PhD Thesis

Computational methods for the analysis of next generation viral sequences

A thesis submitted to the University of East Anglia for the degree of Doctor of Philosophy.

Student:

Sergey Lamzin

Registration No:

4966694

Supervisors:

Dr. Mario Caccamo

Prof. Richard Morris

Dr. Pablo Murcia

February 28, 2016

Abstract

Recent advances in sequencing technologies have brought a renewed impetus to the development of bioinformatics tools necessary for sequence processing and analysis. Along with the constant requirement to be able to assemble more complex genomes from ever evolving sequencing experiments and technologies there also exists a lack in visually accessible representations of information generated by analysis tools.

Most of the novel algorithms, specifically for *de novo* genome assembly of next generation sequencing (NGS) data, are not able to efficiently handle data generated on large populations. We have assessed the common methods for genome assembly used today both from a theoretical point of view and their practical implementations.

In this dissertation we present StarK (stands for k^*), a novel assembly algorithm with a new data structure designed to overcome some of the limitations that we observed in established methods enabling higher quality NGS data processing.

The StarK approach structurally combines *de Brujin* graphs for all possible dimensions in one supergraph. Although the technique to join reads remains in concept the same, the dimension k is no longer fixed. StarK is designed in such a way that it allows the assembler to dynamically adjust the de Brujin graph dimension k on the fly and at any given nucleotide position without losing connections between graph vertices or doing complicated calculations. The new graph uses localised coverage difference evaluation to create connected sub graphs which allows higher resolution of genomic differences and helps differentiate errors from potential variants within the sequencing sample.

In addition to this we present a bioinformatics analysis pipeline for high-variation viral population analysis (including transmission studies), which, using both new and established methods, creates easily interpretable visual representations of the underlying data analysis.

Together we provide a solid framework for biologists for extracting more information from sequencing data with less effort and faster than before.

Contents

List of Figures

List of Tables

Preface

This dissertation is focused on two approaches for the analysis of Next Generation Sequencing (NGS) data. These two approaches are developed in two parts of the dissertation with a discussion chapter covering each one of them. The first part of this dissertation focuses on our methods for presenting raw and processed sequencing data in visually accessible way. While already many methods for analysis exist, few present their results in a visually accessible way. We demonstrate a full analysis pipeline that, given input data samples, generates various visualisations that aid in determining inter- and intra-host variation within the viral population. The pipeline incorporates both established and newly developed methods, which we detail in the following chapters.

In the second part we discuss how similar viral sequencing samples can be assembled *de novo*. We discuss the theoretical concepts behind modern sequence assembly algorithms, putting them into a formal language framework at the same time. We then evaluate their performance on viral sequencing samples and finally present StarK — our own assembler, specifically designed to assemble high-variation viral genome samples and overcome the shortcomings of existing algorithms.

We then discuss the theoretical background of the StarK assembler and how it improves upon the existing *de Brujin* graph assembly approach. Finally we present in depth details on our implementation of a prototype assembler based on the StarK theoretical framework and our efforts on increasing its performance.

Part I.

Viral Population Analysis

1. Introduction

1.1. Motivation

Ribonucleic acid (RNA) viruses constitute a significant source of emerging infections in humans and animals, threatening both public health and food security. They display extremely large population sizes and high mutation rates. This mechanism enables the virus to adapt quickly to new environments and jump host species. The latter has caused various influenza pandemics (epidemics that spread to a large population of humans) throughout history [Bar05] with the Swine Flu pandemic in 2009 being the most recent example [Smi+09]. A better understanding of the dynamics of genetic changes in RNA viral populations is key to gain insight into mechanisms that allow them to jump species barriers, escape host immunity, develop antiviral resistance, or simply become more virulent. The revolution in genomics technologies has resulted in an exponential growth of genetic data at all taxonomic levels. Genetic data can now be generated in most laboratory settings due to the development of powerful and affordable sequencing technologies. The field of virology is one of many areas of research that have benefited from this technological leap: the influenza virus resource (`http://www.ncbi.nlm.nih.gov/genomes/FLU/growth.html`) hosts approximately 16 500 whole genome sequences as of February 2014.

Influenza A viruses (IAVs) are significant pathogens of humans and animals. They have caused four human pandemics since 1918 [Pot01], as well as multiple epizootics (animal/non-human epidemics) with severe mortality, morbidity and socioeconomic costs. Understanding the phylodynamics [HG09] of IAVs is critical in order to de-

vise effective strategies to predict, prevent and/or contain the burden caused by these pathogens. IAVs are members of the Orthomyxoviridae family of viruses, and possess a segmented genome — i.e. it is comprised of multiple pieces of RNA. In the case of IAVs comprising of eight molecules of RNA of negative polarity [BP08], meaning it is negative sense (3' to 5') encoded and positive sense (5' to 3') RNA must be first produced prior to translation. IAVs evolve principally by single-point mutations and reassortment, the former as a consequence of the lack of proofreading activity of the viral polymerase and the latter due to the segmented nature of the genome. Studies on intra- and inter-host influenza variation [Mur+10] have provided insight on the phylodynamics of IAVs, showing that significant genetic diversity is generated during the course of infection, though this diversity is often ignored, instead summarising the genetic information in the form of a consensus sequence (averaged across the population). Also, such studies have shown that a proportion of variants are maintained at low levels along the course of infection and are even being transmitted. Most studies on within-host influenza dynamics have relied on the analysis of sequences derived from the hemagglutinin 1 (HA1) region of the HA gene [Gar+09], generated by capillary sequencing [Smi+85] of cloned Polymerase Chain Reaction (PCR) [Mul+87] products. Although informative, a more comprehensive approach at the whole genome level is required to better understand the evolutionary mechanisms at work during influenza infections.

1.2. Project Objective

The Illumina platform is particularly suitable to study the extent and structure of viral populations as it provides ultra-deep coverage and high accuracy. However, analysing and mining such large sequencing datasets in a computationally efficient and biologically meaningful way is challenging, and requires the development of new computational tools. Probably the biggest limitation of within-host viral population studies is the generation of artefact mutations in the laboratory during the reverse transcription

polymerase chain reaction (RT-PCR) [Bus02], PCR [Mul+87], and sequencing. Identifying these errors is one of the key challenges when screening sequence data. In previous work, our collaborators (Dr. Pablo Murcia and Dr. TJ McKinley) developed a statistical method for screening sequence data for single-site mutations-of-interest [McK+11] based on modelling the observation process rather than the underlying mechanisms driving evolution and/or sequencing error. This method can be used to search and filter for variants that are unlikely to be purely artefacts of the sequencing process. This is particularly important if we consider that a recent study has shown that very few mutations are required to adapt highly pathogenic avian influenza viruses to transmit in mammals [Mur+10]. Importantly, this method uses all available information on genetic diversity obtained from multiple within-sample sequences, rather than simply comparing consensus sequences. The efficiency of this screening algorithm is dependent on the degree of heterogeneity in the observed sequences and the number of longitudinal samples, and is why the analysis of Next Generation Sequencing (NGS) datasets can become computationally intensive and in some cases practically unfeasible.

By modelling the observation process directly (i.e. making no assumptions about the possible origin of an observable deviation from the prior) one can capture a wealth of deviations from the mean. This comes though at the cost of having to apply a filter subsequently to distinguish variants of interest from noise/errors introduced by the sequencing process.

Although many tools already exist that facilitate research in this direction (and we have also used several here), their use often requires advanced bioinformatics or computer science knowledge to

- prepare the data to be processed,
- run the tools (possibly on limited hardware),
- interpret the results (which are often provided in binary or text form).

The objective of this project was to create a fully automated pipeline which, given datasets similar to the ones described in section 1.3, runs a series of analyses fully au-

tomated and produce visually accessible results without requiring special bioinformatics training beyond knowledge of a Linux command line from the user. The visualisations provided allow for both quick visual overview (e.g. whole genome variation clustering) of the data as well as more detailed versions (e.g. per-cite nucleotide counts).

In order to achieve this we have both created new tools and incorporated existing ones into a semi automated analysis pipeline. A flowchart visualisation is shown in Figure 1.6, the details are discussed in the following sections.

We applied this pipeline to characterise the mutational spectra of longitudinal intra-host influenza virus populations derived from pigs experimentally infected with the 2009 H1N1 pandemic virus (pdmH1N1) in two studies (described in section 1.3) and present here our methodology along with a small set of results exemplifying the power of our software.

1.3. Influenza Transmission Studies

1.3.1. Influenza A Viruses (IAVs)

IAV is one of six genera of the family *Orthomyxoviridae*. These are characterised by segmented, negative-sense, single-stranded RNA genomes. Although sharing common ancestry with Influenza B virus, Influenza C virus, Isavirus, Thogotovirus and Quaranjavirus these viruses have genetically diverged throughout evolution. IAVs' host diversity ranges from humans to domestic animals and in rare cases wild poultry. In rare cases recombination between different strains of IAV can cause change in host, which can lead to epidemics as in the case of the 2009 Swine Flu [Smi+09].

IAV genome structure The IAV genome consists of eight negative-sense, single-stranded RNA segments (Table 1.1) [BP08]. Virologists classify different strains of IAVs depending on their variants of HA and NA, as these virus surface proteins are

Segment	Length	Encoded protein	Function
1	2341	PB2	Polymerase subunit
2	2341	PB1, PB1-F2	Polymerase subunit
3	2233	PA	Polymerase subunit
4	1778	HA	Surface glycoprotein; major antigen, receptor binding and fusion activities
5	1565	NP	RNA binding protein; nuclear import regulation
6	1413	NA	Surface glycoprotein; sialidase activity, virus release
7	1027	M1, M2	Matrix protein
8	890	NS1	Interferon antagonist protein
		NEP	Nuclear export of RNA

Table 1.1.: Influenza A virus genome segments.

responsible for the antigenicity of the virus (ability to infect their host). This leads to IAV designations like H1N1, H3N2. The virus classes referred to in this thesis are

- H1N1: 2009 Swine Flu [Smi+09], 1918 Spanish Flu [Gat09]
- N3N2: Common Human Influenza [Rus+08].

Throughout this thesis we will be referring to the segments by the name of the longest encoded protein. Most of the search for sites of interest with variation has only been performed within the protein coding region of the segments.

1.3.2. Experiments & Sequencing

All work detailed in this subsection was done by Dr. Pablo Murcia under GB Home Office Licence following full ethical approval.

In order to study inter- and intra-host virus population dynamics several *in vivo* animal

experiments were performed which we categorise into three studies:

Experimental Infection Study (EI): 8 twelve-week-old piglets seronegative to IAVs of the H1N1, H1N2 and H3N2 subtypes were inoculated with $10^{5.4}$ EID_{50} of A/England/195/09. Nasal swabs were collected for up to eight days after infection. All animals were housed in the same cage.

Transmission Study of pandemic H1N1 (TR): Four naïve twelve-week-old piglets were inoculated with $10^{5.4}$ EID_{50} of A/England/195/09. On day two after inoculation they were housed together with six naïve piglets for 48 hours. At this time point (four days post initiation of the experiment) the recipient piglets were separated from the inoculated donors and further housed together with three naïve piglets. During the course of the experiments nasal swabs were collected on a daily basis. Figure 1.2 shows an overview of the study.

Ferret Transmission Study of endemic H3N2: Two cages, each capable of housing four ferrets, were set up. Two ferrets per virus were inoculated with 1×10^4 Pfu A/Victoria/3/75 or 1×10^5 Pfu Vic-226-228HA. On day 1 post infection, three naïve sentinel ferrets were co-housed with each inoculated donor. Each day, nasal washes collected from the animals were tested for the presence of virus. These studies have been performed by Kim L. Roberts and are documented in the paper "Lack of transmission of a human influenza virus with avian receptor specificity between ferrets is not due to decreased virus shedding but rather a lower infectivity in vivo" [Rob+11]. Figure 1.3 shows an overview of the study. One of the data sets from this study is used as a training set for StarK (chapter 8).

Viral RNA was extracted from the collected swabs, amplified and sequenced on an Illumina HiSeq 2000. Since this work was done before this thesis, it is not further described here.

The contents of this thesis focuses on the methods employed to analyse the data at

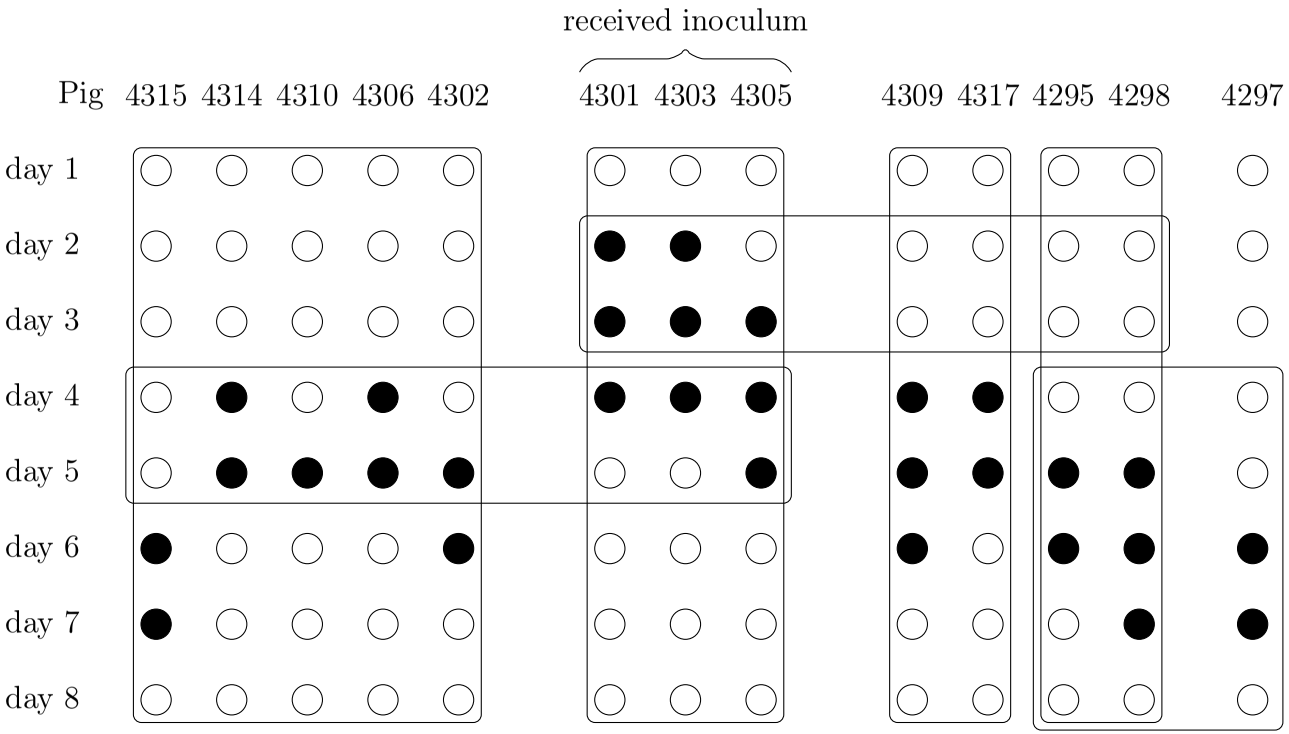


Figure 1.2.: Transmission Study of pandemic H1N1. Columns represent pig identification numbers. Filled circles mark days for which samples of virus have been successfully sequenced. Boxes show which animals were housed together at which point in time. Days are measured in days post infection.

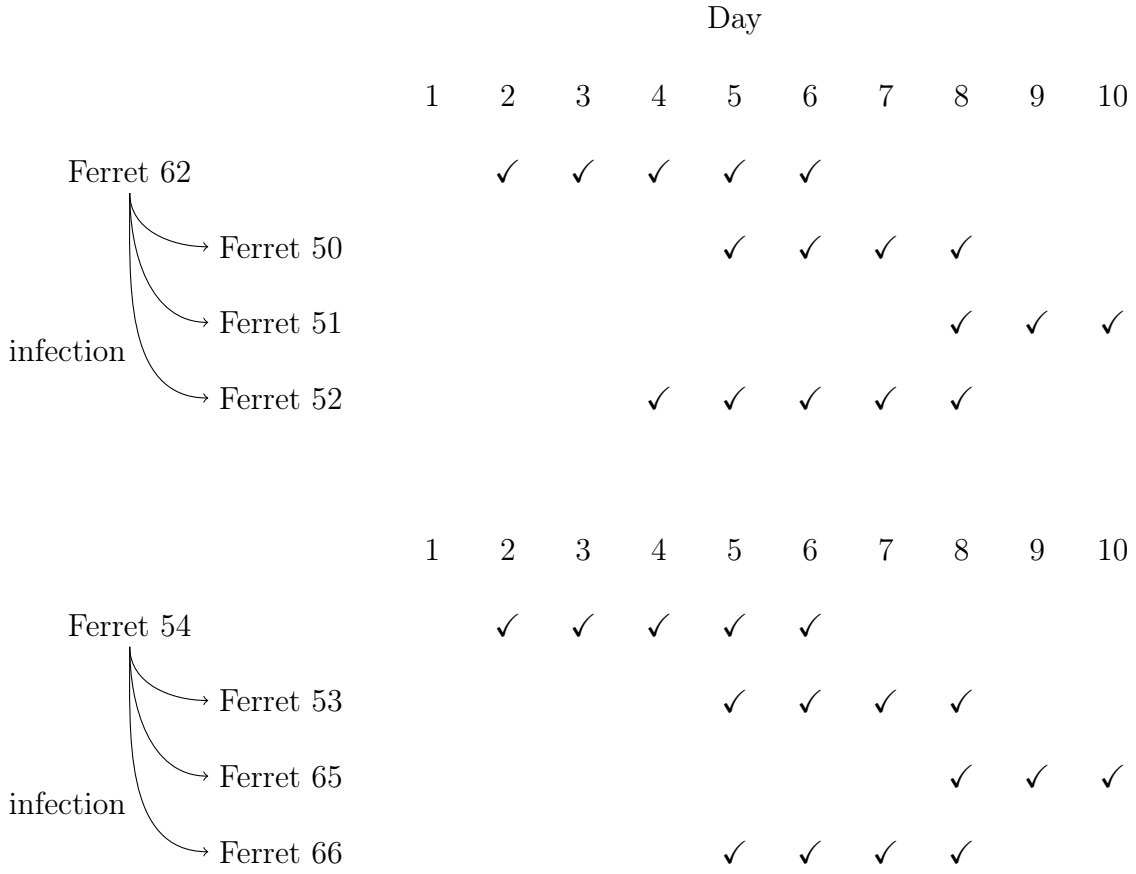


Figure 1.3.: Ferret Transmission Study of endemic H3N2 sample overview. A tick (✓) indicates that enough virus was extracted from a blood sample at that day to carry out sequencing. Ferrets 62 and 54 received the inoculum.

hand and the resulting computational and mathematical tools. The next chapters will explain our analysis strategy and demonstrate the type of information that our analysis pipeline is capable of generating on the example of the Experimental Infection Study with occasional references to the Transmission Study. Data sets from the Ferret Transmission Study are used as training sets for StarK and referred to in chapters 6 and 9.

1.3.3. Studies Summary

Tables 1.4 and 1.5 summarise the type of reads that were used during development of our analysis methods.

In both studies very high average coverage has been achieved (approximately 5000×). Over two thirds of the sequenced reads were not mapped to the National Center for Biotechnology Information (NCBI) [Gee+10] reference genome A/swine/England/453/2006. In order to determine the cause we have assembled the unmapped reads with `velvet` [ZB08] and used Basic Local Alignment Search Tool (BLAST) [Alt+90] to determine their origin. Most contigs aligned against influenza, which is expected given that this is our intended target. One conting aligned against various *Neisseria meningitidis* strains with an E-value of 0.0. Among the remaining contigs, top hits (E-values below 0.1) were contaminants: various bacteria, pig (host), human (lab technician).

All animals, where more then one sample was acquired, displayed variable sites of interest as detected by the Bayesian method `seqmutprobs` (described in subsection 3.3.1) and a significantly higher amount of non-synonymous mutations were found in those sites when compared to synonymous ones.

sample		coverage			genome		reads			sites of	variants	
animal	day	min	avg	max	sites	%	total	mapped	%	interest	nonsyn	syn
3467	2	2	2779	69 686	13 136	99.5%	1 856 640	328 828	17.7%	n/a	n/a	n/a
3468	3	2	4562	107 050	13 177	99.8%	2 481 416	563 104	22.7%	n/a	n/a	n/a
3473	2	9	4204	113 992	13 167	99.7%	2 201 764	523 711	23.8%	59	30	27
	4	4	4023	52 332	13 165	99.7%	1 544 256	499 860	32.4%		43	38
	5	1	3138	48 148	13 138	99.5%	1 146 452	378 840	33.0%		38	24
	6	12	6224	97 813	13 171	99.8%	2 471 686	758 099	30.7%		35	11
3474	3	5	4446	88 911	13 149	99.6%	2 091 920	531 943	25.4%	n/a	n/a	n/a
3475	2	3	1229	35 056	13 169	99.8%	641 228	150 806	23.5%	33	20	8
	3	2	1390	36 887	13 156	99.7%	1 000 906	171 259	17.1%		21	8
	4	1	1410	23 688	13 088	99.1%	605 384	173 024	28.6%		20	9
	5	12	7819	104 641	13 172	99.8%	3 261 914	963 106	29.5%		15	8
3480	2	2	1240	24 937	13 171	99.8%	519 318	155 285	29.9%	31	3	1
	3	4	1397	27 426	13 161	99.7%	613 438	177 278	28.9%		4	1
	4	5	4306	70 535	13 171	99.8%	1 809 068	537 018	29.7%		2	1
4130	3	3	11 752	310 173	13 152	99.6%	6 308 118	1 431 862	22.7%	65	8	5
	4	1	5483	173 813	13 092	99.2%	3 522 180	667 898	19.0%		26	10
4131	3	2	6622	141 191	13 172	99.8%	3 370 884	819 396	24.3%	106	16	5
	4	1	4433	148 027	12 365	93.7%	2 909 944	538 971	18.5%		44	7
	5	6	6018	181 623	13 052	98.9%	3 750 792	746 477	19.9%		38	16
avg		4.1	4340	97 680	13 106	99.3%	2 216 174	532 461	25.1%	56.8	22.7	11.2
max		12	11 752	310 173	13 177	99.8%	6 308 118	1 431 862	33.0%	106	44	38

Table 1.4.: Summary statistics of the experimental infection study.

sample		coverage		reads		sites of	variants	
animal	day	avg	max	total	mapped	interest	nonsyn	syn
4295	5	3516	106 109	2 081 084	20.3%	10	7	3
	6	2107	56 227	970 122	26.4%		6	5
4297	6	2146	86 634	1 296 698	19.1%	n/a		
4298	5	7562	236 451	4 228 184	22.1%	92	6	5
	6	9490	292 089	5 260 926	21.7%		13	10
	7	3419	105 521	2 037 972	20.9%		28	14
4301	2	6134	182 918	3 364 538	23.2%	83	26	7
	3	5837	283 119	5 004 986	14.3%		54	16
	4	4917	187 155	3 742 662	15.9%		80	17
4302	5	11 801	272 213	5 938 080	24.3%	85	25	6
	6	4263	153 625	2 759 880	19.0%		58	19
4303	2	4368	193 575	3 689 764	14.6%	44	16	7
	3	4262	142 116	2 672 464	19.5%		23	7
	4	9730	337 056	6 724 270	17.4%		19	7
4305	3	2276	52 095	1 201 990	23.8%	43	0	1
	4	5337	180 401	3 407 674	19.5%		3	1
	5	2906	74 092	1 448 616	25.2%		1	1
4306	4	7607	145 312	3 141 228	28.8%	48	10	4
	5	2946	94 047	1 613 048	22.1%		57	15
4309	4	6556	163 896	3 267 086	24.1%	161	22	4
	5	12 858	340 998	6 578 830	24.3%		36	9
	6	14 511	432 191	8 548 222	21.8%		74	18
4310	5	4713	104 085	2 109 042	27.6%	n/a		
4314	4	2938	76 784	1 368 756	25.8%	77	12	5
	5	6761	251 885	4 289 118	19.4%		38	10
4315	6	3345	91 857	1 573 180	25.2%	49		
	7	3501	173 828	2 651 266	16.0%			
4317	4	8446	208 334	3 966 866	25.9%	90	16	6
	5	8036	224 396	4 731 434	21.2%		53	26
avg		5941	181 000	3 436 827	21.7%	73.6	27.3	8.9
max		14 511	432 191	8 548 222	28.8%	161	80	26

Table 1.5.: Summary statistics of transmission experiment.

1.4. The Analysis Pipeline

During the course of this project we have designed and implemented a semi-automated analysis pipeline that both incorporates known tools like the Burrows-Wheeler Aligner (BWA) and the `seqmutprobs` Bayesian variant analysis package (both explained in section 2.4 and subsection 3.3.1 respectively) and adds novel analysis methods.

The pipeline (flowchart shown in Figure 1.6) requires minimal input from the user. The input reads and some meta data have to be provided though in order to run:

a) The sequencing reads, can be either as paired end (in which case paired end information will be used) or as single reads.

b) Reference genome for the virus in question,

c) Table containing meta data about the experiment:

   - Mapping of datasets $\leftrightarrow$ animal, day
   - Regions/reading frames within the genome that are of interest.

At this point the analysis can be performed with the invocation of only two analysis scripts (one for within-sample and one for intra-sample analysis) and will produce the following analysis results:

1. Quality control charts,
2. Shannon entropy plots,
3. Coverage plots,
4. Diversity plots,
5. Mutations breakdown Hypertext Markup Language (HTML) tables,
6. Hierarchical clustering of the samples by variation.

Both scripts can be run non-interactively on a cluster and support automatic scheduling

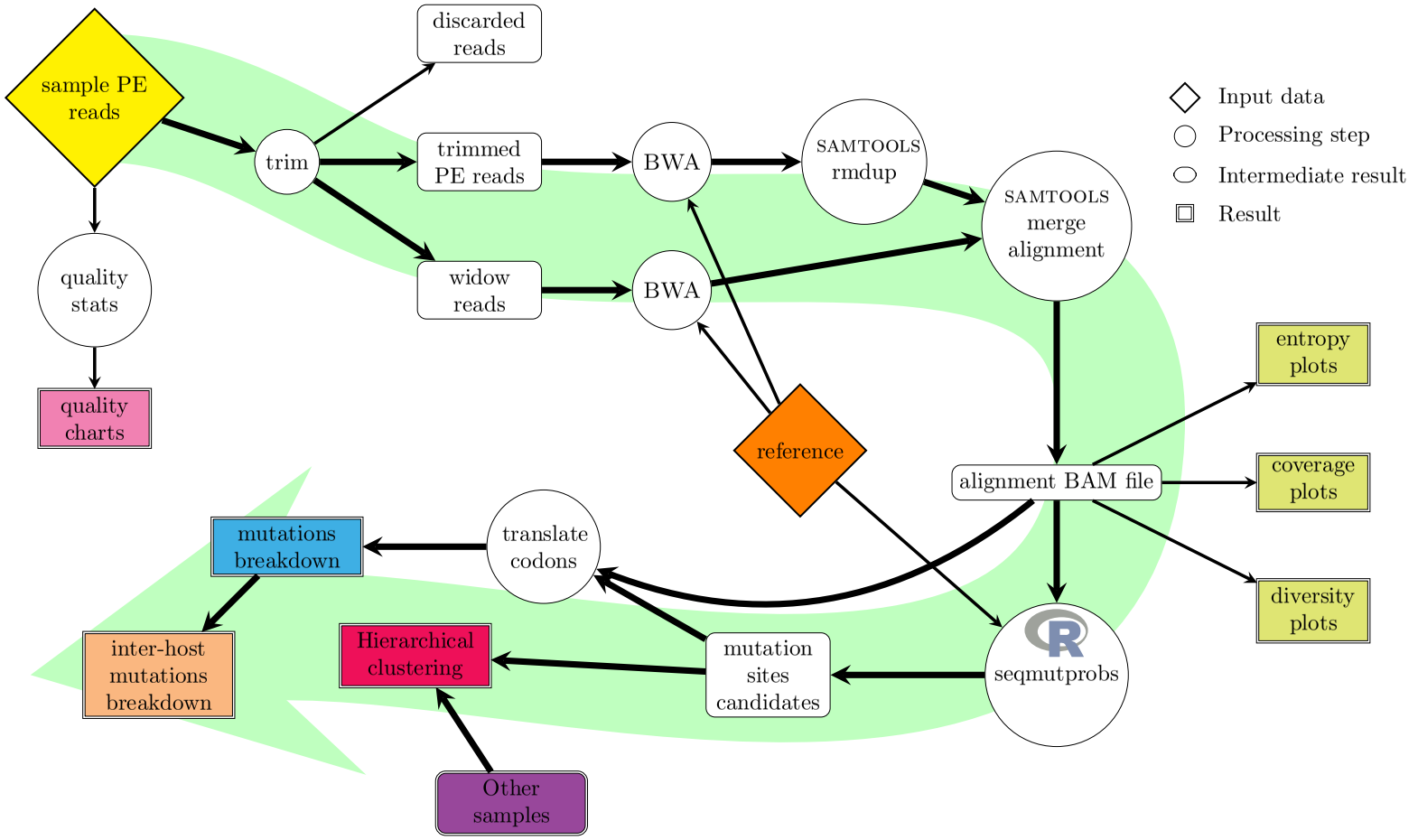


Figure 1.6.: Flowchart visualisation of our semi-automated analysis pipeline. Detailed explanations of the individual parts of the pipeline are discussed in the following chapters.

on Platform Load Sharing Facility (LSF).

In the following chapters we will discuss the methods, that we designed and incorporated into the pipeline, as well as present example outputs and their possible interpretations.

The source code for the analysis pipeline is currently being further developed and maintained by Dr. George Kettleborough and the current version can be obtained by contacting him via email George.Kettleborough@tgac.ac.uk.

We have divided the analysis pipeline into three major parts:

1. Primary Data Preparation. Encompasses everything from quality control, reads filtering to alignments and data condensation.
2. Within-host population dynamics. Single sample and single host variation analysis. As the experiments were conducted as transmission experiments the first step is to analyse what variants the virus displays in each sample and then how the virus population changes within a single host.
3. Inter-Host Variation Analysis. Our attempt at trying to piece information together from the entire study and see how the virus populations change between hosts.

2. Primary Data Preparation

2.1. Introduction

Before we can begin variant analysis on raw NGS reads we have to first determine the quality of the obtained reads, align them (ideally to a reference genome of the same species) and perform sanity checks in order to ensure the most accurate data interpretation.

This chapter focuses on our methods for achieving the above and the visualisations that our pipeline provides to the user for understanding the quality of the reads within data samples.

