytsky, A. Sivachenko, K. Cibulskis, S. Gabriel, D. Altshuler, and M. Daly. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature Genetics*, 43:491–498, 2011.

[GAOC+22] F. García-Alcalde, K. Okonechnikov, J Carbonell, L.M. Cruz, S. Götz, S. Tarazona, J. Dopazo, T.F. Meyer, and A. Conesa.

Qualimap: evaluating next-generation sequencing alignment data. *Bioinformatics*, 28(20):2678–2679, 20122.

[GBH05] M.T.P. Gilbert, H.J. Bandelt, and M. Hofreiter. Assessing ancient DNA studies. *Trends in ecology and evolution*, 20(10):541–544, 2005.

[GNTT10] J. Goecks, A. Nekrutenko, J. Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86, 2010.

[GRH+05] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. C. Miller, W. J. Kent, and A. Nekrutenko. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

[GSK13] G. Gremme, S. Steinbiss, and S. Kurtz. GenomeTools: a comprehensive software library for efficient processing of structured genome annotations. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(3):645–656, 2013.

[HBSB10] S. Heinz, C. Benner, N. Spann, and E. et al. Bertolino. Simple Combinations of Lineage-Determining Transcription Factors Prime cis-Regulatory Elements Required for Macrophage and B Cell Identities. *Mol Cell*, 38(4):576–589, 2010.

[Ill] Illumina, Inc. Technology Spotlight: Illumina Sequencing.

[LD10] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler Transform. *Bioinformatics*, 26(5):589–595, 2010.

[LHW+09] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence alignment/map (SAM) format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[Li11] H. Li. Tabix: fast retrieval of sequence features from generic TAB-delimited files. *Bioinformatics*, 27(5):718–719, 2011.

[Met10] M.L. Metzker. Sequencing technologies - the next generation. *Nature Reviews Genetics*, 11:31–46, 2010.

[MHB+10] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res*, 20:1297–1303, 2010.

[PJ12] R. K. Patel and M. Jain. NGS QC Toolkit: A Toolkit for Quality Control of Next Generation Sequencing Data. *PLoS One*, 7(2), 2012.

[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85:2444–2448, 1988.

[PNTB+07] S. Purcel, B. Neale, K. Todd-Brown, L. Thomas, M. A. R. Ferreira, D. Bender, J. Maller, P. Sklar, P. I. W. de Bakker, M. J. Daly, and P. C. Sham. PLINK A Tool Set for Whole-Genome Association and Populaton-Based Linkage Analysis. *The American journal of Humen Genetics*, 81(3):559–575, 2007.

[QH] A. R. Quinlan and I. M. Hall. BEDTools: a flexible suit of utilties for comparing genomic features.

[RMB+10] M. G. Reese, B. Moore, C. Batchelor, F. Salas, F. Cunningham, G. T. Marth, L. Stein, P. Flicek, M. Yandell, and K. Eilbeck. A standard variation file format for human genome sequences. *Genome Biology*, 2010.

[SNC77] F. Sanger, S. Nicklen, and A.R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74:5463–5467, 1977.

[TRM12] H. Thorvaldsdáttir, J. T. Robinson, and J. P. Mesirov. Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in Bioinformatics*, 14(2):178–192, 2012.

[VdACC+13] G. A. Van der Auwera, M. Carneiro, Hartl C., R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. Garimella, D. Altshuler, S. Gabriel, and M. DePristo. From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline. *Current Protocols in Bioinformatics*, 43:11.10.1–11.10.33, 2013.

[Vil07] P. Villesen. FaBox: an online tool for fasta sequences. *Molecular Ecology Notes*, 7(6):965–968, 2007.

