Representing genomic variation in reference graphs

Johan van Beusekom
s093251

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of the thesis is to represent genomic variation using graphs. The thesis covers the motivation for representing genomes as graphs rather than the classical linear representation. The thesis also covers the design of the genome graph and a discussion of suitable technologies that can be used for the construction of the graph. The thesis discusses how to use the genome graph to sample publicly sharable data from otherwise confidential genetic data by removing rare genetic variants. The challenge of how to replicate the statistical properties of the original data in the samples is discussed as part of this thesis. Futhermore the thesis discusses how the graph can be shared through the *Semantic Web*. As part of the thesis, a tool named *VGTool* has been designed and implemented, with which genome graphs can be constructed from VCF files. The tool can also be used to generate genetic data and share the graph through the Semantic Web. The tool is discussed in terms of design and implementation choices, and a series of tests are performed to analyse the performance of the tool. The quality of the sampled data is also tested, to ensure that the statistical properties of the original data is present in the sampled data. It is concluded that sampling of individuals from a genome graph is a feasible way of sharing otherwise confidential information, but also that the statistical properties of the generated samples are not entirely identical to those of the original data. It is also concluded that the constructed graph can be extended to function in other use cases.

Summary (Danish)

Målet for denne afhandling er at repræsentere genomisk variation med grafer. Afhandlingen begrunder motivationen for at repræsentere genomer som grafer fremfor den klassiske linære repræsentation. Afhandlingen beskriver derudover designet af en genomgraf og diskuterer egnede teknologier, der kan bruges til at konstruere grafen. Det diskuteres, hvordan genomgrafen kan bruges til at generere offentligt delbar data fra fortrolig genetisk data ved at fjerne sjælden genetisk variation. Udfordringen omkring gengivelse af de statistiske egenskaber fra den oprindelige data i den genererede data bliver diskuteret som en del af afhandlingen. Ydermere diskuteres det, hvordan grafen kan deles gennem *The Semantic Web*. Som en del af afhandlingen, er et værktøj ved navn *VGTool* blevet designet og implementeret, med hvilket genomegrafer kan konstrueres fra VCF filer. Værktøjet kan også bruges til at generere genetisk data samt til at dele grafen i *The Semantic Web*. Værktøjet bliver diskuteret i form af dets design- og implementationsvalg, og en række test er udført med henblik på at analysere værktøjets ydeevne. Kvaliteten af det genererede data bliver også testet for at sikre, at de statistiske egenskaber i den generede data er i overenstemmelse med den oprindelige data. Det konkluderes, at det genererede data fra genomgrafen er en mulig måde at dele ellers fortrolig information på, men også at de statistiske egenskaber for den generede data ikke er helt identiske med den oprindelige data. Det bliver også konkluderet at den konstruerede graf kan udvides således, at den kan bruges i andre *use cases*.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with the representation of genetic variation using graphs and the ability to sample data from the graph as a way of sharing otherwise confidential information.

The thesis was supervised by Nicola Dragoni with external supervision by Jette Bork-Jensen and Christian Theil Have.

Lyngby, 09-July-2015

Johan van Beusekom
s093251

Acknowledgements

I would like to thank my supervisor Nicola Dragoni for giving me important guidance and advice when I needed it.

I would like to thank Jette Bork-Jensen and Christian Theil Have, who have guided me in my daily work with the thesis. They have both been available to me at any time and especially their knowledge of genetics has been an important resource for me when writing the thesis. The tests of section 6.3.1.6 were done in collaboration with Christian Theil Have.

I would also like to thank all three, for putting time and effort into providing me with useful feedback in the writing process of the thesis.

I would like to thank The Novo Nordisk Foundation Center for Basic Metabolic Research for providing office space to me and I would also like to thank the students, who I have shared my office with, for their company throughout the five months of the thesis.

Contents

Chapter 1

Introduction

In this chapter we will start by giving a brief overview of the motivation for studying genetics and DNA in the context of a study recently undertaken in Denmark. We will then give an outline of the problems regarding the current format in which the collected data is represented. The presented problems will be used to form a problem description along with goals for this thesis.

Throughout the thesis the reader will be introduced to a number of terms from biology and bioinformatics which will be briefly explained in order to give the reader a better understanding of the context of the problem and the motivation for the thesis. Appendix A contains a collection of these terms.

1.1 Motivation and background

This thesis is motivated by limitations of the current data format in which genetic variation is expressed. In the following, we will explain briefly how and why it is important to study genetics.

1.1.1 Studying genetics

Genes are molecular units that exist within all living organisms and hold information of how to build and maintain an organisms cells and pass genetic traits to offspring. The biological traits of an organism are determined by its genes, and while some traits, such as eye color or height, are easily visible, other traits such as blood type or increased risk of specific diseases are not. The observable traits of an individual is called the *phenotype* of the individual.

> ***Phenotype:*** The observable traits of an organism are called the organisms phenotype. The phenotype of an organism is influenced both by the environment and the genetics of the organism.

Individuals of a species generally have the same collection of genes, but the particular variants of *genotypes* can make an individual carry a certain variation of a trait.

> ***Genotype:*** A genotype is a part of DNA that determines a specific charateristic of the individual. For any position in the DNA, there is a genotype, but it is possible that two organisms have different genotypes for the same "position", resulting in different phenotypes.

As such there are genetic variation associated with having blue eyes and other variations associated with brown eyes. The information stored in the DNA of an organism is present already from birth, and while some genetic traits are visible, other traits such as an increased risk of cancer for an individual are not, but could be inferred from studying the genetic patterns of the individual.

> ***DNA (Deoxyribonucleic acid):*** DNA is a molecule that stores biological information. DNA consists of two strands, which are each made up of sequence of *bases*.

> ***Base (Nucleobase):*** A sequence of DNA is made up of the four bases A (adenine), C(cytosine), G(guanine) and T(thymine).

An early diagnosis of such diseases is desirable because the earlier the diagnosis is made, the earlier actions can be taken to counteract or mitigate the effects of

the disease. Making a diagnosis based on a persons genetics involves comparing an individuals DNA to known DNA patterns, called *reference genomes*, in order to find similarities and infer the biological effects of these. This process will be explained in more detail in section 2.2.

> ***Genome & Reference genome:*** The genome is the entire DNA sequence of an individual. A reference genome is the complete assembled genome of an individual. The term *reference* comes from the fact that it is used as a reference to which DNA can be aligned in order to determine similar regions.

The quality of this comparison relies on genetic similarity between the two DNA sequences, because it can be hard to determine the similarities if the sequences differentiate a lot. In order to be able to choose a suitable reference genome there is motivation for having a database of DNA that resembles the local population; this is because people from the same geographical origin are genetically more similar than people from different parts of the world, in the same way that people who are related to each other are more genetically similar than people who are not.

1.1.2 Context and case study

Besenbacher et al. [12] covers a large study recently carried out in Denmark in which a national *pan-genome* has been created in order to give more insight into the genetics of the Danish population.

> ***Pan-genome:*** A pan-genome is a collection of genetic data that encapsulates the genetic variation of a population.

The study involves the *sequencing* of a *cohort* of 50 trios[1] in order to create a database of 150 individuals' DNA that resembles the danish population as best as possible.

[1] A trio is a family of tree; mother, father, and child

> ***Sequencing:*** Sequencing of DNA is the process of determining the precise order of the bases in a DNA stand as explained in section 2.2.2.

> ***Cohort:*** A group of individuals with a shared characteristic. In the context of the Danish pan-genome, this characteristic is that all involved individuals are Danish.

The study is one of the biggest within the field of genetics carried out in Denmark and Scandinavia and aims, as encapsulated by the quote below, to improve our understanding of the genetics of the Danish population.

> *"Identification of all variants and their frequencies can facilitate an increased understanding of population-specific disease susceptibility and will be important for advancing clinical and public health genetics."* [12, p. 6]

However there are restrictions which mean that the collected data cannot be shared publicly. This of course is a shame because the data is interesting to other research institutes in Denmark, Scandinavia and even the rest of the world.

Working with the DNA of individuals is a delicate matter in the sense that unless the individuals agree that the information of their DNA can be shared publicly, there is a strict requirement to keep the genetic data confidential. Keeping the data confidential is not a problem from a technical standpoint, but it hampers the possibility of publicly sharing interesting findings, allowing researchers in different parts of the world to work with the data. Researching genetics is generally quite expensive due to the required equipment and because the involved techniques are in constant development. This means that due to financial limitations many researchers are not able to gather data and carry out projects on the same scale as for example the Danish pan-genome project. For this reason it would be desireable if the research facilities that have the financial backing to undertake large studies of human genetics were able to share this information with the rest of the world for them to analyse and work with. The required anonymity makes it hard to share genetic information seeing as a persons DNA is like a very detailed fingerprint and therefore could be directly related back to that person. The consequences of sharing a persons DNA are hard to determine, but scenarios have been thought of in which for example an insurance company could refuse to give a life insurance to a person if they were able to tell from

the genetic data that the person is likely to develop cancer later in life. Genotypes are not necessarily statistically independent, but can covary depending on their position in the genome. This means that "critical" genotypes (in the sense that they for example are related to the ability to develop cancer) can be linked statistically with "innocent" genotypes, and revealing the presence of the latter can be used to infer the existence of the former.

In this thesis we consider the Danish pan-genome a case study for the project. This is natural, because the presented problems and the motivation for solving them are a result of the Danish pan-genome project, the limitations of the current data representation and its inability to be shared. The anonymity of the individuals involved in this study is a requirement, but there is motivation for sharing as much information as possible without revealing the identity of anyone involved. The approach taken and the achieved results of the thesis however aim to be of general applicability.

1.2 Problem description

A linear representation of a genome is limited in its ability to express the variation of a population. The genetic variation of populations is interesting because it can be used to identify the consequences of having certain genetic variants, however inidividual level genetic data cannot be published due to ethical ramifications. Sharing beyond summary statistics of single genotypes in a cohort is problematic because interdependency between rare genetic variation may expose the identity of involved individuals and their dispositions.

In this thesis we will examine possibilities for creating a data structure that represents the genetic variation including statistical interdependencies with a population sample without exposing individual genotypes. To test the data structure we implement a tool capable of creating the data structure and sampling data and we evaluate whether it conveys a reasonable statistical reproduction of the original data. The tool will be designed to be capable of handling large amounts of data, to integrate with existing resources and formats and finally to be extendable in anticipation of future use cases of the data structure.

1.2.1 The Novo Nordisk Foundation Center for Basic Metabolic Research

The project has been supervised externally by Jette Bork-Jensen and Christian Theil Have from The Novo Nordisk Foundation Center for Basic Metabolic Research at Copenhagen University. The problems discussed in this thesis are related to their work with the Danish pan-genome.

1.3 Goals & requirements

The overall goal is to redevelop the data representation for genetic variation of a cohort in the context of the Danish pan-genome and to create a method with which this data can be shared while still abiding to the requirements of anonymity for the involved individuals. This process can be divided into smaller goals:

- An analysis of the problems with the current data represention of genetic variation (Chapter 2).
- A discussion of why graphs are suitable data structures for the reprensentation of genetic data and how they can be designed to solve problems of the current representation (Chapter 3 & 4).
- A review of relevant technologies that allows us to build the graph data structure (Chapter 5).
- An implementation of the proposed solution with a justification of the approach used and the technologies involved (Chapter 6).
- An evaluation of the implemented solution and a discussion of the achieved results (Chapter 6, 7 & 8).

1.4 Contribution

The contribution of this thesis is twofold.

Firstly, it provides a motivation for representing genomic data in graphs along with a discussion of how these can be built and used in different scenarios. Along

with the genome graphs is a proposed solution of how to share person sensitive genetic data of a cohort by removing rare genetic variation and sampling individuals with statistical similarity to the original data. This is particularly interesting in the context of the Danish pan-genome as it provides a way of sharing otherwise confidential data from an extensive genetics study.

Secondly, a tool named *VGTool* has been implemented in which graphs can be built from the genetic variation present within a cohort and with which invididuals can be sampled in accordance with the method mentioned above.

1.5 Thesis outline

Chapter 2 will explain briefly about genetics, why and how they are studied. It will then take a look at the current data represention for genetic variation and the limitations of this. After discussing the limitations, we will refine the scope of this thesis, as we are not able to attend each of the presented problems.

Chapter 3 discusses the motivation for representing genome data as graphs by presenting several use cases in which a genome graph would improve upon the current linear representation.

Chapter 4 presents relevant design decisions for a genome graph. The chapter also discusses algorithms for using the graph and analyses these in terms of running time and space complexity.

Chapter 5 provides a review of relevant tools and formats for working with and storing the genome graph. This review provides a reasoning behind the chosen technologies such as programming language and storage format for the graph.

Chapter 6 is concerned with the tool developed as part of this thesis. It is divided into three sections: In section 6.1 the design of the implemented tool is discussed in terms of structure, and the implemented algorithms are explained and analysed in terms of their running time and space consumption. Section 6.2 discusses in more detail the implementation of the tool, with a focus on the key parts of the code. Section 6.3 contains tests of the implemented tool. The section is divided into two parts, one for the performance of the tool, and one for testing the quality of sampled data.

Chapter 7 provides a discussion of the achieved results in relation to the initial goals set for the thesis as well as suggestions for future work.

Chapter 8 concludes the overall findings.

1.6 Summary

In this chapter we have briefly outlined the motivation for studying genetics. We have also discussed the problematics regarding the sharing of data, here in the context of the Danish pan-genome. We used the issues surrounding the sharability of person specific genome data as a motivation for the problem description, which forms the basis of this thesis.
In the next chapter we will take a closer look at how genetics are studied and represented and analyse the problematics regarding this before we provide a new genome representation that seeks to solve the found issues in chapter 3.

Chapter 2

Data representation

In the last chapter we briefly discussed what is involved in the analysis of genetics. In this section we will go into more detail of the usage of the collected genetic information and highlight some of the problematics regarding this. We will also refine of the scope of the thesis.

2.1 Genetics overview

In the introduction we explained why knowledge of the human genome is important, why it must be sequenced and what it can be used for.
Because the general topic of this thesis is within Computer Science, this section will provide an overview that should enable the reader to understand the biological concepts involved regardless of their background. For a more in depth of explanatation the concepts mentioned here we refer the reader to Griffiths [19].

The first concept that we should look at is *DNA*. DNA is a molecule that encodes genetic instructions used in the development and functioning of all known living organisms and many viruses. DNA is made up of *bases* among other things[1]. The four bases in DNA; cytosine (C), guanine (G), adenine (A), and thymine

[1] That are left out of this overview for the sake of simplicity

(T), appear in sequences of DNA. As mentioned, sequencing of an individual is the process of determing the exact order of the four bases in the DNA strand. *Chromosomes* are long sequences of DNA which come in pairs, as individuals inherit one chromosome from each parent. The genome is the collection of the chromosomes, so when we talk about sequencing DNA, it is the process of determining the order of the bases of DNA in each chromosome. An overview of a chromosome is shown in figure 2.1[2].

> ***Chromosome:*** A chromosome is structure that contains DNA. A human inherits one copy of each the 23 chromosomes from each of their parents, giving a total of 23 chromosome pairs.

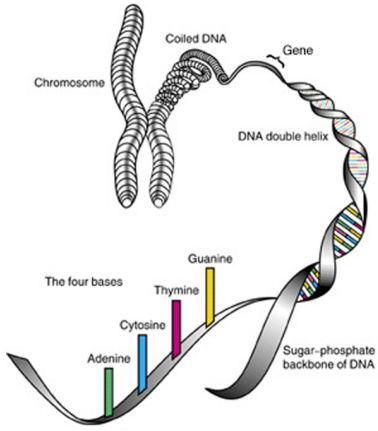


Figure 2.1: Relationship between entities in a chromosome.

2.2 Obtaining and using genetic data

In this section we will look at how genetics are studied. Studying genetics and the effects of certain genetic patterns is an empirical process in which newly sequenced DNA of individuals is compared to a reference genome in order to find their similarities. In the next sections we will start by looking at the nature of genetic variation and then look at how reference genomes are built and then how they are used to identify genetic variation in individuals.

[2]Figure from `http://groups.nbp.northwestern.edu/science-outreach/genome/DNA.jpg`

2.2.1 Genetic variation

Genetic variation is variation of the sequence of bases in DNA from one individual to another. The consequences of having certain genetic variants can range from non-existant to causing quite severe genetic disorders.
Identifying the genetic variation is challenging because it can appear in different forms such as *single nucleotide polymorphisms* (SNPs), small insertions and deletions (*indels*) and larger *structural variants* (SVs) in the DNA [12, p. 2]. Structural variants are larger variations between genetic sequences, such as for example a large insertion of a segment of DNA from one chromosome into another. The simple types of variation are shown in figure 2.2.

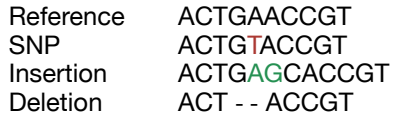


Figure 2.2: The three simplest types of variation in relation to a reference.

> ***Single nucleotide polymorphism (SNP):*** Pronounced "snip". Describes genetic variation of one base.

> ***Indel:*** The term specifies genetic variation that is either a deletion or an insertion. The reason that one term is used for both types of variation, is that depending on the frame of reference, an insertion can also be seen as a deletion and vice versa.

> ***Structural variation:*** A more complex type of variation that changes the structure of the genome, for example by inversing a sequence of DNA.

2.2.2 Sequencing DNA

As mentioned in section 1.1.1, DNA sequencing is the process of determining the precise order of the bases in a DNA strand. The process is central aspect of studying genetics. Given the size of the human genome, it is infeasible to read

an entire chromosome in one go, and it is instead fragmented into *reads* of small length, as illustrated in figure 2.3.
To assemble the original genome, a method called *de novo assembly* [37], can be used in which the reads are compared pairwise in order to find overlaps between them. If reads overlap, they can be merged into a longer sequence until in the end there is only one sequence left for each chromosome which is the reconstruction of that original genome as illustrated in the figure 2.3. The read comparison can be done in parallel to greatly decrease the running time of the process.

> ***Genome assembly:*** Genome assembly is the process of assembling the human genome from smaller reads as shown in figure 2.3.

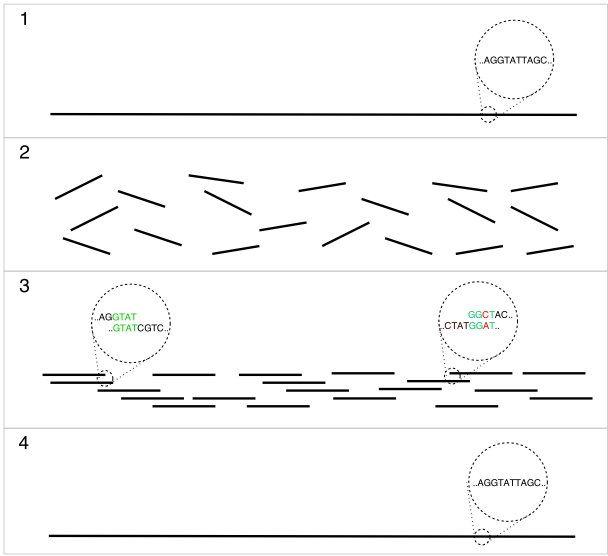


Figure 2.3: Overview of the sequencing process. **1**: The genome as it is in an individual. **2**: The generated reads from the genome. **3**: The reads are aligned to each other by finding overlapping sequences, however some reads do not have a perfect overlap, which can be caused by read errors. Therefore there should be an error margin that allows reads to be paired even though they are not completely identical. **4**: The assembly is constructed by merging the overlapping reads into one sequence. This sequence should be close to or identical to the original genome.

Genome assembly is complicated by the fact that the equipment for generating reads is not perfect, and as such reading errors can occur. This means that reads covering a particular *locus* in the DNA may have different ideas of what base it is.

> ***Locus:*** Locus (plural loci) refers to a specific position on a chromosome.

Therefore the *depth* of the sequencing is important for the result, as it describes how many reads that cover a particular locus. Because errors can occur in the reads, the consensus among the reads of which base is at given position might not be in unison, but if sequencing is performed with high depth, a majority voting scheme can be used to make a qualified guess of the base.

> ***Depth & Coverage:*** The coverage of a locus describes the depth of the sequencing in that locus i.e. how many reads cover the locus. A high depth is desirable because it reduces the risk of sequencing errors.

2.2.3 Identifying variation

The previous section described how to create reference structures. In this section we will look at the usage of these.
Sequencing DNA and building reference structures only solves half the problem regarding analysis of an individuals genetics. Inferring meaning from genetic variation is an empirical study in the sense that variation present only for people with a certain genetic trait can be seen as the cause of the trait. In order to identify the presence and effects of certain genetic patterns in an individuals DNA, the process of *sequence aligment* [37], shown in figure 2.4, can be used in which sequences are aligned to a reference genome in order to find regions of structural similarity.

> ***Sequence alignment:*** Sequence aligment is the process of aligning DNA sequences to a reference to identify regions of similarity, and inferring meaning from these regions, such as for example genetic variation that is related to a certain disease [32]

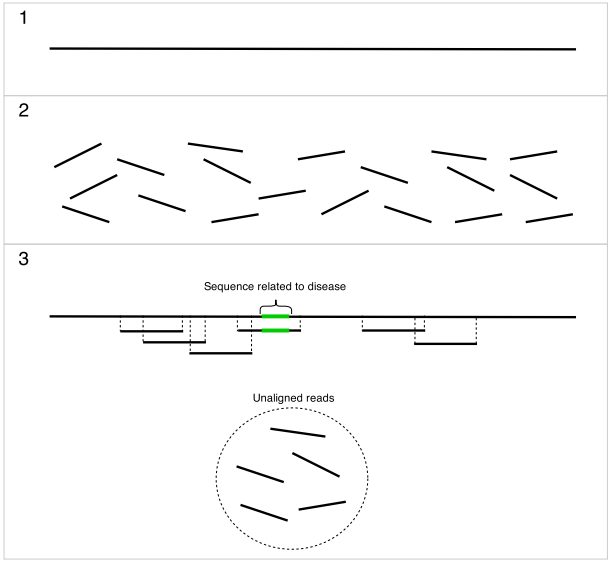


Figure 2.4: Overview of the aligment process. **1**: The reference genome. **2**: The input reads. **3**: The reads are aligned to the reference. **The homologous areas** are then identified. Some areas in the reference genome are known to be associated with certain traits or diseases, and if the sequenced individuals shares this genetic pattern, he/she is likely to have the trait. As mentioned, if the difference between reference and reads is too big, some reads may be unalignable with the reference.

The reference functions as a pattern that the reads should be mapped to, however there is room for a certain error margin, meaning that the alignment does not need to be perfect. This is because reads can contain errors, but also because the reference, as mentioned, is constructed from a different individual than the input and there are therefore bound to be genetic variation between the two. Once the reads have been aligned, the variation between reference and input can be determined and analysed. The method works well when the genetic variation between the sample and reference is small, but for large variations, it can be hard or impossible to perform the aligment [16, p. 5]. This is a problem that stems from the fact that the reference genome is essentially the DNA sequence of another individual, and therefore if the reference is not suitable, the aligment fails.

Regarding mapping Paten et al. [35, p. 2] lists four problems with the current approach:

- There is no standard way of mapping newly sequenced DNA bases to

positions in the reference genome, though a number of algorithms are popular.

- Mapping procedures can suffer from the *multi-mapping problem* where input can map to several locations in the reference due to the fact that identical or nearly identical sequences can appear in different parts of a reference.
- All variation is not currently stored together, but instead spread out over many data bases of genetic variation, both public and private.
- The current format in which a central reference with alternative sequences is used in mapping is problematic, because if the central reference changes, the alternative sequences must change coordinates as well.[3]

Figure 2.5 shows an example of the multi-mapping problem. The problem arises when several positions on the reference provide equally good results in regards to the mapping/aligment. In the example, both mappings have 6 elements that do not match the reference, and as such can be seen as equally good mapping locations.

```
Ref:    AAGCTA--CTAG----CT                  AAGCTA------CTAGCT
Allele: AAGCTAGACTAGGAAGCT       or         AAGCTAGACTAGGAAGCT
    (2 gap opens, 0 mismatch)           (1 gap open, 2 mismatches)
```

Figure 2.5: An example of the multi-mapping problem as seen in [28].