The statistical summary tables (as shown in subsection 1.3.3) are (with the exception of sites of interest) derived from meta data collected during this stage and serve a statistical purpose. Using those we were for instance able to exclude bad samples (bad quality scores or very few reads) or samples with unknown content (less than 1% of reads aligned). Please note that excluded samples are not shown in the summary Tables 1.4 and 1.5.

2.2. Quality Control

Today many more sophisticated NGS quality control tools exist, though at the time this project was started they were not yet available. As such we have designed a simple

program to evaluate the overall quality of the reads obtained from the sequencer. The first step builds a simple histogram of quality score counts across all reads within a sample and produces a plot as displayed in Figure 2.2.

The plot in Figure 2.2 displays a quality score histogram of a "good" sample. The quality scores' distribution is what we expected from the sequencers back in 2010. The decay in the quality scores towards the end of the read is normal and follows our expectations. As a comparison Figure 2.1 shows a quality histogram chart of a failed sample. There are far fewer reads in total (as seen by the scale for the heatmap on the right hand side of the plot) than we would expect and the average nucleotide quality

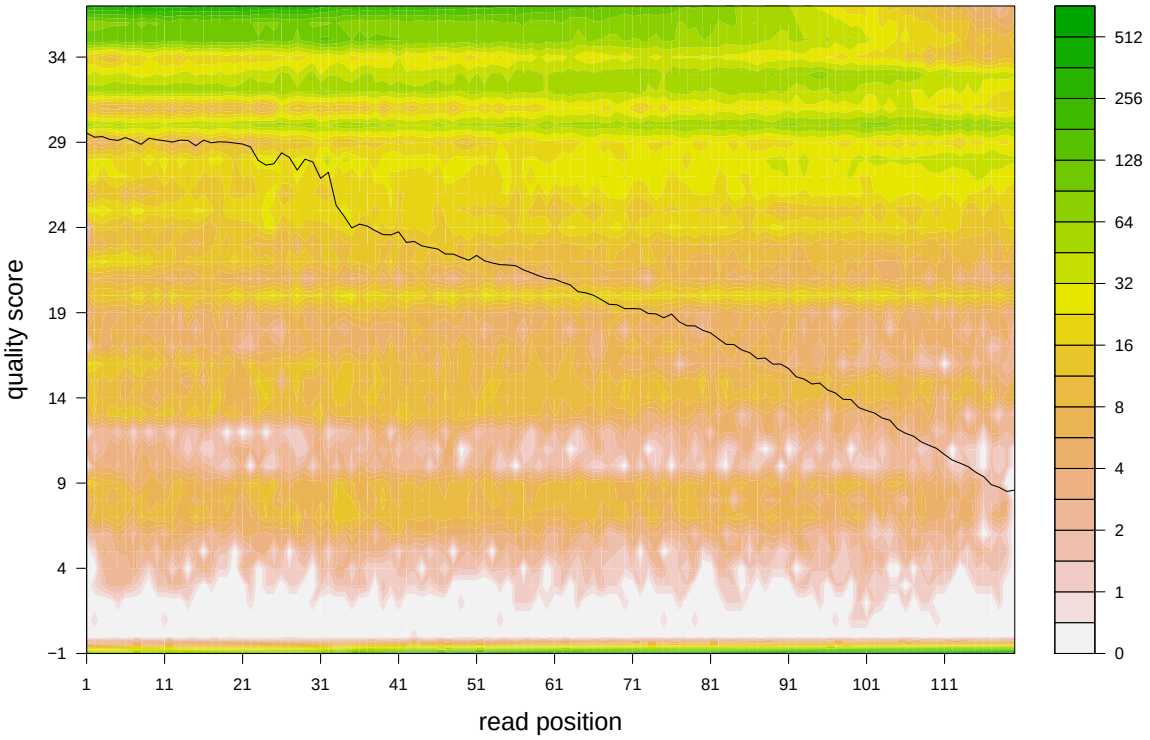


Figure 2.1.: Quality scores histogram of influenza sample from Pig 3468 day 2. This is a failed sample. There are far fewer reads in total than we would expect (please note the scale on the right hand side) and the average nucleotide quality scores are below what was "normal" for the technology at the time (2010).

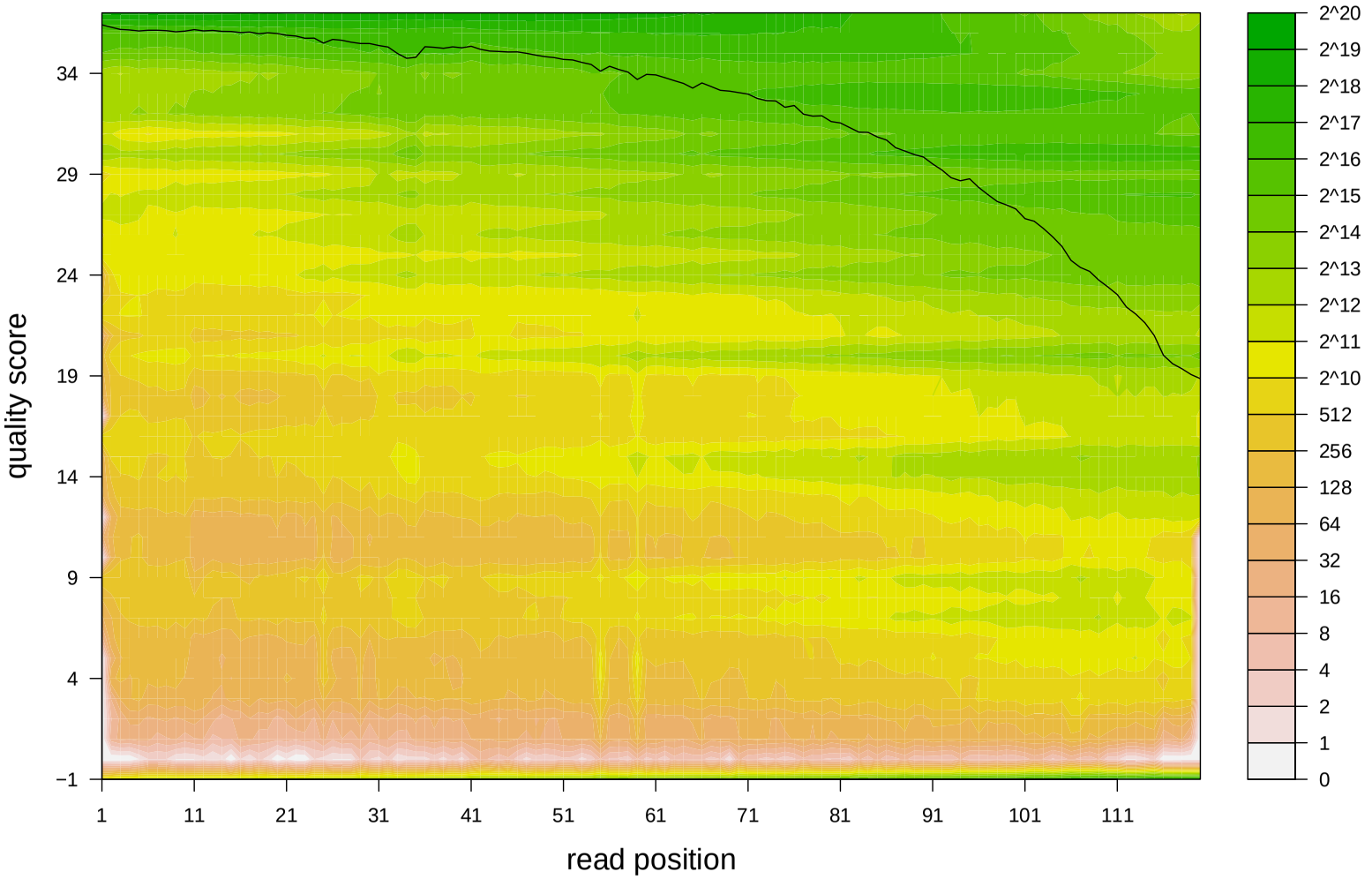


Figure 2.2.: Quality scores histogram of influenza sample from Pig 3473 day 4. The black line shows the average quality at each position. The quality scores correspond to the encoded Illumina score characters minus `'C'`, i.e. -1 corresponds to the Illumina quality score `'B'`. Please notice the abundance of nucleotides scored -1 at all positions.

scores are below what was "normal" for the technology at the time.

These plots are intended as visual aids in determining whether a sequencing sample can be expected to yield useful results in further down analysis. At the time our Swine Flu samples were sequenced no automatic quality control was being done yet by the sequencing team. And no tool for determining the quality of a sample was available beyond simple statistics (mean quality score, standard deviation). We wanted a tool that will visualise the distribution of quality scores throughout the sample, but also putting it into context of position within a read as quality scores tend to drop towards the ends of a read.

We chose represent this three dimensional data using a heat map to indicate quantity of nucleotides with a given quality score depending on their position within a read. This allows us to assess how they are distributed and whether there are any visual anomalies like certain scores being favoured or sudden jumps. The black line showing the mean quality score per position was added to compare with existing mean quality score plots that scripts produced in 2010. We decided to omit variance information (which can be visually implied by a viewer from the heat map) to increase visibility of the overall plot.

Our pipeline automatically generates a quality score histogram chart for each sample to aid the user in tracking down failed sequencing runs.

2.3. Sequencing Read Trimming

As seen in the quality score histograms (Figure 2.2) there are a number of nucleotides scored at -1 (Illumina score `'B'`) which separate from the remainder of the histogram. Based on our observations these are "filler" nucleotides inserted by the sequencing platform to complete a full 100 base pair read. These nucleotides almost never align against our reference and often cause the entire read to be discarded by the aligner as a result.

In order to aid the downstream aligner we have designed a simple read trimmer that removes those score -1 nucleotides from reads following the criteria:

- Remove any continuous leading or tailing sequence of nucleotides that all have a quality score of -1 (`'B'`).

  An aligner will decide whether to align or discard a read based on a score that is determined by the ratio of aligned nucleotides versus unaligned nucleotides. Removing these -1 quality scored nucleotides will raise the aligner score for the remaining ones if they align and may make the aligner keep the remainder of the read.

- If the above process removes more than 50% of the read – discard the read.

  If more than half of the read consists of filler nucleotides then we decided not to trust the remainder either. The underlying aligner does not know that the read was initially more than twice as long as seen by the aligner. We want to reduce the chances of it being misaligned due to short length.

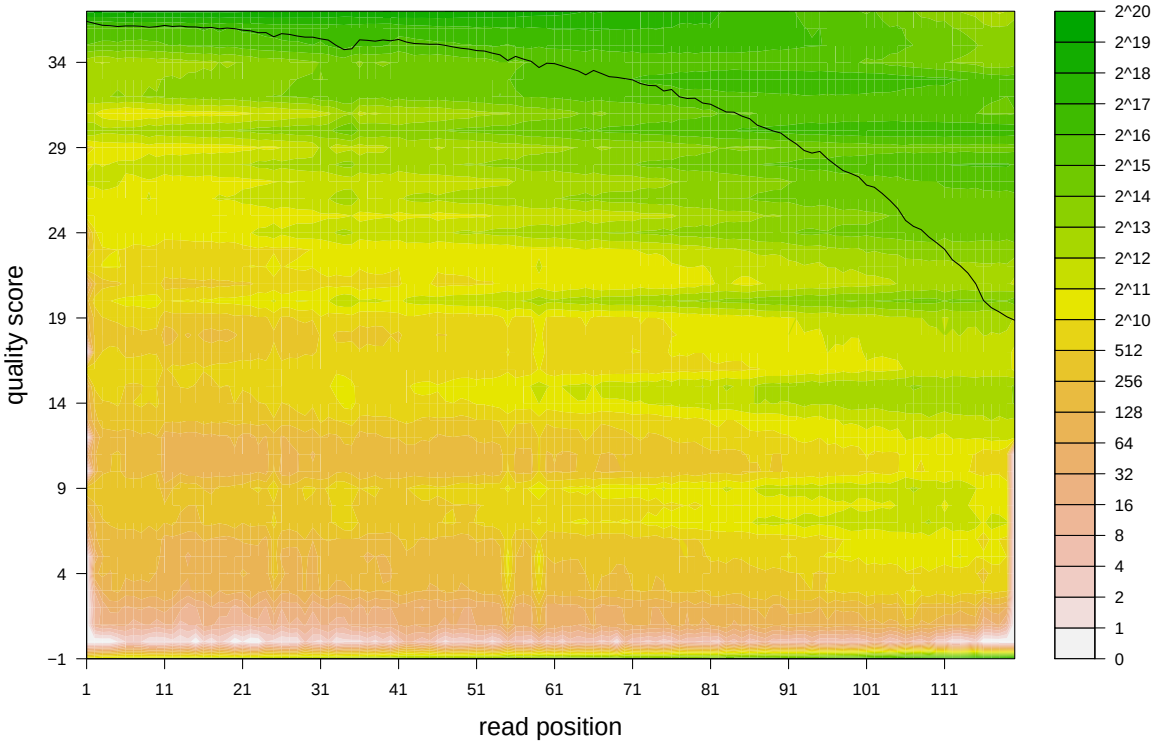
- If the above process discards only one of two reads from a pair (paired end sequencing), then the remaining read is saved as a single **widow** read.

  Sequencing errors affecting one read of a pair have no effect on the other, so there is no harm in keeping only one of a pair, but it needs to be treated as a single read from there on.

After the trimming another quality control step is performed and a second set of quality score histograms is generated. Figure 2.3 shows the quality score histogram plots of sample Pig 3473 day 4 before and after trimming side by side. One can observe a major reduction of nucleotides at score -1 and a significant improvement of the average quality score towards the ends of the reads. This leads to more reads being aligned in our runs during the next step than when using untrimmed reads.

The above parameters can be adjusted at the user's will to have a more strict quality cut-off or preserve shorter widow reads.

(a) Quality scores histogram of influenza sample from Pig 3473 day 4.



(b) Quality scores histogram of influenza sample from Pig 3473 day 4 after trimming.

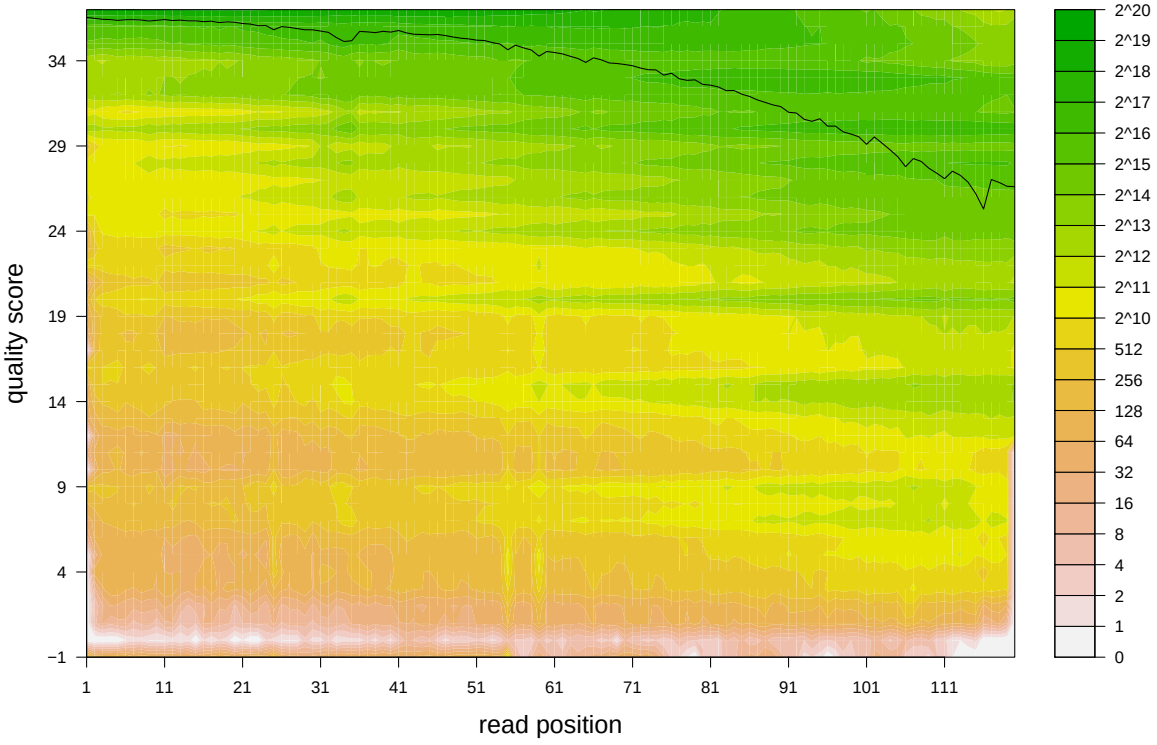


Figure 2.3.: Side by side comparison of quality scores histograms of influenza sample from Pig 3473 day 4 before and after trimming. One can observe a major reduction of nucleotides at score -1 and a significant improvement of the average quality score towards the ends of the reads.

2.4. Alignments

The analysis pipeline presented here requires the reads to be aligned towards a reference genome. The reference genome is not used further down in the analysis, it only provides a means of aligning reads and identifying the necessary read frames for the protein coding regions. Any further references to single nucleotide polymorphisms (SNPs) refer to variants towards the study wide consensus sequence derived from the aligned reads.

For our analyses we have used the Burrows-Wheeler Aligner (BWA) [LD09] (version 0.6.1) with default parameters. The aligner is interchangeable as long as the resulting output is provided in Sequence Alignment/Map (SAM) [Li+09] or BAM format (binary version of the SAM [Li+09] alignment format), i.e. it is simple to replace the BWA [LD09] aligner with the Bowtie aligner [Lan+09] or a more accurate Smith-Waterman based aligner [SW81].

If during the trimming step the reads are split into a set of paired-end reads and a set of single widow reads. `samtools` [Li+09] is then subsequently used to merge the alignments into a single BAM file.

This requirement for an alignment is a major limitation for this type of analysis. A reference genome may not be available for as yet unresearched viral genomes or may be out of date. A first attempt of using assemblies of food and mouth disease virus was attempted before [Wri+11], but even modern assemblers struggle with high variation data sets.

In order to attempt to tackle this limitation we have decided to create a novel genome assembler that not only assembles a consensus sequence, but also does this type of alignment and/or haplotyping during the assembly process. This eliminates both the need for a reference genome and any bias introduced as a result of alignments. This assembler uses a completely new method for assembly that was designed to be able to cope with high variation NGS data.

Our progress towards this goal is presented in Part II of this thesis. Although we did not finish the part of the assembler that does variant analysis and haplotyping, we have created a very fast and accurate general purpose assembler in the process.

2.5. Duplicate removal

NGS data is often polluted by excessive amplification of a few select Deoxyribonucleic acid (DNA) fragments during the library preparation step. This leads to sudden spikes in coverage in these regions of the genome, but does not provide an accurate view of the population diversity.

We often model the selection of fragments from a genome during sequencing using a uniform distribution (compare to subsection 6.2.1). If a genome g is of length $|g|$, and a fragment was selected at a position x then the probability of selecting a second fragment at the same position is

$$\frac{1}{|g|}. \tag{2.5.1}$$

Given n reads on a genome g of length $|g|$ and assuming a uniform distribution of the selection of starting positions for a fragment we will have on average $\frac{n}{|g|}$ fragments starting per position.

The viral genome of Influenza, which we have been working with, is only 14 000 base pairs long. If we would have only sequenced one million reads (in most cases we have more, compare Tables 1.4, 1.5) we would on average observe approximately 71 fragments starting per position.

To help us identify reads originating from a fragment which was amplified excessively from fragments that have randomly originated from the same position we will be using paired-end reads.

Paired end reads are generated by a sequencer that takes a longer DNA fragment than its read length (typically between 300 and 600 base pairs) and sequences it from both

ends up to the sequencer's read length.

Considering this information, in order for two randomly selected fragments to be identical (assuming no repeats of fragment length within the genome) they have to have originated from the same position and have the same length.

Given n reads on a genome g of length $|g|$ and assuming a uniform distribution of the selection of starting positions for a fragment and a uniform distribution of fragment length between 300 and 600 we will have on average $\frac{n}{|g|\cdot(600-300)}$ fragments of the same length starting per position.

Again, given our input data this amounts to only a 23% chance of observing such a pair of fragments. This chance is further decreased by the fact that we are dealing with high-variation viral genomes, erroneous reads and longer DNA fragments than estimated above.

In addition to that, identical fragment reads in abundance do not contribute useful information to variation analysis.

In order to deal with this we use the `samtools rmdup` (remove duplicates) function to scan the alignment files for potential PCR [Mul+87] duplicate reads and remove them [Li+09]. The result is fewer sudden jumps in coverage and data, that should be less biased by the chemistry used during library preparation.

2.6. Alignment pileup

For all our downstream analysis we do not require the alignments themselves, but rather pileup tables derived from those. A pileup table contains for each position in the reference genome the amounts of the different nucleotides that were mapped there. A program that reads the BAM file (interfacing through `libbam` [Li+09]) generates tables for counts of appearance of

- nucleotides per position/sample,

- codons (triplets of nucleotides) per amino acid within protein coding regions.

Those are stored in a SQLite database (`https://www.sqlite.org/`) and are used by all other scripts for downstream analysis. Since in earlier versions of the pipeline these tables were generated on the fly for each subsequent analysis directly from the BAM file, the latter is still used to represent this in the pipeline flowchart in Figure 1.6.

2.7. Consensus Sequences

To determine the level of genetic fluctuation along the course of infection at the whole genome level, we generate a single full genome sequence from each day of each individual sample where coverage permits. The consensus nucleotide at each site is assigned based on a majority rule (i.e the nucleotide that exhibits the highest count).

In addition to this a study-wide consensus sequence is used for all downstream analysis instead of the alignment reference. This is to compensate for any SNPs that may be present in the inoculum compared to the NCBI [Gee+10] reference genome A/swine/England/453/2006.

2.8. Sequence Coverage Visualisation

In order to allow for quick visual assessment of read coverage towards the reference genome the pipeline generates a series of coverage plots depicting a simple numeric pileup of the reads. Figure 2.4 shows the coverage plot of the HA segment of sample from Pig 3473 day 2. This allows for additional post-alignment quality control.

These plots are primarily used as another form of quality control. Severe misalignments or other post-alignment anomalies (i.e. coverage in certain areas) can be identified here.

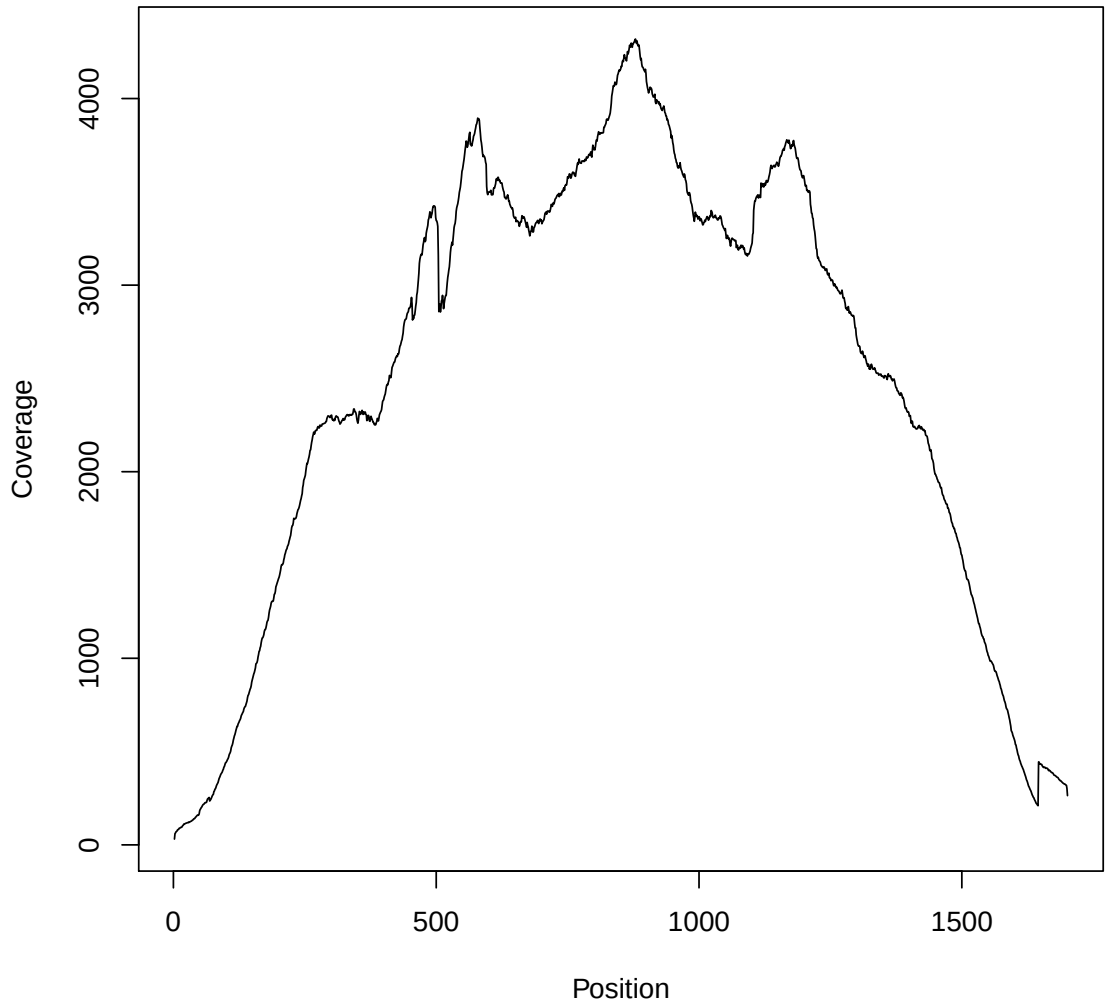


Figure 2.4.: Coverage plot of the HA segment of sample from Pig 3473 day 2. One can make two main observations: The coverage drops at the ends of the segment - this has to do with the lower chance of selecting a fragment for paired end sequencing that covers the ends. The coverage has only few sudden jumps allowing us to deduce read connectivity based on coverage differences as described in subsection 8.2.2.

3. Within-host population dynamics

3.1. Population Diversity Spectrum

Previous studies have shown that IAV within-host populations are highly dynamic even when the consensus sequence remains unaltered [Sta+12]. As such we built our pipeline based around detecting and analysing subtle changes in the viral population that are visible in high coverage deep sequencing data.

The objective of the analysis pipeline is to identify variants within the population. To allow visual assessment of the diversity spectrum within a sample we generate what we call **diversity plots**. A cut out of a sample diversity plot is shown in Figure 3.1. These show both the variants exhibited and their magnitude in a graphical bar plot. The colour coding represents the four different nucleotides. The magnitude of the bars corresponds to coverage at that position. Bars going upwards agree with the study-wide consensus nucleotide at the respective position, bars going down (stacked if more then one) represent variants that disagree with the consensus.

In contrast to the bare coverage plots or the entropy plots (as discussed in the next section, section 3.2) these contain more detailed visual information about the variants at the cost of display space. A more compact and filtered version is later generated for variant sites of interest (see section 4.1).

Since short deletions and insertions within the genome would break the reading frame

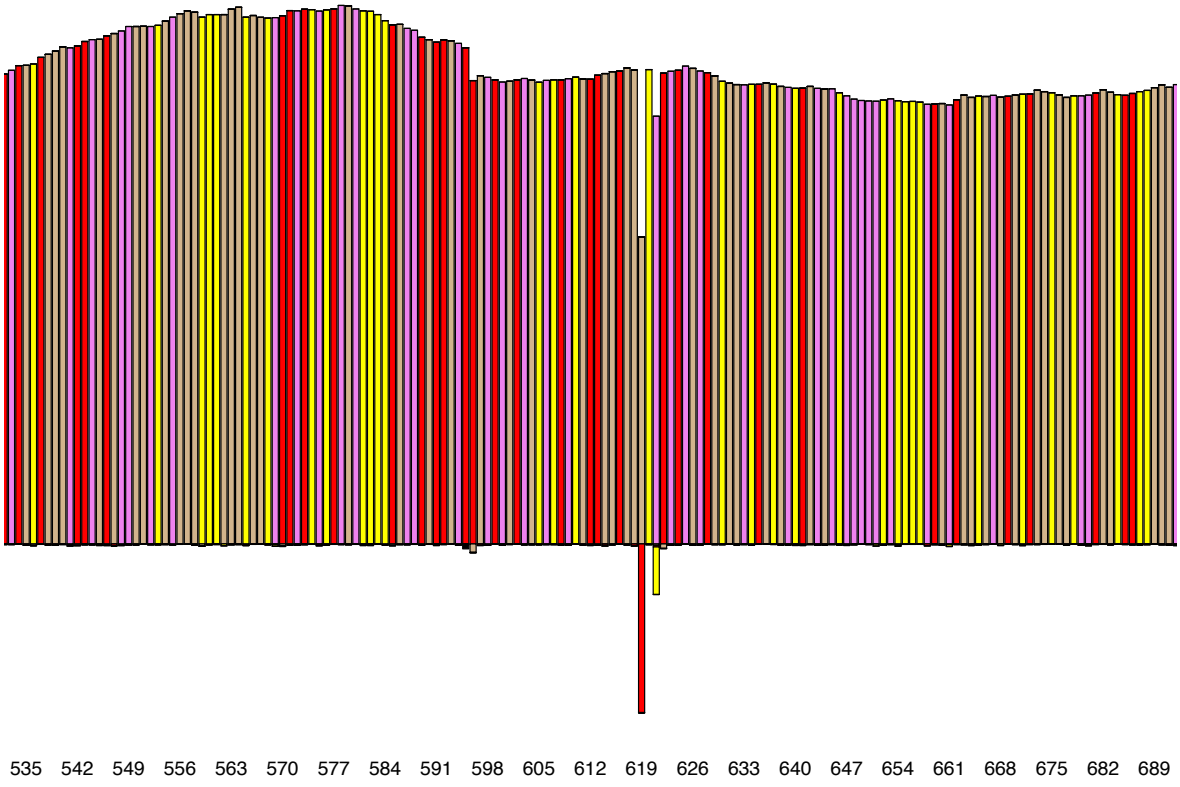


Figure 3.1.: Visualisation of viral genetic diversity. Section of a diversity plot derived from pig 3473 at day 5 post infection. The plot represents the variation present from nucleotide position 535 to 689 in the HA segment. Vertical coloured bars represent individual nucleotide sites and each colour corresponds to a different nucleotide. Bars going upwards are counts of observed nucleotides that agree with the consensus (across all samples), bars going down show those that do not.

for the appropriate protein and thus render the resulting genome non-functional [DSH08] we are not considering short insertions and deletions within our analysis. Longer insertions and deletions are difficult to pick up with short reads and even more so to distinguish from sequencing errors when dealing with an entire virus population.

3.2. Nucleotide Entropy

In order to assign a more visually accessible (and compact) graphical representation than the full-pileup diversity plots to the frequency of a variant at any given site within the genome, we compute the Shannon entropy [Sha48] for each position in each sample.