Appendix A

Sample Outputs of TOPAS' Modules

```
Validation of hs_ref_GRCh37.p10_chrMT.fa (DNA-Sequences)

[NEWLINE_TYPE]
\n
[TOTAL_LINES_IN_FILE]
238
[TOTAL_EMPTY_LINES_IN_FILE]
0
[TOTAL_COMMENT_LINES_IN_FILE]
0
[TOTAL_SEQUENCES_IN_FILE]
1

[TOTAL_AMOUNT_OF_BASES_INCLUDING_N]
16569

[BASE_DISTRIBUTION]
A     C     T     G     U     R     Y     K     M     S
5124  5181  2169  4094  0     0     0     0     0     0

W     B     D     H     V     N     X     -     OtherCharacters
0     0     0     0     0     1     0     0     0

[TOTAL_GC_CONTENT]
9275
[PERCENTAGE_OF_GC_CONTENT_INCLUDING_COUNTED_N]
0.5597803126320237

[WARNING_LINES]

[NOT_UNIQUE_IDENTIFIERS]

[ERROR_LINES] (do have to be corrected before running FastaIndexer)

[SEQUENCE_TYPE        IDENTIFIER        SEQUENCE_LENGTH]
     DNA          chrMT         16569
```

Figure A.1: A sample output of `ValidateFasta` giving statistics information about the validated FASTA file. Also, possible validation errors are displayed (in this case none).

```
TOPAS - TOolkit for Processing and Annotation of Sequence data
topas.CorrectFasta
Use -? for help

Parameters chosen:
Input file(s)        : [hs_ref_GRCh37.p10_chrMT.fa]
Output directory     : ~/bachelor_thesis/Performance/CorrectFasta
Sequence type        : dna
Sequence width       : 80

Correction of hs_ref_GRCh37.p10_chrMT.fa

[NEWLINE_TYPE_IN_CORRECTED_FASTA_FILE]
10
[TOTAL_REMOVED_EMPTY_LINES_IN_FILE]
0
[TOTAL_REMOVED_COMMENT_LINES_IN_FILE]
0

[CORRECTED_WARNING_LINES]

[CORRECTED_SEQUENCE_HEADERS_WITH_NOT_UNIQUE_IDENTIFIERS]

[CORRECTED_ERROR_LINES]


topas.CorrectFasta finished in 0 seconds
```

Figure A.2: A sample output of `CorrectFasta`. As displayed, no empty lines or comment lines in the corrected FASTA file were removed, due to their unpresentness in the file. Also, no error messages can be seen, indicating, that the FASTA file was correct in advance.

```
FASTQ_VALIDATION_OF: leprae_10000_multiline.fastq

[NEWLINE_TYPE]
\n

[TOTAL_LINES_IN_FILE]
100000

[QUALITY_SCORE_ENCODING]
Illumina 1.8, Phred+33, raw reads typically (0, 41)

[TOTAL_READS_IN_FILE]
10000

[OBTAINED_READ_LENGTHS]
200

[OBTAINED_AMOUNT_OF_LINES_IN_A_MULTI_LINE]
4

[NOT_UNIQUE_SEQUENCE_IDENTIFIERS]

[NOT_UNIQUE_QUALITY_IDENTIFIERS]

[HIGHEST_READ_QUALITY]
71.425

[LOWEST_READ_QUALITY]
35.0

[HIGHEST_BASE_QUALITY]
73

[LOWEST_BASE_QUALITY]
35

[MEAN_READ_QUALITY]
65.21538549999997
```

Figure A.3: A sample output of `ValidateFastq` is display. As can be seen, the Phred score encoding is Illumina 1.8, the length of each read is 200 and there are no unique sequence or quality identifier errors. Additionally, the highest/lowest read quality, the highest/lowest base quality and the mean read quality (all in ASCII format) can be obtained from the output.

```
GFF3Validation of ref_GRCh37_chrMT.gff3

[WARNINGS]

[ENTRY_ERRORS]
[line 4] Unrecognised type 'D_loop'.
[line 5] Unrecognised type 'D_loop'.

[UNIQUE_ID_ERRORS]

[RELATIONSHIP_ERRORS]
```

Figure A.4: A sample output of `ValidateGFF3` showing that two GFF3 entries in lines 4 and 5 are of a forbidden type. There are no unique id or relationship errors in the validated GFF3 file.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, den 30.04.2014 Simon Heumos