2.2.4 Population-specific reference structures

In the introduction we mentioned the recent study carried out on 50 trios in Denmark that seeks to provide insight into the genetics of the Danish population. We also mentioned in the previous section how alignment can be complicated or impossible to perform when the reads and the chosen reference differentiate a lot; this is somewhat like solving a puzzle in which the pieces do not fit together. A good alignment relies on two things; having a reference genome that is as similar to the input reads as possible, and having alternative options for alignment if it is not possible to align the reads to the chosen reference. This is where a population specific reference structure comes in. Because having many sequenced individuals means that there are alternatives for regions where the

[3]This is because the alternative sequences are mapped to genome coordinates of the reference, but if the central reference changes, the alternative sequences should have their genome coordinates updated.

reads cannot be fitted onto the primary reference. Currently the process of mapping reads to a reference still relies on having one primary reference, and in the case that it is not possible to map reads to this reference the alternative sequences can be tried. The ideal solution would of course be to have a data structure that contains all the different genetic sequences that have been observed, so that reads have many options for where to map, and the choice of reference is always the best possible choice, because it contains all genetic variation.
Creating a unified data structure that stores the genetic variation observed in a group of individuals would also be interesting if it is possible to extend this structure each time novel variation is discovered. Studies such as Besenbacher et al. [12] aim to build collections of sequenced DNA in order to create an inventory of genetic variation within a population rather than a single person. Besenbacher et al. [12] however studies the sequenced data in the classical linear representation, but highlights the need for unified genetic data structure as seen in the following quote:

> *"Methodological developments in the analysis and representation of sequence data will offer advantages as well as addressing challenges such as storing of variant population frequencies and alternative haplotypes within a reference."* [12, p. 6]

2.3 Current data representation

In this section we will take a look at the current representation of sequenced DNA and some of the problems related to this.
In 2001 the first draft of the human genome assembly was presented [26]. The assembly was monoploid [35, p. 1] which means that the genome in this project was represented by one single sequence of DNA constructed from one individual. The current representation of assemblies uses the principle of a central reference like the one constructed in Lander et al. [26], however with alternative genetic variations for some regions. Small variations are simple, because they can be attached to a locus in the reference, but larger structural variants are more complex and can for example combine one part of a chromosome with another chromosome. Figure 2.6 shows a conceptualization of how the variation is stored in the current model. In the next section we will look at how this is represented the Variant Call Format, which is a file format for storing genetic variation of a group of individuals in relation to a reference genome.

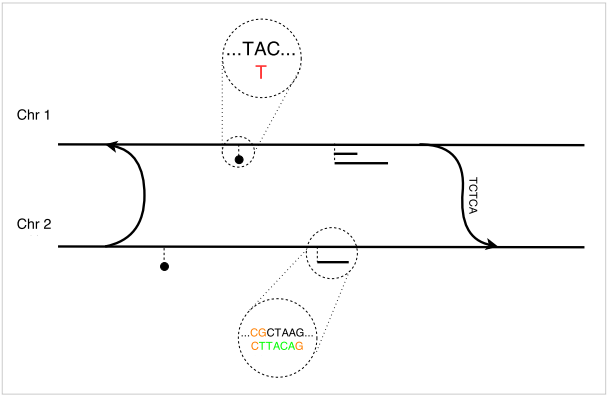


Figure 2.6: A conceptualization of a reference genome to which variation is mapped to positions; for example here is shown a **SNP** as a T instead of the A in the reference, an **insertion** between two **endpoints**, and some structural variants where one chromosome is combined with another.

2.3.1 The Variant Call Format

In this section we will look at the Variant Call Format which is a format for expressing the genetic variation of individuals in a cohort. It is from this data, that we will build a unified data structure of genetic variation in order to solve the problems presented in 1.2.

When genetic information has been collected it can be presented in different formats. The Sequence Aligment/Map[4] format is used to specify how well and where individual reads map to a chosen reference. Once an individual or a group of individuals have been successfully mapped to a reference, the Variant Call Format[5] can be used to express the observed genetic variation. The main purpose of VCF files is to show how the genetic make up of a cohort differs from a reference genome. Each line of the file contains the *allele* from the chosen reference genome, its chromosome and locus, and a number of alternative alleles that have been observed in the cohort.

> ***Allele:*** An allele is one particular variant of a number of possible variations for a certain locus. A genotype consists of two alleles.

Furthermore the type of variation is specified; SNP, indel, or structural variant.

[4]Full specification in [8]
[5]Full specification in [7]

Along with the general information for each locus, the format also holds information specific to each involved individual; this includes the persons genotype and *phasing* information as well as other data such as the coverage of the locus for the particular individual.

> ***Phasing:*** Every individual inherits a copy of each chromosome from both their parents. If a genotype is phased, it means that it is known which allele of the genotype comes from which parent.

Figure 2.7 shows an example of entries in a VCF file.

```
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  ID1       ID2
chr21   123             T       C,G     .       PASS    AC=300  GT:DP   0|1:150   2/0:40
chr21   124             C       A       .       PASS    AC=106  GT:DP   1|0:143   0/1:111
```

Figure 2.7: Example of a simple VCF file.

A VCF file as shown in figure 2.7 file begins with a header followed by a number of entries. In the header there is information about each column of the file; there are columns for the chromosome and locus, for the allele stored in the reference, and for alternative alleles observed among the individuals. There are also columns with information about the particular position followed by the a column for each involved individual. In figure 2.7 there are two individuals with IDs 'ID1' and 'ID2'.
Each entry in the file contains the information specified by the header. In this example there are two loci from chromosome 21. At these loci the reference genome has values T and C respectively, but alternative alleles have been observed among the involved individuals as shown in the comma separated list in the 'ALT' which contains the alternative alleles for the locus. In the example, locus 123 has alternative alleles 'C' and 'G' while locus 124 has 'CT' as an alternative allele.
The value 0|1 for ID1 at locus 123 in the example above means that the individual with ID1 at position 123 in chromosome 21 has the genotype T|C in that position, because 0 corresponds to the value T and 1 corresponds to the value C. The mapping of numbers to alleles is done in incremental fashion from left to right, such that allele represented in the reference is always 0. Individual ID2 has a different genotype in this position; 2/0, which corresponds to G/T. '|' and '/' are used to specify the phasing information of the genotype, '|' meaning phased and '/' meaning unphased. The phasing information is interesting because if we are sure which chromosomes the alleles belong to, we can tell which DNA sequences come from each chromosome.
In this example each genotype is stored along with the depth of the reads for

each individual; this information can be used to determine if rare variants are genuine or due to a poor coverage of the reads in the given location.

```
#CHROM  POS    ID      REF     ALT             QUAL    FILTER  INFO    FORMAT  ID1     ID2
chr21   5              T       TCGA            .       .               GT      0|1     1|1
chr21   501            GAT     G               .       .               GT      1/0     0|0
chr21   577            G       G[chr22:123[    .       .               GT      0|0     0|1
```

Figure 2.8: Example of indels and structural variants in VCF.

The format also handles more complex types of variation. Figure 2.8 has examples of an insertion, a deletion and a structural variant. The first line in the example shows an insertion in which the sequence CGA is inserted after a T. The next line is a deletion in which the reference has the value GAT while an alternative allele is the deletion of AT, leaving G. Insertions and deletions can have arbitrary length. The last type of variant, the structural variant, highlights the limitation or inconvenience of the VCF format. Structural variants are genetic variations which changes the structure of the genome, this can for example be a large insertion of a sequence located on another chromosome, an inversion of a sequence or recombination of two chromosomes. The example above shows a structural variant in which position 577 on chromosome 21 is followed by a stretch of DNA from chromosome 22 position 123 onwards; these two loci are referred to as the *break-ends* of the structural variant.

> ***Break-end:*** The break-ends of a structural variation specifies how two sequences of DNA are connected.

The square brackets determine the direction of this segment '[' meaning forward and ']' meaning backwards.
VCF also allows for specification of uncertaincy regarding the size of genetic variation. This means that instead of for example having a constant size for a deletion, it can be described as an interval of the possible lengths. The same goes for the size of insertions and the breakends of strucural variants.
The full specification of structural variants in VCF can be found in [7, p. 12-20].

The VCF format is quite expressive in terms of the genetic variation and sequenced data. It does however suffer from the fact that it is closely tied to the cohort from which it was created due to the detailed information about the genetics of each involved individual.

2.3.2 Sharing genetic data

The information stored in VCF files is, as we have seen, a specification of the observations made for sequenced individuals. This means that the data cannot be shared freely for reasons we have already discussed. We have also discussed why there is a motivation to share the collected results.

In Denmark, Danmarks Statistik has decided that genetic variation should be common enough for summary statistics to not allow for the identification of individuals with the particular genetic variant [39]. Sharing summary statistics e.g. allele frequencies is desireable to do for the collected data, however a problem arises when those summary statistics approach individual values i.e. rare variation.

> ***Allele frequency (AF):*** AF is an expression of the frequency of each observed allele for a particular locus.

This means that if rare genetic variation[6] is removed, the remaining variation is free to share publicly. However there is the chance that altering the data set by removing rare variation renders it unuseable due a skewing of the statistical properties in the remaining data in relation to the original data. The two primary properties that should be preserved in the data set are the *allele frequency* and the *linkage disequilibrium*.

> ***LD (Linkage disequilibrium):*** LD is an expression of alleles at different loci that are linked together statistically in the sense that they are often both appear together. Figure 2.9 illustrates this.

> ***Imputation:*** In genetics, imputation is the statistical inference of unobserved genotypes. [36]

Sampling data from the graph is useful for the purpose of genetic *imputation*. Imputation is an essential part of studying genetics, as is a cheaper way of determining genetic sequences in an individual by using statistics of genetic patterns and genotypes rather than observing each loci.

[6] In this thesis rare genetic variation will be genetic variation that is observed in less than five individuals of a cohort.

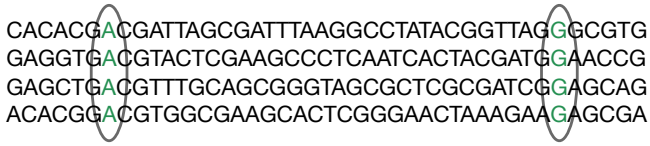


Figure 2.9: If the circled bases in this figure are in linkage disequilibrum, it means that if one of them has been observed, then the other is likely to be in the sequence.

2.4 Unified data structure

We have now discussed how genetics are studied by using a reference genome to find genetic variation of interest in an individual. We have also explained how this information can be expressed using the VCF file format, and the problems regarding sharing this information.
We propose creating a unified data source of genetic variation that can solve the problem of relying on a well chosen reference as discussed in section 2.2.3 by containing all observed genetic variation and thereby providing the best conditions of a successful mapping. The data source could also be used to express the genetic variation of a cohort, and to solve the problem of the inability to share the information of the data structure we propose a scheme for generating sampled individuals that can be expressed in, and shared using the VCF format.

2.5 Restriction of scope

This thesis will be concerned with the creation of a data structure which contains the genetic variation of a cohort as mentioned in the problem description of section 1.2. We have chosen to focus on how this data structure can be used to sample individuals with the same statistical properties as the original data while still ensuring the preservation of anonymity for any of the involved individuals. We will not propose a concrete solution to the mapping problem, although the proposed data structure should be designed with this use case in mind, as it could be a future extension.

2.6 Summary

In this chapter we have looked at how and why genetics are studied. We have also looked at the current data representation and its limitations. With the identified limitations we have proposed using a unified data structure for genetic variation. In the next chapter we will look at how the genetic data could be expressed as a graph.

Chapter 3

Genetic variation as graphs

This chapter contains the motivation for modelling genetic variation as a graph and discusses why a graph is particularly suited for meeting the requirements established in the last chapter. As we shall see, the idea of using graphs to model genetics is not entirely new, and is an intuitive choice of data structure in genetics as expressed in the following quote:

> *"The best data representation for capturing genome data is a graph. Graphs capture the intrinsic connectivity of genome relationships, and they are a comfortable communication medium for biologists and computer scientists."* [18, p. 7]

3.1 Existing graphs in genetics

Graphs in genetics are not a new concept. In the sequencing of DNA for a single individual, graphs have been used for a long time. However the graph in these processes is only a remedy in the assembly, which in the end produces a traditional linear DNA sequence. Graphs such as *De Bruijn* graphs [15] and *breakpoint* graphs [29] *"of local sequence variation and/or prior knowledge of*

small-scale variants are used to aid read assembly by several variant-calling algorithms" Dilthey et al. [16, p. 2].

In recent years more focus has been put on constructing genome graphs for multiple genomes to contain the variation of a cohort in a single data structure. The quote below highlights how the progress in this area:

> *"... the rapidly growing number of sequenced human individuals continueously presses for the necessity of a graphical representation of the existing sequences, which leads to many publications in this direction, both in biology and in computer science."* [28]

An example of one the studies that is concerned with building such a genome graph is [16], which describes the graph in the following way.

> *"A population reference graph is a directed acyclic graphical model for genetic variation that is generated by combining information about known allelic relationships between sequences."* [16, p. 3]

[28] contains a review of related work in regards to containing multiple genomes in a single data structure, including graphs.
Because genome graphs in genetics and genome graphs are not a new concept, we can in the following section discuss the motivation for representing genomic variation in graphs by referring to existing studies as well contribute with our own motivation.

3.2 Genome graphs

Graphs are datastructures that can be used to show the relationship between entities. They appear in many places and have many useful properties. In this section we take a closer look at graphs in the context of DNA and especially their capability of expressing variation.

Throughout this thesis we will refer to a graph that describes genetics with the term *genome graph*.
If we take a look at the representation of assembled sequences explained in section 2.3, we can see that it resembles a graph. The central reference sequence can be seen as a linear graph, and the variation as branches in the graph that

are attached with edges to the start- and end point. Furthermore break-ends can be represented by edges from the source position to the destination.

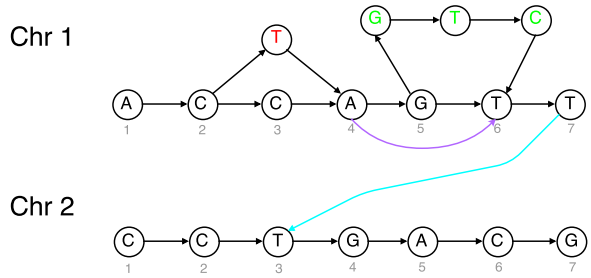


Figure 3.1: An example of how reference genomes can be represented as graphs. The figure shows a **SNP**, an **insertion**, a **deletion** and a **structural variant**.

Figure 3.1 shows and example of how a graph could be used to represent different types of genetic variants.

In the remainder of this section we look at how the nature of genetic variation can be modelled intuitively using a graph.

3.2.1 Vertices and edges

Graphs consist of two fundamental units; nodes and edges, both of which can be used to convey information. In a graph model we want to represent bases of a DNA strand and their relations to each other and we should therefore decide how this should be modelled in the graph. In Dilthey et al. [16] the edges represent bases and the vertices are used to represent a "level", in order to have an unambiguous understand of "positions" in the graph, and as such a level in the graph can be seen as the equivalent to a position in a linear sequence. This concept is shown in figure 3.2.

To overcome the ambiguity of levels shown in 3.2 where the node with level 10 could also have been at level 6 or 7, due to levels of the predeecing nodes, Dilthey et al. [16] introduce a gap node "-" that can be inserted to have persistent levels in the graph.

An example of this model is shown in figure 3.3. Using this scheme we can quickly identify the variation at a level in the graph, however the state space of the graph increases, and it should therefore be determined if the level property

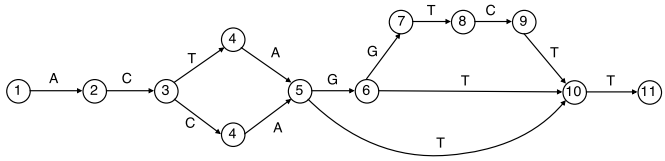


Figure 3.2: Figure 3.12 shown with edges as bases.

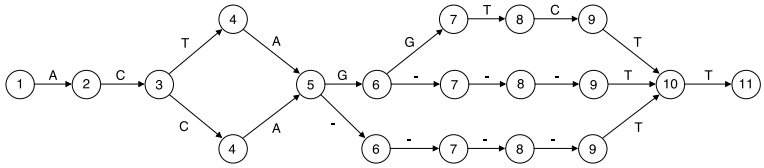


Figure 3.3: Figure 3.2 shown with gap edges to enforce persistent levels.

is an important part of the graph model.

Alternatively we can choose to represent the bases with nodes as shown in figure 3.1. This concept is used in Paten et al. [35] and in this model edges are used to represent the relationship between two bases (predecessor/successor). It can be argued that this model is more intuitive than the one presented in Dilthey et al. [16], because the nodes of a graph are usually used to represent entities, and the edges are used to represent relationships. In Paten et al. [35] there isn't a notion of "levels" in the graph in the same way as in Dilthey et al. [16], however because arbitrary information can be attached to nodes, this concept could be incorporated into the graph, if the need is there.

3.2.2 Break-ends

In regards to structural variation, break-ends are easily expressed in a graph by connecting end points with edges [35, p. 19]. In section 2.3.1 we briefly discussed how there can be uncertainty regarding breakends. Modelling these uncertainties in a graph structure could be done in a relatively simple fashion where instead of having a one-to-one mapping of end points we can instead have one-to-many, many-to-many or many-to-one mapping between nodes with an attached probability of observing each of these edges. The scenario is show in figure 3.4.

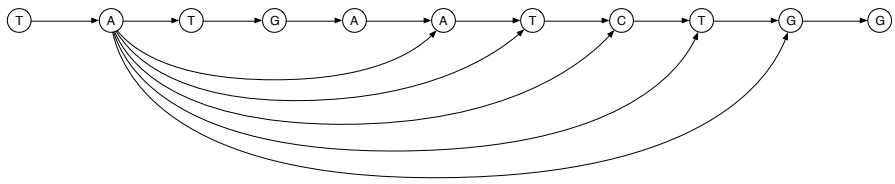


Figure 3.4: Shows a deletion with uncertainty about the end point resulting in a one-to-many mapping from the start of the deletion to the uncertainty interval.

3.2.3 Cycles

We should also consider how to treat cycles in the graph. Cycles can appear if there is a repetition of identical sequences immediately after each other. Cycles can be modelled in the graph intuitively by adding an edge from the end to the beginning of the repeating sequence as shown in figure 3.5.

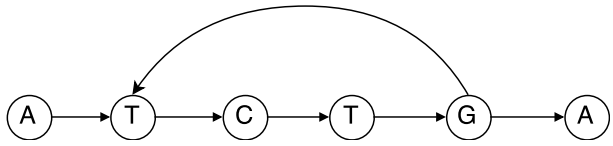


Figure 3.5: An example of a cycle in a graph. This a path in this graph could correspond to the sequence A**TCTG**A, A**TCTGTCTG**A, A**TCTGTCTGTCTG**A, and so on.

Repeating sequences could of course also be expressed by branches of different length, which is the case in Paten et al. [35] where the graph is acyclic.

3.3 Use cases for the genome graph

As shown in figure 3.1, genetic variation can be intuitively represented by using a graph. The graph makes sense because it simplifices the notion of variation and there is less need for a distinction between the types of variation. In the current representation the variation is divided into the groups explained in section 2.3, but these categories of variation are determined by the relationship between the reference and the variation sequence, e.g. an insertion in one perspective is the

same as a deletion in the opposite perspective[1].
In a graph all variation is represented in the same way; as branching paths in the graph. Furthermore Graves [18, p. 5] states that *"an advantage of using graphs for representing concepts is that the definitions can be slowly refined by adding to the existing structure without making changes which would affect previously developed components or queries"*.

For a graph to be a valid choice for modelling genetic variation, the graph should have the same functionality as current genetic data representations, but also provide an improvements in regards to these.
In this section we will justify why a graph is a suitable data structure by discussing its suitability for solving the presented problems.

3.3.1 Mapping

In section 2.5 we restricted the scope of this thesis to not include mapping as part of the solution and implemented tool. However we still want to consider how the genome graph could be used for mapping in the future, and therefore we will discuss the motivation for mapping to a graph here. The idea of mapping to a genome graph is considered in Li [28] and Paten et al. [35] as seen in the following quotes:

> *"A typical use of the graph is to map the sequences of a new individual to the existing variations, in particular large variations."* [28]

> *"To account for natural genetic variation, we consider the more general case in which a reference genome is represented by a graph."* [35, p. 1]

The first thing to note is that if we choose to use a graph as a mapping reference, the concept of having a central sequence dissapears as the different paths through the graph can represent different DNA sequences. If a large enough graph is built with many branching paths of genetic variation, this could be used for a more successful mapping of reads as there is a higher chance of paths in the graph that match the read, and as such a better chance of finding a mapping location with few mismatches as explained in figure 3.6.

[1] An example of this could be the sequences 'A' and 'ACT', where 'CT' can be seen as insertion if 'A' is the reference value, but if 'ACT' is the reference, then 'A' is a deletion in relation to the reference.

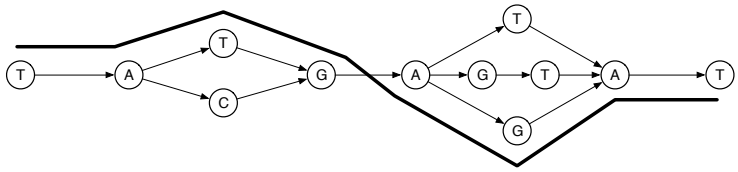


Figure 3.6: Shows how mapping a read (here shown as the thick black line) to a graph could work. The paths of the graph give more possibilities of where to map the read.

A critical part of mapping is to quickly be able to determine if a particular sequence of DNA is present in the reference structure. The way this is done currently as explained in Scheibye-Alsing et al. [37, p. 7] is by splitting the input reads into subsequences of constant length and using a hashing to find out if and where a subsequence is present in the reference genome. However things get more complicated when working with graphs where the context of a base is not necessarily a linear sequence of bases in both directions, but rather trees of contexts. Depending on the number of branches in both direction, the space complexty of storing the context of a base could increase significantly in a graph compared to a linear reference genome. An example of the context for a particular locus in a genome graph is shown in figure 3.7

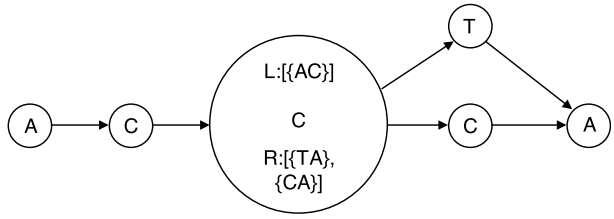


Figure 3.7: The context of the enlarged node with base value "C" is stored in two sets L and R (left and right).

As stated in Paten et al. [35, p. 3] *"base positions within a reference structure should have an aspect of permanence and universality"*. The motivation for this is that entities of the genome graph should be unambigously identifiable, which is an important feature for communicating positions and entities of the graph. In the classical model of reference genomes, this is usually done by stating the position and the chromosome of the base in the reference e.g. [Chr:2 ; Pos: 1000][2]. With the genome graph however, there is no central reference, so the identifier scheme could be defined differently, for example by using UUIDs[3] as

[2]Meaning chromosome 2, position 1000
[3]Universally unique identifiers

in [35] or URIs[4].

Mapping is a very important issue to consider when designing the graph, because one of the future goals for the graph structure we want to build, is that it should be usable in mapping.

3.3.2 Adding variation

Contrary to the classical reference genome, a graph structure does not settle upon one allele for each locus. This is a feature that can help to improve several processes regarding the study of genetics, such as for example mapping, as seen in the previous section. In the graph each locus is a set of alleles which means that the graph can be extended when novel genetic variation is found, and as such there should be schemes for how to merge variation into a graph or even merging two graphs together.

In the context of the earlier mentioned Danish pan-genome in Besenbacher et al. [12], which seeks to create a reference genome for the Danish population, the merging of graphs could be interesting if pan-genomes graphs for distinct populations could be merged into one to create a graph which contains the genetic variation of several populations.

We discussed the problematics of sharing genetic data for example in the form of a VCF file. A graph that contains the same information as the VCF files from which it has been created is of course restricted in its sharability in the same way as VCF files, however a subset of the graph containing only common genetic variation could be shared. As mentioned in 2.3.2, we define rare genetic variation as variation observed in less than five individuals of a cohort. Therefore a subgraph with the alleles observed in more than five individuals can be shared publicly while the complete graph can be stored internally with extra information such as exactly which individuals have which genetic variants for example. Figure 3.8 shows the concept of a sharable subgraph.

The merging og graphs and addition of novel variation are both important features with regards to the sharing of information because through further observations of a rare variants, they can become sharable if we at some point reach the treshold for observations. This principle is shown in figure 3.9.

[4]Uniform resource identifiers

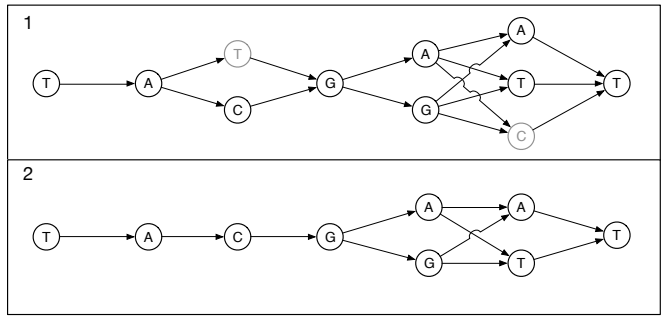


Figure 3.8: The graph in **1** has rare alleles shown with the grey nodes. Because the graph contains these rarely occuring alleles, it cannot be shared, but the graph in **2** which is the subgraph containing only alleles with an observation count above a certain treshold would be sharable.