Shannon entropy is commonly used in information theory for measuring information content within a message. The basic idea is: the more patterns are needed to describe the data the bigger the entropy. In terms of data compression it can be used to measure the minimum amount of bits needed to encode a message. In our case we measure the amount of variation at any given site. The more variation the more entropy — the more information this site carries within the population.

The Shannon Entropy in this case is computed as follows: let A_i, C_i, G_i, T_i be the counts of aligned respective nucleotides and $c_i := A_i + C_i + G_i + T_i$ the coverage at position i. We calculate the nucleotide entropy at position i as

$$H_i = -\left(\frac{A_i}{c_i}\ln\frac{A_i}{c_i} + \frac{C_i}{c_i}\ln\frac{C_i}{c_i} + \frac{G_i}{c_i}\ln\frac{G_i}{c_i} + \frac{T_i}{c_i}\ln\frac{T_i}{c_i}\right) \tag{3.2.1}$$

Using this measure we are assigning a single number (for the purposes of visualisation) to the amount of variation at a single position within the genome. Entropy approaches 0 as all contributing reads display the same variant nucleotide and rises with

- the amount of different variants per position,
- the relative frequency of each variant,

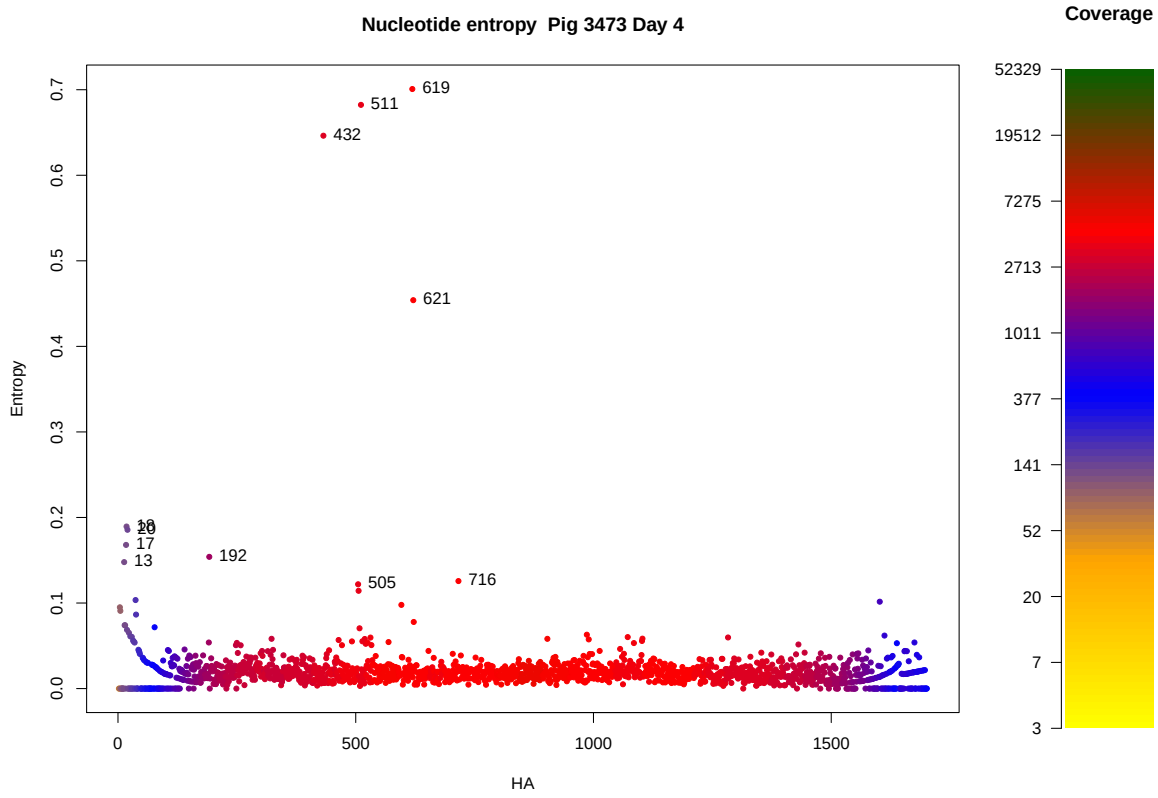


Figure 3.2.: Single sample, single segment Shannon nucleotide entropy plot. The plot shows the entropy levels as computed by Equation 3.2.1 of H1N1 sample from Pig 3473 day 4, HA segment. One can observe clustering of several highly variable mutations between positions 432 and 716. As coverage drops at the ends of the segment, entropy values rise due to the low resolution of the underlying alignment data.

giving us an easily accessible visual representation of variation throughout the genome. Examples are depicted in Figures 3.2, 3.3 and 3.4. Please note that using the natural logarithm the theoretical maximum entropy at a single position is $-\ln {}^1\!/_4 = 1.386$.

As entropy is influenced by the resolution of the data (in the case of genomic pileups – coverage), areas with low coverage are likely to display high entropy values due to background noise having a disproportional effect. We incorporated coverage information to our entropy plots to visualise areas with inflated entropy values. As expected, the ends of the gene segments displayed high entropy due to low coverage (Figures 3.2 and 3.3). In the experimental infection study all but one nucleotide site that exhibited a consensus change displayed high entropy values thus reinforcing this value as a valid measurement for the "magnitude" of variation at a position. The one site where a consensus change did not display high entropy had a distribution of the majority (different) nucleotide above 99%.

Entropy plots are being generated by our analysis pipeline for each sample/segment (Figure 3.2) in addition to full genome (Figure 3.4) and whole-study/segment (Figure 3.3), the latter being a sum of all entropies across all samples of a study. This allows us to identify prominent sites with significant variation across the whole study.

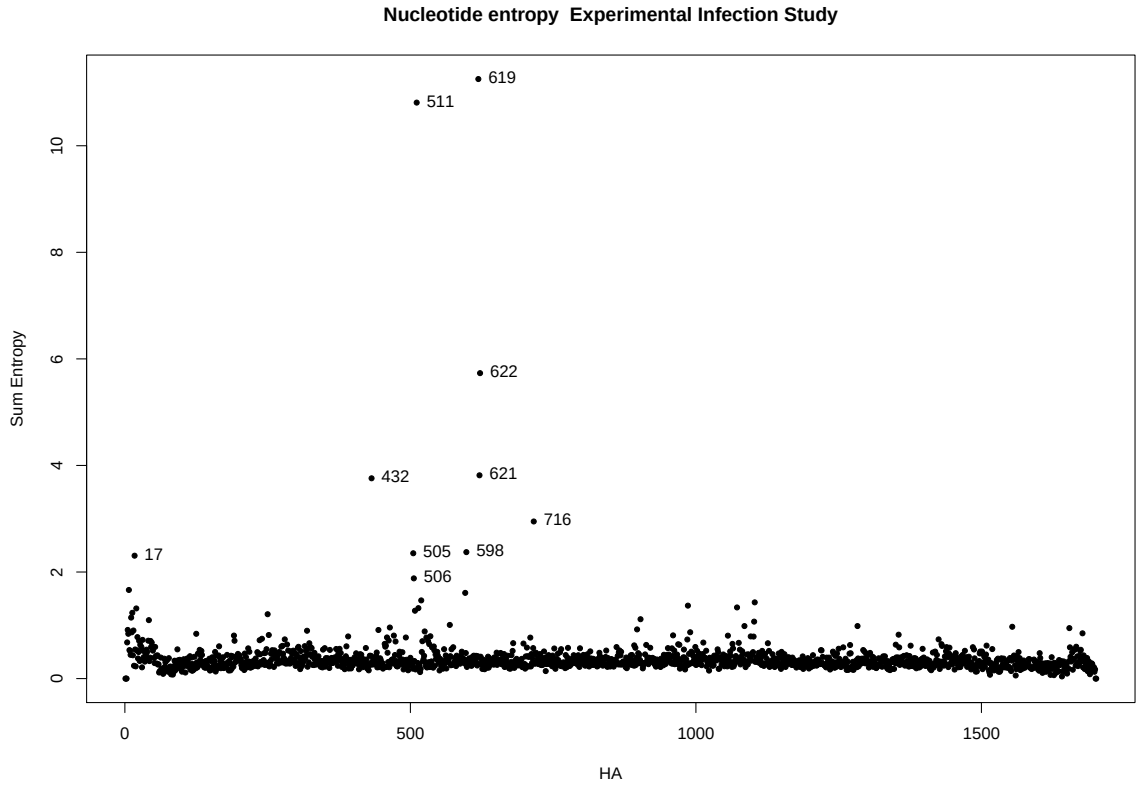


Figure 3.3.: Shannon nucleotide entropy plot for the whole experimental infection study. The plot shows the sum of all entropies across the experimental infection study for the HA segment. As with the entropy displayed in Figure 3.2 one can observe clustering of several highly variable mutations between positions 432 and 716 across the entire study.

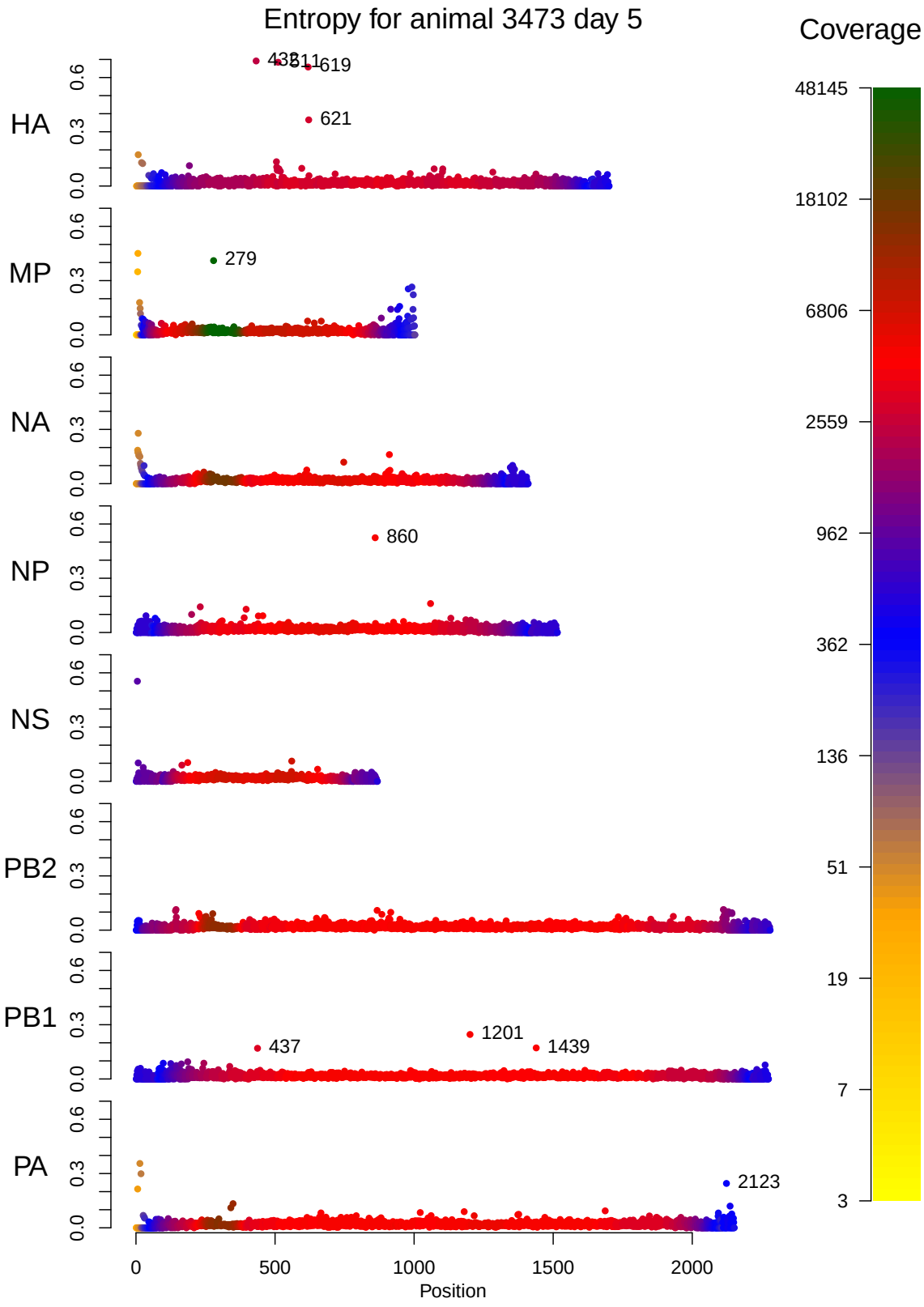


Figure 3.4.: Combined Shannon Entropy plots for all segments of sample Pig 3473 day 5.

3.3. Sites Of Interest

3.3.1. Bayesian statistics

In order to attempt to separate low-frequency variants within the viral population from sequencing errors we have used a version of the "Bayesian Approach to Analyse Genetic Variation within RNA Viral Populations" (as detailed in [McK+11]) that was adopted for the use on high-coverage NGS data. The statistical methodology was adjusted by Dr. McKinley to accommodate the increased amount of information available in NGS data samples versus capillary sequencing. The models were adjusted to allow for a cut-off to be applied to models appearing in very low frequency based on the mean observable variation from the consensus. A beta version of the software that we used for our pipeline, is available for download at the project's home page (`https://github.com/tjmckinley/seqmutprobs`).

In essence the method uses Bayesian statistics to assign a model to the changes in population distributions between samples. If the change in population distribution changes significantly in a position with sufficient coverage (which is a sign of variation), as calculated by Bayesian statistics, then the site is marked as a **site of interest**.

We use the software package (provided as an R [R D10] package), from here on called `seqmutprobs`, as implemented by Dr. McKinley on all samples from one animal combined and then store the results in our SQLite database for easy retrieval.

3.3.2. Comparison with entropy

Although the two approaches are intended for different purposes (`seqmutprobs` detects multi-sample changes in population structure, and the Shannon entropy detects variation in single samples) we have compared the results between the two. High entropy sites were arbitrarily defined as follows:

Segment	Detected by		Exclusive to		Intersection
	`seqmutprobs`	entropy	`seqmutprobs`	entropy	
PB2	156	106	111	61	45
PB1	70	44	51	25	19
PA	180	143	113	76	67
HA	60	70	27	37	33
NP	44	36	24	16	20
NA	48	19	38	9	10
MP	98	7	93	2	5
NS	38	11	30	3	8
$\sum$	694	436	487	229	207

Table 3.5.: Comparison of number of sites detected as variable above noise threshold by Shannon entropy and `seqmutprobs` in the Transmission Study.

- The entropy value must be larger than the study-wide mean entropy + 5 times study-wide entropy standard deviation. This cut-off has empirically shown to almost entirely eliminate noise generated by the sequencing process, but flag biologically interesting sites.
- The site must be at least 120 nucleotides (read length) away from the start/end of the segment. As we have seen on Figure 2.4 coverage drops significantly at the ends of a segment. This artificially increases the entropy in those regions making it difficult to distinguish high entropy due to variation from high entropy due to low sample size.
- The site must exhibit a coverage above 200. This was an arbitrary cut-off to completely exclude low-coverage regions with artificially high entropy due to low sample size in the same way as in the item above.

In the transmission study out of the 694 sites detected by the screening algorithm (same site in multiple animals counts multiple times) 207 displayed high entropies.

Additionally 229 sites with high entropy were not selected by the `seqmutprobs` screening algorithm.

A comparison of the differences in sites marked as sites of interest between `seqmutprobs` and the entropy threshold can be found in Table 3.5. The nucleotide entropy is not intended to distinguish low-frequency variants from noise due to errors in the sequencing process, but to assign a plot-able number to the amount of variation. As such the differences in the sites marked by the two methods is expected.

4. Inter-Host Variation Analysis

4.1. Variant Breakdown Tables

The objective of our analysis pipeline is to provide insights about sites of interest with a virus genome population. Although the diversity plots (Figure 3.1) provide a fair view of all observed variation within a dataset, this information is difficult to perceive without filtering. These plots also provide no insight into the codons and resulting amino acids (synonymous/nonsynonymous mutations). The `seqmutprobs` Bayesian analysis package provides a certain amount of filtering for sites of interest, but lacks a visually accessible representation.

In order to allow us to visually assess and compare the genomic variation at a `seqmutprobs` detected site of interest, we have compiled a script that extracts all pileup data (nucleotide, codon and amino acid) from the SQLite database for those sites and compiles them in easily accessible dynamic tables. In order to allow rendering of the dynamically expandable tables on any system they are represented in form of HTML with JavaScript.

Figure 4.1 shows a static screen shot of a sample table. The first row is expanded and shows the population breakdown of all codon/amino acid breakdowns visually for each animal where that variant was detected and all days (where data is available). This allows for visual assessment of the magnitude of changes. Green parts of the bar represent the majority codon, yellow bars show synonymous variants and red ones represent non-synonymous variants. Hovering the mouse over any of the cells brings

Protein	Codon Start	Aminoacid	Pig 3473	Pig 3475	Pig 3480	Pig 4130	Pig 4131
HA	511	171	Pig 3473	Pig 3475	Pig 3480	Pig 4130	Pig 4131
HA	619	207	Pig 3473	Pig 3475	Pig 3480	Pig 4130	Pig 4131
	Day 2		R S	R S	S R		
	Day 3			R S	R S	S R R	R
	Day 4		R S R	R S	R S	S R R	R S
	Day 5		S R R	S R	R		R S
	Day 6		S R				
HA	622	208	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	190	64	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	199	67	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	430	144	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	508	170	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	598	200	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	715	239	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	766	256	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	895	299	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	991	331	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	997	333	Pig 3473	Pig 3475	Pig 3480		Pig 4131
HA	1243	415	Pig 3473	Pig 3475	Pig 3480	Pig 4130	Pig 4131
HA	1423	475	Pig 3473	Pig 3475	Pig 3480	Pig 4130	Pig 4131

Day: 4
Coverage: 3534
Reference Amino acid: **CGT/R** 65.65%
Synonymous Mutations:
AGG /R 1.95%
AGA /R 0.37%
CGA /R 0.11%
CGC /R 0.11%
CGG /R 0.03%
Non-Synonymous Mutations:
AGT /S 31.35%
CTT /L 0.14%
GGT /G 0.06%
TGT /C 0.06%
ACA /T 0.03%
ACT /T 0.03%
AGC /S 0.03%
ATG /M 0.03%
CAT /H 0.03%
CCT /P 0.03%

Figure 4.1.: Shows a screenshot of the interactive view of mutation sites picked up by `seqmutprobs` in the HA segment of the experimental infection study. The columns display the animals in the study and the rows (when expanded) show the days that were sampled for each animal with percentages of each codon present in the variation site. Green is the consensus codon, red is non-synonymous mutations, yellow is synonymous. Hovering the mouse over any of the cells will display detailed information about all the codon data available for this site.

up a fully detailed breakdown of all observed codons within that sample at the selected position (in the case of Figure 4.1 Pig 3480 day 4).

As an extension to the diversity plots (section 3.1) a whole-sample view (no filtered sites) is also available in the form of similar dynamic HTML tables for viewing either in nucleotide or codon spectrum. The underlying temporary files are used to construct the whole-study selective variant breakdown tables discussed earlier.

4.2. Next Generation Phylogenetics

When studying populations of rapidly mutating viruses we are interested in their phylogenetic relationship. Traditional phylogenetic trees rely on changes in the consensus sequences that occur over longer evolutional time frames. Although we have observed consensus changes within our population, there are far more subtle changes within those which become visible when using deep sequencing. We hypothesize that we can increase the resolution of phylogenetic trees by incorporating the fine distribution changes detected by deep sequencing.

Traditional phylogenetic trees are built using the Jukes-Cantor model [JC69], i.e. one assumes that mutation rates for all nucleotides are the same leading to a distance measure between two samples

$$d(s_1, s_2) = -\frac{3}{4} \ln \left(1 - \frac{4}{3} p \right) \tag{4.2.1}$$

with p being the proportion of sites that differ between the two samples. I.e. the Hamming distance [Ham50] between the two samples divided by the genome length.

For our NGS samples we have extended the notion of the Hamming distance metric to incorporate subtle differences in nucleotide distributions.

To achieve this we created a distance measure between two samples. Let $\mathbf{n}_{s,i}$ be a 4-dimensional vector containing the distributions of nucleotides at position i in sample s.

I.e. if $A_{s,i}, C_{s,i}, G_{s,i}, T_{s,i}$ are the counts of observed nucleotides at position i in sample s then

$$\mathbf{n}_{s,i} := \frac{1}{A_{s,i} + C_{s,i} + G_{s,i} + T_{s,i}} \begin{pmatrix} A_{s,i} \\ C_{s,i} \\ G_{s,i} \\ T_{s,i}. \end{pmatrix} \tag{4.2.2}$$

We then define the distance between two samples as the sum of all Euclidean norms of the distribution vectors differences:

$$d(s_1, s_2) := \sum_i ||\mathbf{n}_{s_1,i} - \mathbf{n}_{s_2,i}|| \tag{4.2.3}$$

If this metric is used on consensus sequences then it is equal to twice the Hamming distance between two samples.

In order to compensate for uncertainties in the nucleotide distributions at low coverage sites and between incomparable sites (missing data, significant differences in coverage) we have used `seqmutptobs` pairwise on all samples to filter for sites that have significant changes in their nucleotide distributions as a filter. This breaks the measure's metric quality (it no longer obeys the triangle law since different sites are used for each pair of samples), but it yields more conclusive results due to significantly less bias from uneven sampling.

Using this filtered distance measure (`seqmutprobs` filtered) we can construct a distance matrix for all samples in a study. In order to get a tree we then use hierarchical clustering [Joh67] on the resulting sample distance matrix. This produces heat maps and dendrograms that attempt to take the concept of phylogenetic trees to the genomic resolution of NGS.

Using this approach we observed that samples that are similar population-wise cluster together, whereas samples that display a significant change in viral population (even

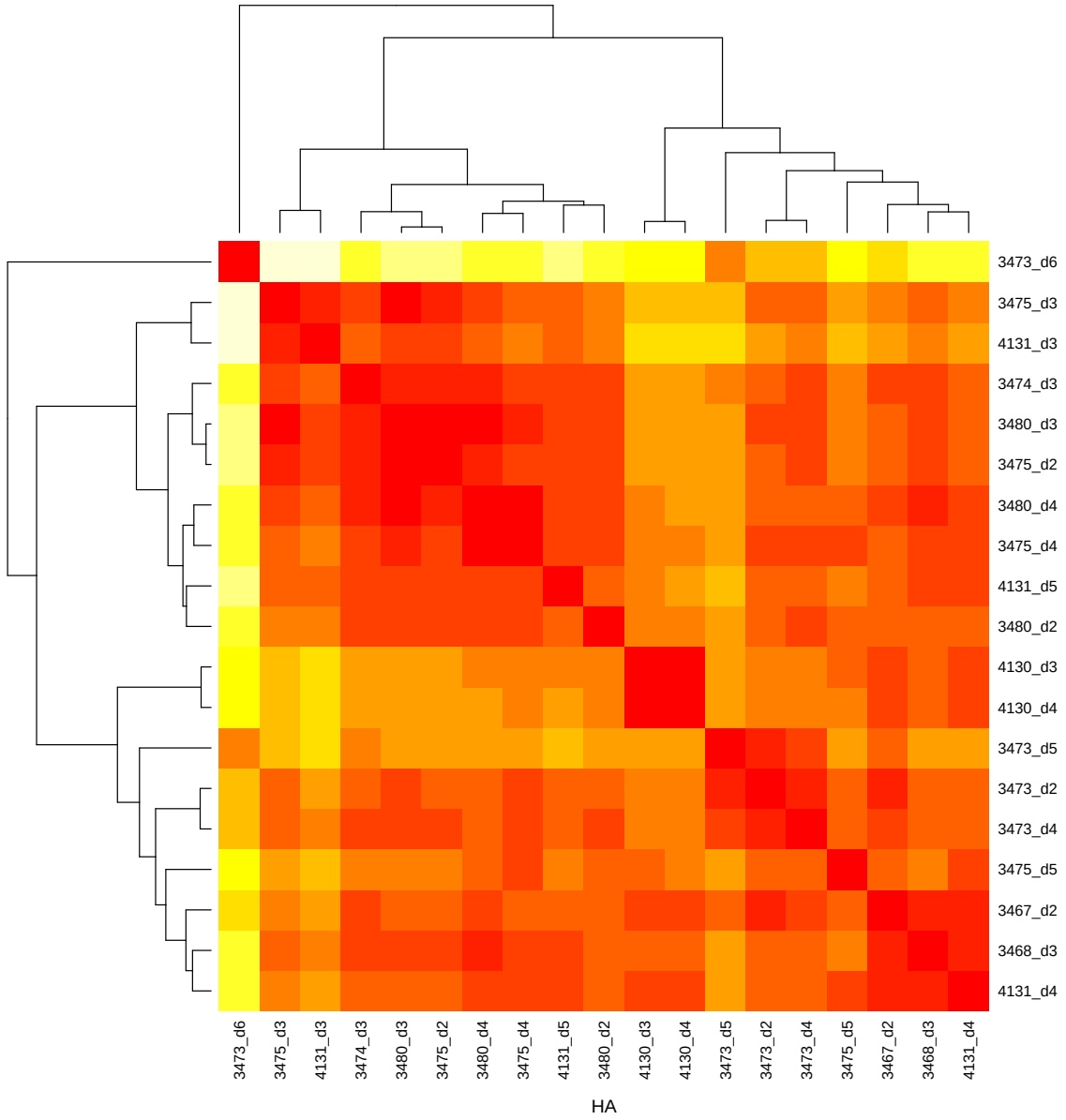


Figure 4.2.: Shows the results of NGS resolution phylogeny of the HA segment of the experimental infection study achieved via hierarchical clustering. The heatmap in the centre is representative of the calculated distance between samples. Red — no difference, white — large distribution changes. The dendrogram on the side is the direct result of the hierarchical clustering algorithm.

without showing changes in the consensus sequence) are separated. Figure 4.2 shows the hierarchical clustering of the HA segment of the experimental infection study.

As a comparison Figure 4.3 shows a phylogenetic tree of the consensus sequences of the HA segment of the experimental infection study. Some similarities, but also differences are readily visible. For example:

- Samples from Pig 4130 (all days) are in the same phylogenetic group and also have been clustered the closest because the distributions of the virus have remained virtually unchanged between them.
- Figure 4.2 shows that although samples 3473 day 5 and day 6 have the same consensus sequence they are distantly related in the dendogram. This is likely due to changes in the frequency and distribution of nucleotides at codon 207 (compare to Figure 4.1) as 87% of the reads for sample Pig 3473 day 6 code for a Serine whereas only 54% of the reads code for the same amino acid in sample Pig 3473 day 5. As a result, the within-host viral population of sample Pig 3473 day 5 relates better to the other samples in that cluster than to sample Pig 3473 day 6.

Overall, our results show that ultra deep sequencing data can reveal patterns of variation that would otherwise not be detected by classical phylogenetic methods based on the analysis of consensus sequences and thus can provide a better insight on the evolutionary dynamics of viruses over short time periods (i.e. days).

There are various other metrics for comparing multinomial samples that we could have tried like the likelihood ratio that is derived from comparing the goodness of fit of two models that can be approximated using Wilks's theorem [Wil38] or Pearson's chi-squared test [Pla83], but given time constraints this was not further pursued.

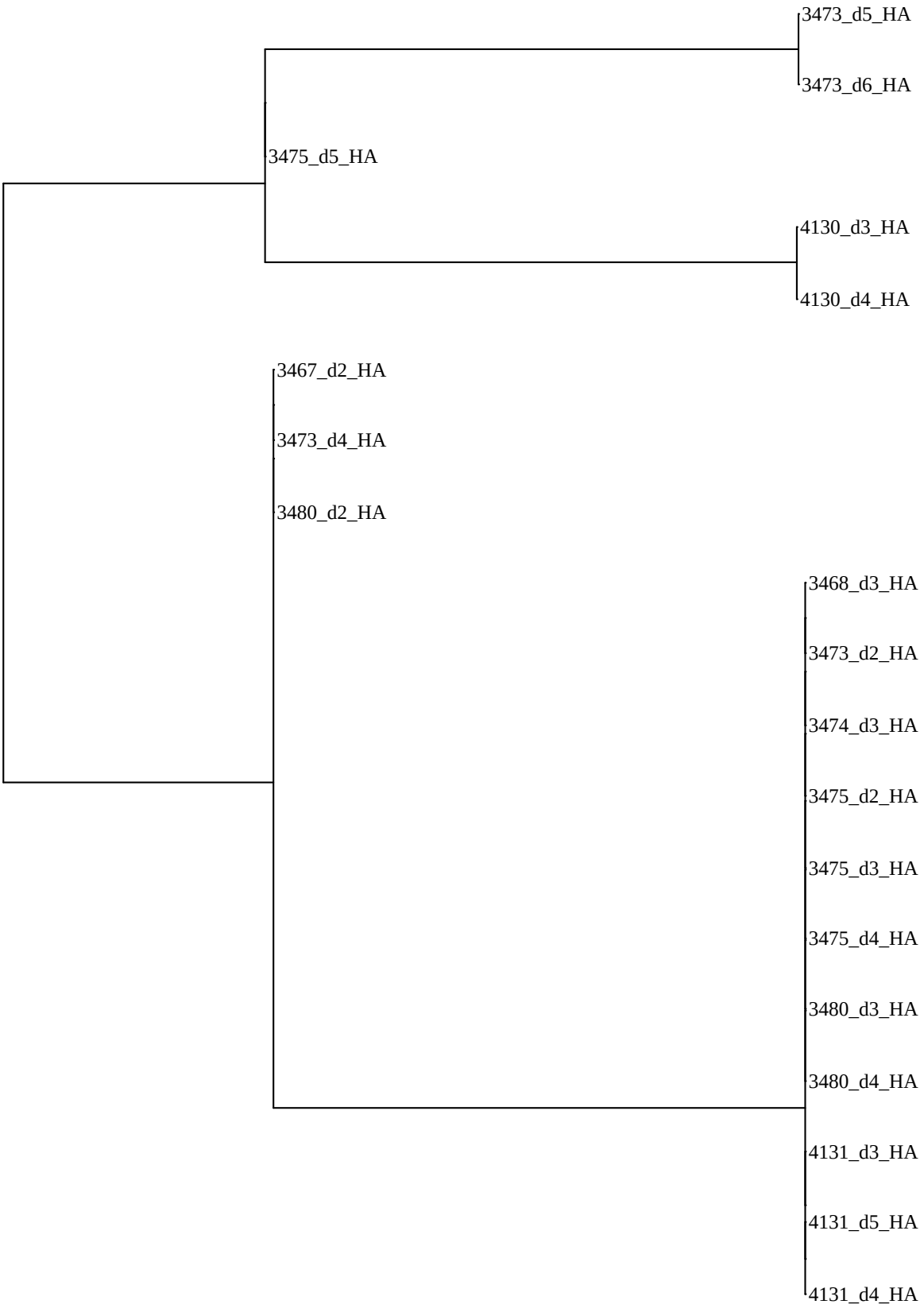


Figure 4.3.: Phylogenetic tree of the HA segment in the experimental infection study built based on consensus sequences (majority rule).

5. Discussion

With our pipeline we have introduced new methods, that specifically target visualisation of data on virus population diversity.

"Use a picture. It's worth a thousand words." - Arthur Brisbane, 1911

This amalgamation of methods has been successfully used to analyse the sequencing data from the three IAV studies mentioned in subsection 1.3.2 (publication manuscript is currently in final stages of development).

We have introduced two new concepts that allow a "Next Generation" view onto deep sequencing data "Beyond the Consensus" [Wri+11]:

- Nucleotide Entropy — a simple, well established method for measuring information content, allows us to visualise and filter out variants based on both number of variants and their magnitude — section 3.2,
- NGS phylogenies — a novel attempt on refining traditional consensus based phylogenetic trees for populations based on deep sequencing data — section 4.2.

These two new methods are only the tip of the iceberg of what kind of information we can extract from NGS deep sequencing data. But already these two simple concepts, when properly visualised, demonstrate the power of NGS data. With more research and development we will be able to bring even more well established consensus based methods (e.g. protein structures, haplotyping, integration of long reads) to the resolution of NGS.

Third generation sequencing technologies (i.e. PacBio [McC10]) can provide cost-effective long read sequencing of full viral genome segments. Our early experiments did not provide data of sufficient quality for our research, but integrating long read information into our tool kit would greatly benefit this kind of analysis. For example we could begin to do reliable haplotying on an entire population.

Along with new methods we have also begun work on web based visualisations of our virus population data (section 4.1). With more work we can create a fully interactive web service for visualising population variation and make it accessible to everyone to use on their own data as it is fully built on portable web technologies HTML and JavaScript.

We have achieved the goal of creating an easy to use analysis pipeline for biologists. Adding a Graphical User Interface (GUI) will be an additional step in this direction as right now the pipeline can only be run from the command line and requires the user to populate a SQLite table with meta data about the study (sample names, animal/day data).

One paper characterising the intra- and inter-host variation of the two Swine Flu studies is based on this pipeline and is currently in its late manuscript stages. Another research group working on the H3N2 IAV studies in Ferrets (Figure 1.3) is currently using and extending our pipeline.

Part II.

Sequence Assembly

6. Introduction

In Part I we were working with reference-based alignments of viral NGS data. In many cases the reference for a virus may not be available yet. In this chapter we will discuss several assembly methods and their performance with regards to viral sequence assembly. We will analyse both their theoretical capabilities and the actual performance of the implementations. We will then introduce StarK, our new assembler specifically targeted to overcome the shortcomings of the other methods with regard to highly diverse viral sequence data.

6.1. Motivation

Determining an organism's DNA sequence has always been a great challenge in biology. Although the first genome was sequenced back in 1977 [SNC77], none of the employed sequencing machines have ever been able to sequence a multiple kilo bases long genome as a single sequence (until the recent announcement of the "Oxford Nanopore" [Eis12]).

Numerous commercially available sequencers have emerged over the years:

- Sanger sequencing was the most widely used sequencing technology for roughly 25 years until the introduction of the Next Generation (NGS) Sequencing technologies below. In 1985 automated capillary sequencing machines were introduced that were capable of generating up to 1kb long reads [Smi+85]. Although being able to generate 500-1kb long reads at high accuracy the technology is slow and costly in comparison to newer NGS and is thus not being used as frequently any

more.

- Pyrosequencing based technology (Roche 454 [Mar+05]) determines the sequence by measuring the release of pyrophosphate molecules during DNA replication. This technology provides reads between 500 – 1kb, but are expensive to produce and often of insufficient quality or quantity to produce a full *de novo* assembly on their own. 454 reads are frequently used to enrich Illumina read datasets for resolving longer repeats.

- Sequencing by synthesis (Illumina [Ben+05; Ben06]) presented the first sequencing method for generating massive amounts of short read at low cost. While initially used for resequencing or alignments the low cost associated with the technology started an influx of *de novo* assemblers attempting to use these reads.

- Sequencing by ligation (SOLiD [Mar08]) is another technology capable of producing short low-cost reads, of comparable length and quality to Illumina at the time. This technology did not see much uptake due to being unable to keep up with longer read lengths of the Illumina platform and lack of paired end reads.

- Single molecule real time sequencing (PacBio [Eid+09]) promises to deliver long read lengths (2-5kb), but does so at a significant hit to the quality of the reads which of their own cannot be reliably used for *de novo* assembly, but have been successfully used to enrich Illumina assemblies as a replacement for previously used 454 reads [Utt+14].

- At time of writing this thesis a new technology called "Oxford Nanopore" [Eis12] had been announced promising read lengths of tens of kilo bases, but was not yet publicly available.

Data sets obtained by current methodology have never been free of errors. This gave rise to the need of computationally assembling larger genomes from small sequenced fragments. Overlap Layout Consensus (OLC) assembly was one of the first developed methods, which worked well with the first generation capillary sequencing [SNC77], that produced genome fragments of roughly ~ 700 bases.

The twenty first century then witnessed a new development in sequencing technology [Mar+05; Ben+05; Ben06] as the so called Next Generation Sequencing (NGS) technology was first made available to the public. NGS was capable of providing scientists with affordable deep sequencing with much higher yields at the compromise of shorter reads. Although initially designed for resequencing the technology soon began to be used as a viable option for *de novo* sequencing as new algorithms for dealing with the shorter fragments became available.

Initially *de Brujin* graph based approaches [IW95; PTW01; ZB08; Sim+09] quickly gained popularity as they produced reasonably good assemblies on low-repeat genomes based on just short reads, but were suffering from serious problems when trying to deal with either repeat-rich genomes or diverse (virus or metagenomics) datasets "out of the box" (i.e. without tweaking the parameters and/or preprocessing the input data). A significant amount of manual work in adjusting the assembler and the input data is required to assemble most genomes using those assemblers. Until today numerous new methods and hybrid approaches [Bra+13] have been developed.

We have assessed the capabilities of the major theoretical assembly models and several representative algorithms on their capability to assemble highly diverse viral sequencing data (see Table 6.5) and have identified their strengths and weaknesses. None of the tested algorithms were capable of assembling our data to our satisfaction (compare to Table 6.5).

In order to address some of the shortcomings of current methods we have developed StarK – a new data structure and algorithm specifically designed to overcome the limitations of single-dimensional *de Brujin* graph based methods for assembly of viral data. We have implemented a prototype of the StarK algorithm in chapter 9 and evaluated its performance in comparison to other algorithms in section 6.6. StarK outperformed all other approaches in both quality of the assembly and speed (see Table 6.5).

6.2. Genome assembly theory

This chapter will use several formal language theory constructs to describe algorithms and data structures. In addition to a set of definitions from the formal language theory field we also introduce new notation which is described in appendix A.

In addition we will be using the following terminology:

sample: Collection of reads generated by a genome sequencer.

target: Desired genome which is to be assembled. Unknown at assembly time.

coverage: the number of reads that have originated (were sequenced) from a given base (or sub-sequence) in the target genome.

contig: A set of overlapping reads that represent a possible sequence from the target genome.

graph: Formal directed graph consisting of vertices and edges between them. Formally written as $G = (V, E)$ where V is the set of vertices and E is the set of edges. We will often refer to a vertex by variables $v, v_i, \ldots$ and to edges by tuples of vertices e.g. (v_1, v_2).

word: A formal language word. Often referred to by symbols w, v, x, y, z, s and used to identify sequences, k-mers and contigs.

Σ^*, Σ^+: Sets of all finite words made of an arbitrary finite alphabet Σ including and excluding the empty word λ respectively. Although our practical application is limited to the genomic alphabet, most of the theory is applicable to arbitrary alphabets.

Γ, Γ^+: The genomic alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and the set of all words made of it (i.e. all possible genomic sequences) respectively.

$\mathcal{R}$: This symbol always represents a finite set of reads $\mathcal{R} \subseteq \Sigma^+, |\mathcal{R}| < \infty$.

$\iota(w, x)$: The set of all positions where the sequence x appears in w as a full subsequence. $\iota(w, x) \neq \emptyset$ is often used to express "x appears in w as a subsequence". For a full definition please refer to definitions A.9 and A.10.

6.2.1. Coverage

When sequencing a genome $g \in \Gamma^+$ we require each base to be covered by at least one fragment in order to be able to reconstruct the whole. We consider a base in the target genome to be covered (by a read) if at least one read has been sequenced that originated from that base. We call the number of those reads that cover a given base the **coverage** of that base. Similarly for sub-sequences of the target genome.

Let $g \in \Gamma^+$ be a genome and $l \in \mathbb{N}$ the length of the fragments that are being sequenced, $|g| >> l$. The sequencing technology will select a number of (not necessarily unique) fragments for sequencing from the genome in accordance to the technology specification and sample preparation. Assuming that the distribution of selected fragments along the genome g is uniform we can estimate that the probability of covering a base at a fixed position $p \in |g|$ with one fragment is

$$\frac{l}{|g|}. \tag{6.2.1}$$

In which case the probability of not covering a base with $n \in \mathbb{N}$ fragments is

$$\left(1 - \frac{l}{|g|}\right)^n \quad \text{[CC76]}. \tag{6.2.2}$$

This expression can be simplified to

$$\left(1 - \frac{l}{|g|}\right)^n \approx \exp\left(-\frac{nl}{|g|}\right). \tag{6.2.3}$$

Where $\frac{nl}{|g|}$ is called the *redundancy*. The actual measured redundancy is then called *coverage*. This type of oversampling is required in order to maximise the probability of covering each base within the target genome.

In practice two additional factors are important:

- Genome sequencers do not actually sample reads from all positions within the genome uniformly due to unpredictable bias in the PCR [Mul+87] primer chemistry [Koz+09],
- The reads generated are not error-free.

Both of these are limitations of today's sequencing technology. Genome assembly algorithms require oversampling in order to distinguish sampling errors from target sequences. In section 7.3 we will be discussing how this affects k-mer based assemblies.

6.3. Overlap Layout Consensus (OLC) Assembly

One of the first methods for *de novo* genome assembly was the OLC assembly method [NP13]. It is based on a simple principle: compare all reads pairwise - then find the best contigs. In practice this is often implemented in the following way:

Overlap: Build a graph $G = (\mathcal{R}, E)$ where the node set $\mathcal{R} \subseteq \Sigma^+$ a finite read set and the edges are the overlaps

$$(v, w) \in E \Leftrightarrow \exists x \in \Sigma^+, |x| > \text{cutoff}, \iota(v, x) \neq \emptyset \neq \iota(w, x). \tag{6.3.1}$$

An edge exists in G between two reads v, w if and only if they share a common subsequence of a minimal length (pre-set overlap length cutoff or another measure of similarity) [Mye+00]. Figure 6.2 shows an OLC graph during the overlap stage.

Although initially this step was limited to overlaps between two reads, more recent assemblers use full alignments between reads instead [Bat+02]. This variant is sometimes referred to as an Align Layout Consensus assembler. We will continue to refer to this group of algorithms as OLC assemblers.

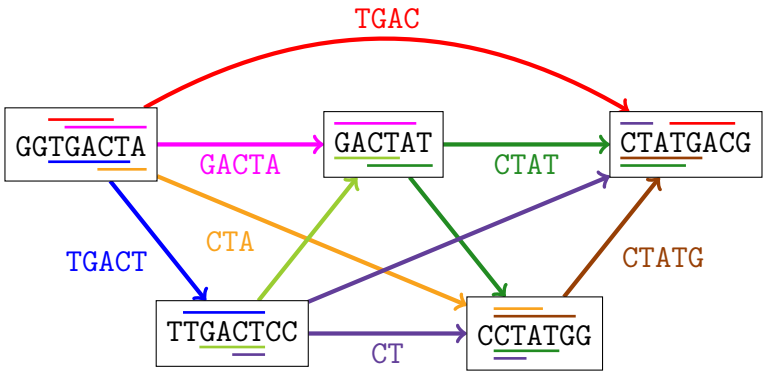


Figure 6.2.: Full OLC graph during the overlap phase. This graph contains five reads which are intended to have been originated from the sequence GGTGACTATGACG. Notice that the repeat TGAC is also the longest overlap between the two reads GGTGACTA and CTATGACG.

Layout: If all reads originated from the target sequence and were error-free, then the ideal contig derived from this graph is a hamiltonian path through G. Unfortunately, it is both computationally unfeasible to compute a hamiltonian path and the assumptions do not apply to real sequences that are generated by sequencers. As a result the algorithms generally try to combine parts of the graph into contigs that can be assembled into continuous sequences with no inner branches. The resulting contigs are an approximation of fragments of the hamiltonian path through the subgraph that contains no erronous nodes.

Consensus: The branchless subgraphs are then compacted into consensus sequences by taking the consensus base at each position.

Although theoretically this method can yield the highest quality results, its computational complexity during the overlap step ($O(|\mathcal{R}|^2)$) limits its use to small data sets.

6.4. *De Brujin* Graph Assembly

Capillary sequencing generates few high-accuracy long (∼700bp) reads. Roughly 6000 reads can be generated per day. Traditionally they can be assembled by pairwise overlapping. NGS technologies generate shorter deep sequencing reads (around 150bp today), but in far greater number. Approximately 5 million reads can be generated per day with NGS sequencing [Qua+12].

One of the biggest projects that used capillary sequencing was the human genome project [Lan+01].

Genome assembly by pairwise overlaps (e.g. OLC) is only feasible with few long reads, typically capillary sequencing (which OLC was designed for) produces several thousand reads. Pairwise overlaps between the large number of NGS reads (typically millions) are computationally not economic. The computational time required for such a number of pairwise comparisons (pairwise comparisons have time complexity of $O(n^2)$) would exceed years on current hardware (e.g. 8 cores at 2.5GHz).

This problem gave rise to an influx of new algorithms for short read processing [IW95; Ear+11; Bra+13; PTW01]. One method in particular became popular quickly: *de Brujin* graph assembly. Two of the most popular assemblers that use *de Brujin* graphs are Velvet [ZB08] and ABySS [Sim+09].

We will be extensively discussing the algorithms used in *de Brujin* graph based assemblers in chapter 7.

6.5. String Graph Assembly

In concept String Graph Assembly is related to both OLC and *de Brujin* graph assembly methods.

A string graph is always derived from a string or sequence. The vertices in the graph

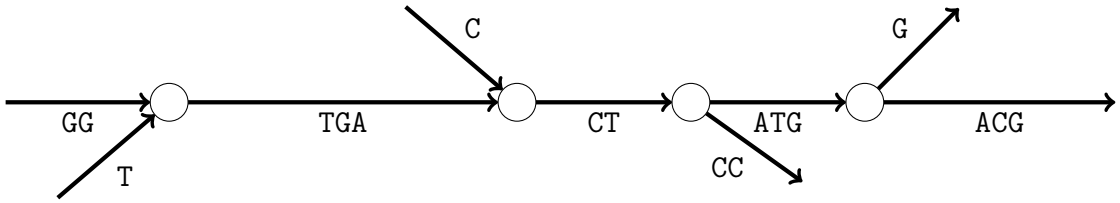


Figure 6.3.: String graph based on the same reads used for Figure 6.2 constructed with overlaps of at least length 3 (with the exception of `CT`, which is displayed to allow the erronous part of the `TTGACTCC` read to be displayed). The rest edges to the top and right correspond to errors in the reads that did not overlap with the main ones. The target sequence `GGTGACTATGACG` is displayed as the central path from left to right. The repeat around `TGAC` is fully resolved by string graph using the given reads.

are positions within the source sequence and the edges are strings — the overlaps within the sequence of an arbitrary minimum length.

Definition 6.5.1. *Let Σ be an alphabet, $s \in \Sigma^+$ a string, $k \in \mathbb{N}$. Then the string graph of s is defined by a graph $G = (V, E)$ and a function $f : E \to \Sigma^{k+}$ such that there exists an Eulerian path (a path that visits all edges once, Definition A.13) $p = (v_1, \ldots, v_n)$ which edges form the string $s = f(v_1, v_2) \ldots f(v_{n-1}, v_n)$.*

In string graph genome assembly the string graph is constructed from the reads and the target sequence is a path within it.

A string graph is constructed by pairwise comparing all reads and creating edges which correspond to an overlap of at least k [Mye05].

In order to reduce the complexity of the resulting string graph transitive edges are compacted such that if edges $(v_1, v_2), (v_2, v_3), (v_1, v_3)$ exist then only the transitive edge (v_1, v_3) is kept. In case v_2 has no other edges connected to it then (v_1, v_3) may also be created as a replacement for $(v_1, v_2), (v_2, v_3)$ [Mye05].

Originally this method was intended to provide similar expressibility to *de Brujin* graph

based assembly with lower memory footprint (as a far lesser amount of sequences are stored in comparison to k-mers in a *de Brujin* graph) at the cost of computational time to calculate the overlaps (initially pairwise alignment, i.e. in $O(n^2)$ time complexity). This fact made the method computationally unfeasible for large sample sizes.

In 2010 Jared T. Simpson and Richard Durbin published a new method of constructing the string graph in linear time using the FM-Index [SD10]. To this day, their assembler SGA remains slow in comparison to modern *de Brujin* graph assemblers due to constant factors in the computation of the FM-Index [SD12].

6.6. Comparison of the above methods

In the following subsections we have summarised some of the theoretical and measured qualities of the different assembly methods. In section 6.7 we will elaborate further on measured run times.

6.6.1. Expressitivity/information loss

The various approaches all work with one or another combined/compressed data structure derived from the reads themselves. Those data structures are intended to facilitate the construction of contigs. This often loses information that may be required in certain circumstances, which may have been present in the reads, but was lost during the conversion.

OLC: The individual reads are preserved as nodes in the overlap graph. Theoretically no information is lost, though often only the best matching overlap between two reads is considered. The remaining is up to the implementation of the Layout and Consensus steps. Repeats within the genome are distinguishable up to the read length.

***de Brujin* graph:** Individual read information is lost and cannot be recovered. Any overlap information longer than the *de Brujin* graph dimension k is also lost. Coverage of individual k-mers is preserved in most implementations, but the connection to individual reads is lost. Repeat resolution of repeats longer than the *de Brujin* graph dimension requires additional algorithms that are not part of the data structure, e.g. paired-end read mapping on intermediate contigs or assisted contig assembly using long reads obtained from another source.

String graph: Individual read information is lost, but can only be probabilistically guessed in some cases. Overlaps up to the read length can be preserved. Repeats up to the read length can be resolved where coverage permits. Coverage information of the reads is lost, but overlap coverage is kept in some implementations.

StarK: It is up to the implementation to preserve or remove individual read information. For memory performance reasons it is not preserved in our implementation. Selective preservation of individual reads is also possible. Overlaps are kept up to the underlying repeat length plus one nucleotide. Palindromes are preserved up to full read length.

6.6.2. Time

Here we mention the theoretical asymptotic runtime of the analysed algorithms during the graph construction step. Real runtime complexity depends on the concrete underlying implementation. Let n be the number of reads. Run times for the actual assembly steps vary between implementations and can be roughly estimated by $O(m \log(m))$ where m is the number of nodes in the resulting graph. This estimation is based on the assumption that many implementations use either sorting on the nodes or a *divide and conquer* approach for contig assembly.

Algorithm	Time	Comment
OLC	$O(n^2)$	Overlap stage requires pairwise comparison of all input reads.
de Brujin graph	$O(n)$	Reads are processed once generating up to read length k-mers. Hash maps (the most common underlying storage) permit constant time access.
String graph	$O(n)$	The introduction of the "Efficient construction of String Graph using the with FM-Index" permits building the string graph in linear time [SD10] .
StarK	$O(n)$	Like with the *de Brujin* graph, reads are processed once generating (read length)^2 k-mers. The design of the data structure permits constant time access.

Table 6.4.: Algorithmic complexity for constructing the data structure for the underlying assembly methods. The Big-O notation is used to express asymptotic runtimes, please refer to Definition A.11.

6.7. Assessment of the tools in respect to viral data

We ran several implementations of the theoretical algorithms mentioned earlier on sample high-population viral data to assess their ability to assemble virus populations. All assemblers that succeeded were ran with 8 and 64 cores respectively. 8 core (one socket) and 64 (8 sockets) core benchmarks were chosen to benchmark assembler capability to utilise more than one socket. Multiple socket deployments (although logically transparent to the program) require additional optimisation to compensate for a slow multi-Central Processing Unit (CPU) interconnect. Times were measured using the Linux utility `time`. The best assembly of the two is shown in Table 6.5.

Assembler	Assembly Size	Number of Contigs	Longest Contig	N50	N90	time	
						8 cores	64 cores
PRICE	199 018	695	5711	450	118	41h	n/a
Velvet	168 273	1304	5385	125	125	7m	73m
Celera	0	0	0	0	0	23m	n/a
SGA	5385	1	5385	n/a	n/a	2h	n/a
IDBA-UD	9949	4	5465	5465	1163	18m	18m
SPAdes	1 213 926	16 448	5441	71	1163	99m	107m
StarK	31 244	37	2331	913	546	26m	12m
Reference	14 720	8	2292	2221	1027	n/a	

Table 6.5.: Comparison of the assembly statistics of various short read assemblers ran on reads from Flu virus H3N2 sample Ferret 52 day 2. The reference genome was obtained through capillary sequencing.

We tried the following assemblers:

- PRICE [RBD13]: *de Brujin*-graph based assembler. PRICE was initially designed to be a virus assembler, but was subsequently made into a general metagenomics assembler tool kit. Was run with 1000 reads as seed sequences with the

options `-nc 30 -mol 30 -tol 20 -mpi 80 -dbmax 100 -dbk 63` and `-a 8/64` respectively. The run on 64 cores ran for 22 days without finishing before we aborted it.

- Velvet [ZB08]: *de Brujin*-graph based assembler. Was run with hash length of 64, no additional options.
- Celera Assembler [Mye+00]: OLC based assembler was ran with a modified options file RunCA Examples - Illumina + 454 Large Genome. After 23 minutes no contigs were produced. This assembler is not intended to be used with short reads only.
- SGA [SD10]: String Graph Assembler. SGA was run with correction k-mer length of 41, minimum overlap of 63 and minimum number of pairs to link two contigs of 10. The assembly produced just one contig. Neither the contig, nor the viral reference sequence aligned against each other. When we ran BLAST [Alt+90] on the contig the top hits were Acinetobacter baumannii , Streptococcus, Coliphage phi and other bacteria all with an E-value of 0. This leads us to believe that a contaminant was assembled instead of our real target.
- IDBA-UD [Qua+12]: IDBA is an assembler that can utilise multiple k-mer lengths of *de Brujin* graphs. It generates multiple assembles based on a range of k-mer lengths iteratively improving on loop resolution by increasing the k-mer length in each step. Given the extreme variation in our virus sequencing sample the assembler has created long contig that contains a concatenation of four of the segments of the target genome. Pieces of the remaining genome are present in multiple copies in the longer contigs.
- SPAdes [Ban+12]: Is a *de Brujin* graph assembler that uses multiple dimensions of *de Brujin* graph. Similarly to IDBA it starts with a smaller k-mer length and slowly works its way up to longer k-mers in order to resolve repeat sections. This assembler similarly to IDBS-UD was not designed to handle the high variation in the test sample. The largest contig is a concatenation of four of the target

segments. The remaining pieces of the target genome are scattered in multiple copies throughout the longer contigs.

- StarK: Our new multi-dimensional *de Brujin* graph assembler. Was run with default options. Out of the 8 target segments 6 were assembled completely. The other two were in two parts each. The other partial contigs are incomplete variant contigs.

PRICE, celera, SGA and StarK were ran on an SGI UV, Intel® Xeon® E5-4600 (8 cores, Hyper-threading off) on Red Hat® Enterprise Linux® on either one or 8 CPU's in a full shared memory environment.

IDBA-UD and SPAdes were run on a SGI UV, Intel® Xeon® E5-4650L (8 cores, Hyper-threading off) on Red Hat® Enterprise Linux® on either one or 8 CPU's in a full shared memory environment.

7. *De Brujin* Graph assembly at work

7.1. Introduction

In this chapter we introduce the basic concepts behind *de Brujin* graph assembly theory in a more formal way. We also conduct a thorough analysis of coverage patterns that are observed when reads are divided into k-mers. This will then be used to explain the concepts behind StarK (our new assembler).

Please note that the theoretical concepts introduced here are not specific to the genomic alphabet (`A`, `C`, `G`, `T`) and can be applied to an arbitrary alphabet.

7.2. Formal *de Brujin* Graph

De Brujin graphs form a family of graphs with two parameters:

- A finite alphabet, which forms its k-mers Σ,
- The *de Brujin* graph dimension $k \in \mathbb{N}$.

Formally a *de Brujin* graph is the graph consisting of all words from the alphabet Σ of length k with an edge between two vertices if one can remove a letter from the beginning of the word represented by the first vertex and append another letter in order to obtain the word represented by the second vertex.

Definition 7.2.1. *Let Σ be an alphabet, $k \in \mathbb{N}$, then a graph $G = (\Sigma^k, E)$ is called a k-dimensional de Brujin graph if and only if*

$$E = \{(w, `wa) | w \in \Sigma^*, a \in \Sigma\}. \tag{7.2.1}$$

In the context of genome assembly any subgraph of a *de Brujin* graph is often referred to as a *de Brujin* graph. In this dissertation we will follow the same reference.

Also the *de Brujin* graph dimension k is often called the **hash length** (due to *de Brujin* graphs often being represented as hash maps in memory) or simply the **k-mer length**.

Example 7.2.2. *Consider the genomic alphabet $\Gamma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and the read* `TGAC`*. The subgraph of the two-dimensional de Brujin graph containing* `TGAC` *is*

$$G = (\{\mathtt{TG}, \mathtt{GA}, \mathtt{AC}\}, \{(\mathtt{TG}, \mathtt{GA}), (\mathtt{GA}, \mathtt{AC})\}). \tag{7.2.2}$$

Figure 7.1.: *de Brujin* graph containing the two-mers of the sequence `TGAC`.

Any path through a *de Brujin* graph can be viewed as a word in $\Sigma^{\geq k}$.

$$\left(\Sigma^k\right)^n \ni (w_i) = (w_1, \ldots, w_n) \mapsto w_{1,1} w_{2,1} \ldots w_{n-1,1} w_n \in \Sigma^{n+k-1} \tag{7.2.3}$$

$$(\mathtt{TG}, \mathtt{GA}, \mathtt{AC}) \mapsto \mathtt{TGAC} \tag{7.2.4}$$

De Brujin graph based assemblers use these paths in order to create contigs. The art of each assembler implementation lies in determining which paths belong to your target sequence and which are spurious connections that are created by errors in the sampled reads.

De Brujin graph assemblers can not resolve repeats within the target sequence that are longer than the graph dimension k based on the graph structure alone.

If a sequence x of length greater or equal to k appears more then once within the target genome, then assembly paths through a k-dimensional *de Brujin* graph have to reuse the same k-mers that assemble x more then once. This requires the graph to have a branch around x (i.e. more than one possible continuation of a path going through x). Unless additional information is provided to the assembler it is impossible to tell within a k-dimensional graph which branches belong together. More about this topic is discussed in example 8.1.2.

Traditionally one of the methods to filter which paths lead to contigs relies on filtering k-mers by their frequency of appearance.

Let $n, k \in \mathbb{N}$, $\mathcal{R} \subset \Sigma^{\leq n}, |\mathcal{R}| < \infty$ a finite set of reads. We assign a weighting function

$$c_{\mathcal{R}} : \Sigma^+ \to \mathbb{N} : w \mapsto \sum_{r \in \mathcal{R}} |\iota(r, w)| \tag{7.2.5}$$

commonly referred to as the k-mer coverage — the frequency of appearance of a k-mer as a sub-word within the read set.

Example 7.2.3. *Consider the same example as in Definition 7.2.2. Having just one read* $\{\texttt{TGAC}\}$ *the values for* $c_{\{\texttt{TGAC}\}}$ *on* 2*-mers are:*

$$c_{\{\texttt{TGAC}\}}(\texttt{TG}) = 1 \qquad c_{\{\texttt{TGAC}\}}(\texttt{GA}) = 1 \tag{7.2.6}$$

$$c_{\{\texttt{TGAC}\}}(\texttt{AC}) = 1 \qquad c_{\{\texttt{TGAC}\}}(w) = 0 \quad \textit{for all other 2-mers} \tag{7.2.7}$$

The assumption for using this method of filtering is that k-mers that belong to your target will have been sequenced in higher frequency than noise. See subsections 6.2.1 and 7.3 for details.

7.3. Observed coverage patterns

In order to determine the quality of a *de Brujin* graph we often build coverage histograms. We count the frequency of occurrence of each coverage value that nodes within a *de Brujin* graph receive after all reads have been processed.

Figure 7.2 shows an artificial example of a coverage plot of a good sample. The k-mers that belong to the target sequence that we want to assemble, clearly appear in

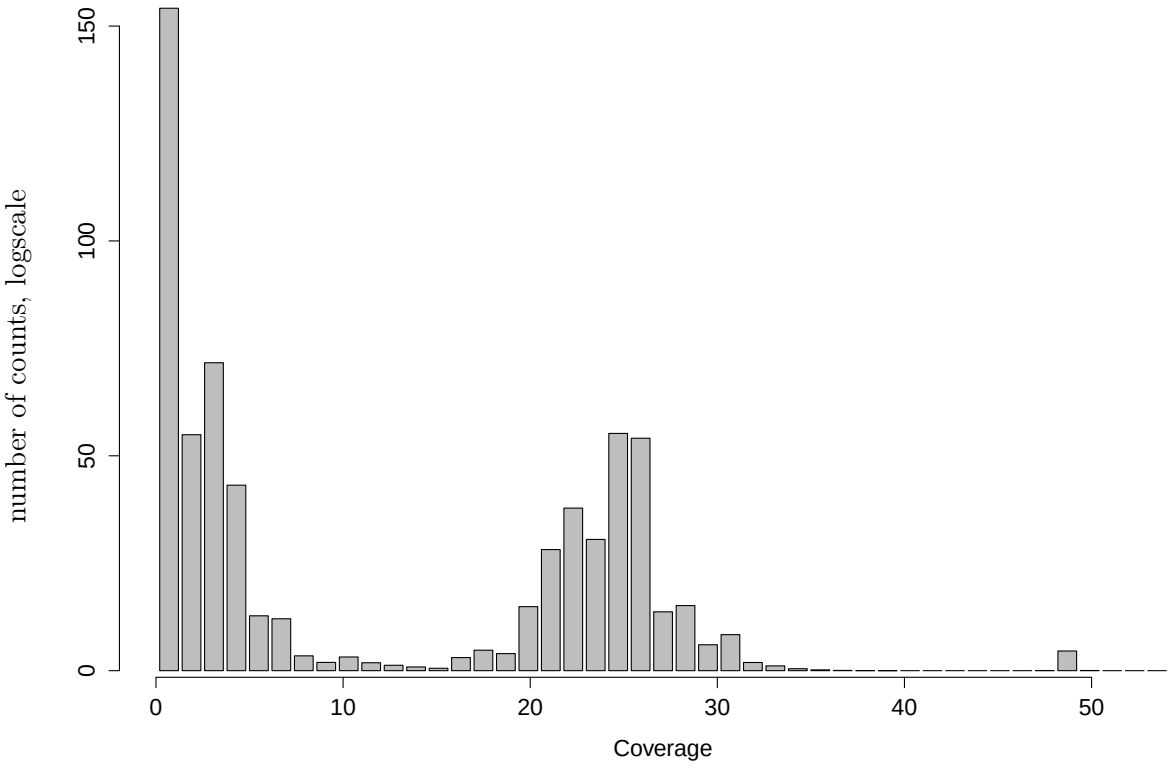


Figure 7.2.: Artificial example coverage plot. The average sampling of the target sequence is roughly 24× as indicated by the highest peak in the center of the plot. The k-mers with low frequency of appearance are likely to be erroneous or artefacts of the sequencing process and can be safely discarded.