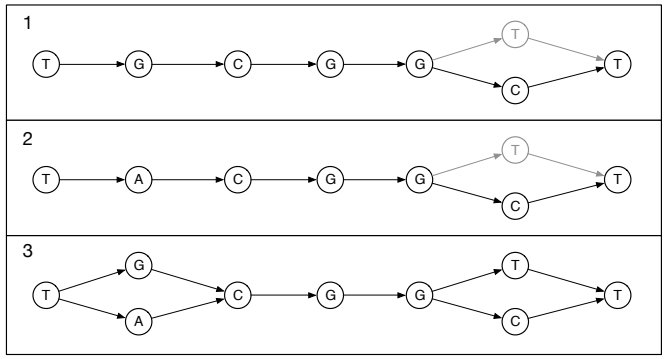


Figure 3.9: The graphs in **1** and **2** have rare variation that cannot be shared, specified by the grey nodes and edges. **3** shows how merging the graph can make the previously rare variant achieve enough observations to be sharable.

3.3.3 Extensibility

So far we have discussed the graph in terms of connecting the variation at different loci. The graph structure however is a very flexible format that allows us to attach different kinds of data to the nodes in the graph. For example it could be interesting to keep a count for the observations of a particular allele in the graph, define regions or bases that relate to specific diseases, or express the probability of observing a particular genetic sequence in the graph. Figure 3.10

illustrates how the probabilities of traversing[5] an edge in the graph depends on the history of the genetic sequence.

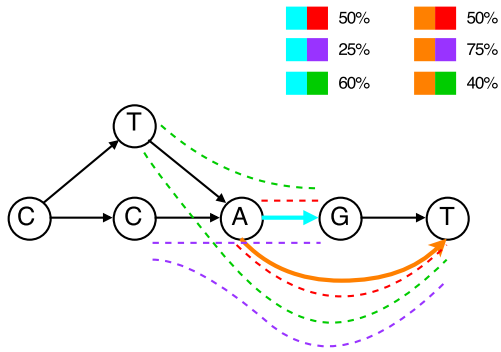


Figure 3.10: The probability of traversing an edge can be related to the previously traversed edges of a sequence. In this example if we look at the outgoing edges from the node with an A base, there is equal probability of traversing the edges to G and T, however if we look at the context, i.e. the previous variants of the path, the probability distributions can look different.

The motivation for storing additional information such as sequence probabilities in the graph reveals that there is a need for having more entities than just bases, and therefore it could be useful to use the entities in the graph for several purposes. As such a central node representing an base could be connected to other nodes with information about that particular node as shown in figure 3.11.

3.3.4 Querying

The ability to link any information to alleles in the graph makes the graph capable of containing a vast amount of information. For example alleles identified to associate with diseases such as diabetes could have edges connecting them to a node regarding this particular disease. This idea is especially interesting if the model is queryable, in which case the relevant parts of the graph could be retrieved through queries such as for example *"Select all alleles assosiated with diabetes"*, *"Select variants present within below 10% of the population"* or *"Select alleles assosiated with both obesity and diabetes."*.

[5]Traversing the graph is interesting in regards to sampling, which is discussed in section 3.3.5.

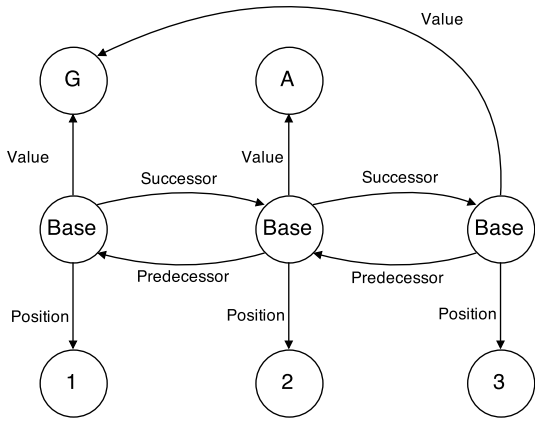


Figure 3.11: Shows how nodes and edges in the graph can serve several purposes.

3.3.5 Sampling individuals

As a means of conveying the information stored in the graph, we can use the aforementioned probabilities to sample individuals. To do this, we traverse the graph according to the probabilities assosiated with edges in the graph and create sampled data that resembles the cohort from which the study was created. Because a graph format is not yet accepted as the primary format for representing genetic variation, the sampled individuals can be stored in VCF files which can then be shared publicly as they only contain common genetic variants. A challenge here is to determine how to generate samples that are statistically similar to the genetics of the cohort, as discussed in 2.3.2, while also removing rare genetic variation. Figure 3.12 shows how DNA sequences can be generated by traversing the graph.

The study in Dilthey et al. [16] also discusses using the graph to sample from. However here the Viterbi algorithm is used, which finds an optimal traversal through a graph, and as such the purpose of this study is to express the "average" individual in terms of genetic variation from a cohort.

3.3.6 Compatibility

A change in the representation of genetic data should be compatible with current collections of data. In connection with this Dilthey et al. [16, p. 2] lists various challenges concerning the use of prior information about variation:

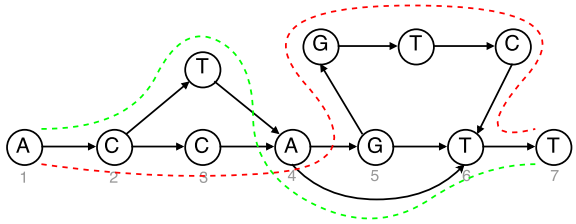


Figure 3.12: Two possible paths in the graph example from figure 3.1. The two sequences corresponding to the paths in the graph would then be **ACCAGGTCTT** and **ACTATT** respectively. To construct the paths shown here, the probabilities attached to the edges of the graph as shown in figure 3.10 could be used.

- There needs to be a data structure for representing genomic variation that can accomodate multiple sources of information.
- As current methods rely on a single primary reference, there needs to be a coherent way of projecting information from a variation-aware reference onto a primary sequence.
- There needs to be methods for validating and comparing output from a variation-aware reference tool and current approaches that rely on a single reference.

In order to create a link between the existing data and a genome graph representation these criterias should be considered.

3.4 Summary

In this chapter we have looked at how genetic sequences can be represented as graphs and what can be gained from adopting this idea.
We summarize the features that the genome graph should be have below.

Mapping - Mapping input reads should be possible in the same way it is possible to map to a reference genome.

Extensibility - Extending the graph with additional information should be possible, and it should be possible to merge graphs together.

Sharability - The data structure should be sharable in order to let the public take advantage of its information. Sampling individuals will be another way of conveying the data contained in the graph.

Queryable - The graph should support querying to allow for the retrieval of entities of interest.

Compatibility - The graph should have a scheme for projecting its data onto a primary reference in order to be usable with the current representation of genetic variation.

Scalability & performance - There should be an emphasis on constructing the graph in such a way that algorithms for creating and working with the graph scale well with the size of the graph. As such the graph and related algorithms should be analysed in terms of time and space complexity.

Looking back to the problem description of section 1.2 and the restriction of the scope of section 2.5, we do not intend to realize the mapping mentioned above. The compatibility will be realized only in the sense that the genetic graphs will be constructed from VCF files and as such can be related back to these. There is not a major focus on querying the graph, but we do intend to realize it in the implementation. The major focus of the thesis are with the scalability of the implementation and the sharability of the graph and the quality generated data.

In the next chapter we look how a genome graph could be designed in order to serve the established purposes.

Chapter 4

Designing the genome graph

In this chapter we will take a closer look at how a genome graph should be designed in order to fulfil the requirements of section 3.4. We will give a brief overview of the entities contained in the graph, and then look at algorithms for constructing and using the graph. As we shall see, the genome graphs contain a lot of data and as such we have an emphasis on reducing both space and time complexities of the algorithms for handling the graph.

4.1 Entities

There is motivation for containing different types of information in the genome graph, which we discussed in section 3.3.3. Before we go into detail of how to construct and work with the genome graph, we should explain the entities that the graph is made up of.

The first thing to note is that a graph is constructed for each chromosome. Chromosomes are separate entities and as such it makes sense from a biological point of view. As we shall see later this separation also makes sense for processing graphs in parallel.

Generally speaking the nodes of the graph are used to represent entities and edges are used to represent the relationship between them. The basic genome graph consists of alleles as nodes with edges between them depending on the locus they belong to as shown in figure 3.1. This means that at each "position"[1] has a number of alleles, which correspond to the observed variation at that position. As we aim to include more information than just genetic variants in the graph, we chose to use nodes to represent alleles rather than edges as in Dilthey et al. [16], because entities representing other information can then be linked by edges to the alleles for example.
As the graph is generated from a number of inviduals which have diffent genomes, we also chose to store the individuals at each position in the graph, divided into sets depending on which genetic variation they have at the position. As we shall see in section 4.5 and 4.6, this information is relevant in regards to sampling individuals.

4.2 Building graphs

The first step for building and using the genome graph is to create the graph. As mentioned the graphs are built from VCF files. Building graphs is a sequential process in which each line of the VCF file is read and the data it contains is added to the graph. Each line in the VCF file contains information about the observed variation of a locus as well as the genotype for each individual. This information is added to the graph by constructing position nodes, and for each of these nodes storing the observed alleles as well as sets of individuals for each genotype. We have chosen to divide the set of individuals observed for each genotype into two; one for observations made with sufficient depth, and one for insufficient depth based on a threshold parameter. Having this information in the data structure can be useful when deciding if rarely occurring genetic variants are genuine or a result of bad coverage.
In terms of running time, the construction of the initial graph depends on both the number of lines in the file e.g. sequenced loci, the total length of genetic variation at each position as well as the number of individuals. If n is the number of positions in the file, k_i is the total length of alleles for position i, and m_i is the number of sequenced individuals for the position we get a runtime of:

$$\mathcal{O}(\sum_{i=1}^{n} k_i + m_i) \;\Rightarrow\; \mathcal{O}(\sum_{i=1}^{n} m_i)$$

[1] We will use the term "position" to refer to a particular locus in the graph.

Because $k_i \leq 2m_i$, as the amount of different variation at a position is at most twice the number of individuals, seeing as each invidual has one genotype consisting of two alleles.

In terms of space complexity we store the genetic variation and the sets of individuals of each position, giving a total size of the graph is:

$$\mathcal{O}(\sum_{i=1}^{n} k_i + m_i) \;\Rightarrow\; \mathcal{O}(\sum_{i=1}^{n} m_i)$$

4.3 Compression and expansion

In an effort to decrease the size the graph, which is desireable given the amount of data involved in sequencing[2] we have looked at compression of the graph. This is a feature that is used in Dilthey et al. [16] for linear parts of the graph, where the edges along the linear path are merged into one edge for that has a sequence of DNA bases rather than just one base. This concept is illustrated with figure 4.1[3].

To further compress a node, an algorithm such as Run-Length encoding [38] could be applied such that repeating sequences of alleles can be expressed by one instance of the allele, and the number of times it appears. Run-length encoding of DNA sequences is discussed in Sirén et al. [38] and concludes that there is motivation for applying this type of compression on genetic sequences. In general we should consider lossless compression algorithms as a means of reducing the space complexity of the graph. We could also choose to use an algorithm like Lempel–Ziv–Welch [40], in which a dictionary of observed patterns is constructed, such that sequences can be expressed by a single resource.

Compressed nodes should however have the ability to be expanded or split at a later point in case we discover novel variation that results in a new branch on a previously linear path. In terms of expanding the graph again, it is important that any compression performed is lossless. In relation to communication positions in the graph, it is also important that positions can be found even if they are part of a compressed stretch.

Another way of compressing the graph, is by collapsing branches that share subpaths. The concept is shown in figure 4.2:1, where two different genetic

[2]Chromosome 21, which is the smallest of the chromosomes has a length exceeding 30 million base pairs [21].

[3]However with merged nodes instead of edges.

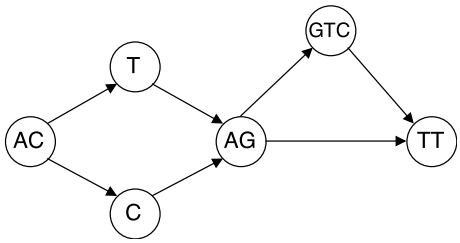


Figure 4.1: Figure 3.12 with compressed nodes.

variants originating from the same position in the graph, share a subsequence of alleles. Here it is obvious that some nodes in the graph can be merged together to reduce the state space of the graph, and instead of having two similarly looking, separate branches in the graph, we can instead have nested branches as shown in figure 4.2:2.

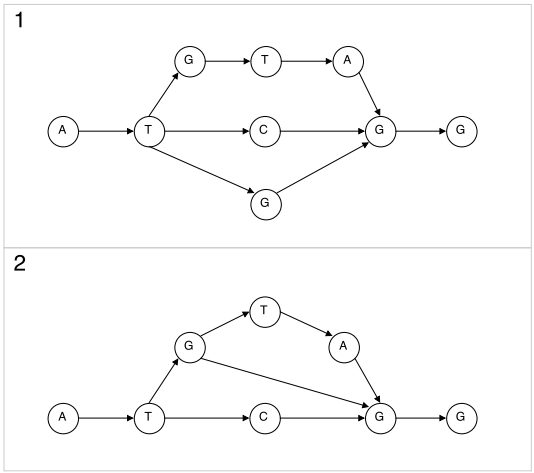


Figure 4.2: The graph shown in 1 can be collapsed into the graph shown in 2, in order to reduce the size of the graph.

Of the two types of compression of the graph, we have only chosen to introduce the compression of linear stretches into the graph. This is due to the nature of graph, which is generally long and shallow, meaning that branches of genetic variation rarely overlap. On the other hand, we mentioned that we store the individuals of the graph for each position, which can be greatly reduced by only storing the set of individuals once for a compressed linear stretch.

For loci without variation i.e. only one allele, all individuals must have the same

genotype as there is only one possibility. This feature allows us to compress linear stretches, because upon expanding the graph we know which genotypes the individuals have for each position of the compressed stretch. This is illustrated in figure 4.3. A compressed node constructed from a linear stretch is positioned in the graph at the position of the first node of the stretch.

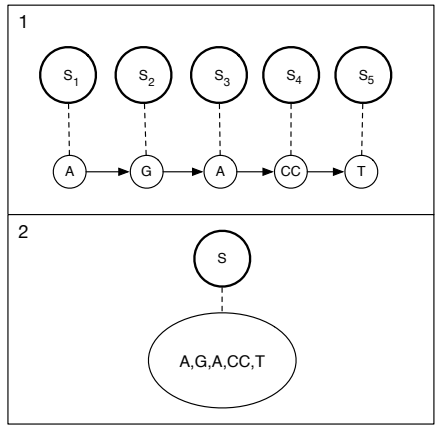


Figure 4.3: The compression of a linear stretch of length five into a single node. Sets $S_1 - S_5$ contain the same individuals, and it is the same individuals that are contained in the set S for the compressed node.

As mentioned, we also divide individuals for each genotype into sets depending on the coverage; one for significant coverage and one for insignificant coverage. This proposes a challenge when compressing, because in order to be able to compress the graph and expand it later without losing information, the division of individuals at each position should remain the same. Therefore an extra constraint is added such that only stretches where the division of individuals into the two sets are the same for all positions of the stretch are compressed. With these constraints, we can ensure that the expansion of a compressed graph contains the same information as the original graph. The original position of alleles contained in a compressed node can be inferred from the position of the compressed node and its internal list of alleles i.e. the reverse process of the compression shown in figure 4.3.

The compression of the graph examines all positions in the graph and merges adjacent positions that contain one allele, which is a runtime of

$$\mathcal{O}(n)$$

where n is the number of positions in the graph. The best case scenario for the

graph compression is that the entire graph is linear, resulting in one compressed node with n internal nodes and one set of m individuals resulting in a total size of

$$\mathcal{O}(n+m)$$

for the graph.

4.4 Merging graphs

In this section we will explain how genome graphs can be merged together. In section 3.3.1 we discussed the motivation for having a graph with as much genetic variation as possible, because it increases the possibility of mapping reads successfully. Furthermore in section 3.3.2 we explained how adding observations of genetic variation can increase the size of the publicly sharable subgraph.

Building the graphs from VCF files makes the merging of graphs relatively simple, because VCF files are constructed in relation to a reference genome. This means that when merging two graphs we have a unambiguous idea of positions in the graph as there is a one-to-one mapping from positions in the graphs to the loci in the reference genome. When merging two graphs, we simply add the genotypes for a position in one graph to the set of genotypes for the corresponding position in the other graph, and in the case that a genotype is already present, we can join the sets of individuals for the genotype in the two graphs. An example of the merging of two graphs is shown in figure 4.4. Upon merging, it must be ensured that the individuals across the two graphs have unique IDs, as otherwise it would be possible for the same individual to have more than one genotype per locus.

The running time of merging two graphs involves iterating through each position of the smallest graph and adding its content to the other graph. This gives a running time of

$$\mathcal{O}(\sum_{i=1}^{min(n_1,n_2)} k_i \cdot m)$$

where n_1 and n_2 are the number of positions in the two graphs, k_i is the total length of alleles for position i in the smallest graph and m is the number of individuals in the smallest graph. In the case that the two graphs do not share

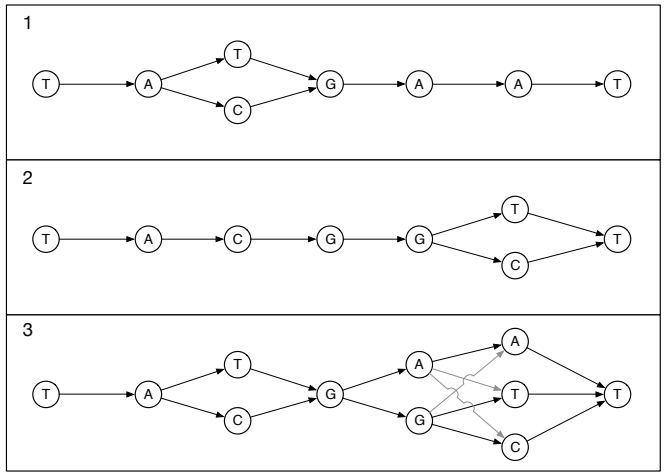


Figure 4.4: **1** and **2** are examples of two distinct graphs for the same set of loci. In **3** the graphs are merged together to form a new graph with the variation represented in both of the graphs. The grey edges show sequences that can be constructed with the graph, but have not been observed in reality.

any genetic variation, the size of the resulting graph will be the sum of the sizes of the two graphs:

$$\mathcal{O}(n_1 \cdot m_1 + n_2 \cdot m_2)$$

m_1 and m_2 being the number of inviduals in graph 1 and 2 respectively, and n_1 and n_2 are the number of positions in the graphs.

4.5 Probability trees

In regards to sampling individuals and in general conveying information about the observed sequences in the data we can take advantage of the observed genetic variation to describe the probability distribution of the genotypes for a position in the graph. To describe these probabilities, we can use the size of the sets of individuals stored for each genotype in the graph as a measure of the genotypes "rarity".
In section 4.6 we will discuss how these probabilities can be used to sample indi-

viduals as a means of conveying the information stored in the genome graph. Alternatively the probabilities of observing certain genetic sequences in the graph can be embedded directly into the graph by using *probability trees*, which we will explain in this section.
In section 2.3.2 we established that there are two statistical properties we would like to be able to convey using the graph, and in turn use to sample individuals with the same statistical properties as the original data.

In each position of the graph there is a number of alleles. The individual of the graph have genotypes made from a combination of two of these alleles. An example of the genotypes for a certain locus is shown in figure 4.5.

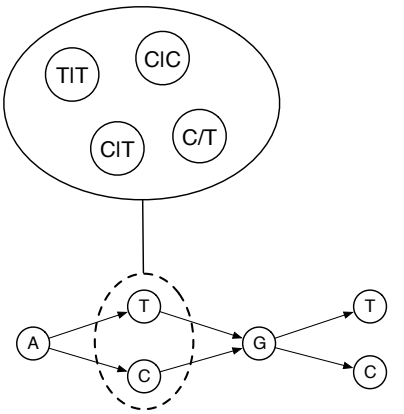


Figure 4.5: Shows the genotypes for a position in the graph with alleles "T" and "C". Four different genotypes have been observed in the individuals in this example.

The probability tree of a position is instantiated with an unnamed root to which children are added depending on the observed genotypes for the position. A node of the tree contains a genotype, the probability of traversing a path from the root of the tree to the node itself, and the number of individuals that have the genotype sequence of the path in their DNA. As we shall see later however, there is a motivation for reducing the space complexity of the probability trees, and as such the probabilities and numbers for individuals are only stored at the leafs of the tree. As the trees aim to be part of the publicly sharable data structure, we do not add rare genotypes. Figure 4.6 the first step of constructing a probability for a position.

We then proceed by adding children to the leafs of the tree by finding the intersection of individuals for the genotypes at the current leafs and the genotypes of the succeeding position in the graph; again we only add children if the in-

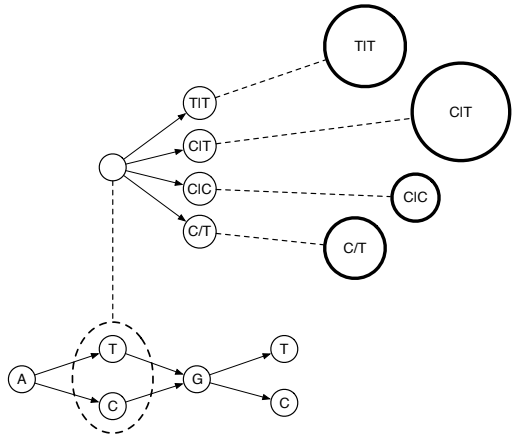


Figure 4.6: The root of the probability tree in the figure has four children, one for each observed genotype. Each child is connected with the set of individuals that have the sequence of genotypes from root to leaf; in this example we have only included the first layer of the tree and therefore the sets for the paths are identical to the sets for the individual genotypes.

tersection of the individuals with the genotypes of the path exceed the chosen threshold. We continue to extend the branches of the tree, as long as the intersection of individuals along the path has a size that exceeds the chosen "rarity" treshold. Figure 4.7 shows an example of this.

In the end we are left with a tree for which each path from root to leaf represents a sequence of genotypes observed in the original dataset, however without rare genetic variation. At each leaf we can keep a count of the number of individuals that have the corresponding sequence of genotypes determined by the path from root to leaf. The numbers across the leafs can then be used to calculate the probability of observing the various paths of the tree, as shown in figure 4.8.

Of course these probabilities are skewed by the fact that rare genetic variation must be removed from the tree. Later we will test if these trees can be used to convey the LD and genotype distribution with a satisfying correlation to the actual distribution.

The probability trees can drastically increase the total size of the graph because they can span many positions in the graph. For this reason it makes sense to prioritize a minimal space consumption at the cost of an increased running time. The first step in doing so, is to only store the number of individuals and

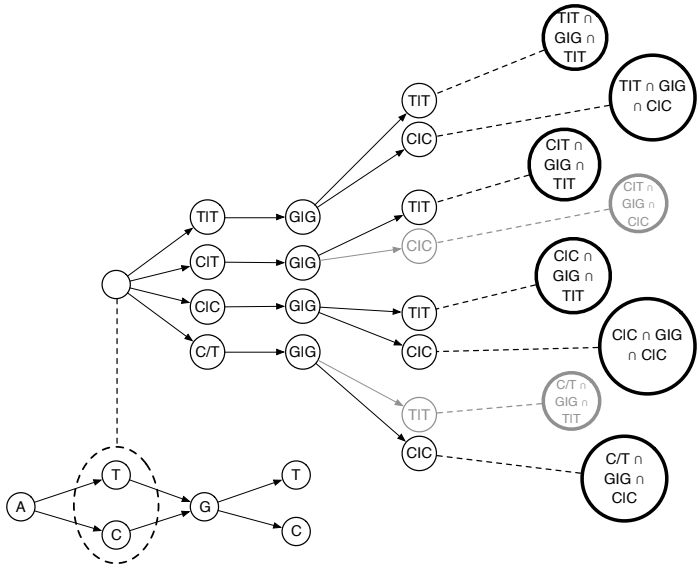


Figure 4.7: An extended version of figure 4.6 with paths for all combinations of genotypes across three positions in the graph. The intersection of the individuals that carry the genotypes of the path is stored at the leafs. The grey leafs have intersections below the chosen treshold and are therefore not included in the tree.

the probability of traversing a path from the root of the tree to that leaf only at the leaf itself. If need be, the number of individuals and the probability of traversing a path from the root to an internal node in the tree can be calculated on demand by iteratively adding the values stored at the leafs of the tree in the above layers.
Another thing that can be done is to compress the tree into a *compressed probability tree*. In the following we will explain the process of constructing a compressed probability tree.

Storing a genotype involves storing each of its two alleles and a phasing type. If all observed alleles and the two phasing types[4] are stored centrally a genotype can be expressed as three resources, one for each allele and one for the phasing type. However the size can be reduced to one, if instead each observed genotype is stored centrally and reused in the graph. This concept is shown in figure 4.9.

Because indels can have an arbitrary size, there is no upper limit on the number of possible genotypes, however SNPs and short indels are the most common

[4]phased and unphased

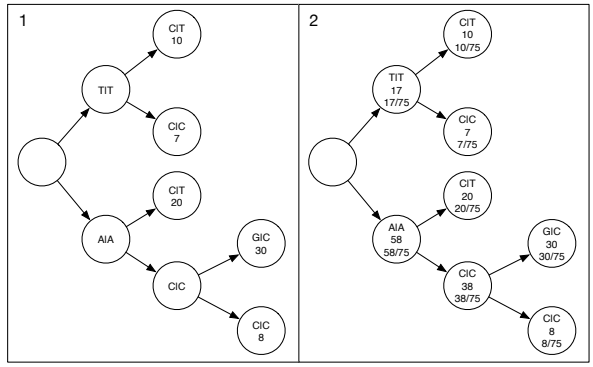


Figure 4.8: **1** shows the initial probability tree with the number of individuals stored at each leafs. **2** shows how the number of invididuals across all leafs in the tree can be used as to specify the probability of traversing each path from root to leaf, and how these values can be accumuluated in higher levels of the tree.

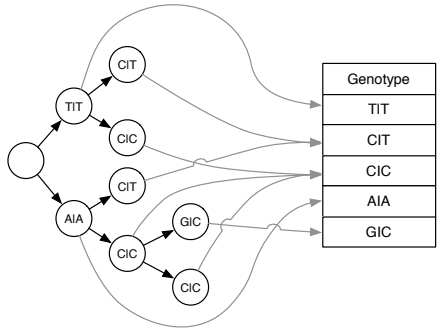


Figure 4.9: Reusable genotypes.

types of variation, meaning that with a reasonable probability the number of distinct genotypes observed can be covered with relatively few resources.

The next step of the compression is concerned with decreasing the amount of nodes in the probability tree. The concept used here is the same as reusing genotypes, but here we reuse sequences of genotypes rather than genotypes individually. The idea behind this is that we can identify a sequence of genotypes of length k with a single global resource. Using this method, the depth of the compressed tree will be reduced by a factor k, thereby decreasing the amount of nodes in the tree. Figure 4.10 illustrates the construction of a compressed tree.

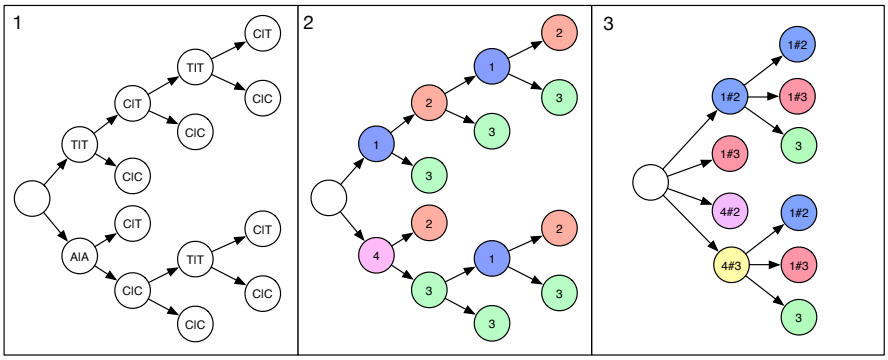


Figure 4.10: Figure showing the construction of a compressed probability tree from a normal probability tree. **1** shows the original probability tree, where each node stores both alleles and the phasingtype. **2** shows the first step of compression where global resources are used to express genotypes. **3** shows the compressed tree. Here the tree is compressed by a value of $k = 2$. A sequence resource is allocated for each color in the tree.

If we consider the case that all genotypes consist of two basevalues e.g. A|G, C|A, G/G etc, there would a total number of 32 different genotypes[5]. The amount of combinations that can be made with these genotypes for a sequence of length k increases exponentially such that a sequence of two genotypes could have $32^2 = 1024$ different combinations and a sequence of three genotypes results in $32^3 = 32.768$ different combinations. This coupled with the fact that the alleles of each genotype can consist of more than one base means that the combinatorial possibilities of genotypes scale drastically as k increases, and thus giving a decreased chance of observing the same sequence in different locations of the probability trees, and thereby being able to reuse a previously constructed resource. On the other hand there is a motivation for increasing the value of k, because it is this factor that decides how much the probability tree is compressed.
One thing to keep in mind when choosing the value of k is that the paths of probability trees in the same region of the genome are likely to overlap. For example choosing a $k = 10$, means that a path of length 200 can be compressed into 20 nodes, each identified by a global resource of 10 genotypes. 19 of these nodes are likely to be reusable at the tree originating 10 positions later on the genome, 18 for the tree at 20 positions later and so on as illustrated in figure 4.11.

[5]Four different base values and two different phasing types gives 32 possible genotypes.

In this thesis we have not compressed the probability trees using classical lossless compression algorithms such as Run-Length encoding or the Lempel–Ziv–Welch algorithm. The latter however could be interesting for the compression of probability trees with a large k because it provides a way of expressing long sequences through combination of shorter sequences. This is a nice feature because the reuse of shorter sequences in the trees is more probable than larger sequences.

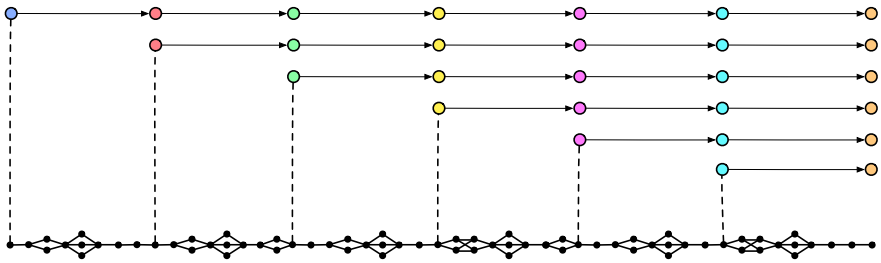


Figure 4.11: Shows how trees at different positions in the graph can reuse the compressed edges. Here a $k = 8$, means that every tree originating a multiple of 8 positions later in the graph is likely to reuse the compressed nodes made in an earlier tree. The color of the nodes in the probability trees specify which nodes have the same sequence of genotypes, meaning that all nodes of the same colour use the same resource. This figure only shows one path for each probability tree.

The worst case for construction of the probability trees in terms of time complexity is that each tree spans all the remaining positions in the graph, which would give a running time of

$$\mathcal{O}(\sum_{i=1}^{n}(n-i)\cdot m)$$

where n is the number of positions in the graph and m is the number of individuals, because for each of the remaining positions the intersection of individuals must be calculated for the current path and the next genotype.
In regards to space consumption the worst case scenario is that $\frac{m}{t}$[6], where t is the threshold for common genetic variation, paths are formed from the root and each of these paths span the remaining positions in the graph. In this scenario the total number of nodes across the probability trees would be:

[6] $\frac{m}{t}$ is the maximum number of sets that can be constructed from m individuals, such that each set exceeds the threshold for common genetic variation.

$$\mathcal{O}(\sum_{i=1}^{n}(n-i)\cdot\frac{m}{t})$$

The time needed for constructing the compressed trees is linear in relation to the number of nodes in the original probability tree:

$$\mathcal{O}(\sum_{i=1}^{n}c_i)$$

Where c_i is the number of nodes for the original tree in position i of the graph. In the worst case, all nodes across the compressed trees are unique, which means that a resource is constructed for each node (rather than having one resource for a set of nodes) resulting in a total size of

$$\mathcal{O}(\sum_{i=1}^{n}\frac{c_i}{k}\cdot k)$$

because the number of nodes in each tree is reduced by a factor k, but each node has a resource size of k.

4.6 Sampling individuals

In regards to sampling individuals we mentioned earlier that there are two distributions of the original dataset that should be reproduced in the sampled individuals, namely MAF and LD.

The study in Dilthey et al. [16] discusses sampling of individuals from a graph in a similar fashion to this project, however here the Viterbi algorithm is used to find an optimal path through the graph, or in other words, to sample an individual that has the most common genetic variation of the observed cohort [16, p. 15-18]. As explained ealier, our motivation for sampling data is that it is a way of conveying a statistical representation of the collected data in a common format while still abiding by the ethical constraints set in place to ensure anonymity.

In this section we propose an algorithm for sampling individuals from a genome graph by using the probability trees discussed in the previous section.

The probability tree is designed such that performing a number of traversals from root to leaf in accordance with the probabilities assigned to the paths in the tree, should convey the LD and MAF distributions for the part of the genome graph that the probability tree covers. As such we can recreate these distributions by traversing through the genome graph via the probability trees. This idea is used for sampling individuals. However the probability trees are unlikely to cover all positions in the graph and as such it is not possible to generate a sample from only one tree. Instead a scheme must be made for how and when to change to a new tree during the traversal of the trees.
There is a motivation for traversing deep into a tree because the traversed path contains information about the likelihood of co-appeareances between genotypes in different positions. However traversing deep into a probability tree is also problematic because it can create a bias towards certain genetic sequences that is not a correct representation of the observed genetics. On the other hand, traversing only short paths of a probability tree means that the history available in long paths is not used to the fullest. The bias that can come with traversing deep into a probability tree stems from fact that rare genetic variation is not present in the data structure. As explained the paths of a probability tree are based on sequences of genotypes observed in the individuals of the graph. However if an individual has a rare genetic variant, the corresponding path will not be included in the probability tree as explained ealier. This means that the individual is excluded from the set of individuals that the lower levels of the tree is built from. As such it is possible to end up with a deep path in the tree based on only a few individuals, and in the scenario where a tree is traversed until the leaf is reached, this will create a bias in the genotypes for the covered positions as shown in figure 4.12.

In the sampling algorithm we first thought of using a FIFO queue of probability trees, where the tree of each position in the graph was added when the sampling algorithm reached that position as shown in figure 4.13.[7]

Choosing a genotype for a position was then done by traversing an edge in the current probability tree, however in order to prevent the bias from figure 4.12 of occuring, a check was made to ensure that possible choices of genotype in the probability tree were the same as those observed in the graph. If it was not the case, we retrieved the next tree from the queue and did the same check until the genotypes of the probability tree were the same as genotypes of the graph. A problem with this approach was that potentially a lot, or all, of the history was thrown away if the trees did not contain the right genotypes and furthermore a strong correlation between genotypes at different positions would be wrongly neglected.

[7]By reaching the position, we mean that the algorithm reached the point where it should determine a genotype for that particular position in the graph.

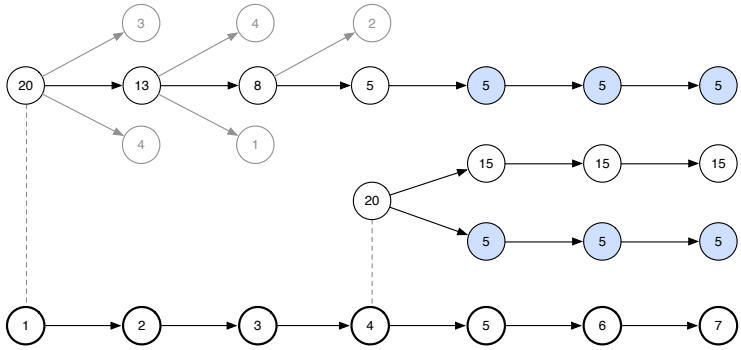


Figure 4.12: Bias can occur as shown in this example. The figure shows the first seven positions of a genome graph and the probability trees for position 1 and 4. In the nodes of the probability trees here is written the number of individuals for the path from root to the node. Grey nodes have been excluded because the number of individuals is below the chosen threshold. The blue paths correspond to the same sequence of genotypes. As can be seen, traversing the tree from position 1 always leads to the blue path, however as can be seen from position 4, it is not the most commonly observed sequence for the particular region of the graph.

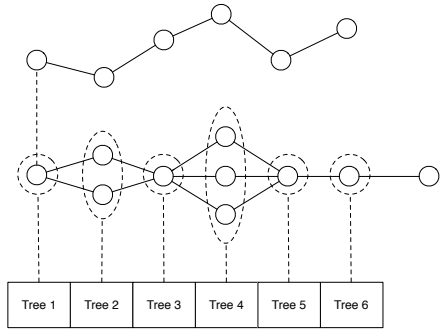


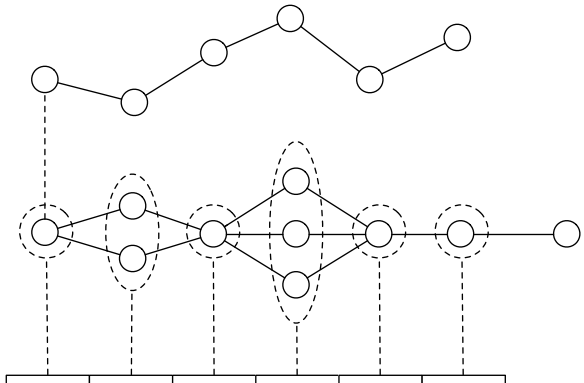
Figure 4.13: Upon traversal of a path in a probability tree, the probabilities for all the covered positions are added to a queue of trees.

To overcome the problems mentioned above, instead of traversing trees to the leafs or assuring that the same genotypes occur in the probability tree and the genome graph, we introduced an aspect of randomization regarding when to change to a new probability tree.
In this algorithm a contant size FIFO queue of probability trees is used. Trees are only removed from the queue in the case where the queue has reached the

maximum size and a new tree is added meaning that if a tree in the middle of the queue is currently in use, there is a chance of going back to an older tree with a more extensive history later on in the sampling. The choice of which tree to use at each position in the graph is decided by a probability p such that the head of the queue is chosen with probability p. The general probability of choosing a tree in the queue is expressed as:

$$p_{tree_n} = (1-p)^{n-1} \cdot p$$

where p_{tree_n} is the probability of choosing the tree at index n as shown in figure 4.14.



Tree 1	Tree 2	Tree 3	Tree 4	Tree 5	Tree 6
p	(1-p) p	$(1-p)^2$p	$(1-p)^3$p	$(1-p)^4$p	$(1-p)^5$p
0.4	0.24	0.144	0.086	0.052	0.031

Figure 4.14: For each tree in the queue, the probability of choosing the tree is based on its position in the queue. The probabilities for choosing each tree when $p = 0.4$ is shown in the figure.

Depending on the choice of p we can ensure that with a high probability a tree with more history is chosen, but we also make it possible to not always rely on the deepest available tree thereby eliminating bias towards certain sequences of genotypes.

A key concept of changing to a new tree is to traverse the path according to the genotypes already determined for the current sample as illustrated in figure 4.15. This is always possible because the set of individuals in a position in the graph is a subset of the set individuals in the succeeding position.

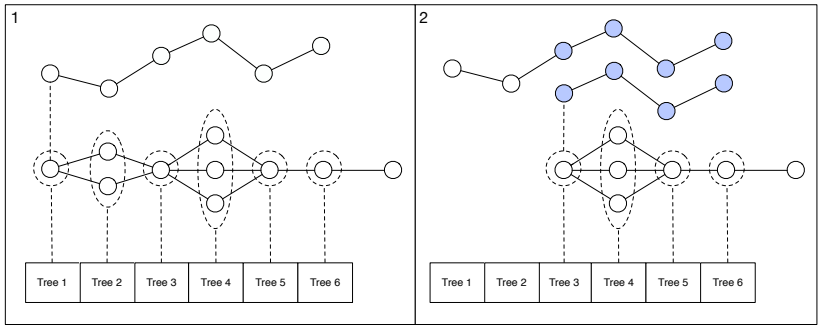


Figure 4.15: The path from **1** is reconstructed in the newly chosen tree in **2**. In this version of the algorithm we only remove trees from the queue when the maximum size is reached, meaning that a later point we can go back to using a deeper tree than the current one. Therefore along with the queue of trees, the traversed path over the corresponding positions is stored.

As the probability trees are compressed as discussed in the last section, they have to be expanded each time they are to be used. Therefore the running time of the sampling algorithm involves expanding the compressed tree for each position in the graph, deciding which tree to use and in the case the current tree is switched, reconstructing the current path in the new tree. In the worst case, the current tree is switched at each position, alternating between the deepest and second deepest trees[8] in the queue, giving a running time of

$$\mathcal{O}(\sum_{i=1}^{n} c_i \cdot k + l) \;\Rightarrow\; \mathcal{O}(\sum_{i=1}^{n} c_i \cdot k)$$

Where n is the number of positions in the graph, c_i is the number of nodes in the compressed probability tree for position i, k is the compression factor of the probability tree and l is the depth of the new tree. $l \leq c_i \cdot k$ is true because the number of nodes in a tree is at least the same as the depth of the tree.

We propose storing the sampled individuals in a "sampled graph" that is similar to the genome graph in that it contains alleles and for each position, sets of individuals according to their genotype. This graph has a size of

$$\mathcal{O}(n \cdot m)$$

[8]This is the worst case because when a new tree is chosen, the stored path has to be traversed in the new tree.

Where m is the number of sampled individuals and n is the number of positions[9].

VCF files can then be created by iterating through each position of the sampled graph and writing out a line to the file with the alleles contained among the genotypes followed by the genotype for each sampled individual in that position.

4.7 Summary

In this chapter we have discussed the design of the genome graph on a theoretical level. Making the right design decisions for the graph is important, because genome data collected in projects such as the Danish pan-genome can reach into the gigabytes and even terabytes. The quote below highlights this importance:

> *"In developing any complex, data-intensive system, the most crucial technical decision to make is the choice of data representation. Data representation affects the ease at which algorithms can be developed, the efficiency at which they perform, the extensibility and maintenance of the software, the accuracy of the data to be captured, and the operations which can be performed."* [18]

We have therefore discussed which measures can be taken to reduce the size of the graph. We have also discussed algorithms for using the graph.

In the next chapter we will look at technologies that allow us to construct, express, work with, and share the genome graph.

[9]The number of positions in the sampled graph is identical to the number of positions in the original graph.

Chapter 5

Relevant tools and technologies

In this chapter we will discuss the existing formats for representation of graphs both in and outside of genetics, and discuss if these formats are suitable for the requirements established in the last chapter. Along with the discussion of graph formats, we will also look at relevant tools and programming APIs in regards to the implementation of the genome graph tool, and based on these dicussions make a choice of which technology to use.

5.1 Graph formats in genetics

Graphs in assembly of genomes have existed for a long time and is an essential part of genome assembly. Graphs as reference structures are also gaining more and more attention, which we discussed in 3.1, but as research in this topic is in a relatively early stage, the focus currently lies on the theoretical side of the graphs and their use cases. Although no standard has been decided for the representation of genome graphs, there are formats which seek to fulfil this role.

One format that seeks to capture the graph attributes of genome assemblies is FASTG [6]. Originally the motivation for creating the FASTG format was

sparked by the limitations of the FASTA format[1]:

> *"This traditional linear representation[2], however, has one deep flaw – it has very limited capacity to express branching that can arise from allelic polymorphism, intrinsic limitations of the data, or even limitations of the algorithms used to assemble them."* [6]

The key feature of the FASTG format is its ability to express the uncertainty that can occur in assemblies by allowing the assembly to have alternative branching paths for regions of uncertainty. Figure 5.1 shows an example of the FASTG format.

```
>Z;
ATAT NNNNNNNNNNNNNNN [13:gap:size=(13,10..35), begin=a, end=g |
                      >a:b,c,d;
                      GGG
                      >b:b,c,d;
                      A
                      >c:e,f;
                      CA
                      >d:g;
                      CAC
                      >e:e,f;
                      T
                      >f:g;
                      AA
                      >g;
                      CCC          ]   GATGAT
```

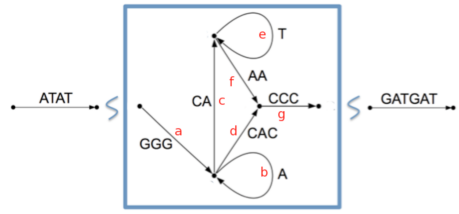


Figure 5.1: An example of of the FASTG format and the corresponding graph as seen in [6].

In terms of expressing the graph we have discussed in chapter 3, the format can be used to express the branching nature of the graph, however it is not flexible in its ability to attach other types of information to the graph such as for example the probability trees discussed in section 4.5.

Another format that focuses on the representation of genome assemblies as graphs is the Graphical Fragment Assembly [27]. The format was created in an attempt to provide a better alternative to the FASTA format [27].

Figure 5.2 shows an entry in the GFA format. We will not go into detail with the syntax here, but as can be seen it is a format that is designed for solving one purpose, namely expressing genome assebly graphs, and as such is not easily extendable with meta information for the graph.

[1]FASTA is a format that represents linear genome assemblies
[2]From the FASTA format.

```
H   VN:Z:1.0
S   1   2   CGATGCAA   *
L   2   3   5M
S   3   4   TGCAAAGTAC   *
L   3   6   0M
S   5   6   TGCAACGTATAGACTTGTCAC   *   RC:i:4
L   6   8   1M1D2M1S
S   7   8   GCATATA   *
L   7   9   0M
S   9  10   CGATGATA   *
S  11  12   ATGA   *
C   9  11   2   4M
```

Figure 5.2: An example of a genome assembly expressed in the GFA format as seen in Li [27].

5.2 Other graph formats

The formats discussed in section 5.1 have certain qualities that are useful in the context of expressing genomic data as graphs. In terms of expressing alternative genetic sequences as branches in a graph, the existing formats have the desired capabilities, even though the motivation for creating them in the first place was to express genome assemblies and the uncertainty involved with such. However the lacking extensibility of the formats means that they are not particularly suited for creating a genome reference graph in which we wish to be able to store other information, and it is therefore natural to look elsewhere for formats that fulfil our requirements.
We established easy extensibility of the graph as one of the main requirements. The reasoning behind this requirement is that we wish to create a genome graph that can serve many different purposes such as being a mapping reference, generating sampled individuals from and linking relevant information to loci or regions of interest on the genome. It is therefore important that we choose a format in which we can add the needed information to make the graph serve new purposes and that added information does not change the usability of the graph in the already established use cases. With this in mind it makes sense to consider formats where the data itself is represented in a graph where entities are represented with nodes and relations are represented with edges.

Formats such as GML [24], DOT [17] or GXL [25] are usually used in graph visualisation. However these formats do allow for the specification of meta data in nodes and edges, and as such could be used to express a graph structure without being related to its visualisation. These formats come with programming interfaces, and as such it would be possible to work with the graph in a programming environment and storing it in a graph format.

It would be possible to express the genome graph in a designated graph format such as the ones mentioned above, and store the graph persistently in a specialized graph database such as Neo4J. Storing genetic graphs in graph databases, which is discussed in Have and Jensen [22], would allow queries to performed on the graph, however this approach does not satisfy the needs for sharability of the graph and integration with other resources. In the next section we will discuss a format that supports this.

We could consider defining our own graph format, but it is hardly a desirable solution, as it only limits our possibility of creating an extensible and commonly used data source and would also require a lot of work to develop.

5.3 The Semantic Web and Linked Data

In the previous sections we have looked at formats that capture the interconnectivity in a graph. Some of the formats are even extensible, platform independent and suited for representing an arbitrary data structure. In terms of creating a universally usable genome graph it is desirable that the data structure can be extended by anyone to suit any purpose. This way, one central data structure of genomic information can be used in many different settings. Furthermore if the data structure supports querying it would make it easier for users to retrieve relevant data from the graph depending on their use case.

For this reason it makes sense to look at *Linked Data*. Linked Data describes a web of data, the *Semantic Web*, that is accessible and extensible by anyone. In the Semantic Web, data available in different sources on the web is linked together in a graph like structure as shown in figure 5.3[3].

> *"Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked to from external data sets."* [14]

The linked nature of the Semantic Web is interesting in several regards when it comes to the genome graph presented in this project. Firstly it is interesting because the linkage of information forms a graph structure and is therefore an intuitive format for describing a graph. Secondly the interconnectivity and extensibility of the linked data means that the graph can be extended by linking

[3] As seen in `http://linkeddata.org/static/images/lod-datasets_2009-07-14_cropped.png`

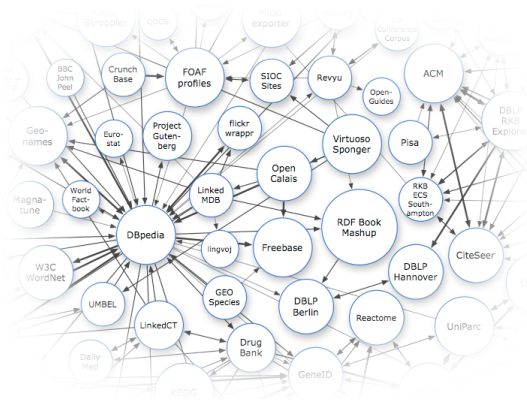


Figure 5.3: The idea behind the semantic web where data available on the web can be linked together to form a graph of connected data sources.

new information to and from the graph without altering the already existing nodes and edges of the graph. Lastly the Semantic Web fulfils our requirement of making our data sharable and supports querying.
Sharing and linking data between sources across the net is of course something that requires a common standard for communication. The discussion of whether this common standard should be part of the Semantic Web is discussed in Miller [33] where it argued that the Semantic Web as a concept should be distinguished from the involved technologies. In this thesis we will not go into the discussion, but instead discuss the format in which resources are described in the Semantic Web. In relation to that Berners-Lee [11] lists four principles of Linked Data and the Semantic Web:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs. so that they can discover more things.

In the following we will take a look at the proposed standard for representing resources on the semantic web, namely RDF.

5.3.1 Resource Description Framework

RDF is a standard model for data interchange on the web [3]. It links data through *triples* which are entities that consists of three resources; a subject, a predicate and an object. The triple can be seen as directed graph consisting of two nodes, the subject and object with an edge going from the subject to the object labelled with the relationship of the two entities as shown in figure 5.4.