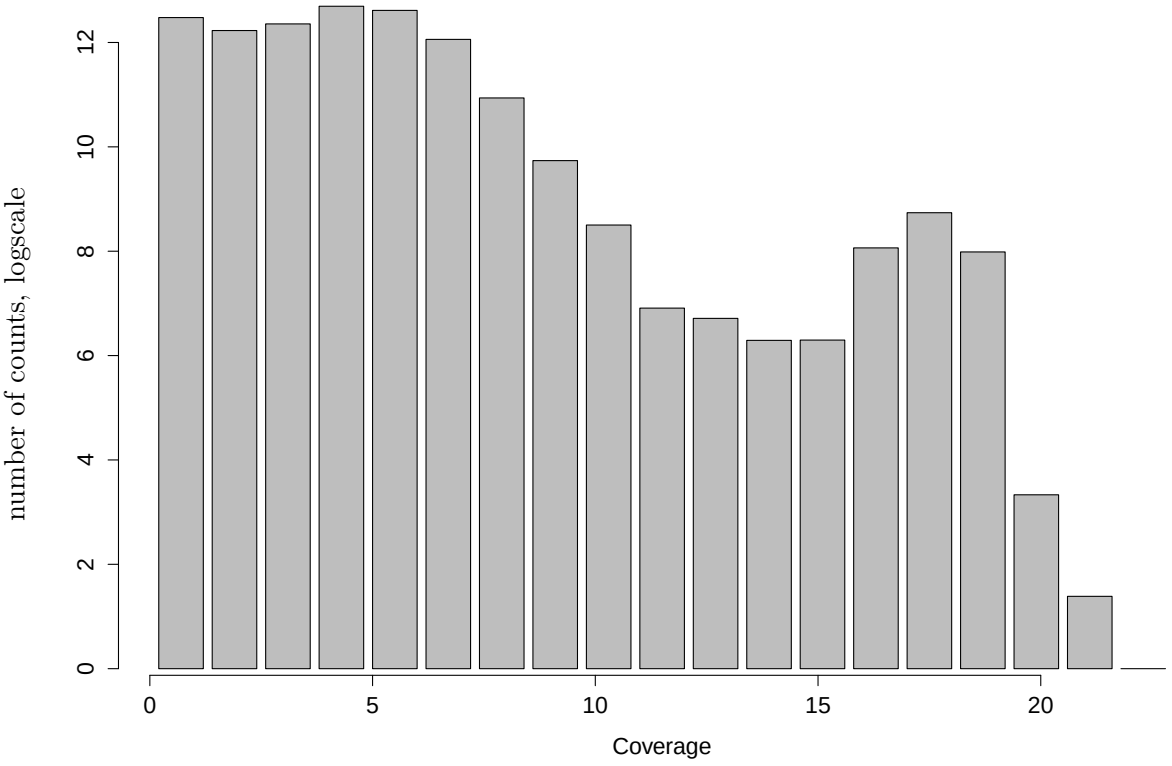


Figure 7.3.: Coverage plot of influenza virus sample from Ferret 54 day 2 at k-mer length 37.

far higher frequency than the erroneous k-mers that appear in frequencies of $1 - 10$. Most assemblers would remove all k-mers with frequency of appearance below 10 and assemble the remaining ones into contigs.

This approach is often impractical as real coverage plots often more resemble Figure 7.3. This kind of coverage distribution makes it very difficult to distinguish between k-mers that belong to the target sequence, k-mers that belong to variants and k-mers that belong to errors.

In order to attempt to explain these observed plots we will attempt to model them using stochastic distributions. We use the following assumptions and simplifications in order to achieve this.

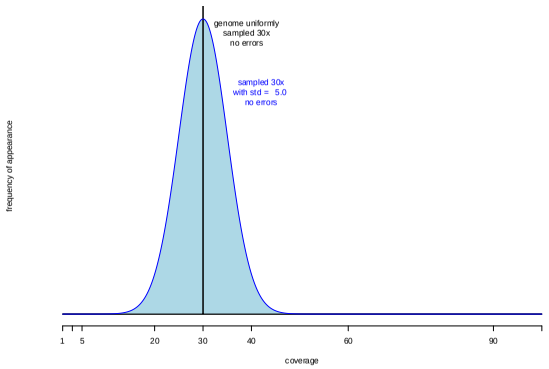
- If we were to sample k-mers from a genome uniformly with no errors, then we would expect to see only one peak in the coverage plot at exactly the sampling rate like shown in Figure 7.4a.
- We have to expect uneven sampling, so we will model this using a poisson distribution with a certain standard deviation (Figure 7.4b). Let s be the sampling rate and c the coverage. Then the frequency of appearance for a given coverage number can be modelled as

$$\frac{s^c \cdot e^{-s}}{c!}. \tag{7.3.1}$$

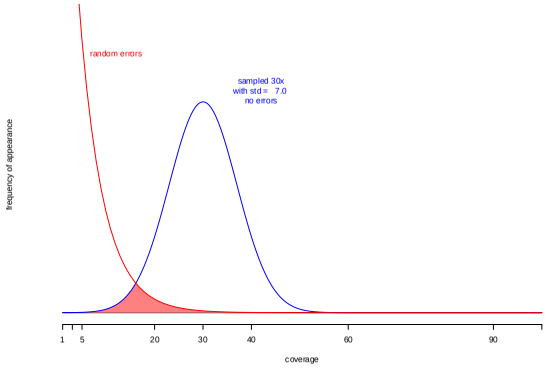
- In addition to that we expect the sequencing platform to generate spurious errors which will result in new k-mers that appear only very few times. Model these using a Boltzmann distribution (σ is the spread, Figure 7.4b):

$$\exp\left(-\frac{c}{\sigma_2}\right). \tag{7.3.2}$$

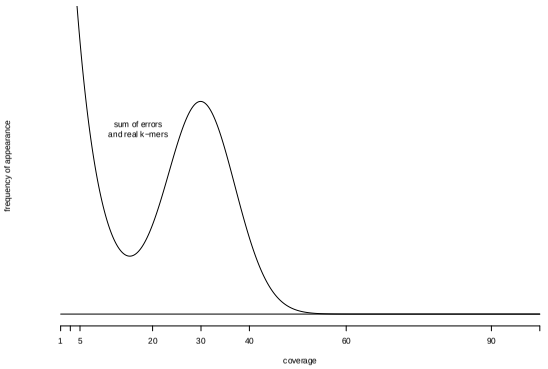
- Combining the terms (7.3.1) and (7.3.2) we receive a coverage distribution similar to the observed one as shown in Figure 7.4c.
- In addition to that there are often repeats within a genome. As such we expect



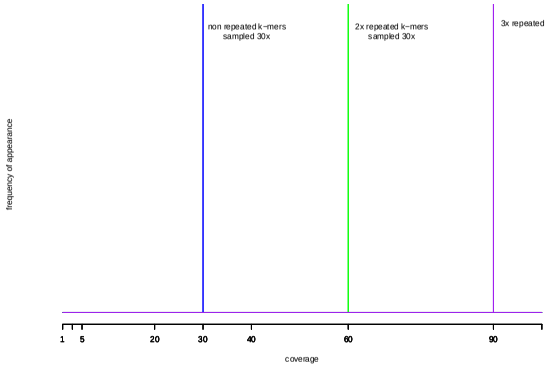
(a) No errors, sampling with variance



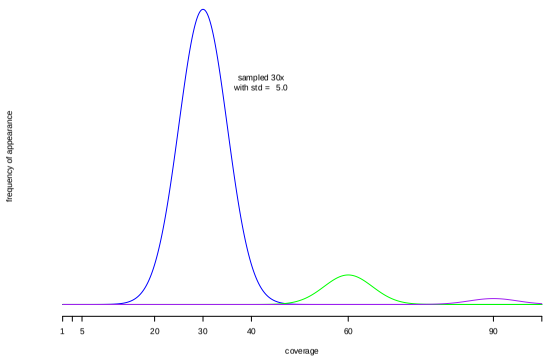
(b) Sampling with variance + separate error curve



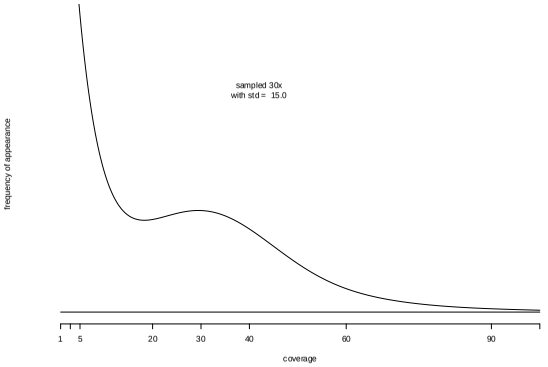
(c) Sum of errors + target as observed



(d) No errors, uniform sampling with $2\times, 3\times$ repeats



(e) No errors, uneven sampling with $2\times, 3\times$ repeats



(f) Sum of errors + uneven sampling with $2\times, 3\times$ repeats on high variance as observed

Figure 7.4.: Shows the different models for the origin of coverage plots.

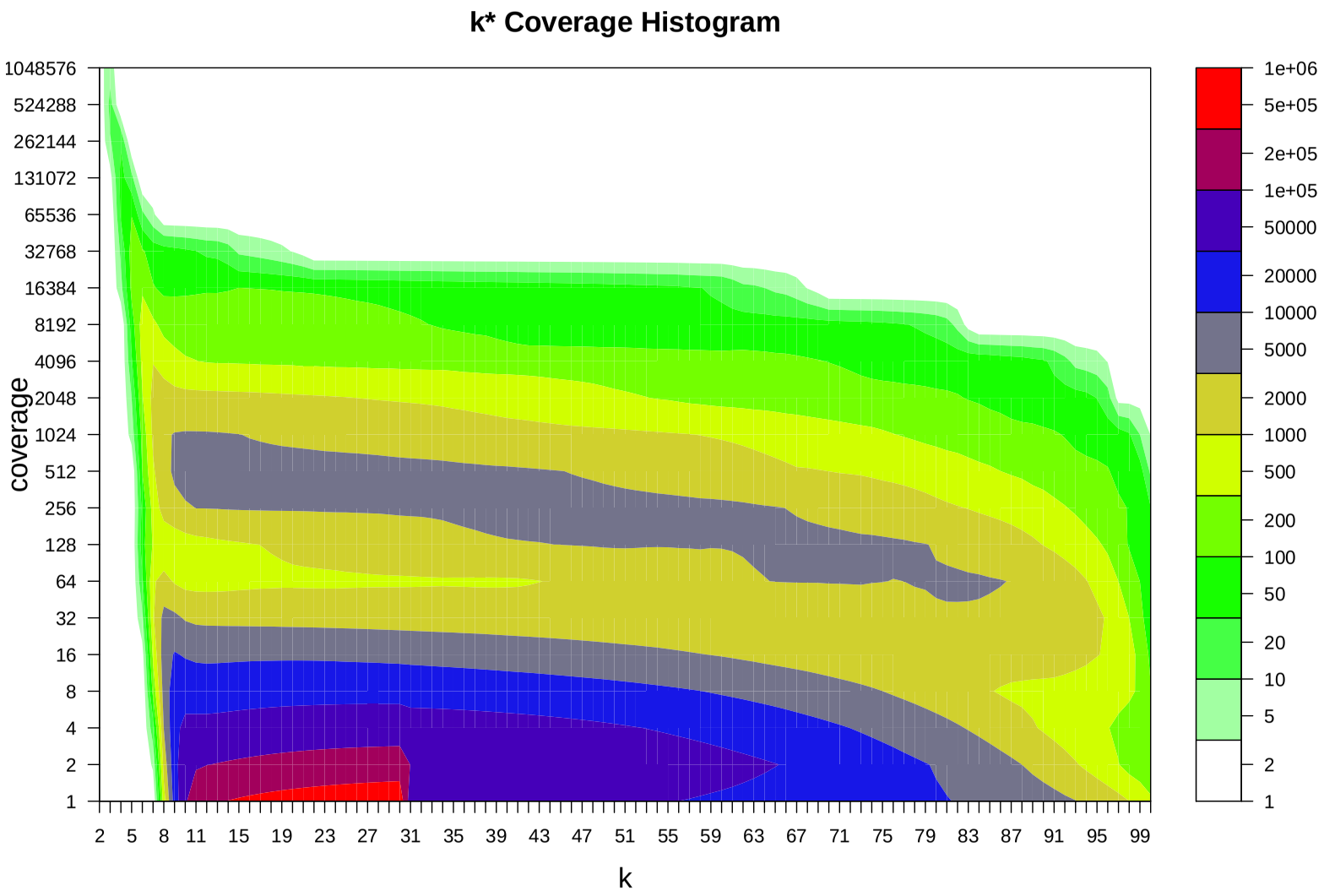

k* Coverage Histogram
coverage
1048576
524288
262144
131072
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2 5 8 11 15 19 23 27 31 35 39 43 47 51 55 59 63 67 71 75 79 83 87 91 95 99
k
1e+06
5e+05
2e+05
1e+05
50000
20000
10000
5000
2000
1000
500
200
100
50
20
10
5
2
1

Figure 7.6 *(previous page)*: Multi dimensional coverage histogram of influenza sample from Ferret 54 day 2 (100bp reads). The x-axis shows the dimension of the *de Brujin* graph used and the y-axis the coverage in log-scale. The heat map displays the frequency of appearance of each coverage at each hash length in log-scale. One can see the high frequency of appearance of low coverage across most hash lengths. Below hash length 10 one observes saturation of the k-mer universe, thus the high frequencies. Closer towards 100 the resulting graphs become more and more disconnected and amount of overlaps drops.

the k-mers within those repeat sections to appear in a multiple of the sampling rate. See Figures 7.4d, 7.4e.

- The final observed coverage plot is a sum of the individual components as shown in Figure 7.4e.
- In addition to that the coverage distributions change with the chosen hash length as seen in Figure 7.6.

These distributions appear to model observed k-mer coverage patterns fairly accurately and analyses based on similar (but more detailed) models are being used in the Kmer Analysis Tool (KAT) [CB13].

This composition of k-mers makes it difficult to distinguish k-mers that originated in errors from those that are part of the target sequence. We have found a new way of separating those as discussed in subsection 8.2.2.

7.4. *De Brujin* graph assembly

As discussed in the previous section 7.3 erroneous reads create a large amount of k-mers that are not part of the target genome. As a result the erroneous k-mers in practice outnumber the target k-mers by at least a factor of 100 (for $k > 15$, based on empiric measurements in StarK graphs). Each *de Brujin* graph assembler implements the extraction of contigs from the graph in their own way based on various heuristics that the developers find to work well on their given training sets. Also various specialised assemblers exist (for e.g. metagenomics assemblies [RBD13; Bak+13]). Here we will explain the most basic approach as details of the different assemblers are beyond the scope of this comparative chapter.

The first draft of *de Brujin* graph assemblers was to simply use a coverage cut-off and assemble the remainder of the graph. This is based on the assumption that the k-mers generated by erroneous reads will separate in their frequency of appearance from k-mers that belong to the target genome in a fashion similar to what is displayed in Figures 7.4c and 7.2.

The steps for the most simplistic assembly algorithm are as follows:

1. Select all k-mers that have coverage greater than a cut-off (chosen as an assembly parameter or based on some heuristic).
2. As long as there are unused k-mers in the graph:
    - Begin at a random k-mer.
    - Extend a path from there until a dead end or a branch is encountered.
    - Mark used k-mers.
    - If the contig associated with the marked path is of a minimum length – print it.
3. Run your favourite scaffolder with the contigs.

We will not be further elaborating on *de Brujin* graph assembly here. The above algorithm is mentioned for reference and in order to explain the transition to the

StarK graph assembly in chapter 8 subsection 8.2.3 Graph partitioning.

8. StarK – locally adaptive graph assembly

8.1. Motivation

As shown in Table 6.5 we experienced difficulties trying to assemble our high variation viral samples using existing assemblers, primarily due to various limitations in the underlying algorithms and implementation. In addition to that none of the assemblers at the time this project began (2010) were capable of assembling variants together with consensus contigs. As such analysis that requires this information like in Part I rely on alignments towards a consensus sequence which is in itself biased.

Most short-read assemblers then were *de Brujin* graph based and were all suffering from the limitations implied by choosing a fixed graph dimension. String graph assemblers were not yet capable of run times that could be applied in practice.

As such we chose the *de Brujin* graph as the base for our new method due to widespread use, good general algorithms for it and the speed that modern implementations allow. The name StarK stands for k^* (k star) — all dimensions.

The design focus behind the StarK assembler initially was to

- Create an assembler for high-variation viral samples,
- Overcome the limitations of single-dimension *de Brujin* graph assemblers.

During development of the algorithm and the StarK software we found that it can also

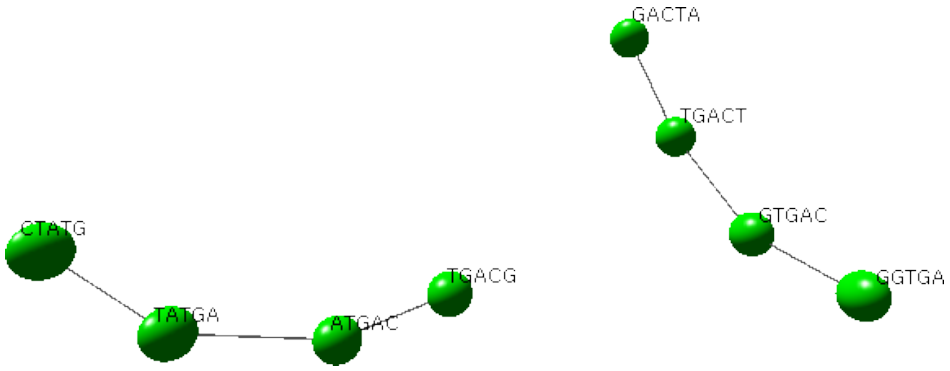


Figure 8.1.: The reads `GGTGACTA` and `CTATGACG` as 5-mers.

be used for general-purpose assemblies and is capable of outperforming current NGS short read assemblers. The focus then slightly shifted towards being able to cope with larger genomes and sample sets. As such the part of the assembler that identifies and maps variants onto contigs has its backbone implemented, but is unfinished as of today.

The main sequence assembly part is implemented, works and our training sets are being assembled into sensible contigs.

8.1.1. Limitations of *de Brujin* graph assemblers

Before we introduce the theoretical concepts behind StarK, we will recap the limitations of *de Brujin* graphs in this section and then address them with StarK.

While *de Brujin* graphs are extremely fast to build they come with significant limitations due to information loss originating from the representation.

The major limitations of *de Brujin* graph assembly is that it introduces an artificial parameter: the *de Brujin* graph dimension k (also referred to as hash-length, k-mer length or just k).

This leads to two major problems:

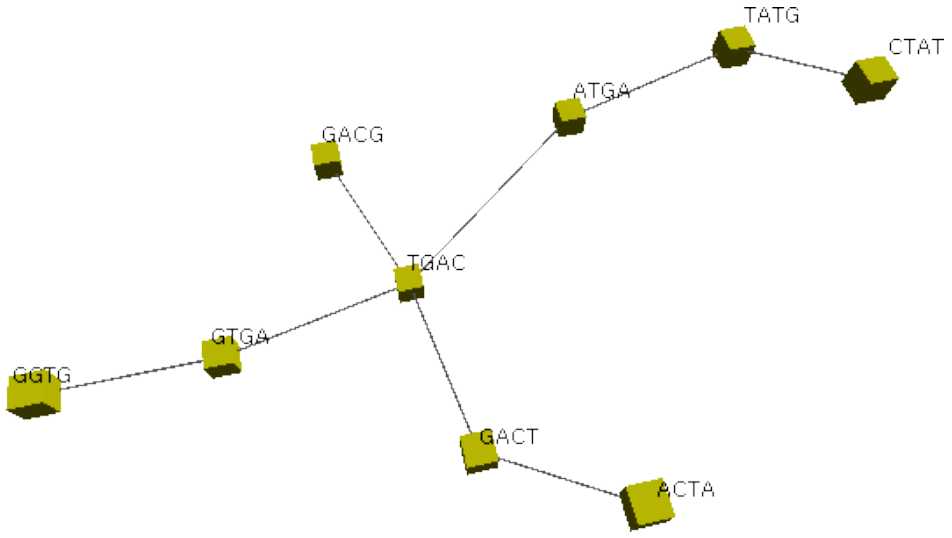


Figure 8.2.: The reads GGTGACTA and CTATGACG as 4-mers.

Short overlaps cannot be assembled Let $s \in \Sigma^*, |s| \gg k \in \mathbb{N}$ a long sequence, $w \in \Sigma^k$ a k-mer and $\iota(s, w) \neq \emptyset$ (w appears in s as a subsequence). If a read set $\mathcal{R} \subset \Sigma^{\geq k}$ does not contain a k-mer that is part of s

$$\forall r \in \mathcal{R} : \iota(r, w) = \emptyset \tag{8.1.1}$$

Then no contig containing s can be constructed.

Example 8.1.1. *Let* $s =$ GGTGACTATGACG, $\mathcal{R} = \{$GGTGACTA, CTATGACG$\}$. *For all* $k > 3$ *it will be impossible to reconstruct* s *with a de Brujin graph formed from* $\mathcal{R}$ *because the overlap* CTA *can not be represented by any* 4*-mer or longer. See Figures 8.1, 8.2.*

One could argue that in order to overcome the above limitation one could use a smaller *de Brujin* graph dimension. But indefinitely reducing the *de Brujin* graph dimension is not an option because of the second drawback:

Repeats shorter than the hash length cause spurious joins of the graph

Let $s \in \Gamma^*$ a sequence and $|s| \gg k \in \mathbb{N}, w \in \Gamma^k$ such that

$$|\iota(s, w)| > 1 \tag{8.1.2}$$

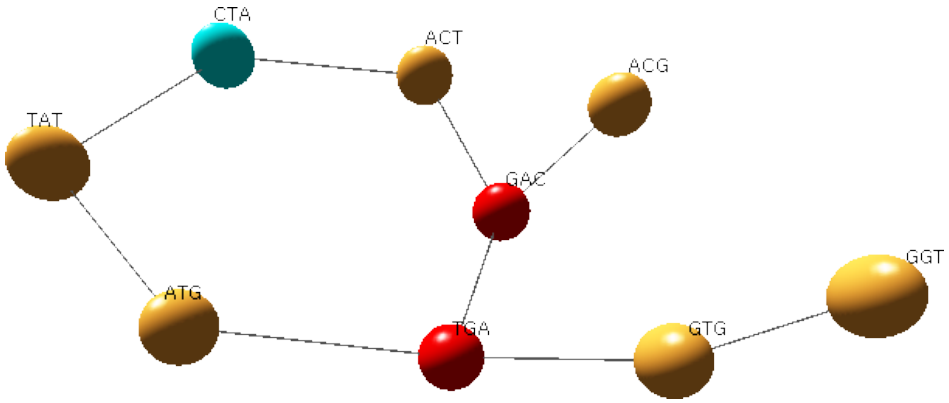


Figure 8.3.: The reads GGTGACTA and CTATGACG as 3-mers.

i.e. the k-mer w repeats within s.

Then any assembler will require additional information on how to join the ends of the graph leading from w.

Example 8.1.2. *Let* $s = \texttt{GGTGACTATGACG}$, $\mathcal{R} = \{\texttt{GGTGACTA}, \texttt{CTATGACG}\}$. *For all* $k > 3$ *the sequence* TGAC *will have branches in the graph. Figures 8.2 and 8.3 show the de Brujin graphs for dimensions 4 and 3 respectively.*

Combining those two drawbacks leads us to the conclusion that without additional information a traditional *de Brujin* graph assembler can not reconstruct GGTGACTATGACG uniquely from the reads GGTGACTA and CTATGACG with any fixed hash length $k \in \mathbb{N}$ even though the reads themselves provide sufficient information to do so.

Although a common approach today is to "scan" multiple k-mer sizes and then pool the contigs together (e.g. the Trinity Assembly package [Gra+11]) it will rarely work because of repeats like those shown in examples 8.1.1 and 8.1.2.

In order to address the above-mentioned limitations we have designed an algorithm that does not require a fixed hash length and is capable of adjusting it on the fly.

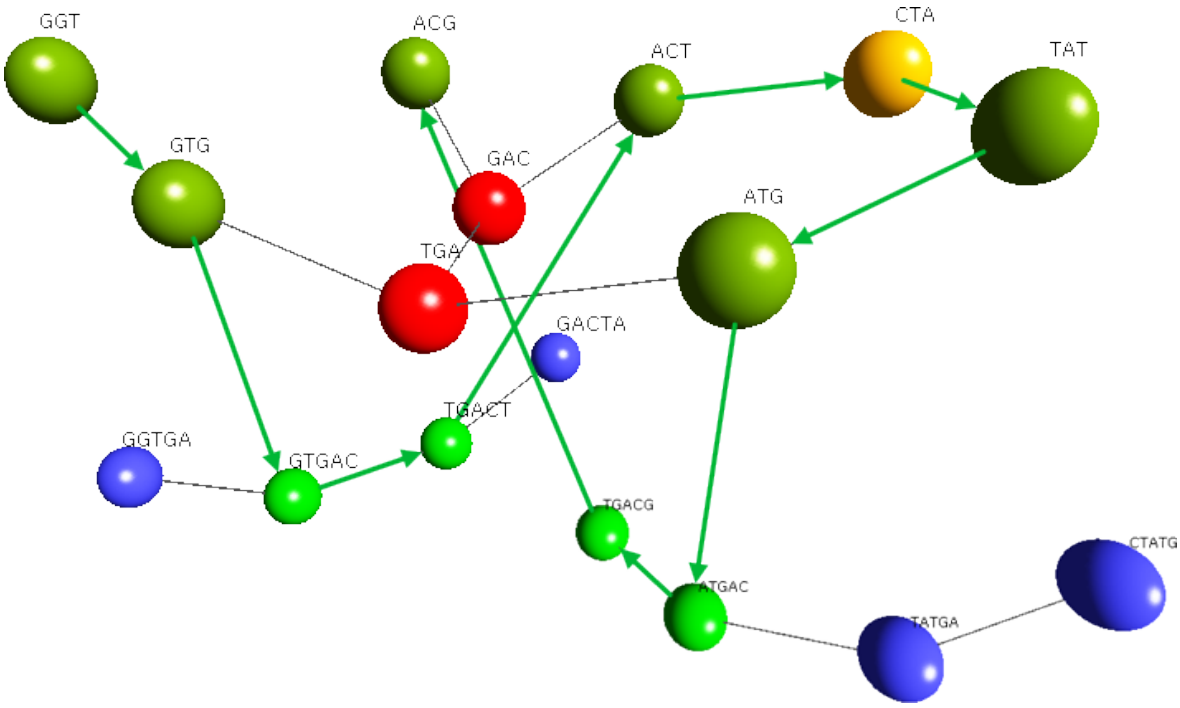


Figure 8.4.: Multi-dimensional *de Brujin* graph assembly example. The green path shows the assembly path for the contig GGTGACTATGACG avoiding the repeat section TGAC by using 5-mers at those positions.

8.1.2. Multi-dimensional solution

Regular (single-dimensional) *de Brujin* graphs are unable to efficiently overcome the limitations described above and demonstrated in examples 8.1.2 and 8.1.1. But the reads in those examples provide sufficient information to assemble them into the sequence GGTGACTATGACG, assuming we have a hint that they need to overlap on CTA, which can be for example in the form of coverage information.

If we were able to use information from different *de Brujin* graph dimensions then we could follow an elegant assembly path as shown in Figure 8.4. The displayed path uses 5-mers to go around the 4-mer repeat TGAC, but reverts to 3-mers in order to join the path at the 3-mer CTA. In the next section we will be discussing the theoretical framework for achieving this.

De Brujin graphs of the same data on multiple dimensions can be linked together by overlaying them with suffix and/or prefix trees. (i.e.) A k-mer w and a $k+1$-mer wa (for $w \in \Sigma^+, a \in \Sigma$) would have an edge between them in a prefix tree. Suffix trees are being used for alignments [Alt+90]. Combining the power of of *de Brujin* graphs, prefix and suffix trees allows us to link short read fragments more efficiently.

We have earlier benchmarked the capabilities of two *de Brujin* graph based assemblers that do utilise multiple dimensions (IDBA-UD [Pen+12] and SPAdes [Ban+12]), but they both do so by gradually increasing the k-mer length. StarK differs from that approach in such that we allow the assembler to access any k-mer length at any given time allowing the information content from all dimensions to be available during all stages of assembly.

8.2. StarK Theoretical Framework

The essence of the StarK approach is to load *de Brujin* graphs of all possible dimensions into main memory (RAM) at once and allow the assembler to use any path through the graph changing dimensions as necessary. The graph resembles an affix tree[Maa00] and we will be using affix tree like connections between the nodes to connect them, but apply metrics implied from the underlying *de Brujin* graphs to choose which ones to use. Figure 8.5 displays all k-mers of the two reads `GGTGACTA` and `CTATGACG` as they appear in the StarK graph.

Reminder of notation: 'w is the word w truncated by one letter from front, w' is the word w truncated by one letter from the end.

Definition 8.2.1. *Let Σ be an alphabet. A graph $G = (\Sigma^*, E)$ is called a (full) StarK graph (over Σ) if and only if*

$$E = \{(w, wa), (w, \text{'}w), (w, \text{'}wa) | w \in \Sigma^*, a \in \Sigma\} \tag{8.2.1}$$

Please note that a full StarK graph has only one parameter – the underlying alphabet Σ. Although our current application and implementation (chapter 9) use only the genomic alphabet, the theory discussed here applies to generic alphabets.

In practice we will only be interested in the subgraph of a StarK graph for which the coverage (see Equation 7.2.5) in relation to a set of reads is above zero, thus the full StarK graph (as is the case with the full *de Brujin* graph) is only of theoretical interest here.

As is also the case with practical implementations for *de Brujin* graph assemblers out StarK graph over the genomic alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ is implemented by folding the graph in such a way that a k-mer is stored together with its the reverse complement. This turns a practical implementation often into an undirected graph where each node has twice as many edges.

When such a folded graph is traversed it is till only traversed following a direction

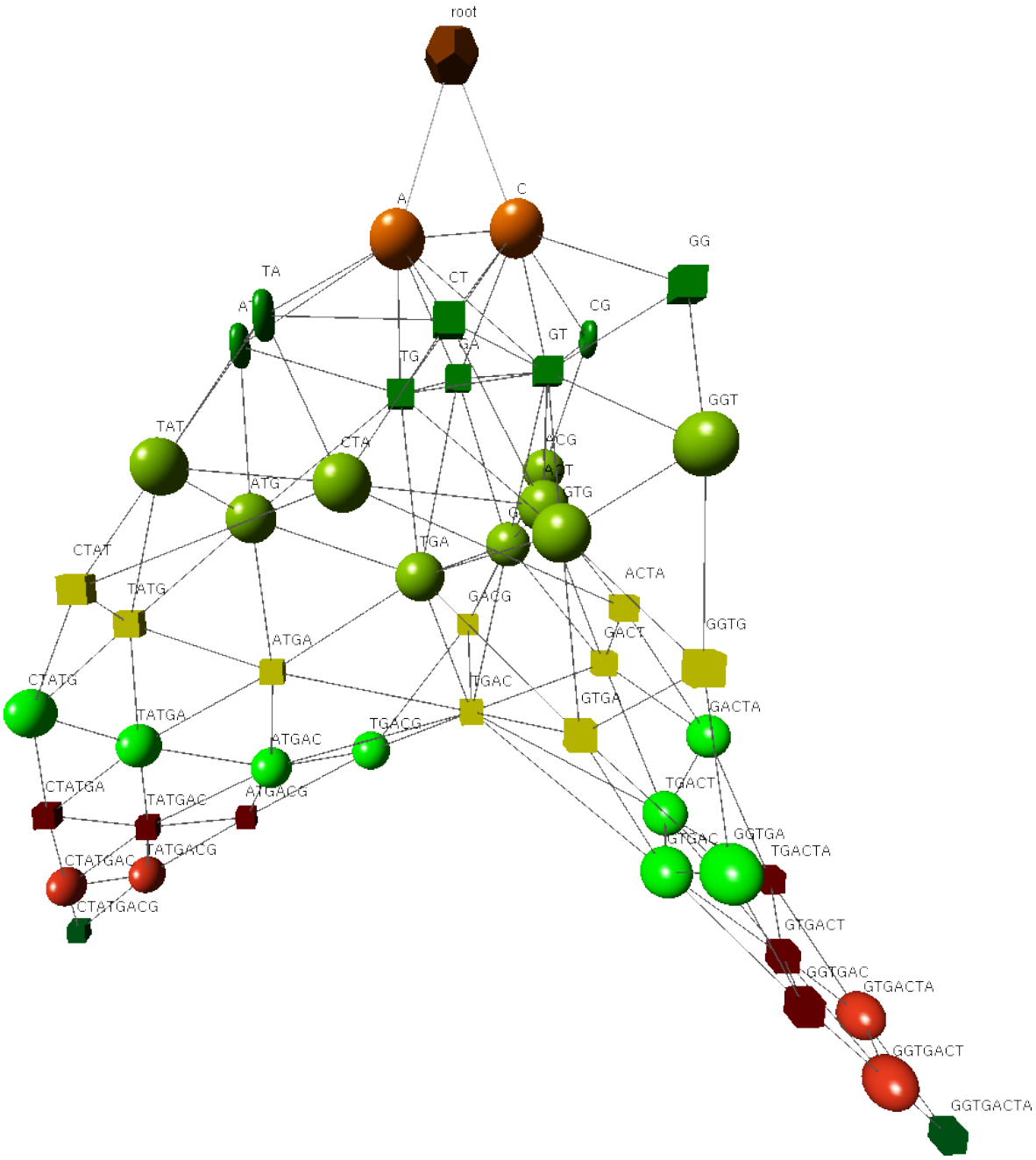


Figure 8.5.: All possible k-mers of the two reads `GGTGACTA` and `CTATGACG` as they appear in the StarK graph. Each *de Brujin* dimension has a separate colour. Circular vertices are odd dimensions, cubical ones are even. Oval shaped nodes identify palindromic sequences (here displayed are `AT`, `TA` and `CG`).

that identifies only one of the two k-mers represented by a node in the graph. While describing the theoretical framework here we will stick to the more general directed graph.

Proposition 8.2.2. *Let Σ be an alphabet. Let $G = (\Sigma^*, E)$ be the StarK graph over Σ. Then the following hold true:*

1. *The empty word λ is a vertex in G.*
2. *For every $k \in \mathbb{N}$ the de Brujin graph $G_k = (\Sigma^k, E_k)$ of dimension k is a subgraph of G.*
3. *For every word (k-mer) $w \in \Sigma^*$ and every character $a \in \Sigma$ there are two edges $(w, wa), (wa, w)$ in E connecting the different de Brujin graphs of dimensions $|w|$ and $|w| + 1$.*

This new theoretical data representation has the practical advantage to be related to *de Brujin* graphs and as such several features common to *de Brujin* graph assemblers apply here too, namely coverage distributions as discussed in section 7.3. But the new structure also comes with new features distinct from *de Brujin* graphs that need to be considered:

- The same k-loop-free sequence may be assembled by more than one path through the graph,
- In a *de Brujin* graph erroneous reads typically result in short, low-coverage offshoots (called **tips** in the *de Brujin* graph jargon) from a high coverage path within the graph, in a StarK graph those offshoots rejoin the graph at a shorter k-mer length creating new spurious connections.

In the next two subsections we will be discussing two new theoretical concepts that deal with these new features.

8.2.1. Surface paths

One of the main differences which makes traditional *de Brujin* graph assembly approaches inappropriate for a StarK graph is the fact that the same sequence may be reconstructed by more than one path though a StarK graph.

This makes it impossible to use the conventional "longest possible sequence" assembly algorithm (compare to section 7.4). In order to extract contigs from a StarK graph we require a method to tell the assembler which *de Brujin* dimension to use at a given position in the contig.

For this reason we use so called surface paths. A surface path consists exclusively of vertices (k-mers) (of a subgraph of a StarK graph) such that for any two connected k-mers of the same dimension no $(k+1)$-mer exists in the subgraph that could replace them in the path and still represent the same overall sequence.

We call them surface paths, because they go only along the surface of the appropriate StarK subgraph (when being drawn) — there are no k-mers on a higher "k-mer dimension" than those in the surface path.

Definition 8.2.3. *Let Σ be an alphabet, $G = (\Sigma^*, E)$ be a StarK graph over Σ and $S \subseteq \Sigma^+$ a subgraph in G. Let $p = (v_1, \ldots, v_n), n > 1$ be a path in S. For $i \geq 2$ we call*

1. *v_i a decent node of p if $|v_{i-1}| - 1 = |v_i|$*
2. *v_i an ascend node of p if $|v_{i-1}| + 1 = |v_i|$*
3. *v_i a level node of p if $|v_{i-1}| = |v_i|$*

This categorises all nodes in a path except for the first as they can always only change length by one.

Definition 8.2.4. *Let Σ be an alphabet, $G = (\Sigma^*, E)$ be a StarK graph over Σ and $S \subseteq \Sigma^+$ a subgraph in G. Then a path $p = (v_1, \ldots, v_n)$ in S is a **surface path** with the sequence $s_1 \ldots s_m$ if and only if for every subpath q of p the sequence represented by q cannot be represented by a shorter path in S or a path using nodes or larger k-mer*

dimension.

Example 8.2.5. *If two vertices* `CTGA`, `TGAC` *appear consequently in a surface path then* `CTGAC` *cannot be in the same subgraph the surface path was derived in as it would represent the same sequence and be both shorter and contain larger dimension k-mers.*

Proposition 8.2.6. *Let Σ be an alphabet, $G = (\Sigma^*, E)$ be a StarK graph over Σ and $S \subseteq \Sigma^+$ a subgraph in G. and $p = (v_1, \ldots, v_n)$ in S a* ***surface path*** *with $n \geq 2$. Then any sequence of decent nodes in p is always followed by a level node.*

Proof. A surface path cannot be terminated by a sequence of decent nodes as the start of such a sequence will be a node that represents the same word in Σ^* as the entire sequence. As such it could be truncated away from the path forming a shorter path thus violating the definition of a surface path.

If sequence of decent nodes $v_i \ldots v_j$ in a surface path cannot be followed by an ascension node v_{j+1} because in that case the node v_j could be removed from the path and it would still represent the same word in Σ^* thus violating the definition of a surface path.

$\Rightarrow$ any sequence of decent nodes in p is always followed by a level node. □

Proposition 8.2.7. *Let Σ be an alphabet, $G = (\Sigma^*, E)$ be a StarK graph over Σ and $S \subseteq \Sigma^+$ a subgraph in G and $p = (v_1, \ldots, v_n)$ in S a* ***surface path*** *with $n \geq 2$. Then any sequence of ascend nodes in p is always preceded by a level node.*

Proof. Same as proof for Proposition 8.2.6 just reversed. □

Proposition 8.2.8. *Let Σ be an alphabet, G be a StarK graph over Σ, $S \subseteq \Sigma^+$ a subgraph in G, s a word in Σ^+ that has a surface path p representation in S. Then p is unique (i.e. p is the only surface path representing this word).*

Proof. Proof by contradiction.

Let $p = (v_1, \ldots, v_n), p' = (v'_1, \ldots, v'_m)$, $p \neq p'$ be two different surface paths in S that represent the same word. They have to be of the same length ($n = m$), otherwise the longer one violates the definition of a surface path.

Induction over $k \leq n$.

Initial Step: If $v_1 \neq v'_1$ then $|v_1| \neq |v'_1|$ which violates the definition of a surface path that demands that it has to use the vertices of larger k-mer dimension.

Inductive Step: Let us assume for all $k \leq o < n : v_k = v'_k$. Then if $v_{k+1} \neq v'_{k+1}$ then $|v_1| \neq |v'_1|$ which violates the definition of a surface path that demands that it has to use the vertices of larger k-mer dimension. □

Now we know that once we have a surface path, no other surface path will give us the same sequence. And we can uniquely map all loop-free paths of a subgraph of a StarK graph to contigs. Just like splitting a *de Brujin* graph into paths that represent contigs, we will be splitting StarK graphs into subgraphs, such that each contains only one surface path and such a path is either a contig, a variant of another contig or an assembly of errors.

The subsection 8.2.3 will discuss how we partition a StarK graph in such a way that we get meaningful contigs from their respective surface paths.

8.2.2. Link strength

In a traditional *de Brujin* graph we would just select the parts of the graph with sufficient coverage (compare to section 7.4). Using a similar cutoff in a StarK graph leads to contigs that are not much longer than the input reads. This is due to the fact that the majority of reads generated by the NGS short-read sequencers contain at least one error and thus on the k-mers closer to read length we can rarely find a StarK subgraph that contains only one longest path. On the smaller dimensions (shorter k-mers) we receive too many spurious joins and the graph begins to "clump up" into one large weakly connected component making it almost impossible to get branch-free surface paths.

When analysing Figure 7.6 we have found that most connections between k-mers are steady (i.e. coverage differences are small). Having observed that we have analysed

relative k-mer coverage between adjacent graph vertices further and are now using the following measure for inter k-mer connectivity:

Definition 8.2.9. *Let $c_1, c_2 \in \mathbb{N}$ be the coverage values for two neighbouring vertices (may be neighbouring dimensions). We define the* ***link strength****:*

$$l : \mathbb{N}^2 \to [-1, 1] : (c_1, c_2) \mapsto \frac{c_1 - c_2}{\max(c_1, c_2)}. \tag{8.2.2}$$

In order to measure how link strength is distributed in a StarK graph we have computed a histogram of link strength absolute values (from between 11-mers in a StarK graph of influenza sample from Ferret 54 day 2) as displayed in Figure 8.6. The results are clear: this measure separates parts of the graph with steady inner coverage changes from those with radical coverage differences.

This approach uses two assumptions:

1. Coverage in the target is steady (except in repeat sections),
2. No single error variant appears in abundance.

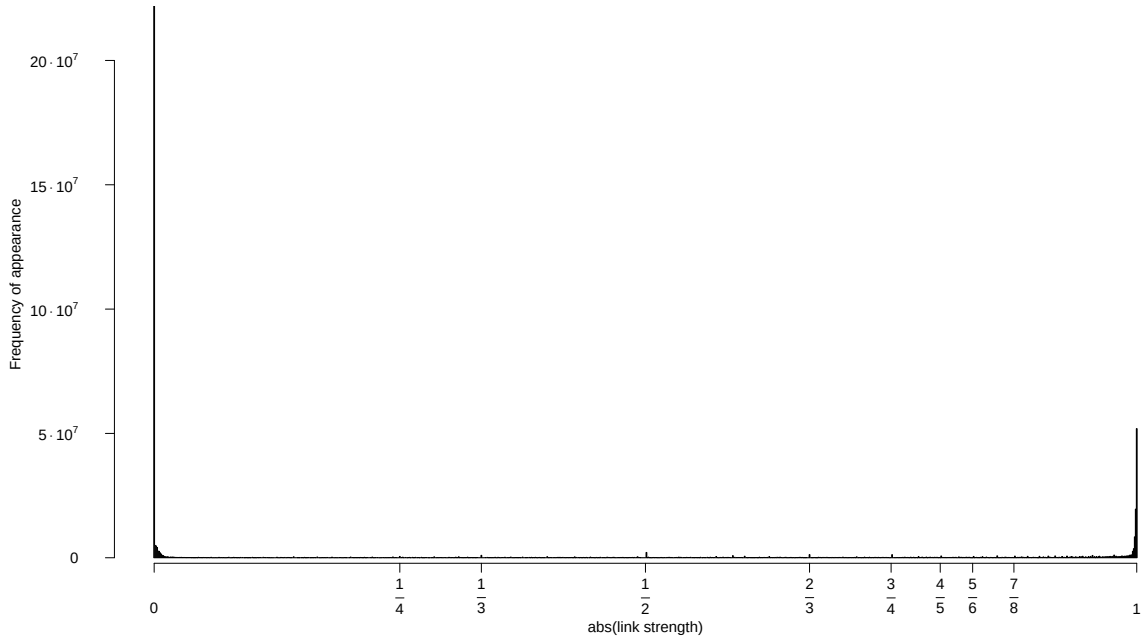
Based on our observations these assumptions are not unrealistic, especially if some preliminary filtering on the reads was done (e.g. PCR [Mul+87] duplicate removal as described in section 2.5).

We use this metric to partition the StarK graph.

For implementation reasons we use a symmetric weighting function derived from the link strength:

$$\mathfrak{l} : E \to [0, 1] : (v_1, v_2) \mapsto l(c(v_1), c(v_2))^2. \tag{8.2.3}$$

Unlike in Figure 8.6 we use squares here instead of absolute values. This has performance reasons tied to how x86 CPUs implement floating point operations; multiplying a double precision float is faster than unsetting the sign bit if the value is already in


Frequency of appearance
20 · 10^7
15 · 10^7
10 · 10^7
5 · 10^7
0
0
1/4
1/3
1/2
2/3
3/4
4/5
5/6
7/8
1
abs(link strength)

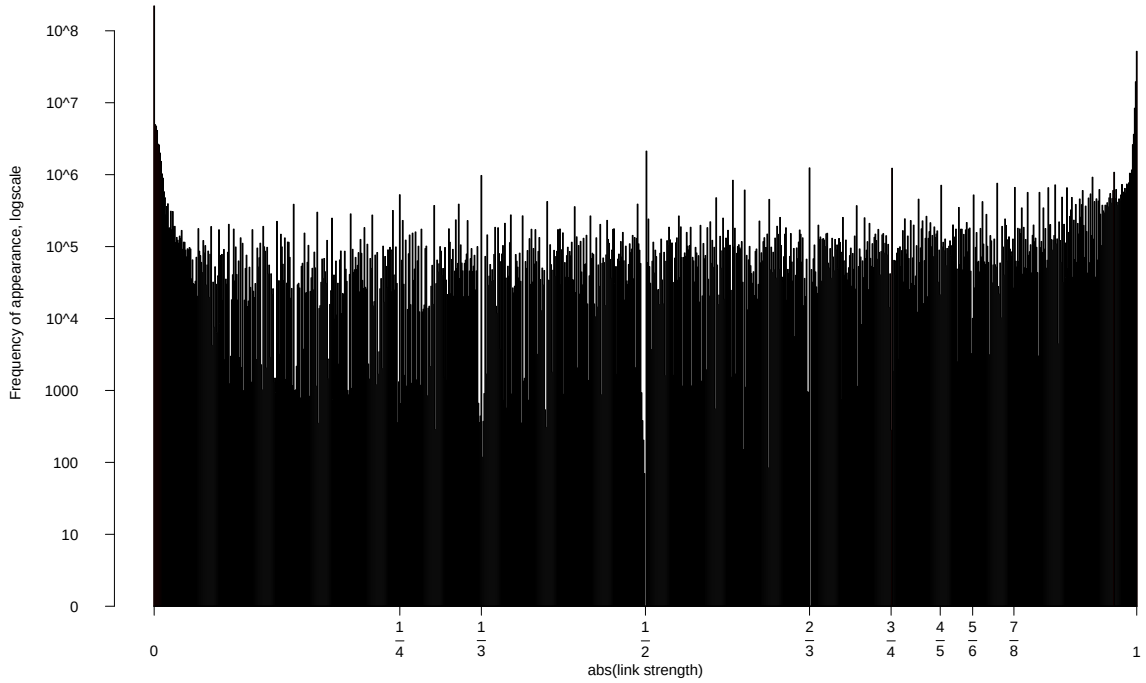

Frequency of appearance, logscale
10^8
10^7
10^6
10^5
10^4
1000
100
10
0
0
1/4
1/3
1/2
2/3
3/4
4/5
5/6
7/8
1
abs(link strength)

Figure 8.6 *(previous page)*: The plots show link strength absolute values distributions between 11-mers in a StarK graph of Influenza sample from Ferret 54 day 2. The upper plot shows how well the link strength separates to the two extremes 0 (similar coverage) and 1 (total difference) while the bottom plot shows the same in log scale. Small peaks can be observed at the various fractions, although one might suggest that those are entries and exits into repeat sections, they are in reality abundant links between erroneous k-mers of discrete coverages 1–2, 2–3, etc.

a floating point register. Since squares are a monotone function this does not change the sorting order compared to absolutes, which is all we are interested in.

We have benchmarked this performance by running a simple benchmark program 10^10 times in a loop compiled with `clang` on a Mac running an Intel Core i5@2.4 GHz. Squaring the result of a floating point subtraction each time required 72 seconds versus 81 seconds when unsetting the sign bit.

This has to do with the fact that the CPU needs to copy the contents of the floating point register into a general purpose register before unsetting a bit incurring a delay, while a multiplication will take place on the floating point stack directly.

8.2.3. Graph partitioning

The idea is to partition the StarK graph into subgraphs in such a way that each subgraph contains exactly one surface path. Each surface path then corresponds to either

- (part of) a haplotype of the target,
- an assembly of errors in the reads (equivalent to a tip in a *de Brujin* graph).

Since we can no longer apply the same approach to identify errors as *de Brujin* graphs use (looking for low-coverage offshoots), we need to find some other means of doing

this. This is where the link strength comes in.

Example 8.2.10. *Assume the reads* TCAGATGCCACT *and* CCACTCCATCAT *were sampled in more than one copy each. In addition to that the read* AGATGCCACACCA *was sampled once a with an error.*

```
TCAGATGCCACT
        CCACTCCATCAT
AGATGCCACACCA
```

In a 5*-dimensional de Brujin graph these would form a single high coverage path with a low-coverage tip that is created by the* k*-mers containing the error.*

In a StarK graph the tip could be rejoined with the main contig path on the 3*-mer* CCA *in two positions making it more difficult to distinguish* k*-mers introduced by errors.*

At this point we will use the assumption (as explained in subsection 8.2.2) that we expect no radical changes in coverage between k-mers that make up our target contigs.

To explain the theoretical goal of our approach we introduce two new concepts:

Definition 8.2.11. *Let* $G = (V, E)$ *be a StarK graph and* $G' = (V', E')$ *a subgraph of* G *with only one weakly connected component. We define the* ***internal link strength*** *of* G *as the maximum of all link strengths within the subgraph:*

$$l(G') := \max_{e \in E'} |l(e)|. \tag{8.2.4}$$

The link strength between two disjoint subgraphs $G' = (V', E'), G'' = (V'', E'')$ *is defined as the maximum link strength of any edges connecting them:*

$$l(G', G'') := \max_{\substack{v' \in V', v'' \in V'' \\ (v', v'') \in E}} |l(v', v'')|. \tag{8.2.5}$$

Definition 8.2.12. *Let* G *be a StarK graph. We call* $\mathcal{G} = \{(V_i, E_i)\}$ *a* ***family of sub-contigs*** *of* G *if*

1. $\forall (V_i, E_i) \in \mathcal{G} : (V_i, E_i)$ *weakly connected component of* G*,*
2. $\forall (V_i, E_i), (V_j, E_j) \in \mathcal{G} : V_i \cap V_j = \emptyset$ *(pairwise disjoint),*

3. $\forall(V_i, E_i) \in \mathcal{G} : (V_i, E_i)$ *has one unique (except for reversing) maximal surface path that contains all other surface paths.*

In practice construction of contigs during genome assembly is a heuristic process during which we create the longest possible sequences that, based on some assumptions, have most likely been the origin for the underlying reads. In StarK the contigs are derived from sub-contigs (StarK subgraphs) and we assume that extreme changes in coverage are unusual.

As such the ideal partition of a StarK graph into a family of sub-contigs that we would aim for should have two features:

1. Joining any number of sub-contigs will not result in a new sub-contig (i.e. the resulting graph has no unique maximal surface path and would introduce branches),
2. The internal link strength of each sub-contig is lower than the link strength to any of its neighbouring sub contigs. Lower in this case means less radical changes — more steady transitions.

Although we do not enforce the latter (weaker) feature in our implementation, it is a factor that plays an important role in how we choose to merge sub-contigs.

Since the time complexity for achieving an ideal version of this goal is probably beyond anything that can be reasonably computed, we will be trying to approximate this ideal partitioning.

The steps are as follows:

1. Generate an initial partitioning $\mathcal{G}$ of a StarK subgraph with coverage of at least 1 into a family of sub-contigs. A trivial partitioning would be one where each sub-contig contains exactly one k-mer.
2. Compute the link strength between any two adjacent sub-contigs.
3. In the order of ascending absolute link strength:
    a) Attempt to join any two sub-contigs.

b) If a merge results in a new sub-contig with a longer sequence represented by any of the two origin sub-contigs:

   Keep the merged sub-contig instead, recompute link strength to neighbouring sub-contigs.

4. Repeat the previous two steps until no more merges are done.

The resulting family of sub-contigs is an approximation of the ideal target family.

We observed that the restriction to apply this merging in ascending order guarantees that contigs, which likely belong together, get merged first instead of joining errors/tips with contigs.

The output then contains all contigs longer than a certain preset (runtime parameter).

Remark 8.2.13. *Although building a similar family of sub-contigs could have also been achieved through a top-down approach by splitting the original graph iteratively we have chosen the bottom-up approach (merging) because it is easier to parallelise.*

In the next chapter we will discuss the practical implementation of this theoretical framework into software and the challenges met regarding to parallel execution.

9. Implementation and parallelisation of the StarK assembler

This chapter focuses on the technical implementation of the StarK assembler theory into a working software prototype. The theoretical construct only explains the algorithmic value of the approach. Here we explain the challenges met when implementing the StarK algorithm in C [KRE88] and how we tackled them. Special attention is given to parallelisation.

9.1. Introduction

Over the years we have seen various implementations of genomic *de Brujin* graphs. The one most frequently used is to store all k-mers in a hash map by sequence. Edges are stored implicitly in form of existence or non-existence of neighbour nodes. Testing for a constant number of neighbour nodes requires $O(1)$ constant time assuming the hash map provides constant time access. This development has also led to the term **hash length** to be used synonymously with k-mer length or *de Brujin* graph dimension.

This data representation requires a lot of memory, as the sequence, even when compressed to two bits per nucleotide, uses up a large amount of memory per k-mer. Considering that the majority of stored k-mers (in practice over 99%) originate from

errors and contaminants in the reads this can quickly exceed today's typical hardware capabilities (e.g. 256 GiB Random Access Memory (RAM)). For example storing a 95-mer (not unreasonable given read lengths of up to 160) would require $2 \cdot 95 = 190$ bits or 24 bytes to store the sequence alone. The human genome requires approximately 2^{32} k-mers to represent the target (without erroneous k-mers) as a *de Brujin* graph. At 24 bytes per k-mer this brings us to 96 GiB for the target k-mers alone at a single dimension.

Recently an implementation was suggested using a bloom filter to decrease the amount of memory used per k-mer to less than two bytes [CR+12], but this implementation comes at the cost of not being able to attach payload data to a k-mer (i.e. coverage information).

Since the StarK graph essentially has to store up to the read length number individual *de Brujin* graphs we have chosen a different in-memory representation.

We have chosen to represent a StarK graph in memory in the form of a directed graph. Edges in the graph exist only between nodes representing k-mers of length difference of 1. The graph resembles a eight-ary tree-like structure where each node has exactly two parents and up to 8 children.

Each node represents a k-mer and its reverse complement. The first draft of the data structure (`starknode_t`) uses 48 bytes per k-mer. It illustrates well the functionality of the data structure and allows superior processing speed, but is not intended for production deployment. Using the new compressed data structure (section 9.8) data representation each k-mer consumes on average less than 10 bytes independent of k-mer length, while sacrificing performance.

9.2. Data structure

The heart of the StarK implementation is our representation of a k-mer in memory — the StarK node type `starnode_t`. The design of this data type focuses on both representing the StarK graph with as much detail as is necessary for operation and constant time transition between neighbouring graph vertices.

9.2.1. k-mer representation (`starknode_t`)

Each `starknode_t` struct represents both a k-mer and its reverse complement. They are identified by a "forward" and a "reverse" k-mer. The "forward" k-mer is always the lexicographically smaller one. A bit identifying node relative **rotation** allows selection of a specific k-mer. If a k-mer is equal to its reverse complement (i.e. it is a palindrome) then the software enforces its child nodes to be identical in both directions and the rotation bit has no meaning.

Throughout this chapter we will refer to a `starknode_t` object as a **node** (dual k-mer storage container, rotation independent) and only refer to k-mers if we wish to identify a specific *de Brujin* k-mer (rotation dependent).

Since a StarK graph in memory resembles a tree-like layered structure (all shortest paths to the node representing the empty word have the same length), we will also be using tree-related terminology:

- Graph **root** – in this case the node representing the empty word λ.
- Node **child** – a directly connected node, which is one depth level lower from the root.
- Node **parent** – reverse relation to child.

Table 9.1 Shows the `starknode_t` data layout as used by our implementation.

Although this is not the most memory efficient form of representation (see section 9.8

	4 bytes				4 bytes			
	1	2	3	4	5	6	7	8
0x00	parent[0]				parent[1]			
0x08	edges	flags	debug		coverage			
0x10	child[0][0]				child[0][1]			
0x18	child[0][2]				child[0][3]			
0x20	child[1][0]				child[1][1]			
0x28	child[1][2]				child[1][3]			

(a) The `starknode_t` struct data layout.

`parent[*]`	Offsets into for the two parents.
`child[*][*]`	Offsets for four forward and four reverse children. These fields double as an indicator for a child's presence and as a lock when the child is created (refer to subsubsection 9.3.2.2)
`edges`	Bit field of 8 bits signalling the existence of the four neighbours '$w\Gamma, \Gamma w$'.
`flags`	Two times 2 bits indicating the transition nucleotide for each of the two parents. This is necessary to be able to deduce the k-mer sequence. Two bits indicating whether the parent is reverse-complemented in relation to this k-mer (relative rotation). Two flags for temporary use by downstream implementation.
`coverage`	32-bit integer holding the current coverage count.
`debug`	Two bytes that were initially used for debugging purposes and are only present for padding right now. Can be assigned any function if necessary.

(b) Description of the `starknode_t` fields.

Table 9.1.: The `starknode_t` data structure.

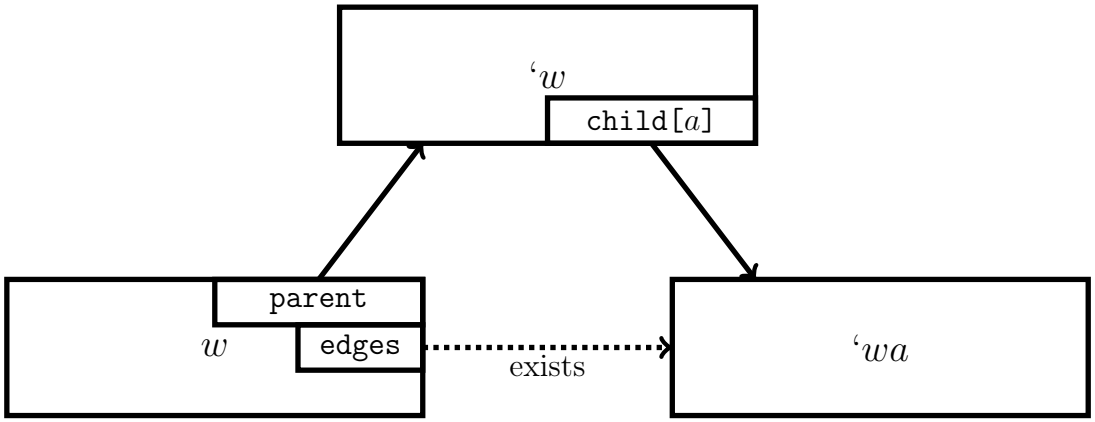


Figure 9.2.: `starknode_t` node – parent – neighbour relation.

for improvements) it has several advantages over hash maps (as used by most graph-based genome assemblers):

- k-mer sequence is not stored in the struct allowing for arbitrary length k-mers to have constant memory footprint.
- Transitions $w \rightarrow$ ‘w, w’, $w\Gamma, \Gamma w$ (node $\rightarrow$ parent, node $\rightarrow$ child) can be performed in constant time via only two pointer lookups.
- The struct allows for additional information to be stored like coverage. It can be extended with extra fields if necessary.