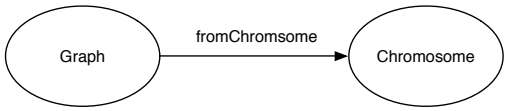


Figure 5.4: A triple showing the relationship between a genome graph and a chromosome.

The subject and predicate are URIs and the object can be either a URI or literal. The smart thing about identifying resources through URIs is that anyone with access to the internet can look up these resources and other relevant resources they are linked to. Sometimes a URI is not necessary to represent a resource, which is why literals can be used instead. Figure 5.5 shows an extension of figure 5.4 where the resources are identified by URIs and literals for simple data such as names and numbers.

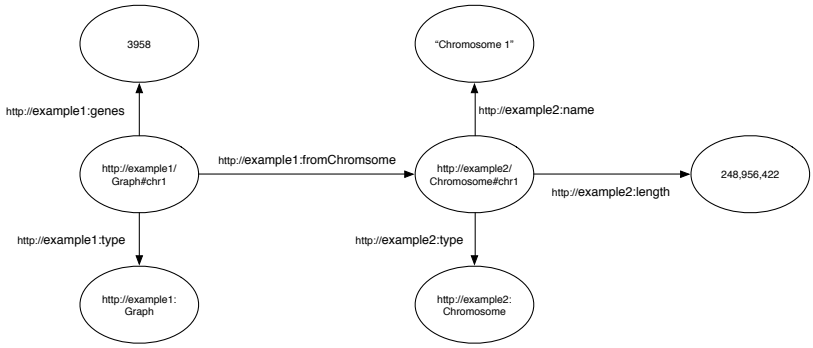


Figure 5.5: An RDF graph for the relationship between a genome graph and the chromosome it is built from.

In the example the resources are located at different hosts, *http://example1* and *http://example2*, but because HTTP URIs are used to identify them, it is possible to link them together. With RDF and the Semantic Web we are able to

share our data as part of a bigger context, where relevant parts of the genome graph can be linked to other sources in the Semantic Web and others can extend the graph by linking their own data to it. As seen in the following quote, the current size of the Semantic Web already exceends several billion triples:

> *"... as of September 2010 there are 28,562,478,988 triples in total published by data sources participating in the Linking Open Data Project"* [30]

Several *serialization* formats exist for RDF graphs, which are formats for writing RDF resources and triples to a file. In section 6.2.5.2 we will show an example of the genome graph written in the Turtle serialization.

5.3.2 Integration with the Semantic Web

In order to get the most out of the Semantic Web, the resources should be part of a context. This means that a clear definition of the resources themselves and the relationship they have to other types of resources in the Semantic Web should be made. Ontologies are used for expressing this relationship between concepts, and with a clearly defined ontology, the users of the Semantic Web will know exactly what type of data is linked to the different resources. In order to describe the entities of the genome graph and their relations, we could construct our own ontology, but seeing as one of the main principles of the Semantic Web is to reuse existing data, it makes sense to find existing ontologies that can be used and extend these if necessary.
BioInterchange is a *modern data processing application that brings genomics data to the cloud* Baran et al. [10], which has a defined ontology of genetic entities that can be used in the Semantic Web. In the implementation it would make sense to bind as many of the entities in the graph as possible to concepts of this ontology as a first step of integrating into the Semantic Web.

In this thesis we are mainly concerned with the modelling and usability of the genome graph, and there is therefore not a major focus on integrating the graph into the Semantic Web. However in the following we will discuss the choices of relevant technology that provide the best solution in terms of working with the graph and sharing it in the Semantic Web.

5.4 Tools for the Semantic Web

Currently there exists a number of tools regarding working with RDF. For persistent storage there are triple stores which are databases for triples that can execute semantic queries to retrieve entities of the graph as discussed in Wilkinson et al. [41], and alternatively RDF serializations can be used to write resources and triples to a file. There also exists a number of Semantic Web frameworks for different programming languages that allow programmers to extract data from and write to RDF graphs.

5.4.1 Semantic Web APIs

To provide a seamless transition from objects in a programming environment to RDF graphs we can make use of Semantic Web frameworks. The Semantic Web of course has many users with different backgrounds and therefore there are many APIs in various different programming languages.

Apache Jena is an open source Semantic Web framework for Java. The framework has a number of functionalities that make it possible to work with RDF in a programming environment. The central object in Jena is the "Model" which holds the RDF graph and can be sourced with data from files, databases, URLs or a combination of these. The Model object can be queried through SPARQL, a query language for the Semantic Web. The model object can also be easily be written to files or databases.
Jena also includes a rule-based inference engine [1] that can perform reasoning based on Web Ontology Language[4][5] and RDFS ontologies which can be used to infer relationships between entities beyond what is explicitly stated.
Handling triples in Jena can be tedious as shown in the example in listing 5.1 from [2] and instead of thinking in resources, predicates and objects, Jena can be used along with Jenabean which is a library for persisting JavaBeans [5] to RDF. This way we can create our graph data structure as Java objects and use a Bean2RDF writer to convert it to RDF. Likewise Jenabean allows us to create Java objects from RDF, which means that once the graph has been stored in RDF, it can be recreated as Java objects.

Sesame [9] is an alternative to Jena, and the two do not differ particularly. However Sesame does not have support for Web Ontology Language. Like Jena, Sesame also offers querying and Java binding tools though Alibaba to generate RDF from Java objects and vice versa.

[4] Known as OWL.
[5] JavaBeans are objects that follow a defined design pattern and are easily serializable [20].

JRDF [34] is another Java framework for RDF. It offers query handling, disk and memory based storage of RDF graphs like Jena and Sesame. However the framework does not currently support RDF to Java object and Java object to RDF generation.

More APIs exist in both Java and other programming languages, however we will not go over all of them here, but instead use the remainder of this section to motivate a choice of one of the mentioned Java APIs.

```
1  // some definitions
2  String graphURI    = "http://somewhere/Graph";
3  String graphName    = "Chr1";
4  int geneCount    = "3958";
5
6  // create an empty Model
7  Model model = ModelFactory.createDefaultModel();
8
9  // create the resource
10 // and add the properties cascading style
11 Resource genomeGraphChr1
12   = model.createResource(graphURI)
13         .addProperty(GraphOntology.Name, graphName)
14         .addProperty(GraphOntology.GeneCount, geneCount);
```

Listing 5.1: Jena example inspired by [2].

5.4.2 Persistent storage

> *"Graphs are ubiquitous in bioinformatics and frequently consist of too many nodes and edges to represent in random access memory. These graphs are thus stored in databases to allow for efficient queries using declarative query languages ..."* [22, p. 1]

A crucial part of RDF is how to store the linked data. RDF graphs can become quite big, and it is therefore important to have persistent storage that scales well and performs queries efficiently. Storing massive RDF data sets is a science in and of itself as discussed in Luo et al. [30], however we will briefly touch upon some possible choices here.

The Java frameworks of interest (Sesame and Jena, because JRDF does not provide Java object to RDF support and vice versa) we have mentioned both come with triple store solutions[6]. In Bizer and Schultz [13] an extensive study can be found for the performance and scalability of the triples stores TDB, SDB for Jena and the Sesame triple store. The study comes to the conclusion that in terms of loading triples, Jena TDB is more than one order of magnitude faster

[6]Sesame, Jena TDB and Jena SDB

than SDB and Sesame. However in terms of performing queries on the data, Sesame has the best performance. In terms of scalability TDB demonstrates the best results.

In this project we have not integrated persistent storage using triple stores, but it is a priority in the future. Instead we have stored the generated RDF graphs to text files by using the RDF serialization formats. Storing the graph in a triple store makes sense in the future because it is more efficient in terms of disk space usage than an RDF serialization, and can efficiently execute SPARQL queries.

5.5 VCF parsers

We mentioned earlier how we are going to build the genome graph from VCF files, and therefore it makes sense to choose a programming language which has libraries for handling VCF files. Luckily there exists a number of VCF parsers in different languages such as Java (GATK), Python (PyVCF), C++ (vcflib) and Perl (VCFTools).

5.6 Chosen tools

On the background of the discussed tools, we have found that RDF and the Semantic Web fulfils our needs in terms of providing a format in which we can easily share the genome graph. It is also appealing that the graph can be extended by other developers across the world. We have discussed a number of tools regarding working with RDF. We choose to work with Jena, because it allows us to work with Java objects and provides well performing persistent storage through a triple store (which can become relevant given the amount of data collected in a project like the danish pan-genome). Furthermore the choice of using Java means that we have access to a VCF parser through GATK.

5.7 Summary

In this chapter we have looked at the possibilities of storing graphs in formats that are capable of expressing a genome graph which consists of many different entities. A major influence on the choice of format was its ability to be shared

easily. The existence of relevant tools for handling and sharing the graph influenced the choice of technology. For this reason we have chosen to represent the graph as triples, which can be shared as part of the Semantic Web. Java has been chosen as the programming language because it provides a number of useful tools such as a VCF parser, Java object to triple generation and triple to RDF serialization.

Chapter 6

VGTool

6.1 Design

We have now discussed the modelling of genetic variation as graphs, the motivation for doing so and some interesting use cases regarding genome graphs in chapter 3. In chapter 5 we have discussed and decided upon suitable technologies that will allow us to build an RDF graph from VCF files, using Java along with tools for parsing VCF files and serializing Java objects to RDF.
In this thesis a tool called *VGTool* has been designed and implemented. In this chapter we will go over the design of the tool from a structural standpoint, and then explain more in detail about the design of the implemented algorithms.

A number of decisions have been made in order to make VGTool perform well when working with large amounts of data. There is also a focus on making the implementation easily extensible. In the following section we will discuss the structure of the design and go into detail with some of the main priorities for the design.

6.1.1 Tool overview

Figure 6.1 provides an overview of the VGTool and its collaboration with other technologies. As can be seen in the figure, the input to VGTool can either be VCF files which are parsed with the GATK Java library, or a previously constructed VariantGraph loaded from triples. Persistently storing the graphs can be done with Jena and Jenabean to either a triple store database or through a serialization of RDF such as for example the Turtle format.

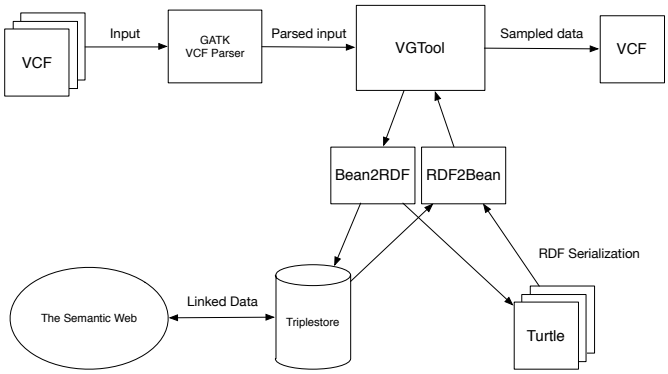


Figure 6.1: The different technologies of the implementation and their interaction.

Triple store and Semantic Web integration has not been implemented in the tool in this thesis, but would be an obvious extension for the future.

6.1.2 Design requirements

Before we go into the details of the implementation in chapter 6.2, we can take a look at the overall design of VGTool. The design of the tool focuses on a few key points:

Modularization of graph handling features. The motivation for this is to avoid a tightly coupled system such that the algorithms discussed in chapter 4 can be changed or replaced independently of each other.

Parallelization is a key part of VGTool and working with genome data in general. The motivation for parallelizing processes comes from the sheer size of the collected data.

Minimization of memory allocation is important for the same reasons as parallelization. The highly repetitive nature of the human genome and DNA in general means that several entities in the genome graph can be represented using references to global resources.

In the following we will discuss these design requirements further.

6.1.3 Modularization

VGTool has a number of implemented features and has been constructed with a long term goal of fulfilling other purposes such as serving as a read mapping reference as discussed in 2.2.3. This need for extensibility should be reflected in the implementation and as mentioned there is also a motivation for being able to easily change and improve the already implemented features. For this reason the implementation has been modularized such that each feature in regards to working with the genome graph is a separate module/component. The list below gives a short description of the different components that perform the different functionalities of VGTool.

Graph reader: The component for parsing the input. In case a different format than VCF was to be used, this module could be changed to support that format.

RDF reader: The component for reading a previously constructed graph stored as triples.

Graph compresser: The component for compressing a graph in order to reduce memory allocation.

Graph expander: The component for expanding a compressed graph.

Graph merger: The component responsible for merging graphs.

Probabilities builder: The component that builds probability trees in a graph.

Individual sampler: The component that can generate sampled individuals from a graph.

Samples writer: The component that stores the constructed samples to VCF files.

RDF writer: The component that serializes entities of the graph and stores them persistently.

6.1.4 Parallelization

In chapter 4 we discussed the different algorithms for building and using the genome graph and ensured that both the asymptotic running time and space complexity of the graph were kept at a reasonable level. However with the amount of data involved in studies such as the Danish pan-genome, there is a clear motivation for increasing the performance of the data handling. In this section we will discuss how parts of the designed algorithms have been parallelized to improve their performance, and which measures have been taken for avoiding race conditions when several threads manipulate the graph concurrently. Later in section 6.3, we will perform a series of tests to analyse the effect of the parallelization.

6.1.4.1 Compressing, expanding and writing graphs

The simplest form of parallelization has been performed for the tasks of compressing, expanding and writing graphs to files. The parallelization here is achieved by assigning a thread to each graph object, and have that thread perform the chosen task. Remember that we construct distinct graphs for each chromosome, meaning that a total of 23 sets of graphs can exist at the same time. Even more graphs can exist, as graphs for the same chromosomes can be constructed from different input, however when more than one graph exists for a given chromosome, they are automatically merged together.

As we shall see later in section 6.3, neither compression nor expansion are very time demanding tasks in comparison to the other graph algorithms. For this reason, we have not chosen to parallelize the tasks further, even though this would be possible by delegating parts of the graph to different threads and have them perform the task of either compressing or expanding the given subgraph. As for writing the graph to a file, we use an external library, and as such do not have control over the parallelization of the task. With the use of a triple store however, it would be possible to persistently store and retrieve the genome graph and its entities in parallel.

6.1.4.2 Merging graphs

Separate graphs are constructed for each chromosome which means that the graphs for each chromosome can be merged independently of each other. To take advantage of this fact the merging of graphs is done on separate threads for each chromosome as shown in figure 6.2.

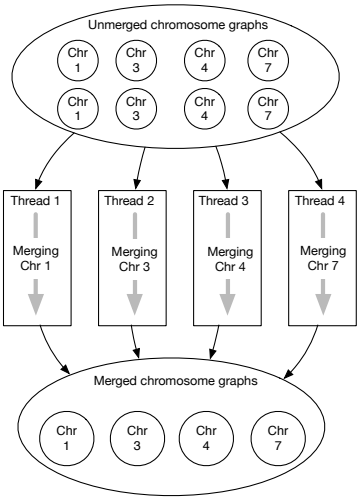


Figure 6.2: Merging different chromosome graphs on different threads.

Furthermore, if many graphs exist for one chromosome and are to be merged together into one graph, these graphs could be merged pairwise iteratively on different threads in order to speed up the merging process as shown in figure 6.3.

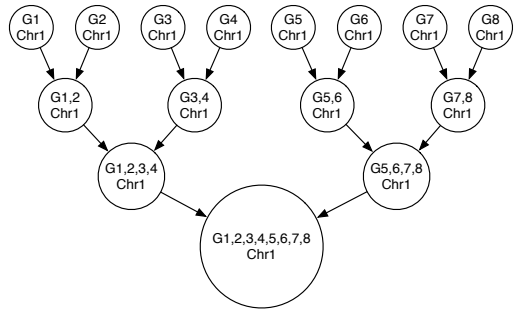


Figure 6.3: Pairwise merge of graphs.

Pairwise merging of graphs has not been implemented, but depending of the running time of the merging algorithm, there could be motivation to implement it in the future.

6.1.4.3 Generating probability trees

A major concern when constructing the probability trees is the time and space consumption of the process. For the construction of the trees we can take advantage of parallelization by dividing the graph into smaller pieces and then assigning different threads to building the probability trees for each part as shown in figure 6.4.

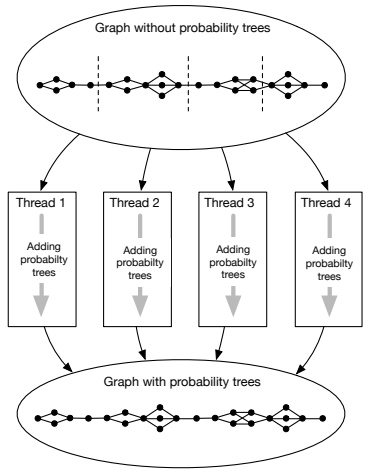


Figure 6.4: Parallelization of probability tree construction.

However the entire graph should be available to access for all threads because the probability trees can span positions exceeding the interval given to the thread. Fortunately there is no critical section regarding accessing the graph among the threads, because information is only read from, and not written to the graph.

The process of generating the probabilty trees, or more specifically, the compressed probability trees on different threads proposes two critical sections; the first critical section is concerned with reusing genotypes at the nodes of the probabilty tree as explained in chapter 4. As several threads are assigned with constructing trees, and thereby storing unique genotypes, we must ensure that there is mutual exclusion when writing to the hashmap of genotypes. Here it is important to remember, that if a genotype has already been stored, retrieving it is not a critical section. Instead when a thread retrieves a genotype, it first checks if has been stored previously, and if not, it enters the critical section to save it in the shared hashmap. Upon entrance in the critical section it checks again that genotype still does not exist, as the current thread could potentially have been waiting to enter the critical section due to another thread that was

in the process of saving the same genotype.
The other critical section is in regards to storing genotype sequences as central resources, which allows them to be reused to reduce the space complexity of the compressed probability trees. The principle is the same as above, where a global hashset of resources is kept and threads use the hash value of a genotype sequence to retrieve the global resource.
Deadlocks will not occur as the conditions of *hold and wait*, *resource holding* and *circular wait* do not hold for the algorithm, which is required for the possibility of a deadlock occuring as explained in Havender [23].

In section 6.3 we perform tests of how this parallelization affects the performance of the generation of probability trees. Depending on the outcome of these tests, there might be a motivation to solve the issue of having a bottle neck in the shape of a critical section. For this, a scheme could be used where instead of using one hashmap for storing genotypes and one for genotype sequences, each of these could be divided into several smaller maps.
For example if we have 100 threads that all work on storing global genotype references, we could share the genotypes out across 100 hashmaps. Another hashmap can then be used to point threads to a genotype map, for example by taking the current genotypes hashvalue modulo 100 to get an even load for each genotype hashmap. Writing to the map would still be a critical section and require mutual exclusion of threads, but the probability of threads accessing the same hashmap could be reduced with this method. This is illustrated in figure 6.5. In terms of asymptotic time and space complexity, this approach is not different than the single hashmap approach.

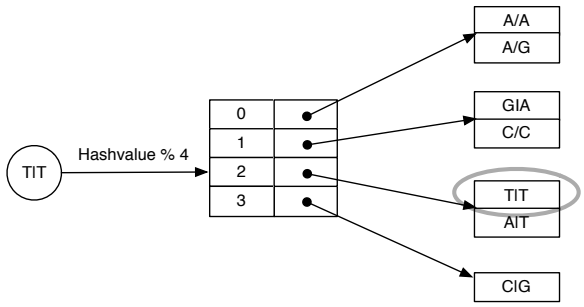


Figure 6.5: Nested hashing.

To speed up the sampling, a thread is assigned to the creation of each sample. Because the threads for each sample insert into the same sampled graph, there is a critical section for which a monitor has been used to ensure mutual exlusion upon insertion. A different design here could allow threads to insert more efficiently which we will explain further in section 6.3.1.6

During the sampling the compressed probability trees explained in section 4.5 are expanded before they can be used. Currently each thread does this individually and releases it again when it is no longer relevant. However instead of letting each thread expand and later release the trees, a scheme could be constructed where trees are expanded once, used by all threads, and then released again once all threads are done using them. For this to work, the threads should traverse the graph at the same pace, which could be achieved by inserting barriers at fixed intervals in the graph such that no thread traverses too far ahead of the others as illustrated in figure 6.6.

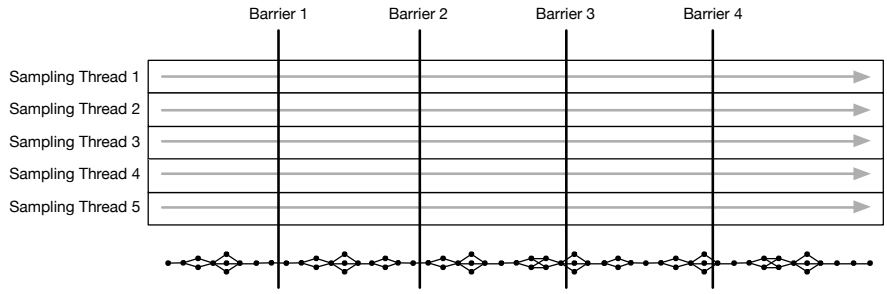


Figure 6.6: Barriers to ensure side by side traversal of the genome graph for the sampling threads.

6.1.5 Minimal memory allocation

We have already touched upon the steps taken to reduce the space complexity of the graph. In this section we will briefly sum up those steps.

6.1.5.1 Global resources

The genome graph contains many elements that represent entities. As such we have made an effort to store reusable entities as global resources that can be linked to any part of the graph. This is the realization of the idea presented in figure 3.11. The entities we have chosen to represent as global resources are the individuals of the graph, the four base values that alleles are made up of and the observed genotypes. In terms of Semantic Web integration, reusing entities of the graph as much as possible makes sense, because it allows users of the Semantic Web to link to those particular entities. For example each of the four

DNA bases[1] in the RDF graph could be linked to external data sources with information of their chemical compound.

6.1.5.2 Compressing

The other aspect of reducing the space complexity of the graph comes from compression. We have already discussed how compressing linear stretches of the graph is desirable, because it allows us to express the genetics of the observed individuals for that part of the graph with only one set rather than a set of individuals for each position. We also covered how compressing probablity trees to reduce the amount of nodes in the tree is desirable, because the overlapping nature of the trees makes it possible to reuse the compressed genotype sequences in several trees.

As we shall see in the tests of space consumption for genome graphs in section 6.3.1.5, the major source for space allocation is the probability trees. However in a graph without these trees[2], the main part of memory comsumption comes from the set of individuals that are stored at each position. We therefore propose a method for storing sets of individuals in a compressed manner, which takes advantage of their unique IDs. The compressed set of individuals relies on the individuals being alphabetically sorted according to IDs. This way, a set can be expressed by the ID of the first individual in the set, and a number determining how many of the next individuals in the sorted individual collection are in the set, similar to the Run-Length encoding algorithm. Figure 6.7 shows how a set can be expressed using this method.

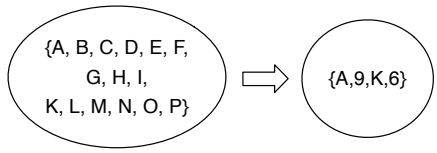


Figure 6.7: Expressing a subset of individuals A-Z in a compressed set.

[1] adenine, cytosine, guanine, thymine

[2] For an internal version of the graph the probability trees are not strictly necessary, because we can store the individuals from the original observations instead. Furthermore a graph without probability trees could function in other use cases, such as mapping for example.

6.1.6 Design summary

In this section we have taken a look at the overall structure of VGTool and its connection to other technologies. We have also discussed some of the key design focuses that makes the implementation both easily extendable and the performance of the algorithms well.

In the next section we will take a closer look at the implementation and describe key parts of the code.

6.2 Implementation

In this section we will take a closer look at the implementation of VGTool. We will provide an overview of the program and look at relevant parts of the code. Figure 6.8 shows the state space of VGTool.

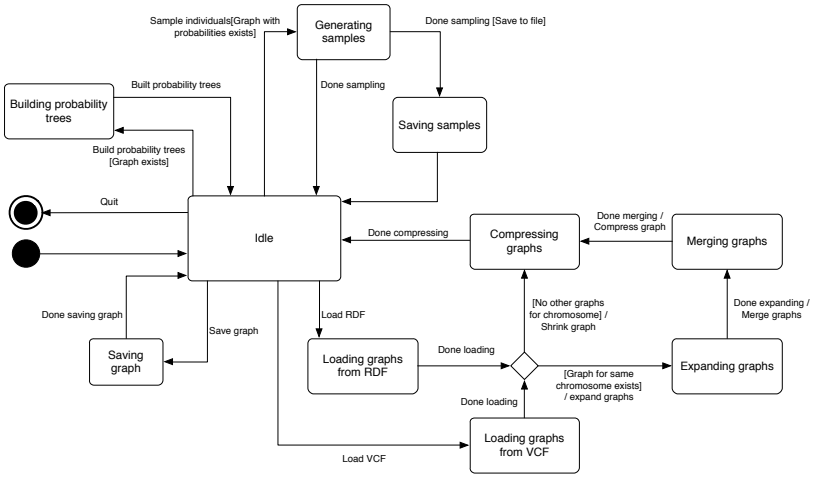


Figure 6.8: State chart showing the various states of the VGTool.

6.2.1 Implementing the graph

In the following we will go over the implementation of the graph structure in the light of the analysed requirements of chapter 3 and the graph design discussed in chapter 6.1.

The central object of the implementation is the VariantGraph. A VariantGraph object is created for each chromosome from the input. The individual chromosomes can reach lengths spanning hundreds of millions of positions[3]. By creating several graphs instead of one we can easily manipulate one chromosome at a time or all chromosomes in parallel. Due to structural variants however the graphs should be able to be linked together.
The structure of the entities contained in the VariantGraph is shown in figure 6.9.

[3]For example chromosome 1 has a length of 248,956,422 bases [4].

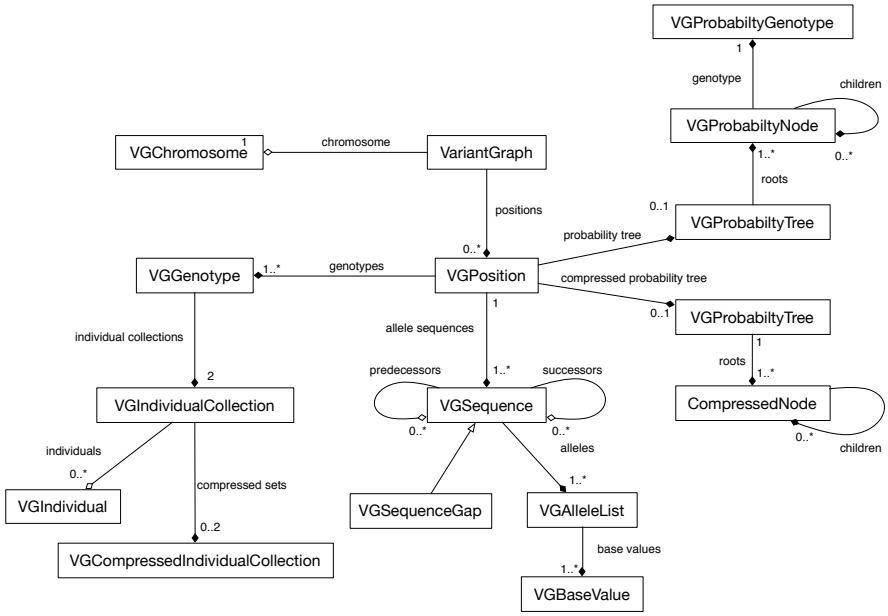


Figure 6.9: Class diagram of entities connected to the VariantGraph.

Figure 6.9 shows a class VGSequenceGap which is a subclass of the VGSequence class. We implemented this class to be able to connect positions in the graph that are not adjacent. However we do not currently use it in the implementation, as the sorted list of positions in listing 6.1 can express the same relationship.

```
1  @Namespace(JenaObject.ns)
2  public class VariantGraph extends JenaObject {
3      private VGChromosome chromosome;
4
5      public ArrayList<VGPosition> positions;
6      public Map<Integer, VGPosition> positionsMap;
7
8      public boolean hasBeenShrinked;
9      public boolean hasProbabilities;
10
11     ...
12
13     public VGChromosome getChromosome() {
14         return chromosome;
15     }
16
17     public void setChromosome(VGChromosome chromosome) {
18         this.chromosome = chromosome;
19     }
20 }
```

Listing 6.1: A selection of the VariantGraph object.

Figure 6.9 also shows how a position can contain two types of probability trees; a compressed and an uncompressed version. The same principle applies to the VGIndividualCollected, which can either contain a set of individuals or a compressed set, which was explained in section 6.1.5.2.

The fields contained in the VariantGraph object are shown in listing 6.1, and as can be seen, the graph contains both a list and a map for the positions due to a need for deterministic iteration through the graph and constant time lookup of positions. As a graph can also be compressed and extended with probability trees, booleans are used to store if this has been done. Each position in the graph holds information in regards to the observed variation at the position, the individuals and their genotype, and also a probability tree which is built from the position itself and succeeding positions as explained in 4.5. The VGPosition object used to represent a locus can be seen in listing 6.2. Each VGSequence in the sequence set specifies distinct allele observed at the position and for each observed genotype, a set of the individuals with that genotype is stored. Furthermore a positions probability tree is either represented as a VGProbabilityTree or a VGCompressedProbabilityTree, however only one of these is allocated at a time and they can be converted to one another.

```
1  @Namespace(JenaObject.ns)
2  public class VGPosition extends JenaObject implements Comparable<VGPosition> {
3
4      private VariantGraph graph;
5      private int position;
6
7      public Set<VGSequence> sequencesSet;
8
9      private Map<Pair<VGSequence, VGSequence>, VGIndividualCollection>
           individualsForGenotypeUnphased;
10     private Map<Pair<VGSequence, VGSequence>, VGIndividualCollection>
           individualsForGenotypePhased;
11
12     private VGProbabilityTree probabilityTree;
13     private VGCompressedProbabilityTree compressedTree;
14
15     ...
16 }
```

Listing 6.2: A selection of the VGPosition object.

The variation stored for the positions in the graph is represented by the VGSequence object. The VGSequence object has information about its predecessors and successors because although we have a sorted collection of positions in the graph, there is the possibility of structural variation resulting in edges between non-neighbouring positions. The VGSequence object can also represent compressed parts of the graph as discussed in 4.5, and as such contains a list of alleles rather just a single allele. Upon expansion these alleles can be reinserted into the graph in incremental fashion from the origin of the compressed sequence.

```
@Namespace(JenaObject.ns)
public class VGSequence extends JenaObject {

    public List<VGAlleleList> allelesLists;

    public Set<VGSequence> predecessors;
    public Set<VGSequence> successors;

    VGPosition position;
    private short i;

    ...
}
```

Listing 6.3: A selection of the VGSequence object.

6.2.2 Project settings and global resources

In the implementation we use static globally accessible resources to define settings such that the implemented algorithms can easily be tweaked.

```
public class Constants {
    //RDF url prefix
    public static final String SOURCE = "http://vg";
    public static final String NS = SOURCE + "#";

    //Threshold that determines what is good coverage
    public static int coverageThreshold = 0;

    //Threshold that determines what is rare genetic variation
    public static int individualsThreshold = 5;

    //Limit of probability trees depth. -1 is no maximum depth.
    public static int maxDepthForProbabilityTrees = -1;

    //Size of tree queue for sampling
    public static int treeQueueSize = 20;

    //Probability of choosing head of tree queue.
    public static float treeChoiceProbability = 0.5f;

    //Should compress
    public static boolean compressTrees = true;
    public static boolean compressIndividualSets = false;

    //Persistent storage type. Database or RDF serialization.
    public static boolean shouldUseDatabase = false;
}
```

Listing 6.4: Settings for the tool

Listing 6.4 shows some of the different settings that can be used in the tool, for example the base URI for resources in RDF.

We also use global resources for entities in the graph that should be reused. Examples of these are the individuals of the graph, the base values of alleles, genotype IDs and compressed probability paths as shown in listing 6.5.

```
1  public class GeneticsUtils {
2
3      public enum PhasingType {
4          UNPHASED, PHASED
5      }
6
7      private static HashMap<String, VGBaseValue> baseValues = new HashMap<String,
           VGBaseValue>();
8      private static HashMap<String, List<VGBaseValue>> alleleListHashMap = new
           HashMap<String, List<VGBaseValue>>();
9      private static HashMap<String, VGProbabilityGenotype> genotypeHashMap = new
           HashMap<String, VGProbabilityGenotype>();
10
11     private static HashMap<String, String> probabilityPaths = new HashMap<String,
           String>();
12
13     private static HashMap<String, Short> uniqueGenotypeMap = new HashMap<String,
           Short>();
14     private static HashMap<Short, String> uniqueIdToGenotypeMap = new
           HashMap<Short, String>();
15
16     private static short uniqueGenotypeCount = 0;
17
18     ...
19
20 }
```

Listing 6.5: Some of the global resources used in the graph.

6.2.3 Algorithm components

In section 6.1.3 we discussed dividing the functionality of the program into several components and listed which parts of the tool could be a seperate component. In order to fully utilize this design, a clear definition of the protocols for communication with the components should be defined, such that developers can change a component without affecting other functionalities of the program. Figure 6.10 shows the components of VGTool along with the input they take and the output they produce. All that is required for changing a component in this design is that the new component follows the same specification of input and output.

Some components rely on graphs with certain properties, for example the individual sampler component requires that the graph which is to be sampled from has probability trees. Currently it is specified in the VariantGraph object if the graph contains probability trees with a boolean, however we could consider changing the component protocols to only require specific types and as such a new graph object (*ProbabilitiesVariantGraph* for example) could be

Component	Input	Output
Graph reader	VCF file	VariantGraph
RDF reader	RDF graph	VariantGraph
Graph compresser	VariantGraph	VariantGraph
Graph expander	VariantGraph	VariantGraph
Graph merger	Set<VariantGraph>	VariantGraph
Probabilities builder	VariantGraph	VariantGraph (with probability trees)
Individual sampler	VariantGraph #Samples	SampledGraph
Samples writer	SampledGraph	VCF file
RDF writer	VariantGraph	RDF graph

Figure 6.10: The different components of the VGTool with a specification of the input and output. The RDF writer component produces an RDF graph, which can be written to an RDF serialization format or a triple store for example.

created, which inherits all the properties of the VariantGraph, but also ensures that probability trees have been constructed.

6.2.4 Threads and critical sections

Several parts of the implementation have been parallelized to improve performance. The ThreadHandler class handles the instantiation and delegation of tasks to threads. Once threads are started with the purpose of each solving a small part of a bigger problem, a barrier is put in place such that the execution of the main thread continues when all threads have finished their assigned task. Figure 6.6 shows an example of how the threadhandler instantiates threads to solve a task[4] where each thread is given a list of positions for which it creates probability trees.
There is a motivation for being able to control the amount of threads, such that we do not switch between contexts of different threads too often. As such it

[4]Here it is the task building of probability trees in the graph

makes sense to not instantiate more threads than there are processors. Tasks such as compression currently uses one thread per chromosome at maximum, but the sampling of individuals instantiates threads for each individual concurrently, such that for example sampling of 1000 individuals creates 1000 threads. A future change to the implementation that could result in an improved performance could be to have a fixed size threadpool, which is at most equal to the number of processors.

```
1  public static void startGraphProbabilityThreadsForGraphs(Set<VariantGraph>
       graphSet) throws InterruptedException {
2
3      int splitBy = Runtime.getRuntime().availableProcessors();
4
5      for (VariantGraph g : graphSet) {
6
7          List<List<VGPosition>> choppedPositions =
               Lists.partition(g.getPositions(), g.getPositions().size() /
               splitBy);
8
9          CountDownLatch latch = new CountDownLatch(choppedPositions.size());
10
11         for (List<VGPosition> positions : choppedPositions) {
12
13             GraphProbabilityInitialiserRunnable runnable = new
                   GraphProbabilityInitialiserRunnable(positions, latch);
14             Thread thread = new Thread(runnable);
15             thread.start();
16
17         }
18
19         latch.await();
20         latch = new CountDownLatch(choppedPositions.size());
21
22         for (List<VGPosition> positions : choppedPositions) {
23             GraphProbabilityTreesRunnable runnable = new
                   GraphProbabilityTreesRunnable(positions, g, latch);
24             Thread thread = new Thread(runnable);
25             thread.start();
26         }
27
28         latch.await();
29
30     }
31 }
```

Listing 6.6: Initialization of threads that build probability trees for different regions of the graph.

In section 6.1.4 we discussed the critical regions that can occur from manipulating the graph in parallel and how mutual exclusion should be achieved for those regions. One example of this is in the creation of probability trees in the graph, where each newly observed genotype is assigned an ID. As responsibility of generating probability trees, and thereby observing genotypes, is delegated to several threads, the assigning of the genotype IDs is atomized. To avoid a bottle neck in regards to accessing the shared hashtable of genotypes and IDs, only the assigning of IDs has been atomized as shown in figure 6.7.

```
1  public class AlleleUtils {
2
3      private static HashMap<String, Short> uniqueGenotypeMap = new HashMap<String,
           Short>();
4      private static HashMap<Short, String> uniqueIdToGenotypeMap = new
           HashMap<Short, String>();
5
6      private static short uniqueGenotypeCount = 0;
7
8      public static Short uniqueStringForGenotype(VGProbabilityGenotype genotype) {
9          if (!uniqueGenotypeMap.containsKey(genotype.toString())) {
10             mapGenotype(genotype);
11         }
12
13         Short id = uniqueGenotypeMap.get(genotype.toString());
14         return id;
15     }
16
17     private static synchronized void mapGenotype(VGProbabilityGenotype genotype) {
18         if (!uniqueGenotypeMap.containsKey(genotype.toString())) {
19             uniqueGenotypeMap.put(genotype.toString(), uniqueGenotypeCount);
20             uniqueIdToGenotypeMap.put(uniqueGenotypeCount, genotype.toString());
21             uniqueGenotypeCount++;
22         }
23     }
24
25     ...
26 }
```

Listing 6.7: The assigning of IDs to genotypes is a critical section. The synchronized method *mapGenotype* prevents race conditions from occuring.

6.2.5 RDF and Jena

In this section the use of Apache Jena and Jenabean to work with RDF graphs in Java will be covered.

6.2.5.1 The JenaObject

Any class that should be part of the RDF graph subclasses the abstract class JenaObject which is shown in figure 6.8. This object ensures that an ID, and thereby a unique URI for the resource is created. The URI is created as concatenation of the base url, Constants.NS, the name of class and a unique ID for the object. The assigning of IDs is particularly important, because a clash of IDs between objects means that only one will be stored. For example assigning an ID to a VariantGraph object is trivial, as there is only one graph for each chromosome (if there are more than one, they are automatically merged),

but for resources such as the variation of a position, care must be taken when assigning IDs.

```
@Namespace(JenaObject.ns)
public abstract class JenaObject {
    public static final String ns = Constants.NS;
    public String id;

    public JenaObject(String id) {
        setId(id);
    }

    @Id
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    ...
}
```

Listing 6.8: The abstract JenaObject.

The general principle that has been used for IDs is that each graph entity uses the ID of the entity it is contained in with an extension for the object itself. In an effort to reduce the memory allocation that would occur if every JenaObject stored its own ID as String object, instead objects are given the minimal required resources from which the ID can be generated on demand. This means that if an objects ID is generated as an concatenation of a parent objects ID and an identifier for the current object, a reference to that parent object is given to the current object along with the identifier extension.
It is also important here that when for example merging, compressing, or expanding graphs, we are change the entity IDs contained in them, meaning that some scenarios require a new assigning of IDs.

6.2.5.2 Serialization and storage

Conversion from Java objects to a Jena Model object which contains the triples of the RDF graph (and in turn can be saved to persistent storage) can be done with the use of Jenabean. We already discussed how subclassing the JenaObject ensures that every entity to be stored in RDF is assigned a URI. The triples of RDF are inferred from beans which have fields that are other beans or primitive types and follow a getter/setter pattern. The example in figure 6.1 shows the VariantGraph object which has a field of type VGChromosome with a getter and setter. Jenabean interprets this pattern and from it generates the triple that binds the graph and the chromosome together. Listing 6.9 shows an example of the triples where VariantGraph is the subject, written in the Turtle format.

```
<http://vg#VariantGraph/chr1>
        a                       vg:VariantGraph ;
        vg:chromosome           <http://vg#VGChromosome/chr1> ;
        vg:hasBeenShrinked      true ;
        vg:hasProbabilities     true ;
        vg:id                   "chr1"^^<http://www.w3.org/2001/XMLSchema#string> ;
        vg:positions            [ a
             <http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq> ;
                                  <http://www.w3.org/1999/02/22-rdf-syntax-ns#_1>
                                          <http://vg#VGPosition/chr1:11905665> ;
                                  <http://www.w3.org/1999/02/22-rdf-syntax-ns#_2>
                                          <http://vg#VGPosition/chr1:11905793> ;
                                  <http://www.w3.org/1999/02/22-rdf-syntax-ns#_3>
                                          <http://vg#VGPosition/chr1:11906733> ;
                                ] .
```

Listing 6.9: Triples where a VariantGraph entity is the subject. Here the base url for the entities in the graph is `http://vg`.

The Bean2RDF class of the Jenabean library allows a set of related bean objects to be written to a Jena model easily. This is done by instantiating a Jena Model object and a Bean2RDF object that writes resources to the model. For writing objects as triples to the model there are two methods; `saveDeep(Object bean)` and `save(Object bean)`. The basic save method writes a single resource to the model, and the saveDeep method traverses through the graph of JenaObjects originating from the object that the method is called with, saving all objects as resources and generates the triples according the relation between the objects. This traversal depends on whether or not the setter method for a JenaObject has been used; if the method has not been called, it is assumed that the object is not instantiated and therefore shouldn't be written to the file.

```
public void writeToFile(VariantGraph vg) throws IOException {

    Model m = ModelFactory.createDefaultModel();
    m.setNsPrefix("vg", Constants.NS);

    MyBean2RDF writer = new MyBean2RDF(m);

    File file = new File("VCFDataBase/VCF_Files/" + vg.getId() + ".ttl");
    file.getParentFile().mkdirs();
    if (!file.exists()) {
        file.createNewFile();
    }

    writer.saveDeep(vg);

    FileOutputStream fop = new FileOutputStream(file);
    RDFDataMgr.write(fop, m, RDFFormat.TURTLE);
}
```

Listing 6.10: The writeToFile method loads the VariantGraph and related beans into a Jena Model and saves them to a Turtle file.

However sometimes it is desirable not to keep parts of the graph in memory at all time, such as for example with probability trees that are only kept in a compressed version, and therefore the getter method for the probability tree should generate the tree on demand from the compressed tree. Listing 6.10 and 6.11 shows how the graph can be saved to, and loaded from RDF.

```
1  public static void readGraphFromFile() {
2          Model m = ModelFactory.createDefaultModel();
3          RDFDataMgr.read(m, "VCFDataBase/VCF_Files/chr1.ttl", Lang.TURTLE);
4
5          Jenabean b = Jenabean.instance();
6          b.bind(m);
7
8          Collection<VariantGraph> vgs = b.reader().loadDeep(VariantGraph.class);
9          for (VariantGraph vg : vgs) {
10             EnclosingModel.putNewGraphForKey(vg.getId(), vg);
11         }
12 }
```

Listing 6.11: Loading a graph that has been serialized in the Turtle format.

Once the beans have been written to the Jena Model, this model can be saved to persistent storage, such as a triple store or written to a file in one of the many RDF serializations.

6.2.5.3 Querying the model

Once the VariantGraph has been loaded into the Jena Model it is possible to query it using SPARQL queries. Exposing this functionality through a SPARQL endpoint means that users of the Semantic Web can perform queries on the graph. Alternatively it can be used locally in the tool for example for testing purposes. Listing 6.12 shows how a query is performed using a String.

```
1  private void performQuery(String queryString) {
2        try {
3            Query query = QueryFactory.create(queryString);
4            QueryExecution qexec = QueryExecutionFactory.create(query, model);
5            ResultSet results = qexec.execSelect();
6            System.out.println("Rownumber: " + results.getRowNumber());
7            for (; results.hasNext(); ) {
8                QuerySolution soln = results.nextSolution();
9                RDFNode x = soln.get("varName");
10               Resource r = soln.getResource("VarR");
11               Literal l = soln.getLiteral("VarL");
12           }
13       } catch (Exception e) {
14           System.out.println("Bad query " + e.toString());
15       }
16   }
```

Listing 6.12: Querying the Jena Model.

6.2.6 Implementation summary

In this section the implementation of VGTool has been covered with a focus on the overall structure of the implementation and key parts of the code such as the multithreading, algorithm, entity and RDF implementation. The next section will test VGTool in terms of its performance, scalability and also with regards to the quality of sampled data.

6.3 Tests

We have now covered the design and implementation of VGTool. In order for VGTool to be usable in real world scenarios, there is however a need for testing the scalability, performance and correctness of the implementation. For example the sampling algorithm is constructed from assumptions and intuition by using the probability trees as discussed earlier, but because the data is altered as a result of the removal of rare varition we should test if the sampled data correlates satisfactorily with the actual data.
It is also crucial that the algorithms of the tool scale well given the amount of data that can be contained in the input VCF files.

This chapter will be divided into two kinds of tests of VGTool; one for the scaling and performance of the tool and one for the quality of the sampled data in relation to the actual data.

6.3.1 Scaling and performance

In this section we will perform a series of tests to analyse the performance of VGTool and test that the asymptotic running times of chapter 4 are correct.
We will test running time and space consumption for varying sizes of graphs, as well as how multithreading affects performance for graphs of constant size. VGTool contains many different algorithms, that can be adjusted with different settings, and to limit the amount of tests, we have picked tests that show the performance of the discussed algorithms of chapter 4. We also test the I/O regarding the tool. These things are tested to provide insight into which parts of the tool that are the bottle necks in regards to performance such that we will know where future efforts for optimizing the tool should be applied.

Each different type of test in the following section has not necessarily been run on the same machine, so the result of the tests should be viewed in isolation e.g. the running time of one algorithm should not be compared to the running time of another algorithm.

For generating test data we wrote a small script which can take VCF files as input and modify things such as the number of lines in the file, i.e. the number of loci, the phasing of genotypes, the observed alleles for loci or the chromosomes to which loci belong. In the tests, a set of files were created as extracts of different sizes from the same VCF file.
The file sizes we used are listed in figure 6.11.

File	Size	Rows	Individuals	Variation only
File 1	3.1 MB	5000	150	Yes
File 2	6.5 MB	10000	150	Yes
File 3	67.2 MB	100000	150	Yes
File 4	168.8 MB	250000	150	Yes
File 5	676.1 MB	1000000	150	Yes
File 6	11.6 MB	500	6000	No
File 7	24.3 MB	1000	6000	No
File 8	48.1 MB	2000	6000	No
File 9	123.5 MB	5000	6000	No
File 10	250.4 MB	10000	6000	No

Figure 6.11: The used test files.

As can be seen in figure 6.11 we used two types of files in the testing; files where only variation is stored and files with all observed positions. Which files have been used depends on each test, for example compressing graphs requires the graphs to have linear stretches and therefore file 6-10 have been used here.

Generally testing the memory allocation of a Java application is hard because of the automatic garbage collection performed by Java. We therefore tested the minimal feasible amount of memory to perform a task by decrementing the maximum heap size until the program gets an OutOfMemoryError or the garbage collecter uses more than 98% of the CPU cycles, also resulting in an error.

6.3.1.1 Loading

The loading and construction of the VariantGraph is currently sequential and as such is not improved by the number of processors available. Figure 6.12 shows the performance of loading and constructing the initial graph.

As can be seen in figure 6.12, constructing the graph can be done with a heap size about 15 times that of the initial file size. This is for files that only contain variation and therefore cannot be compressed. Currently the required heap size for loading a VCF file which contains positions without variation would require similar memory size because the compression is done after the entire graph has been loaded. Because files with all variation are significantly larger than files with only variation, the problem of loading them into memory could be solved by performing the compression during the loading.

File	Max heap size	Time (ms)	Time / file size (ms/MB)	Status
File 1	30 MB	N/A	N/A	Out of memory
File 1	40 MB	1800	~ 580	Success
File 1	100 MB	800	~ 260	Success
File 2	75 MB	N/A	N/A	Out of memory
File 2	80 MB	3000	~ 460	Success
File 2	500 MB	1200	~ 180	Success
File 3	800 MB	N/A	N/A	Out of memory
File 3	850 MB	17000	~ 250	Success
File 3	2 GB	8000	~ 120	Success
File 4	2 GB	N/A	N/A	Out of memory
File 4	2,1 GB	45000	~ 270	Success
File 4	5 GB	19000	~ 110	Success
File 5	8 GB	N/A	N/A	Out of memory
File 5	9 GB	176000	~ 260	Success
File 5	14 GB	76000	~ 110	Success

Figure 6.12: Performance of the loading process on a single core.

From the figure we can also see that garbage collection takes up a significant portion of the CPU cycles if the max heap size is close to the minimal possible heap size. It is therefore important to take this into consideration when running the program. From the figure we can infer that the running time is linear in relation to the file size[5] which corresponds to the analysed running time of section 4.2.

6.3.1.2 Compression

Determining the gain in memory due to compression of a graph with linear strecthes is hard because of the limited control over Javas garbage collector. Recall that the main motivation for compressing graphs was due to the set of individuals stored for each node in the graph, however of course these individuals would not be part of the RDF graph and therefore compression should not affect the disk space to persistently store the graph. Figure 6.13 shows the running time of the compression for files with all loci included i.e. not only loci with variation.

[5]Given that the max heap size is large enough for garbage collection to not interfere too much with the running time.

File	Time (ms)	Time / file size (ms/MB)
File 6	19	~ 2
File 7	30	~ 1
File 8	50	~ 1
File 9	84	~ 1
File 10	185	~ 1

Figure 6.13: Performance of the graph compression process on a single core. All tests succeeded and were performed with a 50 GB max heap size.

As can be seen, the compression of graphs is quite fast. The running time corresponds with the asymptotic linear running time of section 4.3, however because the measured times are so small they are more affected by random spikes in performance.