The inclusion of the `edges` bit field (which holds a bit for every exiting neighbour k-mer) seems redundant here. This information could be retrieved by querying child existence on the appropriate parent node as displayed in Figure 9.2. This field becomes necessary after the redundancy cleaning step section 9.4 and provides performance until then.

Figure 9.3 shows a StarK graph in memory with all virtual links through node offsets in order to illustrate inter-node link complexity.

9.2.2. Explanation

The StarK-graph is built in memory as a full graph containing every k-mer present in the reads starting with the empty word λ. By convention the graph grows down and we call the k-mer length **depth**.

The StarK master struct `stark_t` contains an array `level` holding 255 pointers to arrays of `starknode_t`. Almost every function that operates on the StarK graph receives a pointer to the master struct.

For memory efficiency we do not store pointers to StarK nodes, but rather their offsets in the `stark_t->level` array. The former would require 8 bytes of storage space for a 64-bit pointer, the latter is confined to a 32-bit offset and thus only requires half as much memory. Although this only allows for $2^{32} - 2$ total k-mers to be stored per depth, this was sufficient for our tests that we ran on viral sequencing data. The offsets were only saturated to 22 bits. See section 9.8 for an alternative data structure which allows for more k-mers to be stored at a smaller memory footprint.

Although `edges` is an optional field and we could merge `coverage` and `flags` into just four bytes, we have decided to keep all fields to preserve 8-byte struct alignment. Assigning only 2 bytes for the `coverage` has proved to be insufficient for viral sequences.

9.2.3. k-mer sequence retrieval.

The `starknode_t` struct does not carry the sequence of the represented k-mer itself in order to conserve memory as discussed above. In order to retrieve the sequence we have to traverse the graph from the node in question all the way to the root by following the parent node links. The nucleotide of each transition to a parent is stored in the `flags` field and allows together with the relative rotation full reconstruction of the sequence. The function `stark.h:get_sequence` can be used as reference for this process.

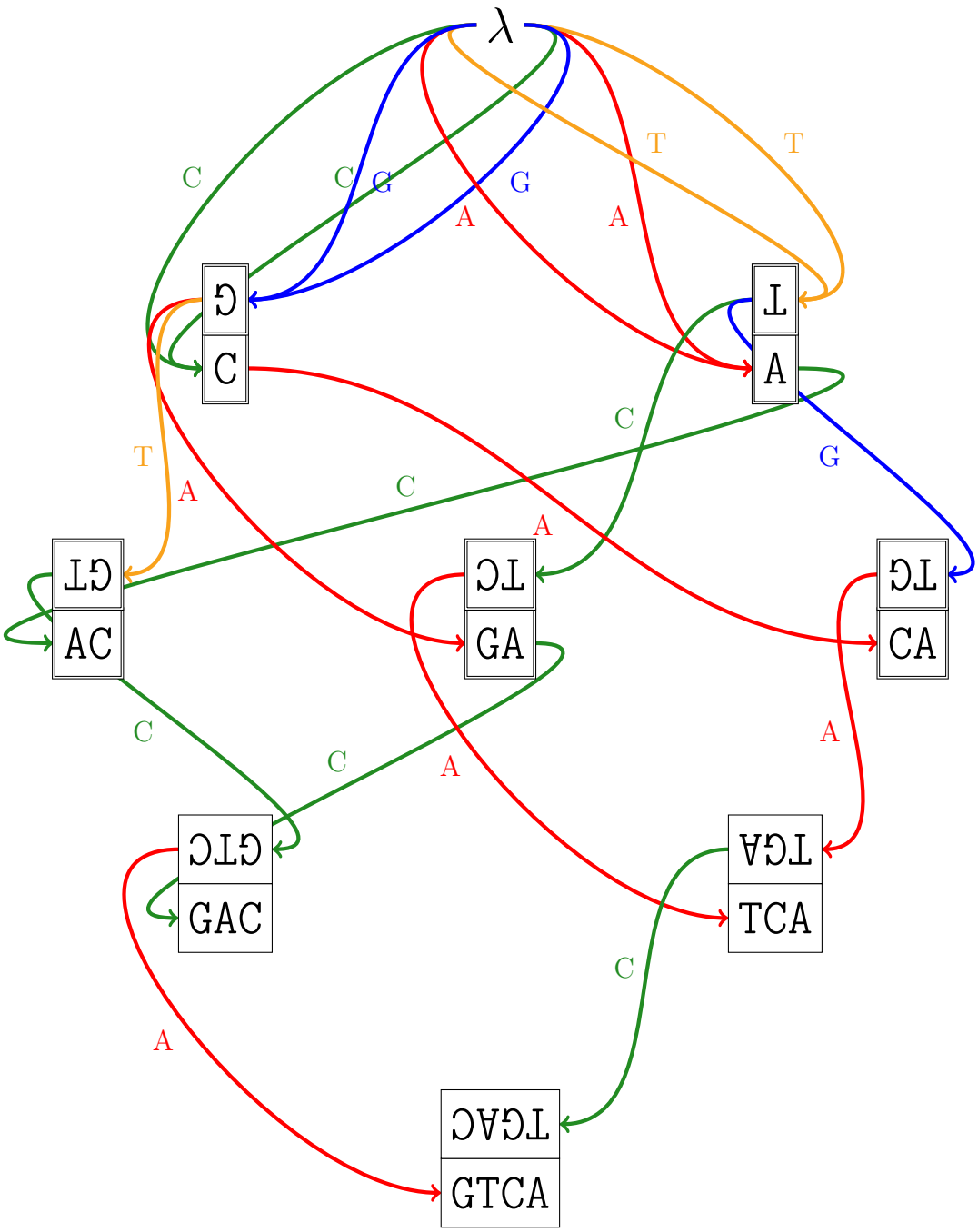
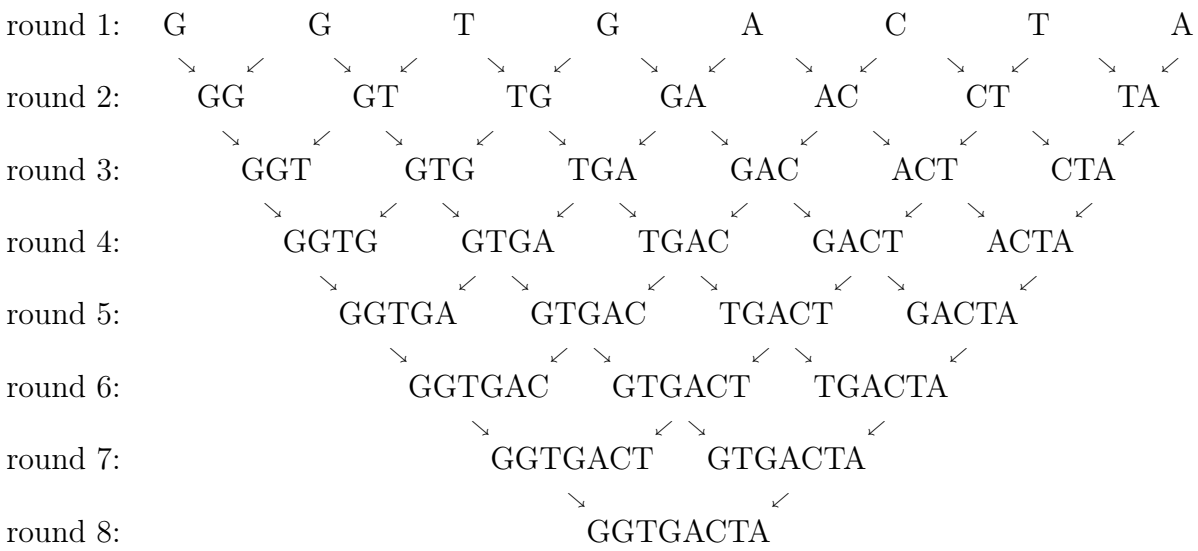


Figure 9.3.: Shows the StarK data structure concept when the read `TGAC` is inserted. The sequences are not actually stored, but implied by the structure of the graph. One can acquire the sequence associated with a k-mer by following the graph to the root node.

9.3. Building a StarK graph

9.3.1. Inserting reads

When reads are inserted into the StarK-graph they are parsed in ascending k-mer sizes. If $l = |w|$ is the read length of a read w, then w contains $\frac{1}{2}l \cdot (l+1)$ k-mers that need to be processed. As such this operation has time complexity of $O([\text{read length}]^2) = O(l^2)$. This process needs to recognise nodes for k-mers that already exist in the graph, create new ones, when necessary, and link them together. In order to minimise computation time a dynamic programming algorithm is used that caches the parent nodes for longer k-mers in each round. As an example the k-mers of the read `GGTGACTA` would be processed in the following way:



Identical k-mers are automatically mapped onto the same StarK nodes in the process. If a given node does not exist, it is created, otherwise the coverage counter is incremented.

If this were the first read to be inserted into a new StarK graph representation, then the resulting graph would look like shown in Figure 9.4.

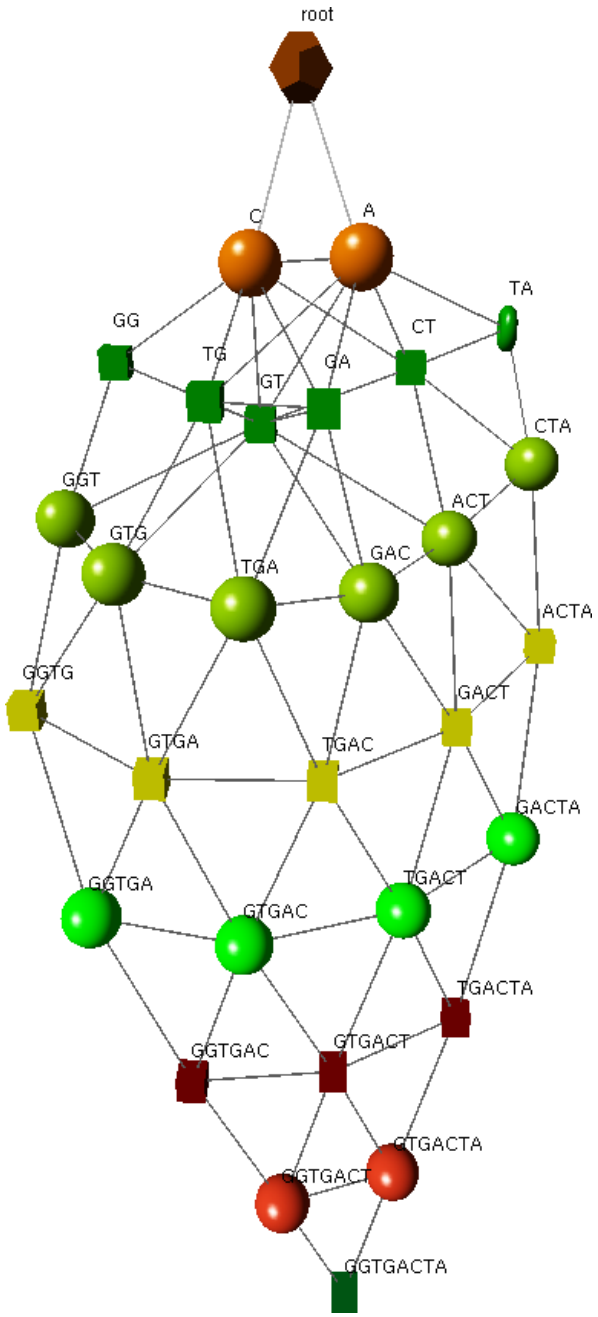


Figure 9.4.: Illustration of a StarK-graph containing only the k-mers of the read `GGTGACTA`.

9.3.2. Parallelisation

Although a StarK graph can be trivially constructed by a single thread, parallel construction of the graph is not straightforward. We use OpenMP [DM98] for parallel dispatching. Still, there are three challenges which cannot be trivially addressed by the middleware:

1. Arrays holding the nodes at each depth have to be enlargeable without interrupting concurrent updates to existing nodes on the same depth.
2. It should be possible to create new nodes without blocking access to their parents.
3. Cumulative updates to the coverage counters updates have scale to an infinite amount of threads.

In the following subsections we will address those challenges and discuss our solutions.

9.3.2.1. Arrays

The objective here was to allow for each array to grow dynamically on use without invalidating any pointers another thread may have acquired into the array.

In the first draft of the implementation we used a data structure similar to C++ `std::vector` [Str13] with a thread lock (or mutex) in front of it. This has proven to scale badly for several reasons:

- Using a readers-writers mutex (a lock that permits multiple concurrent readers, but only one writer) allows for multiple threads to update, but not construct, nodes on the same depth simultaneously. Unfortunately this proved to be slow because in our runs threads switch depths roughly three times faster then it takes to acquire the reader's lock. In addition to that acquiring a writer's lock often took until over half of the active threads went into idle state waiting for a lock themselves.

- Making each depth mutex thread exclusive causes the threads to work in a serial nullifying the benefits of parallelisation.

The shared memory resizeable list (section 10.1) would have solved these issues, but was not used here because most modern Operating Systems (OSes) do not allow for sufficient shared memory to hold our data. Modern Linux is normally only configured with a few hundred megabytes of shared memory, while we require tens of gigabytes.

The final solution we have used was to allocate unfaulted Virtual Memory (VM) pages for as many nodes as we intend to hold and let the kernel provide empty pages on first write. This relies on the fact that `mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);` always provides zeroed-out pages which is the default initialisation state for `starknode_t`. This type of allocation does not use any physical RAM until it is written to, but reserves sufficient space in the processes VM without the need for relocation. This approach requires kernel support for full memory overcommit (which is a far less intrusive configuration than increasing shared memory size and is default on most modern OSes). As a drawback this removes any kind of runtime memory checking and should the process run out of memory it will just crash. For our purposes this is sufficient as the data structure is unusable if it cannot be fully constructed.

9.3.2.2. Node creation

Parallel node creation is not trivial in our case because:

- Two threads need to be prevented from attempting to create the same new node simultaneously,
- This contention has to be resolved once one thread finishes initialising a new node,
- Other threads need to be able to continue working (ideally) on all other existing nodes and should not be prevented from creating other different new nodes even

on the same depth.

The reason why this problem is non trivial is because:

- Two parents need to receive consistent offsets to a new child,
- Having each node carry a mutex that needs to be locked is unfeasible for memory reasons,
- Having a range (depth and/or modulus) mutex will introduce too much contention.

This required us to develop our own localised locking mechanism.

To make sure two threads lock both parents consistently (to avoid a deadlock) each will collect the pointers to the two to four offsets, determine the numerically smaller one and promote this to the primary offset. Offsets start initialised to zero. The thread that is creating a new node will then compare and swap (synchronised processor instruction) a lock value of `0xFFFFFFFF` ((`unsigned int`)−1) into the primary offset preventing any other thread from acquiring it. If this fails this means that another thread has acquired the lock, so this one will wait until a valid child offset is written to the primary. The thread will then proceed to create the new node and write the child offset first to all other offset locations before overwriting the primary offset.

```
input offset_t * child_offsets[] =
      all offsets that need updating in parents;

volatile offset_t * primary_offset = min(child_offsets);

if ( *primary_offset == 0
    && compare_and_swap(primary_offset, 0, -1)) {
   offset_t child = new child();

   for ( all offsets in child_offsets )
      if (offset != primary_offset)
```

```
            *offset = child;

    *primary_offset = child;

} else {
    while (! (valid *primary_offset)) {};
}
```

This locking mechanism does not use any additional memory and only removed one offset from the pool of valid offsets. Also it issues only one hardware lock (plus contention) per new node. Any thread that does not need to access the new node is free to continue updating the parents and any other nodes in the graph.

9.3.2.3. Updating coverage

This may seem simple at first. An atomic update the style of `__sync_fetch_and_add(coverage,1)` (gcc) or `lock inc (%rdx)` (x86-64) makes sure no update is missed, but it introduces a surplus of hardware locking. On a single CPU (tested on Intel® Xeon® CPU E7- 8837) this works without major stalls in the instruction pipeline because Intel CPUs only issues hardware locks to caches when working on a single socket (physical chip). As soon as the program is run across multiple CPUs or over a Non Uniform Memory Architecture (NUMA) link these updates stall execution due to limited bandwidth of the NUMA interconnect and become a major bottleneck. Each instruction with a `lock` prefix will issue cache invalidations on all other CPUs and issue a lock on the NUMA link. We measured that those updates are issued on average every 70 nanoseconds if not stalled. A lock on the NUMA link lasts longer then that, and as such these updates quickly become a scalability problem when deployed across multiple sockets.

After trying various approaches including special hardware accelerated atomic operations (SGI UV Global Reference Unit (GRU)) we have decided to serialise access to

the coverage counters removing the need for locking.

To allow some simultaneous counter updates we divided the counters equally into 16 buckets (can be extended to more). Each bucket has a unique access token which is granted to one thread at a time. Each time a tread wants to update a coverage counter it checks whether it holds the token associated with that bucket, if so it flushes all queued updates in the bucket and then passes the token to the next thread in line. If the thread does not currently hold the token then the update is queued until later.

```
input num_buckets;
input bucket_tokens[num_buckets];
input bucket_queues[num_buckets];

function update_counter(int * counter) {
    int bid = hash(counter) % num_buckets;
    enqueue(bucket_queues[bid], counter);

    if (this thread is assigned to bucket[bid]) {
        flush(bucket_queues[bid]);
        pass_to_next_thread(bucket[bid]);
    }
}
```

This code can be made to run completely lock-free and as such does not cause the problems discussed above. It comes at the cost of keeping bucket queues per thread, but this time and memory overhead dwarfs in comparison to cross-socket locking delays.

9.4. Redundancy cleaning

Although having all k-mers of all reads in a single data structure has its advantages, it also comes at a significant cost in memory. The example in Figure 9.4 requires a total of 30 nodes to represent. For our application (contig assembly) we are not interested in individual reads, but in how we can put them together into longer sequences. The StarK graph in Figure 9.4 has only one surface path — `GGTGACTA`. In order to minimize memory usage we have implemented a procedure that will give us the smallest subgraph of a StarK graph, that contains the exact same surface paths (the theory behind surface paths was developed in subsection 8.2.1).

Essentially this procedure checks for every node whether removing it will create a branch in the new surface path. The only way this can happen is if any of a node's parents have more then one child in either direction or the node itself represents a palindromic k-mer.

This algorithm cannot leave a node "orphaned" (i.e. a node gets deleted that still has children) because

1. We work through the graph starting from the longest k-mers first
2. If a node with the sequence awb of length $k \geq 3$, awb not a genomic palindrome is the only child of its parents aw, wb in the appropriate direction then no node exists of a sequence $awc, c \neq b$ or $dwb, d \neq a$. This means that all children on awb are either $xawb$ or $awbx$ with $x \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.

   Now let us assume that awb has two children $xawb$ and $yawb$, $x \neq y$. But in this case the node aw would have to have at least two children in one direction xaw and yaw which contradicts the requirement for both parents to have at most one child in each direction.

The algorithm in pseudo code:

```
input stark;
```

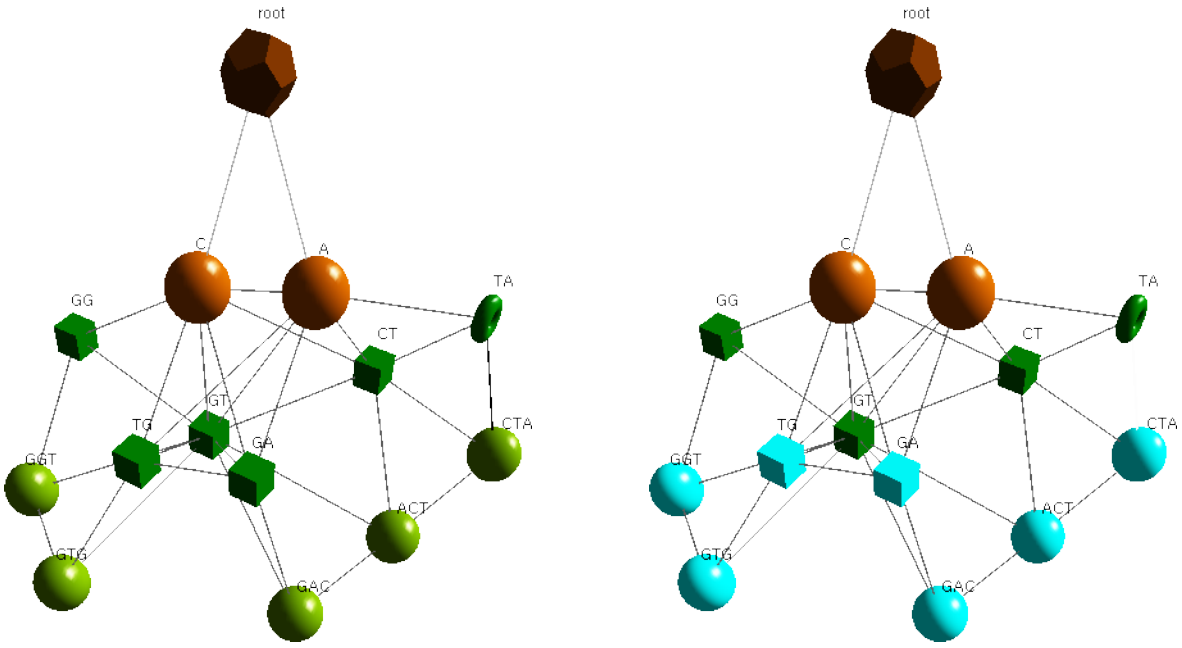


Figure 9.5.: This figure shows the "cleaned" StarK-graph of `GGTGACTA` (Figure 9.4) with all redundant nodes removed. The remaining graph contains exactly one surface path (cyan nodes) at the cost of only 15 nodes.

```
for (i = max(depth); i > 0; i--) {
    for (all nodes n of length i) {
        if (n is
                not a palindrome
            and
                both parents have at most one
                child in either direction) {
            remove n;
        }
    }
}
```

In practice this step often reduces the memory footprint by at least 60%. As an example: when ran on the Ferret 54 day 4 influenza H3N2 sample this step reduced the memory footprint from 56 to 18 gigabytes.

Parallelisation The redundancy clearing step is implemented together with a memory compaction step. The latter is implemented as concurrent parallel tasks.

9.5. Assembly

The sequence assembly step implements the theoretical algorithm described in subsection 8.2.3. Please refer to chapter 8 for a detailed explanation of the algorithm. This section will focus on how we achieve the goals set in the theoretical constructs in practical software.

After all reads have been imported into the StarK graph (section 9.3) and the graph has been redundancy cleaned as described in section 9.4 the next step is trying to approximate the ideal sub-contigs as described in subsection 8.2.3 (please refer there for terminology).

This part consists of three sub-steps:

1. Initialise sub-contigs,
2. Merge sub-contigs,
3. Export assembled sequences.

9.5.1. Initialisation

We initialise a starting set of sub-contigs by dividing the StarK graph into groups of connected nodes such that the internal link strength is less than a set cut off. The default starting link strength cut off is 0.05, but this may be adjusted via a parameter. We use Depth First Search (DFS) to determine inter-node link strength and group nodes into connected sub-graphs with low link strength.

If at any point a constructed group is not a sub-contig (i.e. it does not contain a unique maximal surface path), the group is subsequently split with a lower maximal internal link strength until all sub-groups are sub-contigs.

Algorithm 9.5.1. *The following pseudo code shows approximately the implementation of the initialisation step:*

```
input stark, link_strength_cutoff;
lsc ← link_strength_cutoff;

sub_contigs ← ∅

while (node n (not assigned to) a sub-contig) {
   sub-contig s ← sub_contigs.new();
   s.assign(n);

   for(all nodes m in DFS(n)) {
      if (m (not assigned to) a sub-contig
         && link_strength(s, m) < link_strength_cutoff
         ) {
         s.assign(m);
      }
   }

   if (s does not have a unique surface path) {
      split(s, link_strength_cutoff/2) until it does;
   }
}
```

9.5.1.1. Parallelisation

The code shown in Definition 9.5.1 will in its current state only work reliably when executed sequentially. Although this stage takes only roughly 5% of the run time we still parallelised it because many of the constructs here are used in the next step: Merging sub-contigs.

Note: due to the way the initial sub-contig set is constructed the results are deterministic (except for group IDs) independent of the amount of threads constructing them.

The challenges in parallelisation during this step are:

1. Resolve contention when two (or more) threads attempt to combine the same group simultaneously from different starting points.
2. Allow more then one DFS to be run simultaneously on the same graph (each thread needs its private "seen" flag per node).
3. Allow threads to expand the array holding each group's control structure while other threads work on an existing group. This has been resolved by using our generic list in section 10.1 with shared memory back-end.

In the following paragraphs we will discuss how we tackled the issues 1 and 2 mentioned above.

Group contention resolution. When threads are assigned a starting node, they are only allowed to pick nodes which have not yet been assigned a group id > 0. This eliminates having two threads attempting to use the same starting node.

If during DFS traversal a thread detects that a node is within the link strength limit, but already has been assigned to a different group then the group IDs are compared. The thread processing the other group will eventually come to the same point. The thread with the numerically smaller group ID gets to keep going on and the other aborts essentially allowing a group to "absorb" another half-constructed group. This comes at the cost of limited recalculation, but redundant execution like this is less expensive than synchronising the threads using locks and group merges.

The numerically larger group ID gets recycled by the aborting thread and is reused for a new group.

Parallel DFS. Depth First Search based algorithms require a "seen" flag to be set on every visited graph node in order to avoid going down the same branch twice. When running multiple independent parallel DFS searches with different starting nodes on the same graph, then each thread requires its own private "seen" flag on each node.

Since we cannot afford to allocate any additional memory for each thread on each `starknode_t` (plus synchronisation overhead) we need a method to keep track of this "seen" flag for each thread in a different way.

We solved this issue by giving each thread a private hash map as discussed in section 10.2. This special implementation of a hash map stores pointers to each visited node for fast lookup and insertion. In addition since the constructed groups vary in size from a few nodes to millions of nodes the efficient recycling implementation of this hash map significantly boosts performance. Please refer to section 10.2 for details on design and performance.

9.5.2. Merging sub-contigs

Conceptually this step is trivial. During construction of the initial sub-contig set each group's list of "neighbour" groups (groups where there is at least one edge between them connecting nodes from different groups) is already created. These pairs of groups are sorted in ascending inter-group link strength and each pairing is then tested to see whether a merge results in a new sub-contig that is longer than both original sub-contigs.

This can then be repeated in multiple rounds, recalculating the pairings before each round, until no more merges are possible.

Algorithm 9.5.2. *Basic pseudo code algorithm for the merge step:*

```
input sub_contig_set;
do {
   adjacent_pairings ← pairings of adjacent groups
                                  from sub_contig_set;
   adjacent_pairings.sort(by link strength, ascending);

   for (each pair p in adjacent_pairings) {
      if (merge of groups in p gives longer sub-contig) {
         merge groups in p;
      }
   }
} while (at least one merge was done)
```

9.5.2.1. Parallelisation

This part of the software requires roughly 70% of the total CPU time, so parallelisation of this step is crucial for performance.

The biggest challenge during parallelisation of this step was to preserve a deterministic output while maximising the amount of parallel processing.

To achieve this we had to determine which pairings are dependent on a previous pairing succeeding or failing before they can be tested. As soon as we can detach independent pairings which are guaranteed to be tested independently, we are able to run those tests in parallel.

To achieve this we have divided the threads into two categories:

1. One dispatcher,
2. Workers.

The dispatcher thread is responsible for determining independent lists of pairings which can be executed by the workers in parallel, keeping track of which groups are being tested by which worker and recycling the workers' testing queue.

The concept for determining pairing test independence is simple:

A pairing $(g_1, g_2, l(g_1, g_2))$ (where g_i are group IDs and $l(g_1, g_2)$ is the square of their inter-group link strength, please compare to Definition 8.2.11) is independent from a pairing $(g_3, g_4, l(g_3, g_4))$ with $l(g_1, g_2) < l(g_3, g_4)$ if and only if

- g_1, g_2, g_3, g_4 are pairwise different,
- There is no chain of pairings $(g_{i_1}, g_{j_1}, l(g_{i_1}, g_{j_1})) \dots (g_{i_n}, g_{j_n}, l(g_{i_n}, g_{j_n}))$ such that $l(g_1, g_2) \leq l(g_{i_1}, g_{j_1}) \leq \dots \leq l(g_{i_n}, g_{j_n})$ and $g_{i_1} \in \{g_1, g_2\}, g_{j_n} \in \{g_3, g_4\}$. In essence if merging any pairings between those will affect the last pairing.

In practice we do it the following way:

- Each worker thread receives a queue of pairing to process.
- The dispatcher iterates the list of sorted pairings one by one. For each pairing it tests:
  - Are both groups in this pairing not yet assigned to any groups. If so, assign it to a worker with the least load.
  - If both groups are assigned to the same worker or one is assigned and the other is not yet assigned – assign this pairing to that worker.
  - If both groups are assigned to different workers, mark both groups as being invalid for any further assignments.

  This ensures that the same merges are done as if pairings would be tested in the same order as they would be if worked through sequentially.
- The dispatcher periodically tests whether a worker's queue is empty. If so, it releases all group assignments, reassigns groups that are still in other workers' queues and rewinds the previous step. This ensures that load is balanced through-

out most pairing tests. In our trial runs on sample Ferret 54 day 2 all workers were busy 90% of the time.

9.5.3. Exporting contigs

Once the merging step has been run for (either a fixed number of rounds or) as many rounds as it can be the remaining groups are tested for their contig length and contigs longer than a threshold (set via runtime parameter) are printed in FASTA format. This step was intended to include alignment comparisons of the contigs in order to map variants to each other, but was not implemented due to lack of time.

No parallelisation was done on this part as it does not require much CPU time compared to the other steps.