6.3.1.3 Expanding

The expansion is the reverse process of compression. This is reflected in the running time, which does not differ much from compression, and also has a linear asymptotic running time as expected from section 4.3.

File	Time (ms)	Time / file size (ms/MB)
File 6	62	~ 5
File 7	81	~ 3
File 8	124	~ 4
File 9	155	~ 1
File 10	194	~ 1

Figure 6.14: Performance of the graph expansion process on a single core. All tests succeeded and were performed with a 50 GB max heap size.

6.3.1.4 Merging

For merging we constructed a new set of files. These files are the same as file 1-5 except that the genotypes for each position have been changed. The files, named 1.1, 2.1, 3.1 and so on, have the same sizes as file 1-5. We test the

merging process by merging two files of the same size with the same loci, but with different individuals and alleles. Memory allocation of the merged graph should therefore be the sum of the memory allocations for each graph. Figure 6.15 shows that there is a constant overhead with regards to the graph merging, which affects the running time of smaller graphs more than large graphs, but from the figure we can see that the running time for the large graphs correspond to the analysed linear running time of section 4.4.

Files	Time (ms)	Time / file size (ms/MB)
File 1 & 1.1	187	~ 60
File 2 & 2.1	247	~ 38
File 3 & 3.1	790	~ 11
File 4 & 4.1	1563	~ 9
File 5 & 5.1	5872	~ 9

Figure 6.15: Performance of the merging process on a single core. All tests succeeded and were performed with a 50 GB max heap size.

6.3.1.5 Building probability trees

A big challenge in regards to running time and space consumption is the construction and storing of probability trees. In this section we will take a look at the measures taken to counteract the memory and time needed for constructing and storing the probability trees by looking at the space needed to store compressed and uncompressed probabilty trees and also at how the multithreading can improve the running time of the construction of the trees.
Table 6.16 shows the performance for the construction of probability trees on a single core without compressing the trees.

From figure 6.16 we can see that the memory usage for the construction of trees is 150 times the size of the original file, which is an undesirably high factor. Considering that the files we used for testing only contain 150 individuals, it is likely that files with more individuals will scale even worse when adding the trees because the size of the trees are related by the number of individuals on the branches, which will naturally rise as the total number of individuals in the tree rises.
From figure 6.16 it also follows that the performance for constructing the trees is improved if the max heap is significantly larger than the minimally needed amount for constructing the trees. From the test it does not look like the analysed linear running time of section 4.5 was achieved, and as such a review of the implementation would make sense in the future, as we saw from the

File	Max heap size	Time (ms)	Time / file size (ms/MB)	Status
File 1	250 MB	N/A	N/A	Out of memory
File 1	500 MB	29418	∼ 9500	Success
File 1	10 GB	21181	∼ 6800	Success
File 2	800 MB	N/A	N/A	Out of memory
File 2	900 MB	78945	∼ 12000	Success
File 2	10 GB	44208	∼ 6800	Success
File 3	50 GB	2767987	∼ 41000	Success

Figure 6.16: Performance of the probability tree generation on a single core without compression of trees.

analysis of the algorithm that linear time should be achievable. However there is the chance that the test with File 3 did not have a large enough max heap size, meaning that the garbage collection affected the running time significantly.

Next we test the compression factor of the trees. We discussed in section 4.5 how the compression of trees and the use of globally stored genome sequences for the trees could help reduce the memory needed to store the trees due to the overlapping nature of trees in the same region of the graph. The results for the compression can be seen in figure 6.17.

Figure 6.17 shows that the memory needed to store the probability trees can be reduced significantly be compressing the trees. For example building trees for File 1 with a max heap size of 100 MB is not possible when the trees are uncompressed, and if the trees are compressed by a factor 4, the memory is still insufficient. At compression factor 8 the trees can be built, but as it is shown in the figure, choosing an even higher compression factor improves the construction time significantly. However the gain from compression flattens out as the compression factor increases. This is likely due to the fact that on average the probability trees are not as deep as the chosen compression factor, such that increasing the compression factor does not change the compression of the trees. As graphs based on more individuals are likely to have deeper probability trees, it is likely that a higher compression factor is better.
From the results observed in figure 6.17, we chose a fixed compression factor of 32 for the next test in which we test how the number of threads for generating the trees affects on the overall performance.

Figure 6.18 shows how increasing the number of threads can improve the running time of the probability tree construction. The machine which we tested on had a total of 64 processors. It follows from the figure that the time is not

File	Compression factor	Time (ms)	Time / file size (ms/MB)	Status
File 1	2	N/A	N/A	Out of memory
File 1	4	N/A	N/A	Out of memory
File 1	8	80191	∼ 26000	Success
File 1	12	29287	∼ 9500	Success
File 1	16	24634	∼ 8000	Success
File 1	32	22028	∼ 7000	Success
File 1	50	23326	∼ 7500	Success
File 2	4	N/A	N/A	Out of memory
File 2	8	174627	∼ 27000	Success
File 2	16	51775	∼ 8000	Success
File 2	32	49103	∼ 7500	Success
File 2	50	48529	∼ 7500	Success

Figure 6.17: Performance of the probability tree construction process on a single core. For file 1 we used a max heap size of 100 MB, and for file 2 we used a max heap size of 200 MB.

File	Threads	Time (ms)	Time / file size (ms/MB)
File 1	1	45257	∼ 14500
File 1	2	30102	∼ 9500
File 1	4	16358	∼ 5500
File 1	8	9629	∼ 3000
File 1	16	6792	∼ 2000
File 1	32	6137	∼ 2000
File 1	64	5815	∼ 2000
File 3	16	296911	∼ 4500
File 3	32	210690	∼ 3000
File 3	64	142423	∼ 2000

Figure 6.18: Performance of the probability tree construction process on multiple threads, on a machine with 64 total cores. All tests succeeded and were performed with a 50 GB max heap size and a compression factor of 32.

reduced linearly as the number of threads increase. We discussed in section 6.2.4 how there are critical sections involved with the construction of the trees, as all threads read and write to shared hashmaps of genotypes and strecthes of genotypes. Therefore increasing the number of threads that generate trees,

means that more threads access this critical region, which can be the cause for sub-linear improvement as thread number increases. To increase the performance gain from multithreading, each thread could keep a seperate hashmap in which they store the compressed sequences because a compression factor of for example 32 means that the compressed sequences are likely to only be reused by other trees in the same region, and therefore threads handling different parts of the graph do not need these threads. This would result in a slight memory allocation increase in the case that the threads observe the same sequences, but on the other hand would allow each thread to write to its own hashtable without causing a critical section.

From the testing of the probability tree generation we can see that there is a motivation for building the trees with a significant margin of memory in regards to the minimally required amount. There is also a motivation for using compressed trees, as this allows the graph (including probability trees) to be contained in significantly less memory than a graph with uncompressed trees. An increase in compression factor beyond 16 does not seem to greatly affect the memory used to store the trees, however as the tested graph contains 150 individual, the number could be higher for a graph with more individuals. As for parallelization of the process, there is a motivation for using multiple threads, however it seems that the performance gain is sublinear in relation to the number of threads used, which can be caused by the requirement of mutual exclusion for critical regions.

6.3.1.6 Sampling

With the probability trees in the graph it is possible to sample individuals. We test the sampling in this section in three ways. First we test the running time of creating samples in a sequential manner using only one thread for a varying number of samples. Next we test for a fixed number of samples with a varying number of threads and lastly we test the generation of VCF files from sampled graphs.

As can be seen from figure 6.19, the sampling algorithm is linear both in regards to sampling multiple individuals from the same graph and in regards to sampling individuals from graphs of different sizes which is in accordance with the analysed running time of section 4.6.

Figure 6.20 shows that performing the sampling of individuals in parallel can improve the overall running time. However increasing the number of threads for sampling does not result in a linear decrease in running time. This has to do with the shared sampled graph which all threads add the samples to. To improve upon the overall performance of the sampling, each thread could

File	Samples	Time (ms)
File 1	1	6082
File 1	2	12355
File 1	4	22759
File 1	8	41720
File 2	1	12527
File 2	2	24148
File 2	4	46545
File 2	8	91543

Figure 6.19: Performance of the sampling process on a single thread with a max heap size of 50 GB. All tests succeeded.

Threads	Time (ms)	Time / samples (ms/sample)
1	181522	$\sim$ 5500
2	101481	$\sim$ 3200
4	63658	$\sim$ 2000
8	50347	$\sim$ 1500
16	46886	$\sim$ 1500
32	45132	$\sim$ 1500

Figure 6.20: Performance of sampling 32 individuals on multiple threads. All tests succeeded and were performed with File 1 and a max heap of 50 GB.

instead have a sampled graph that it can write to without a critical section. After the construction of the sampled graphs for each thread, these could be merged together in pairs following the same principle as in figure 4.4.
Figure 6.21 shows the data structure in which the sampled data is kept, and which is accessed by the sampling threads. Instead of having a map where for each position the genotypes and individuals are stored, the data structure could be changed to a map where the positions and genotypes are stored for each individual, thus mimicking the principle of a seperate sampled graph for each thread. This would allow threads to iterate the graph and generate the sampled individual without critical sections.

Another major influence on the running time is the fact that each thread expands the compressed tree of every position in the graph, when it is used. We discussed in 4.6 how a scheme could be constructed, such that trees are only expanded once and used by every thread. When all threads are done using it, it can then

```
1 //<Position <Genotype, <Individual, Phased>>>
2 public Map<Integer, Map<String, Map<SampledIndividual, Boolean>>>
      genotypeIndividualsFirst;
3 public Map<Integer, Map<String, Map<SampledIndividual, Boolean>>>
      genotypeIndividualsSecond;
```

Figure 6.21: The sampled graph.

be released from memory again.

Time (ms)	Samples	Time / samples (ms/sample)
551	1	~ 550
448	2	~ 225
535	4	~ 550
431	8	~ 130
644	16	~ 40
1040	32	~ 30
1994	64	~ 30
3747	128	~ 30

Figure 6.22: Performance of the VCF writing process. All tests succeeded and were performed with File 2 and a max heap of 50 GB.

Next, we test the performance for generation of VCF files from a sampled graph. This process writes to a file and as such is single threaded.
From figure 6.22 we can see that the time for writing the sampled graph to a VCF file is mainly affected by the meta data columns mentioned in 2.3.1, such as the position, variation ID and sequencing depth, which are contained in every line (for generated VCF files the two latter are hardcoded). However once the number of samples increase, it affects the running time such that it approaches a linear time for the saving.

6.3.1.7 RDF

In order to persistently store the created graph, Jenabean can convert Java objects and their relations to resources and triples in the Jena model. This model can then be written to a triple store or an RDF serialization. In this section we will test the performance of the I/O of VGTool.
Figure 6.23 shows the performance of writing the graph to a file in the RDF serialization format Turtle.

File	Time (ms)	With trees	RDF file size	Zipped file size
File 1	9893	No	6.7 MB	287 KB
File 1	41660	Yes	634.5 MB	37.2 MB
File 2	29906	No	13.9 MB	593 KB
File 2	99337	Yes	1.26 GB	74.5 MB

Figure 6.23: Performance of the RDF serialization process. All tests succeeded and were performed with a max heap of 15 GB.

From the figure it follows that graphs with probability trees use significantly more space when they are stored. Here it is important to point out a few things. Firstly the probability trees are stored completely uncompressed, which was a choice taken in order to make the graph easier to use for other users of the Semantic Web. Similar to the in-memory compression performed on the probability trees, the persistently stored trees could be compressed. Secondly the RDF serialization used here does not store the data in a very compressed manner; an example of this can be seen in figure 6.24, which shows how a single node in a probability tree is represented in Turtle. Figure 6.23 also shows the redundancy of the RDF serialization as compressing the generated files can reduce their size by more than a factor 10. In this project we have not tested triple stores, but it is likely that storing the graph in a triple store rather than in a RDF serialization can reduce the storage space needed as well, because in principle each URI of a resource is only stored once.

```
1 <http://vg#VGProbabilityNode/chr1:838638:331>
2         a             vg:VGProbabilityNode ;
3         vg:children   <http://vg#VGProbabilityNode/chr1:838638:332> ;
4         vg:genotype   <http://vg#VGProbabilityGenotype/G;G> ;
5         vg:id         "chr1:838638:331"^^<http://www.w3.org/2001/XMLSchema#string>
              .
```

Figure 6.24: An example of a probability node in Turtle. For the genotype here, the phasingtype "|" has been replaced with a valid XML character ";".

Figure 6.25 shows the performance for the loading of a graph from RDF.

The loading of the RDF graph is similar to the saving, both of which are quite time consuming in comparison to the other parts of VGTool.

File	Time (ms)	With trees	RDF file size
File 1	16008	No	6.7 MB
File 1	74698	Yes	634.5 MB
File 2	60807	No	13.9 MB
File 2	217144	Yes	1.26 GB

Figure 6.25: Performance of loading the graph from the Turtle format. All tests succeeded and were performed with a max heap of 15 GB.

6.3.2 Performance summary

In this section we tested the performance of the different functionalities of VG-Tool.

Loading the graph into memory can be done with a heap size about 10-15 times the size of the input file for files containing only variation.
Constructing and storing the probability trees both in memory and on the disk provides a major challenge. As we have seen, the size of the uncompressed trees take up about 90% of the space needed for storing the graph in memory and almost 99% when the graph is written in an RDF serialization like Turtle.
Given an internal version of the graph, which is not to be shared with the public, it is possible to store the observed individuals in the graph instead of the probability trees, and only generate the trees when needed. This could significantly reduce the size needed to store the graph without losing information. However for sharing the graph the key point is that the graph contains information about genetic patterns without revealing identities and as such it is necessary to have the probability trees as part of the RDF graph. To make this feasible a number of things can be done. A triple store or a more compact RDF serialization could be used to store the graph. As the produced RDF is highly repetitive, there is the possibility of compressing the RDF file to significantly reduce the file size. Lastly the probability trees could be compressed in the RDF in a similar way to how they are currently being compressed in memory.
Another way to circumvent the problems regarding the storage of the RDF graphs is to simply sample VCF files and share these instead. Currently the main use case for the genome graph is to generate VCF files, and as such it is a feasible solution if the needed disk space for persistently storing the graph is unrealistic.
Some processes regarding the graph have shown to be quite time consuming. As such it was shown that a significant performance gain could be achieved by splitting those tasks into smaller tasks which each could be handled in seperate threads.

6.3.3 Sampling quality

In this section we will look at how the sampled individuals correlate with the originally observed data for both different tree queue sizes and different p[6] as discussed in section 4.6.

6.3.3.1 Allele frequency

The first thing we tested was the allele frequency (AF)[7] correlation for different queue sizes of the probability tree queue explained in 4.6. Because the samples are generated from the probability trees and rare variants have been removed it is a challenge to replicate the distribution in the sampled data.
Figure 6.26 shows the correlation between AF of the original data and the sampled data for different queue sizes with a fixed $p = 0.75$.

In the figure the R^2 correlation is noted along with a line that is fitted to the points according to the least squares method. The closer these values get to 1, the more correlation there is between the original and sampled AFs. From the figure it follows that increasing the size of the tree queue leads to a better correlation between the data sets, meaning that using the history contained in probability trees leads to a more accurate AF in the sampled data.

Figure 6.27 shows how the choice of p affects the AF. As can be seen, the correlation between the original and sampled AF is generally quite good, but having a high probability of choosing a deep tree leads to a slightly better correlation between the data. This is in accordance with figure 6.26, which also showed that deeper trees lead to a better correlation of AFs.

6.3.3.2 Linkage disequlibrium

The other distribution that should be in accordance with the original data is the linkage disequilibrium. The main purpose of the probability trees is to convey this connection between alleles at different loci, and intuitively it makes sense that deeper trees describe LD between alleles that are far apart better than shallow trees. However the LD of alleles that are further apart than the span of a tree can still be found to be statistically associated if there are alleles in between them which they are both associated with.

[6] p is the probability of choosing the head of the tree queue as explained in section 4.6
[7] The reason it is shown as MAF in the graphs is that to express the frequency of two variants, only the frequency of one of them (the minor frequency) has to be expressed.

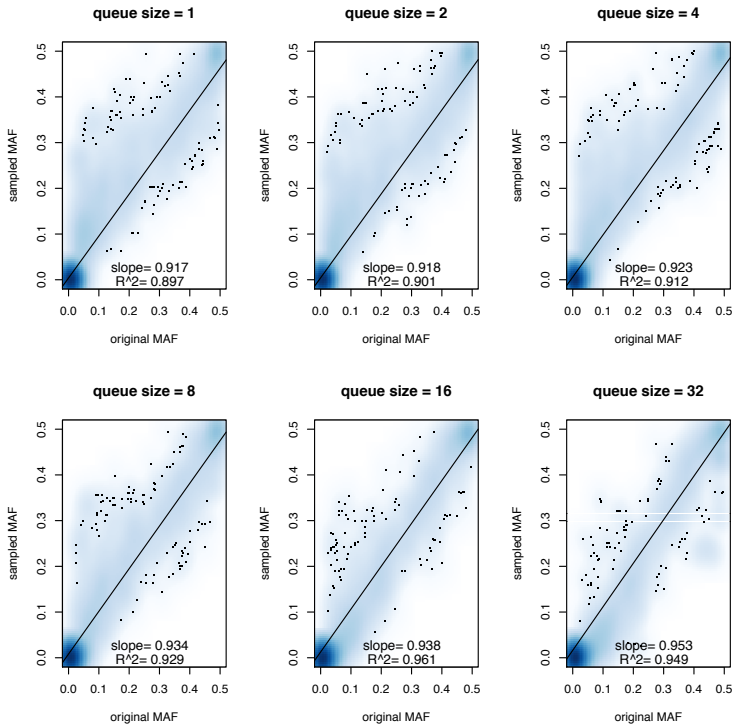


Figure 6.26: Correlation between original AF and sampled AF with different queue sizes.

Figure 6.29 shows the correlation between LD of the original data and LD of the sampled data. We tested with a varying tree queue size of 1, 2, 4, 8, 16 and 32. We calculated the LD for a window size of 8, i.e. the LD of alleles within 8 positions of each other. The color of the dots in the plot determine the distance between the alleles as shown in figure 6.28.

In the plots, only alleles with R^2 correlation above 0.2 are shown. From figure 6.29 it can be seen that with a small tree queue size i.e. little history, most of the correlating LDs found are for LD windows of size 2, with a few exceptions of size 3 and 4. As the tree queue size increases, the correlation between LD of alleles further apart also increases which can be seen by the increasing number of orange, yellow and green dots. The example with a queue size of 4 has some green dots which shows that alleles further apart than the size of the tree queue can still be found to correlate, which as mentioned earlier can occur when there are alleles in between that are assosiated with both. From the plots it also shows

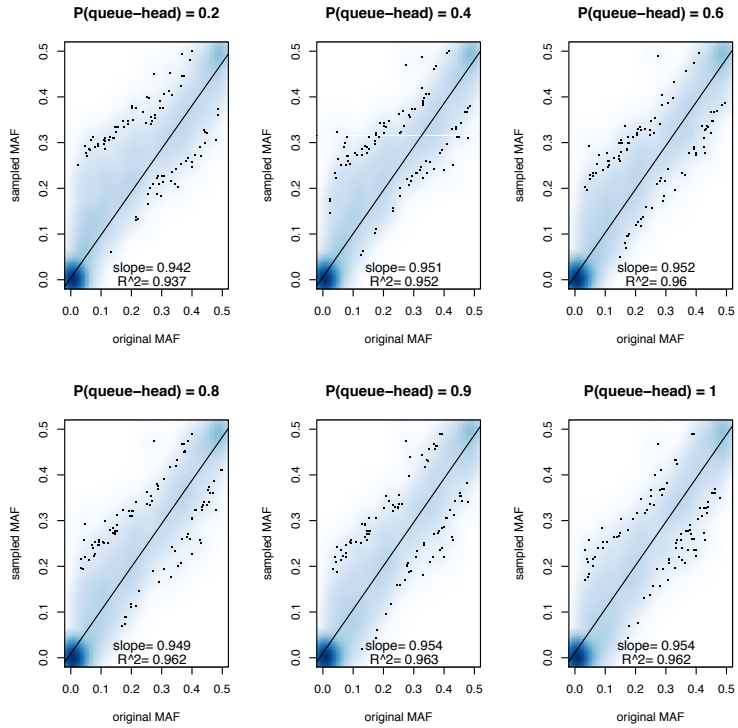


Figure 6.27: Correlation between original AF and sampled AF with different p.

Distance between SNPs	LD window size	Color
1	2	
2	3	
3	4	
4	5	
5	6	
6	7	
7	8	

Figure 6.28: The colours that specify the distance between SNPs.

that increasing the queue size beyond the size of the chosen LD window does not increase the overall LD correlation significantly, which will be tested further in figure 6.31.

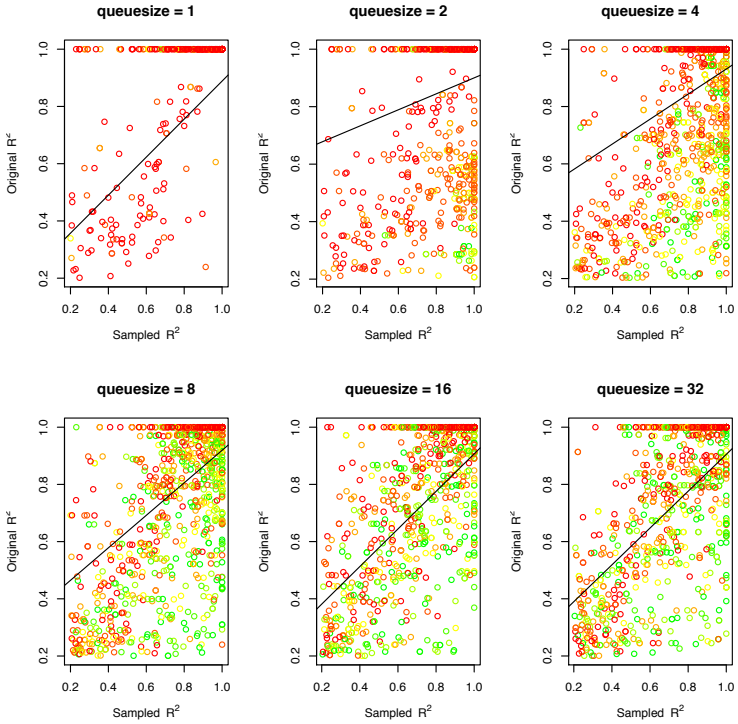


Figure 6.29: LD correlation for different tree queue sizes.

Like with the AF, we also tested the correlation between LD in the data sets for a varying p in figure 6.30.

Figure 6.30 shows again that the best results can be achieved by taking advantage of the deeper history of the first trees in the queue. In the figure, the best results, in terms of R^2, were achieved by having a 90% probability of chosing the first element in the tree queue. Good results were also found by having a 100% probability of choosing the first element in the queue, although as discussed earlier, this scheme can lead to scenarios with bias towards certain variation.

Figure 6.31 shows how an increased tree queue size affects the correlation between sampled and original LDs, for LDs in the range 2-8.

From figure 6.31 we can be see that the correlation between sampled and actual LD for the different window sizes improves as the tree queue size increases, but also that it quickly flattens out, here around a queue size of 16, meaning that

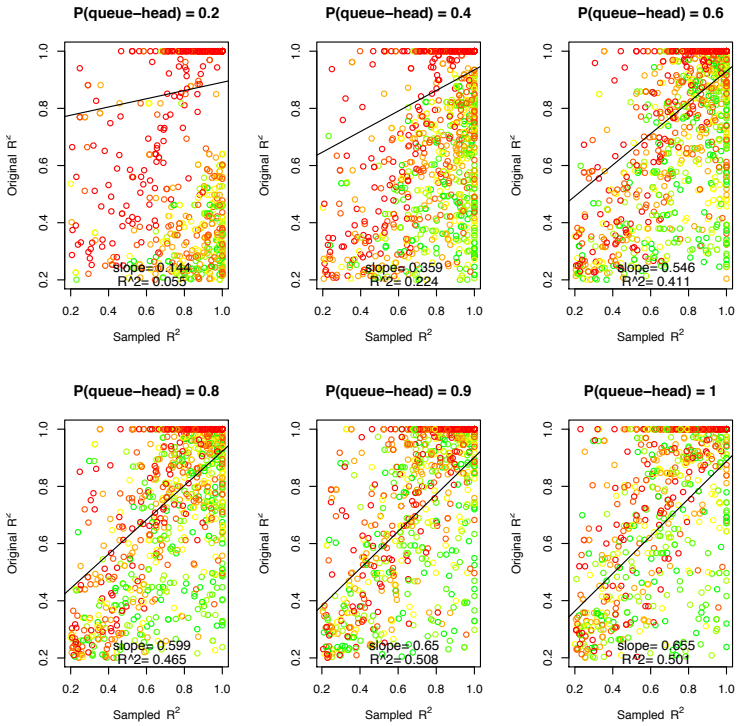


Figure 6.30: Correlation between LD of original and sampled data for varying p.

beyond that size there is not much to gain from increasing the queue size. It also follows from the figure that the replication of LD is more accurate for lower LD windows, which makes sense as the longer the paths are in probability trees, the more individuals are likely to have been sorted out, due to rare variation.

6.3.4 Sampling quality summary

In this section we covered the quality of the generated samples by comparing different distributions in regards to the sampled data versus the original data. We found that the AF distribution could be improved by increasing the size of tree queue, which is convenient because an increased size in tree queue also improved the quality of the LD in sampled data. However we also found that increasing the tree queue beyond the size of the LD window with which the tests were performed did not increase the overall result significantly, and that the LD between SNPs further apart than the size of the tree queue could still be found

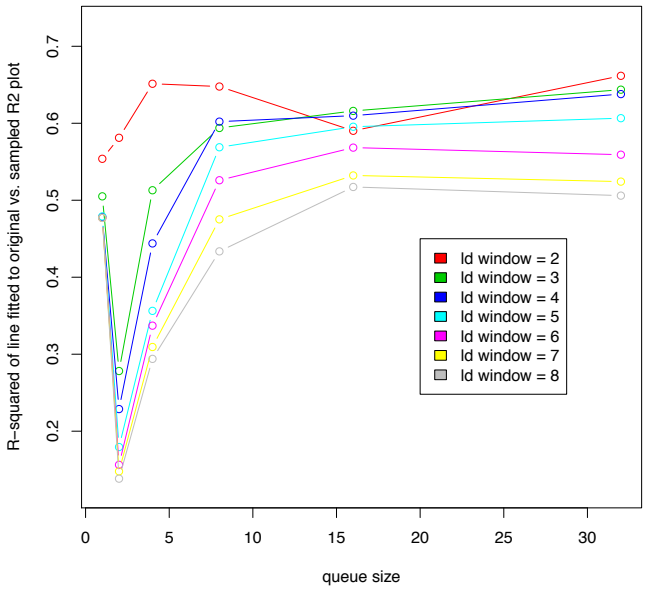


Figure 6.31: The relationship of entities in the VariantGraph.

in the sampled data.
Furthermore we found that replicating the LD of the original data set became increasingly more difficult for alleles further apart, which makes sense intuitively as the data used in probability trees is only a subset of the actual data and decreases in size as the depth of the tree increases. As for the value of p which determines the probability of choosing elements in the queue, we found that a higher p, i.e. higher probability of choosing a tree of the first positions in the queue, led to a better correlation between sampled and original data, both for AF and LD, which indicates that using deeper trees with more history for the sampling gives better results than shallow trees with little history.
A test which we did not perform in this section, but could be interesting, is to find out how the number of sampled individuals affects the correlation between the sampled and original cohort. Intuitively it makes sense that as larger cohorts are generated, the correlation between the datasets improves. Also as the main motivation for sampling of individuals in the first place is to use the generated individual for imputation of genetics, it would make sense in the future to test how imputation with sampled data performs.

In the next chapter we discuss of the achievements of the thesis as well as future

improvements that can be made to both improve on the obtained results and in regards to extend the functionality of VGTool.

6.4 Applicability in case study

The original motivation for constructing VGTool was due to the Danish pan-genome project. As mentioned this project aims at giving an *"increased understanding of population-specific disease susceptibility and will be important for advancing clinical and public health genetics"* [12, p. 6]. With the genome graph and VGTool we have created a way of sharing non person referable data constructed from the observations made in the Danish pan-genome project. As such it is possible to share summary statistics of the study publicly. This is significant because the sampled data can be used in research in processes such as imputation [31]. Furthermore constructing a genome graph from 150 Danish individuals provides an excellent reference structure for read-mapping the DNA of Danes as the number of genetic variants encapsulated in this graph makes it a better representation of the Danes' genetics than a single linear reference genome.

As we mentioned, a VCF file of only the loci with variation in the cohort of the Danish pan-genome study is about 10 GB in size. This raises a question of whether it feasible to build a genome graph for the entire data set which covers all 23 chromosomes. Especially when considering that a file that contains all loci for the cohort reaches into the terabytes. Currently the graph construction and handling is done all in memory on a cluster computer of 1 TB RAM and 64 CPUs. Given the results of the tests, it would be feasible to create genome graphs on the cluster computer for each chromosome and possibly all chromosomes at once if they are built from the file that only contains variation. However building the graph from a file where all loci are included seems infeasible in memory given that the file size of the VCF exceeds the amount of RAM on the cluster computer. To overcome this, compression of the graph could be performed during load, graphs could be built for each chromosome seperately and a scheme for storing parts of the graph on disk, while keeping the other part in memory, could be constructed.

6.5 Tool summary

In this chapter we covered the design and implementation of VGTool. We also performed tests to see that the tool scaled in a satisfiable manner and that the sampled data from the graph had reasonable correlation to the original data. In the next section we will discuss the different steps of the thesis and how we achieved our initial goals and requirements.

Chapter 7

Discussion

In this chapter we will evaluate the achieved results and compare them to the initial goals of the thesis. As such this section will serve as discussion of problems we initially set out to solve in the problem description of section 1.2. We will also discuss how the implemented graph structure could function in other use cases in the future.

7.1 Genome graphs

The first step of this thesis was to identify some of the problems regarding the current representation of genomic data. As discussed in chapter 2 the current linear representation of DNA that is used in read-mapping is problematic because it relies on the similarity between the chosen reference genome and the input reads. The current solution to this problem is for the reference to have a number of alternative sequences that can be used if the reads do not map to the original reference. We have not solved the problems regarding mapping, but the constructed genome graph is easily extended and could in the future be designed to work as a mapping reference. We also highlighted the problematics regarding the inability to share the collected genetic data due to its close link to the involved individuals, and used this as a motivation for creating a scheme

where sampled individuals could be constructed from the commonly observed genetic variants.

We found that a graph was an ideal data structure for expressing the genomic variation of a cohort, because it allows us to express alternative genetic variation as branches in the graph. Graphs are also very flexible data structures in which anything can be linked together. This is a nice feature for a genome graph that consists of many different entities.

7.2 Choice of technologies

After establishing the requirements for the graph we reviewed relevant technologies that enabled us to express and work with the genome graph. We chose to represent the graph as triples in RDF because of its flexibility, extensibility, query support, and because sharing the graph as part of the Semantic Web means that everyone who should desire to use the graph can easily access it. We chose to develop the graph tool, VGTool, in Java because there exists a number of relevant libraries and frameworks that allowed us to read VCF data, work with the graph as Java objects and express it in RDF.
Due to the widespread use of the Semantic Web, triple stores and VCF files, there are other valid programming languages that would have allowed us to construct and share the genome graph.

7.3 Computational challenges

When constructing the genome graph we realised that some parts of the graph were problematic in terms of space consumption. To counteract this we proposed a number of ways for compressing the graph. On the other hand several of the implemented algorithms for handling and using the graph proved to be very time consuming. Beyond ensuring that the asymptotic running time of the designed algorithms were acceptable, we made an effort to parallelize the tasks as much as possible. Because the genome graph proposes challenges in regards to both space and time consumption, decisions have to be made carefully because working towards decreasing one of the two usually means an increase in the other, exemplified by for example compressing the probability trees of the graph, which allows the graph to reside in less memory, but also means that the trees need to be expanded every time they are to be used.
The current implementation works with the entire graph in memory, however

in the future if VCF files for entire genomes with all loci are present, it could be a good idea to change the design such that only part of the graph resides in memory and the rest is stored on the disk, and can be retrieved if needed.

7.4 The Semantic Web

We achieved the goal of creating an extensible graph structure by expressing the graph in RDF such that any resources of the graph can be linked to the Semantic Web. Through testing we found that storing the graph in an RDF serialization such as Turtle is not the best option for persistent storage because the file size of a graph with probability trees could reach about 200 times that of the original VCF file. As RDF serializations are largely verbose however, we also found that compressing the produced files could decrease the size to about 10 times that of the original file. Instead of using RDF serializations, it makes sense to look at triple stores, which is not something we have done in this project, but it is very likely that triple stores store the data more efficiently than an RDF serialization.

7.5 Sampling individuals

One of the main goals of the thesis was to be able to share the data that has been collected in the Danish pan-genome study. We have covered why it is not possible to share the data in its unaltered form, and therefore instead wanted the functionality of extracting summary statistics from the data that could not be related back to individuals.
With this motivation in mind, we constructed the genome graph in such a way that it contains information about observed genetic patterns which can be used to sample data from. We have tested the quality of the sampled data by correlating it to the original data for a number of different settings, and found that using the history contained in a deep probabilty tree improves the overall quality of the result.
We also had a focus on making it possible to merge genome graphs together. This is an important feature because it allows us to add more observations of genetic variation to the graph, and more importantly it can make previously rare genetic variants reach a sufficient number of observations thereby exceeding the threshold for when those variants can be shared.

In its current state while the genome graph is sharable through the Semantic Web, there are no tools for using the graph, other than the tool developed in

this project. For this reason we chose to create to an algorithm for generating sampled individuals using the graph and storing this data in the VCF format which is a standardized format that is widely used as a way of expressing the genomic variation of a cohort.

Currently structural variation is not supported in VGTool, meaning that only adjacent positions in the graph are connected. Structural variants are however quite rare, which makes it unlikely that these variations are observed in enough individuals to be considered common variation and therefore it is not a necessity for generating samples.

The sampling was tested with a hypothesis that rare genetic variation is variation present in less than five individuals of a cohort. However Danmarks Statistik have a set of guidelines [39, p. 13] in which this number is identified as 10. Raising the threshold for genetic variation would of course decrease the quality of sampling, as the probability trees will be constructed from less individuals, but this can be mitigated by constructing the graph from more individuals.

7.6 Future work & limitations of VGTool

In the problem description[1] we mentioned a number of problems that we have worked with in the thesis. However we have also discussed other problems such as for example using the graph as a mapping reference. The following list provides ideas for future work that could be done to extend the usefulness of the genome graph and VGTool in general.

Mapping - A natural extension to the genome graph would be to use it for read-mapping. We have already discussed earlier why there is a motivation to do so. With the flexible graph format, adding base contexts to alleles in the graph would certainly be possible, but it would pose a number of challenges due to the increased amount of data that the graph would contain.

Merging - The merging algorithm that is implemented in VGTool relies on the graphs to be constructed from the same reference. Therefore the merging is done solely based on the positions of the two graphs which can be assumed to be the same. Instead a more sophisticated merging algorithm could be designed that uses the principles of mapping to identify regions and positions that contain the same genetic patterns and merge these regions.

[1] Section 1.2

Performance enhancements - In section 6.3.1 we saw that the running time and space consumption of some of the implementated algorithms in VG-Tool were quite large, and as such future work could be put into designing more efficient compression algorithms, faster algorithms and parallelization without bottlenecks.

Structural variation - We have already discussed how sampling data should only be done for common variation. Generally structural variation is quite rare, and in a sampling scheme it is unlikely that a structural variant is observed in enough individuals for it to occur in the sampled data. For this reason we have not focused much on structural variation in this thesis, however the nature of structural variants can be expressed very intuitively using a graph structure where loci that are far apart or reside on different genomes can be connected together. Traversing a genome graph with structural variation poses new challenges however, as the direction of the traversal can change.

Triple store & the Semantic Web - In this thesis we have built the graph with a focus on being easy to share through the Semantic Web. However the only persistent storage we have tested here has been RDF serializations. In order to integrate the graph as part of the Semantic Web it is natural to store it in a triple store. The advantage of doing so is that resources of the graph can be located, and thereby linked to, through their URIs and relevant data can be retrieved from the graph through Semantic Queries.

Ontologies - The main idea behind the Semantic Web is that data can be linked together from different sources. Currently the entities of the genome graph are not linked to anything in the Semantic Web, however to fully utilize this feature, existing ontologies regarding genetics could be linked to entities of our genome graph.

Worldwide pan-genome - Much of the motivation for this thesis came from the Danish pan-genome study recently undertaken in Denmark. With a genome graph in the Semantic Web, instead of national pan-genome, there is the potential to create pan-genomes of even larger cohorts, such as for example Scandinavia, Europe or the entire world.

Chapter 8

Conclusion

In this thesis we have analysed the problems of the classical representation of genome data. The analysis lead to the motivation for using a graph structure to represent genomic data. We discussed the design of the graph in the light of all the presented problems, although only a subset of these problems were solved in this thesis. The design of the graph has a focus on flexibility, which means that in the future it could be extended to handle all of the presented problems. After discussing the design of the genome graph, we implemented a tool named VGTool, with which we could represent the genomic data of a cohort as a graph. The tool also achieved the goals of being able to merge genome graphs and sample data with a statistical similarity to the original data.
VGTool was designed with a focus on modularization of tasks, such that the different components for handling the graph can be changed or modified without affecting other parts of the tool. The implementation process proposed a series of challenges in regards to memory and time consumption, and as such efforts were made to reduce both. Through testing we found the critical parts of the program in terms of the memory and time consumption and discussed which measures could be taken to improve upon these parts.
We chose to express the genome graph as triples in the Semantic Web because it is an easy way of sharing the graph and allowing others to extend it with their own information. For this reason we ensured that the genome graph of VGTool could be both exported to and imported from RDF triples. The decision to express the graph as part of the Semantic Web was also motivated by the

fact that several studies have already taken their own approach to representing genomic variation in graphs, but with the Semantic Web, we feel that we have provided a reasonable standard for expressing genome graphs, rather than have every individual study use their own representation.

As a concluding remark we can say that we with this thesis and the implemented tool achieved the goals creating a unified data structure of genetic variation from which sampled data can be created and made it easily extendable in anticipation of future use cases, which was the initial goal stated in the problem description of section 1.2.

Appendix A

Terminology

This appendix contains a collection of the terms explained throughout the thesis.

Phenotype - The observable traits of an organism are called the organisms phenotype. The phenotype of an organism is influenced both by the environment and the genetics of the organism. [Page 2]

Genotype - A genotype is a part of DNA that determines a specific charateristic of the individual. For any position in the DNA, there is a genotype, but it is possible that two organisms have different genotypes for the same "position", resulting in different phenotypes. [Page 2]

DNA (Deoxyribonucleic acid) - DNA is a molecule that stores biological information. DNA consists of two strands, which are each made up of sequence of *bases*. [Page 2]

Base (Nucleobase) - A sequence of DNA is made up of the four bases A (adenine), C(cytosine), G(guanine) and T(thymine). [Page 2]

Genome & Reference genome - The genome is the entire DNA sequence of an individual. A reference genome is the complete assembled genome of an individual. The term *reference* comes from the fact that it is used as a reference to which DNA can be aligned in order to determine similar regions. [Page 3]

Pan-genome - A pan-genome is a collection of genetic data that encapsulates the genetic variation of a population. [Page 3]

Sequencing - Sequencing of DNA is the process of determining the precise order of the bases in a DNA stand as explained in section 2.2.2. [Page 4]

Cohort - A group of individuals with a shared characteristic. In the context of the Danish pan-genome, this characteristic is that all involved individuals are Danish. [Page 4]

Chromosome - A chromosome is structure that contains DNA. A human inherits one copy of each the 23 chromosomes from each of their parents, giving a total of 23 chromosome pairs. [Page 10]

Single nucleotide polymorphism (SNP) - Pronounced "snip". Describes genetic variation of one base. [Page 11]

Indel - The term specifies genetic variation that is either a deletion or an insertion. The reason that one term is used for both types of variation, is that depending on the frame of reference, an insertion can also be seen as a deletion and vice versa. [Page 11]

Structural variation - A more complex type of variation that changes the structure of the genome, for example by inversing a sequence of DNA. [Page 11]

Genome assembly - Genome assembly is the process of assembling the human genome from smaller reads as shown in figure 2.3. [Page 12]

Locus - Locus (plural loci) refers to a specific position on a chromosome. [Page 13]

Depth & Coverage - The coverage of a locus describes the depth of the sequencing in that locus i.e. how many reads cover the locus. A high depth is desirable because it reduces the risk of sequencing errors. [Page 13]

Sequence alignment - Sequence aligment is the process of aligning DNA sequences to a reference to identify regions of similarity, and inferring meaning from these regions, such as for example genetic variation that is related to a certain disease [32] [Page 13]

Allele - An allele is one particular variant of a number of possible variations for a certain locus. A genotype consists of two alleles. [Page 17]

Phasing - Every individual inherits a copy of each chromosome from both their parents. If a genotype is phased, it means that it is known which allele of the genotype comes from which parent. [Page 18]

Break-end - The break-ends of a structural variation specifies how two sequences of DNA are connected. [Page 19]

Allele frequency (AF) - AF is an expression of the frequency of each observed allele for a particular locus. [Page 20]

LD (Linkage disequilibrium) - LD is an expression of alleles at different loci that are linked together statistically in the sense that they are often both appear together. Figure 2.9 illustrates this. [Page 20]

Imputation - In genetics, imputation is the statistical inference of unobserved genotypes. [36] [Page 20]

Appendix B

VGTool source

VGTool is an open source project, which can be cloned from GitHub using the command:

```
git clone git@github.com:Johanvb/VGTool.git
```

The project uses Apache Maven for handling plug-ins, and can be run by opening the `VGTool` folder in IntelliJ and specifying the correct main class, `Main.Main`.

VGTool is open source under the GPL license.

Appendix C

Using VGTool

Using VGTool is simple. When the project is run, the user is presented with a menu of different options. The user interacts with the program through textual input.

```
Choose option:
(LF) Load file from path.
(R) Load from rdf.
(W) Write graphs to files.
(P) Add probabilities to graphs.
(S #) Create # samples.
(PGS) Print graph sizes.
(PG) Print graphs.
(Q) Quit
```

Listing C.1: VGTool menu options.

The user is presented with the following options:

Load file from path. Loads a VCF file from specified path, builds and compresses a VariantGraph from the input.

Load from rdf. Loads a graph previously stored in the Turtle RDF serialization.

Write graphs to files. Writes the current VariantGraphs and contained entities to Turtle files.

Add probabilities to graphs. Constructs probability trees in the graph, and compresses them depending on the settings of the project.

Create # samples. Creates a specified number of samples. Which chromosome the samples should be created for can be specified, however the default is all chromosomes. Only works if probability trees have been added to the graph. Once the samples are created, the user can specify a file name and the samples will be saved to `"VCFDataBase/generated_samples/" + fileName`.

Print graph sizes. Prints the number of entities in the current graphs.

Print graphs. Prints relevant information about the graph, such as genotype observations for example.

Quit. Quits the program.

Bibliography

[1] What is Jena? https://jena.apache.org/about_jena/about.html [Online; accessed 10-June-2015], . The Apache Software Foundation.

[2] An Introduction to RDF and the Jena RDF API. https://jena.apache.org/tutorials/rdf_api.html [Online; accessed 10-June-2015], . The Apache Software Foundation.

[3] Resource Description Framework. http://www.w3.org/RDF/ [Online; accessed 1-June-2015], 2014. RDF Working Group.

[4] Chromosome 1 summary. http://useast.ensembl.org/Homo_sapiens/Location/Chromosome?r=1, 2015. [Online; accessed 2-July-2015].

[5] *OWL 2 Web Ontology Language Document Overview (Second Edition)*, December 11, 2012. W3C OWL Working Group.

[6] The fastg format specification (v1.00). http://fastg.sourceforge.net/FASTG_Spec_v1.00.pdf [Online; accessed 1-July-2015], December 12, 2012. The FASTG Format Specification Working Group.

[7] *The Variant Call Format (VCF) Version 4.2 Specification*, version 4.2 edition, January 26, 2013. https://samtools.github.io/hts-specs/VCFv4.2.pdf [Online; accessed 7-July-2015].

[8] *Sequence Alignment/Map Format Specification (SAM)*, May 11, 2015. https://samtools.github.io/hts-specs/SAMv1.pdf [Online; accessed 7-July-2015].

[9] Sesame. http://rdf4j.org/about.docbook?view, October 16, 2014. [Online; accessed 6-July-2015].

[10] Joachim Baran, Kevin B. Cohen, Geraint Duck, Michel Dumontier, Begum Durgahee, and Jin-Dong Kim. BioInterchange - Cloud and NoSQL integration of genomics data made easy. http://www.biointerchange.org/about.html. [Online; accessed 28-May-2015], 2015.

[11] Tim Berners-Lee. Linked Data and Design Issues. http://www.w3.org/DesignIssues/LinkedData.html, 2009. [Online; accessed 20-May-2015].

[12] Søren Besenbacher et al. Novel variation and de novo mutation rates in population-wide de novo assembled danish trios. *Nature Communications*, 6:5969, 2015.

[13] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web Information Systems*, 5:1–24, 2009. Freie Universitat Berlin, Germany.

[14] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5 (3):1–22, 2009.

[15] Compeau, Phillip E C, Pevzner, Pavel A, and Glenn Tesler. How to apply de bruijn graphs to genome assembly. *Nat Biotech*, 29(11):987–991, 11 2011. URL http://dx.doi.org/10.1038/nbt.2023.

[16] Alexander Dilthey et al. Improved genome inference in the MHC using a population reference graph. http://biorxiv.org/content/early/2014/07/08/006973 [Online; accessed 1-March-2015], 2014. BioRxiv is and archive unpublished preprints in the life sciences. This article was published in Nature in 2015 (http://www.nature.com/ng/journal/v47/n6/full/ng.3257.html [Online; accessed 30-June-2015]).

[17] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*, January 26, 2006.

[18] Mark Graves. Graph Data Models for Genomics. urlhttp://www.xweave.com/people/mgraves/pubs/tods-unpub.pdf [Online; accessed 20-March-2015], nd. Technical report from the Department of Cell Biology, Baylor College of Medicine.

[19] Anthony J. F Griffiths. *Introduction to genetic analysis*. W.H. Freeman, 2012. ISBN 9781429276344 1429276347. URL http://www.worldcat.org/search?qt=worldcat_org_all&q=1429276347.

[20] Graham Hamilton. *JavaBeans*, August 8, 1997. Version 1.01-A.

[21] M. Hattori et al. The DNA sequence of human chromosome 21. *Nature*, 405:311–319, 2000.

[22] Christian Theil Have and Lars Juhl Jensen. Are graph databases ready for bioinformatics? *Oxford Journals Bioinformatics*, 29:1–2, 2013.

[23] J.W. Havender. Avoiding deadlock in multitasking systems, 1968.

[24] Michael Himsolt. *GML: A portable Graph File Format*, nd. UniversitaÌt Passau, Germany.

[25] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Graph eXchange Language. `http://www.gupro.de/GXL/Introduction/background.html`. [Online; accessed 15-May-2015].

[26] Lander et al. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[27] Heng Li. A proposal of the Grapical Fragment Assembly format. `http://lh3.github.io/2014/07/19/a-proposal-of-the-grapical-fragment-assembly-format/`. [Online; accessed 2-June-2015], 2014.

[28] Heng Li. On the graphical representation of sequences. `http://lh3.github.io/2014/07/25/on-the-graphical-representation-of-sequences/`. [Online; accessed 2-June-2015], 2014.

[29] Yu Lin, Sergey Nurk, and Pavel A Pevzner. What is the difference between the breakpoint graph and the de Bruijn graph? *BMC Genomics*, 15 (suppl 6), 2014.

[30] Yongming Luo et al. *Semantic Search over the Web*, chapter 2. Springer, 2012.

[31] Jonathan Marchini and Bryan Howie. Genotype imputation for genome-wide association studies. *Nature*, 11:499–511, 2010.

[32] Mark P. Miller and Sudhir Kumar. Understanding human disease mutations through the use of interspecific genetic variation, 2001. Department of Biology, Arizona State University, Tempe, AZ 85287-1501, USA.

[33] Paul Miller. Does Linked Data need RDF? `http://cloudofdata.com/2009/07/does-linked-data-need-rdf/`, 2009. [Online; accessed 20-May-2015].

[34] Andrew Newman. JRDF - An RDF Library in Java. `http://jrdf.sourceforge.net`, 2011. [Online; accessed 6-July-2015].

[35] Benedict Paten, Adam Novak, and David Haussler. Mapping to a Reference Genome Structure. ArXiv `http://arxiv.org/abs/1404.5010` [Online; accessed 15-April-2015], 2014.

[36] Paul Scheet and Matthew Stephens. A Fast and Flexible Statistical Model for Large-Scale Population Genotype Data: Applications to Inferring Missing Genotypes and Haplotypic Phase. *The American Journal of Human Genetics*, 78, 2006. Department of Statistics, University of Washington, Seattle.

[37] Karsten Scheibye-Alsing, Steve Hoffmann, Annett M. Frankel, Peter Jensen, and Peter F. Stadler. Sequence Assembly. Technical report, Santa Fe Institute, 2009. SFI Working Paper 2009-04-010.

[38] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections. 2008. Dept. of Computer Science, Univ. of Helsinki, Finland and Dept. of Computer Science, Univ. of Chile.

[39] Danmarks Statistik. *Datafortrolighedspolitik i Danmarks Statistik*, January 28, 2015. Version 1.03.

[40] T.A. Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, June 1984. ISSN 0018-9162. doi: 10.1109/MC.1984.1659158.

[41] Kevin Wilkinson, Craig Sayers, Harumi A Kuno, Dave Reynolds, et al. Efficient rdf storage and retrieval in jena2. In *SWDB*, volume 3, pages 131–150. Citeseer, 2003.