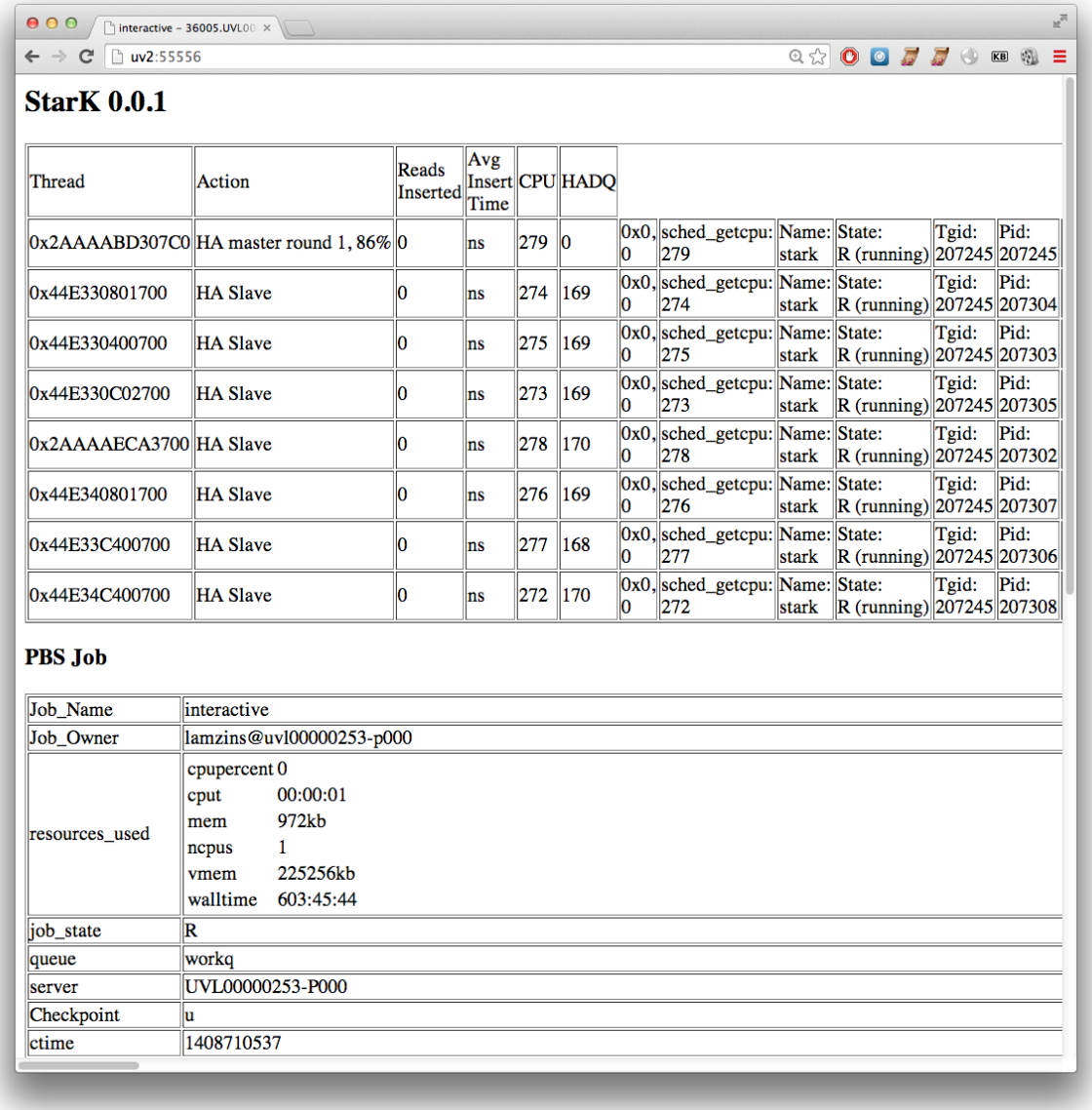
9.6. Monitoring

Our implementation of StarK provides a built-in minimal web interface for progress monitoring. The interface was initially developed for debugging purposes, but is available to anyone and provides information like:

1. How many threads are running and their status.
2. Sequence statistics, current assembly progress.
3. Thread scheduling, dispatched CPU, memory usage and other diagnostic statistics.
4. If StarK was compiled with `libpbs` support and is running on a Portable Batch System (PBS) scheduler then PBS job statistics are shown in the web interface.

Figure 9.6 shows a screenshot of the status website on a running assembly process. The plan was to extend the web interface to allow the user to alter execution parameters, but this was never implemented because of time constraints.

Access to the web interface is provided through a Transport Control Protocol (TCP) port that can be set as a parameter at start or through a UNIX domain socket, which is created automatically in the user's `TEMP` directory.



interactive - 36005.UVL00

uv2:55556

StarK 0.0.1

Thread	Action	Reads Inserted	Avg Insert Time	CPU	HADQ						
0x2AAAABD307C0	HA master round 1, 86%	0	ns	279	0	0x0, 0	sched_getcpu: 279	Name: stark	State: R (running)	Tgid: 207245	Pid: 207245
0x44E330801700	HA Slave	0	ns	274	169	0x0, 0	sched_getcpu: 274	Name: stark	State: R (running)	Tgid: 207245	Pid: 207304
0x44E330400700	HA Slave	0	ns	275	169	0x0, 0	sched_getcpu: 275	Name: stark	State: R (running)	Tgid: 207245	Pid: 207303
0x44E330C02700	HA Slave	0	ns	273	169	0x0, 0	sched_getcpu: 273	Name: stark	State: R (running)	Tgid: 207245	Pid: 207305
0x2AAAAECA3700	HA Slave	0	ns	278	170	0x0, 0	sched_getcpu: 278	Name: stark	State: R (running)	Tgid: 207245	Pid: 207302
0x44E340801700	HA Slave	0	ns	276	169	0x0, 0	sched_getcpu: 276	Name: stark	State: R (running)	Tgid: 207245	Pid: 207307
0x44E33C400700	HA Slave	0	ns	277	168	0x0, 0	sched_getcpu: 277	Name: stark	State: R (running)	Tgid: 207245	Pid: 207306
0x44E34C400700	HA Slave	0	ns	272	170	0x0, 0	sched_getcpu: 272	Name: stark	State: R (running)	Tgid: 207245	Pid: 207308

PBS Job

Job_Name	interactive
Job_Owner	lamzins@uvl00000253-p000
resources_used	cpupercent 0 cput 00:00:01 mem 972kb ncpus 1 vmem 225256kb walltime 603:45:44
job_state	R
queue	workq
server	UVL00000253-P000
Checkpoint	u
ctime	1408710537

Figure 9.6.: Example screenshot of the StarK status monitoring web page. The assembler finished loading a previously generated snapshot of a StarK graph and is in its first round of merging sub-contigs.

9.7. Performance Test

We ran our StarK prototype implementation on the Ferret 54 day 4 influenza H3N2 sample to determine its performance. We used the Allinea Map (`http://www.allinea.com/products/map`) 5.0-40932 profiler to determine the activity of the threads spawned by the program and the memory usage curve.

Figure 9.7 shows the CPU and memory usage of the profiling run while Table 9.8 shows the timings for the individual steps.

The green line in Figure 9.7 shows CPU usage averaged over the 8 cores through a full assembly run while the red line shows memory usage. One can clearly see that most of the time is spent in parallel sections where all cores are busy. The two longest episodes of parallel work are the step that builds the StarK graph from the reads (in the first third) and the assembly part (in the second half) are both executed fully in parallel. The first long parallel section has a little jitter in the CPU usage as there are sometimes wait times due to synchronisation routines. The second parallel section (the assembly part) has occasional short breaks in parallelism at the end of each round that merges contigs where threads need to resynchronise. Even though the redundancy cleaning step (where the drop in memory occurs) is parallel, the amount actual of parallelism is minimal.

The red line shows memory usage. It shows the build up of memory while the StarK graph is built and the sharp drop during the redundancy cleaning step. In this particular run the cleaning saved over 60% of the memory. The memory build-up during the assembly step is only marginal.

The assembler performed the steps detailed in Table 9.8 on 8 cores on a Intel(R) Xeon(R) CPU E5-4650L 0 @ 2.60GHz (Check mark indicates that this step is capable of running in parallel across all 8 cores).

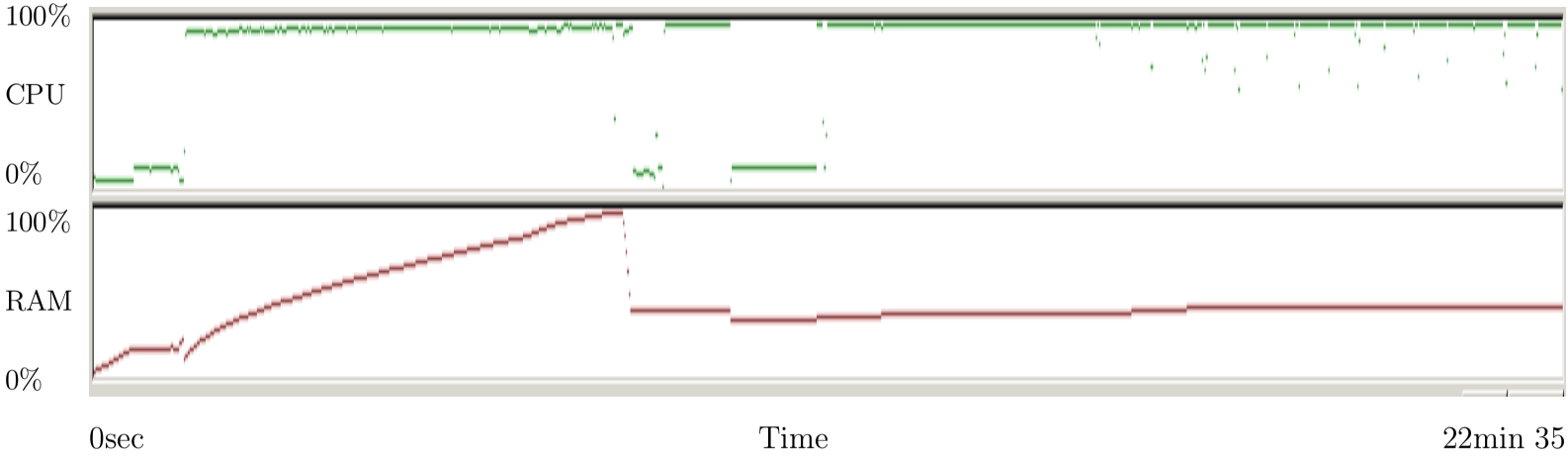


Figure 9.7.: Shows The CPU and memory usage charts usage. The green line shows CPU usage averaged over the 8 cores through a full assembly run while the red line shows memory usage. One can clearly see that most of the time is spent in parallel sections where all cores are busy. The red line shows memory usage one can clearly see the build up of memory while the StarK graph is built and the sharp drop during the redundancy cleaning step.

Task	Parallel	Real time
Read I/O (read sequences)	X	14sec
Remove duplicates	X	18 sec
Build StarK graph	✓	6min 33 sec
Redundancy cleaning & compact memory	X	126sec
Compute statistics	✓	1min 36sec
Prepare graph for assembly	X	1min 33 sec
Merge contigs Round 1	✓	4min 24 sec
Merge contigs Round 2	✓	57 sec
Merge contigs Round 3	✓	32 sec
Merge contigs Round 4	✓	30 sec
Merge contigs Round 5	✓	30 sec
Merge contigs Round 6	✓	33 sec
Merge contigs Round 7	✓	30 sec
Merge contigs Round 8	✓	33 sec
Merge contigs Round 9	✓	30 sec
Merge contigs Round 10	✓	30 sec
Merge contigs Round 11	✓	30 sec
Merge contigs Round 12	✓	30 sec
Merge contigs Round 13	✓	30 sec
Merge contigs Round 14	✓	31 sec
Merge contigs Round 15	✓	30 sec
Total		22min 35 sec

Table 9.8.: The prototype assembler ran on on 8 cores on a Intel(R) Xeon(R) CPU E5-4650L 0 @ 2.60GHz assembling the Ferret 54 day 4 influenza H3N2 sample. Please note that the timings are different from those in Table 6.5 because the same machine the initial tests were conducted on was unavailable for further profiling.

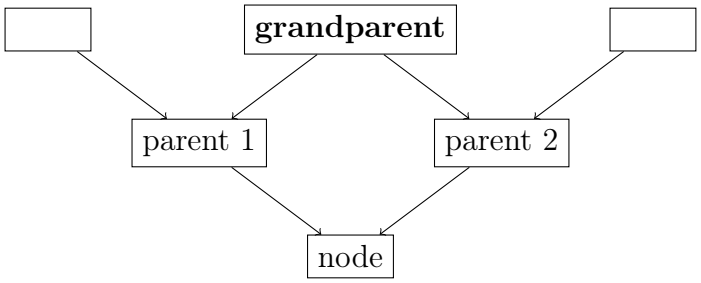


Figure 9.9.: Shows node → parents → grandparent relation within the StarK graph.

9.8. Compressed data structure

The `starknode_t` (subsection 9.2.1) is a robust data structure for the purposes of our implementation and provides superior performance when deployed in a parallel environment, but comes at the expense of 48 bytes per node (two k-mers). When importing all the reads from our training sample (human Influenza H3N2 from ferret 54 day 2) peak RAM usage was 56 gigabytes. The majority of the space (83%) is used up for parent and child offsets within the struct.

Benchmarks have shown that out of the eight entries for child offsets only two are used in over 92% of the nodes within the graph.

Additionally we have observed that a full 32 bits for child offsets may not be necessary as grouping nodes that share certain features within memory may give us the ability to decrease this offset size. The common feature that we will be exploiting here is that although each node has two distinct parent offsets, making offsets based on parent unfeasible, both of those share one common (grand)parent.

Definition 9.8.1. *Let awb be a k-mer in the genomic alphabet. $a, b \in \Gamma, w \in \Gamma^*$. The parent nodes are representatives for the $(k-1)$-mers aw and wb respectively. We will be calling the common ancestor of those (node that is representative for the $(k-2)$-mer w) the **grandparent**. See Figure 9.9 for clarification.*

Combining the fact that in the majority of nodes only two child offsets are necessary

and that we can use part of a grandparent's offset for addressing a node led us to the design for a compressed data structure – StarK node version 2. Please note that in its current state this new data structure is only capable of delivering the first two phases of the StarK assembly process – importing reads into the StarK graph and redundancy cleaning. Due to this it is not yet used by the main algorithm, but will eventually replace the base `starknode_t`. A temporary solution is to convert the StarK node version 2 graph into a `starknode_t` based graph after the redundancy cleaning step, thus avoiding the peak memory usage of a `starknode_t` based graph before redundancy cleaning.

9.8.1. Design

The design of what we internally call `stark_node_phase1` (StarK node for phase one) is divided into two structs: a small version and an extension.

The small node has one of two states:

1. It stores two child offsets directly.
2. It was extended and stores the location of the extension, which in turn contains up to eight child offsets.

Base state 1. In the base state where a `stark_node_phase1` small node stores only two child offsets it holds enough data to store the familiar `edges` (8-bit) field from `starknode_t`, a new `flags` (8-bit) field, two 20-bit child offsets and two 4-bit identifiers for those children.

If a small node receives a third child, then an extension is created, flags are changed to indicate the presence of an extension and the two times 24 bit, which previously stored child data, are re-purposed to hold the offset of the extension struct.

The small node struct is just 8 bytes in size and (according to earlier mentioned benchmarks) will be sufficient to represent 92% of nodes. The remaining nodes will addi-

bytes	bits 7	6	5	4	3	2	1	0	bits
0x00	lock	c1 direction	c1 nucleotide		c1 offset[16..19]				8
0x01	lock	c2 direction	c2 nucleotide		c2 offset[16..19]				16
0x02	flags: n/a		palindrome	reverse	first nucleotide		last nucleotide		24
0x03	edges, at most two bits set								32
0x04 0x05	child 1 offset[0..15]								48
0x06 0x07	child 2 offset[0..15]								64

Table 9.10.: Data layout of `struct stark_node_phase1_small_s` with up to two offsets. c stands for child.

tionally have an extension struct of eight times 20 bits = 20 bytes. On average this brings us to less than 10 bytes per node. This is a significant improvement in storage in comparison with the original `starknode_t` structure, which requires 48 bytes per node – an improvement of almost 500% in memory efficiency.

Table 9.10 shows the data layout of the small node (`struct stark_node_phase1_small_s`) with up to two offsets.

The child offsets are split over multiple non-adjacent fields. The four least significant bits of the first two bytes hold the most significant bits of the offsets and the last two words (2 bytes in Intel terms) hold the 16 least significant bits. The bits 4, 5 and 6 in each of the two first bytes identify which of the 8 possible children are stored.

The first two bytes also function as a lock indicator in their most significant bit for parallel execution. A value of `0xFFFF` in the two first bytes (a value that does not make sense for child storage) indicates that another thread is currently modifying this node.

20 bits are by far not enough to store sufficient nodes (recall that our training sample

required 22 bits of offsets). This is why we splice the final child offsets together from the common grandparent's (refer to Figure 9.9) offset (up to 20 bits may be used) and the offset stored within the node. The resulting final offset is calculated as

```
final_offset =  ((grandparent_offset & 0xFFFFF) << 20
                | child_offset) & mask;
```

where `mask` is an offset mask calculated depending on the runtime's available VM space and the depth (small depths can not produce sufficient nodes to justify large offsets). On our SGI UV this was set to up to 38 bits, which is a significant improvement over `starknode_t` maximum of 32 bits. It is also way above anything current hardware can actually accommodate. When virtual memory address space will be increased (currently 48 out of 64 bits), this maximum final offset length will rise to 40 bits, which is orders of magnitude more then is needed for today's NGS sequencing samples, not to mention way beyond any current hardware.

Extended state. When a `stark_node_phase1` small node would receive a third child it is instead extended. An extension struct (eight times 20 bits) is created and the small node is altered to hold the extension's 48-bit offset in place of the memory previously occupied by the two prior children. At this point all child offset information is outsourced to the extension.

The two states are differentiated by the amount of bits set in the edges field. More than two bits set indicate the presence of an extension. Table 9.11 shows the data layout for this state.

As explained above, this two-stage approach to storing StarK nodes significantly improves memory footprint due to the fact that in practice often not more than two children are attached to each node.

bytes	bits 7	6	5	4	3	2	1	0	bits
0x00	extension offset[32..48]								
0x01									16
0x02	flags: n/a	palindrome	reverse	first nucleotide	last nucleotide				24
0x03	edges, more than two bits set								32
0x04	extension offset[0..32]								
0x05									
0x06									
0x07									64

(a) Data layout of `struct stark_node_phase1_small_s` in the extended state.

bytes	byte: quartet	quartet	byte: quartet	quartet	bits
0x00	c0 offset[16..19]	c4 offset[16..19]	c1 offset[16..19]	c5 offset[16..19]	16
0x02	c2 offset[16..19]	c6 offset[16..19]	c3 offset[16..19]	c7 offset[16..19]	32
0x04	child 0 offset[0..15]				48
0x06	child 1 offset[0..15]				64
0x08	child 2 offset[0..15]				80
0x0A	child 3 offset[0..15]				96
0x0C	child 4 offset[0..15]				112
0x0E	child 5 offset[0..15]				128
0x10	child 6 offset[0..15]				144
0x12	child 7 offset[0..15]				160

(b) Data layout of `struct stark_node_phase1_extension_s` – the small node extension.

Table 9.11.: Combined data layout of `struct stark_node_phase1_small_s` in the extended state with its extension.

9.8.2. Parallel read parsing

As with section 9.3 we have taken into account parallelisation when designing this data layout. `starknode_t` uses local per-offset locking when children are being added. Although this allows for two threads to append two different new children to the same node, in practice this almost never happens. Also because the two data fields for child offsets in a `stark_node_phase1` small node are no longer associated with specific children, the entire node has to be locked down during addition of new children.

Taking this into account we have specifically designed the `stark_node_phase1` small node in such a way that it is only 8 bytes long and can thus be loaded atomically by a modern 64-bit CPU (memory alignment provided). Thus each time a thread wishes to modify a `stark_node_phase1` small node, it loads it atomically onto its stack, performs updates and then pushes the full struct in an atomic compare-and-swap back into memory. This guarantees a fully atomic update of the entire struct in just one write.

This method ensures that only a single locked instruction (compare-and-swap) is used to limit memory bus lock down and that the struct is always kept in a consistent state without the use of locks and unnecessary strain on the hardware memory bus.

When a small node is extended, it can no longer be updated with just one write. At this point an internal locking mechanism is used. This is acceptable due to the significantly smaller amount of extended nodes.

9.8.3. Node meta data

The reader may have noticed that the `stark_node_phase1` small node struct has no fields for meta data (e.g. coverage information). This type of data is independent of the actual structure of the StarK graph representation in memory. As such we have decided to move this part to a separate array. Any meta data is now held in separate arrays that are addressed with the same offset as the `stark_node_phase1` small node itself and thus uniquely identified. Our current implementation supports coverage data only, but can easily be extended to any additional information as it becomes necessary.

The idea behind this is to run the graph creation and redundancy cleaning steps before any memory for meta data is commissioned.

10. Libraries

During the implementation of StarK we encountered the need for several helper libraries with specific requirements that were unavailable otherwise. Below are the helper libraries listed that we implemented and the reasoning behind them.

10.1. Generic Lists

Relatively early in development we started to require a list type structure similar to the C++ `std::vector` class [Str13].

The reasons for not using an off-the shelf implementation were:

- We wanted to rely on a custom allocator making use of Linux `realloc` and Mach `vm_remap`,
- We require support for atomic size increments,
- We require (in some use cases) the ability to enlarge/reallocate the memory region to allow for new elements **without** invalidating any pointers another thread may have acquired into the array.

Since C does not directly support template/generic programming we have implemented much of this in preprocessor macros.

The resulting library is self-contained in just one header `list.h` and implements the following functions:

`#define list_t(type)`

Macro that unfolds into a struct representing the container meta data. Used as the type for the list. Provides access to the members

- `type * list;` the pointer into the array that holds the list contents.
- `size_t size;` Current amount of elements in the list. Setting this to `0` effectively erases the list.

`list_init(list_t *)`

Simple list initializer, also available with optional parameters initial `size` and `flags`.

`list_init_size(list_t *, size_t)`

`list_init_flags(list_t *, int)`

`list_init_size_flags(list_t *, size_t, int)`

So far the only implemented flag is thread safe memory expansion (see below).

`list_free(list_t)`

List deinitialiser, deallocates the underlying array and any helper memory. The list meta structure is unmodified and should not be used unless reinitialised.

`list_new_empty(list_t *)`

Makes sure that sufficient memory is available for a new element in the array, zeroes it out and returns its index.

`list_insert(list_t *, object)`

Makes sure that sufficient memory is available for a new element in the array and copies the passed object in. Types of the list generic and the object have to match at the penalty of unpredictable results.

`list_push(list_t *, object)`

Alias for `list_insert` for stack semantics.

`list_pop(list_t *)`

Decrements list size for stack semantics. No consistency checking is done and the popped element is not zeroed out. Users should make sure that `list->size > 0` before calling this.

`list_compact(list_t *)`
: Compacts the underlying allocated array to use only exactly as much memory as necessary within the bounds of the used allocator.

`list_qsort(list_t *, int (*compare)(const void *, const void *))`
: Quicksort convenience macro, calls libc `qsort`.

`list_heapify(list_t *, int (*compare)(const void *, const void *))`
: Heapifies the list for use as a heap with array backbone.

Most of the above functions are implemented as preprocessor macros for fast inlining.

Five memory allocator models are supported by the implementation, three of which are fully implemented and tested. The Mach-specific allocators are implemented in a testing build, but not yet fully tested and are not currently distributed with the code. The allocators are:

1. `libc malloc, realloc`
2. POSIX `mmap` with Linux `mremap`
3. Mach (OS X) `vm_alloc`, `vm_remap`
4. POSIX shared memory
5. Mach shared `vm_remap`

The first three are selected transparently within the initialiser, the fourth and fifth ones are necessary for a special feature discussed below.

`list_init` allows a flag to be passed to it which sets the list up in such a way that `list_insert` and `list_new_empty` can concurrently enlarge the array while another thread is modifying existing elements. On Linux this effect is achieved by binding an instance of `list_t` to a shared memory file descriptor (memory model 4). Each time the array is enlarged a new VM mapping is created from the underlying shared memory preserving the previous one. This permits other threads to continue working on the obsolete mapping while a new larger one is available at the next access. Sadly

this comes with a limitation: OS wide maximum shared memory (as discussed in subsubsection 9.3.2.2). On Mach-based VM implementations (OS X) this problem can be avoided by using the Mach `vm_remap` without binding the memory to a file descriptor (memory model 5).

10.2. Hash maps

During various stages of the StarK implementation we encountered the requirement for a hash map with special requirements:

- Space efficient.
- Map has to store only the key (which in itself is an integer) and thus needs only to say whether a key is present or not. A key of zero (`0`) is invalid.
- Needs to support the usual hash map features of fast lookup, insertion and resizing.
- Needs to support a fast zeroing out operation which is called frequently on sparsely populated maps.

The reasoning behind the above requirements is that the hash map will be used primarily by a re-entrant graph traversal function that needs to store visited node flags. Thus every thread will receive a private copy of such a hash map which will be recycled on every new function call. The map stores addresses of visited nodes. Thus the ability to quickly zero everything is essential.

Since only integers are stored and zero values are invalid we do not need additional meta data to store in the hash map container. As the underlying contention resolution we have chosen chained hashing and limited the map's load factor to $\frac{1}{2}$ (although usually values around 0.7 are recommended [Knu73]).

The default hash function for integer types is the identity function. As the chaining hash function we have used

$$h : x \mapsto x + 1 \mod [\text{map size}]. \tag{10.2.1}$$

Although a more sophisticated hash function can be used for increased spread, this was sufficient during our tests due to the nature of our input data (VM addresses) being

relatively uniformly distributed in the first place.

We identified that it is a significant performance bottleneck when the functions that use this special hash map often only sparsely populate it before requiring a new one.

We benchmarked different approaches for recycling the memory. Table 10.2 shows the results of those benchmarks.

The slowest was to deallocate and reallocate a completely new map using `munmap/mmap`. This is because it involved to two system calls and the new memory needs to be faulted in on use.

The first attempt which gave a small performance increase was to remap fresh anonymous VM on top of the old ones using `mmap` with the `MAP_FIXED` flag to overwrite the existing mapping. Although this reduced the amount of system calls to one and performed adequately if always only few keys were inserted, but was still very slow if most of the map was actually used. This is due to the cost associated with faulting in new pages on each use.

At this point we have divided the memory used by the map into buckets of VM page size (this can be altered if necessary). Each time a bucket is written to a flag is set that this bucket is dirty (non-zero). When the map has to be zeroed out we will then only zero out the dirty buckets.

This has two main advantages:

- Pages of large maps with only few entries that remained unused do not have to be touched.
- Pages do not have to be faulted in again. Surprisingly on our test Systems (Linux Intel® Xeon® E7- 8837, OS X Intel® Core™ i5-2435M) faulting a new empty page in is slower than zeroing a dirty page with `memset`. This may be due to expensive context switches.

We note that the most consistent way of using the Direct Memory Access (DMA) controller to fulfil this job is yet to be found.

Fill amount	Method	Map size						
		4KiB	8 KiB	16KiB	32KiB	64KiB	128KiB	256KiB
1‰	`u;m`	6ms	7ms	12ms	32ms	182ms	59ms	116ms
	`m/F`	6ms	6ms	9ms	17ms	29ms	72ms	163ms
	`i/s`	**730μs**	**1344μs**	**1663μs**	**3ms**	**6ms**	**20ms**	**27ms**
2‰	`u;m`	6ms	10ms	11ms	20ms	35ms	63ms	119ms
	`m/F`	4ms	5ms	9ms	79ms	40ms	59ms	106ms
	`i/s`	**390μs**	**744μs**	**1507μs**	**3ms**	**6ms**	**15ms**	**36ms**
3‰	`u;m`	7ms	6ms	12ms	29ms	42ms	87ms	145ms
	`m/F`	7ms	9ms	10ms	17ms	31ms	66ms	**125ms**
	`i/s`	**542μs**	**1000μs**	**2ms**	**4ms**	**8ms**	**19ms**	197ms
5‰	`u;m`	9ms	12ms	25ms	17ms	39ms	67ms	119ms
	`m/F`	4ms	6ms	12ms	22ms	40ms	62ms	127ms
	`i/s`	**807μs**	**2ms**	**4ms**	**7ms**	**19ms**	**28ms**	**57ms**
7‰	`u;m`	4ms	7ms	13ms	24ms	38ms	216ms	140ms
	`m/F`	4ms	10ms	13ms	25ms	46ms	77ms	124ms
	`i/s`	**954μs**	**1926μs**	**3ms**	**8ms**	**17ms**	**39ms**	**73ms**
1.1%	`u;m`	5ms	6ms	11ms	22ms	40ms	78ms	161ms
	`m/F`	6ms	9ms	12ms	22ms	41ms	72ms	142ms
	`i/s`	**1476μs**	**2ms**	**5ms**	**15ms**	**25ms**	**53ms**	**99ms**
1.7%	`u;m`	5ms	7ms	13ms	26ms	56ms	90ms	166ms
	`m/F`	6ms	9ms	14ms	270ms	44ms	85ms	168ms
	`i/s`	**2ms**	**4ms**	**8ms**	**19ms**	**43ms**	**82ms**	**152ms**
2.5%	`u;m`	6ms	9ms	18ms	34ms	204ms	108ms	245ms
	`m/F`	5ms	9ms	15ms	29ms	52ms	120ms	198ms
	`i/s`	**3ms**	**6ms**	**15ms**	**67ms**	**65ms**	**107ms**	223ms
3.8%	`u;m`	6ms	10ms	20ms	**33ms**	**71ms**	**112ms**	**226ms**
	`m/F`	5ms	**9ms**	**18ms**	**33ms**	240ms	128ms	233ms
	`i/s`	**4ms**	**9ms**	24ms	43ms	83ms	164ms	482ms
5.7%	`u;m`	7ms	**15ms**	**22ms**	**43ms**	**75ms**	151ms	307ms
	`m/F`	27ms	17ms	24ms	49ms	76ms	**143ms**	**282ms**
	`i/s`	**6ms**	19ms	31ms	60ms	120ms	423ms	482ms
8.6%	`u;m`	**8ms**	**15ms**	30ms	**55ms**	**95ms**	322ms	**377ms**
	`m/F`	10ms	16ms	**29ms**	56ms	103ms	**214ms**	410ms
	`i/s`	10ms	24ms	48ms	87ms	208ms	584ms	882ms

Table 10.2 *(previous page)*: Benchmarking of three methods to zero sparsely used memory arrays. Fill amount is the amount of non-zero bytes in the map. Methods are: `u;m` – `munmap` the mapping and remap it using `mmap`, `m/F` – remap the mapping using `mmap` with `MAP_FIXED` flag overwriting the previous mapping, `i/s` – keep a bookkeeping bitfield indicating dirty pages and set those to zero using `memset`. The bold entry marks the fastest approach for each fill amount/map size.

The resulting library is contained in a single header `hashmap.h` and implements the following functions:

`#define hashmap_t(type)`
: Macro that unfolds into a struct representing the container metadata. Used as the type for the hash map. `size_t size;` is the only member that should be used and holds the current count of members in the map.

`map_init(hashmap_t *)`
: Simple map initializer, also available with an optional parameter specifying the initial map `size`.

`map_free(hashmap_t *)`
: Hash map de-initialiser, deallocates the underlying map and any helper memory. The map meta structure is unmodified and should not be used unless reinitialised.

`map_free(hashmap_t *, size_t size)`
: Shrinks the memory container for the map to the specified value. Only usable on an empty map.

`map_insert(hashmap_t *, uint64_t object)`
: Inserts a new non-zero integer value into the hash map. Expands the underlying storage if necessary.

`map_get(hashmap_t *, uint64_t object)`
: Test whether an object is in the map. Returns a copy of the object on success, zero on failure.

`map_zero(hashmap_t *)`

Empties the map as discussed above. The underlying memory container is retained for new use.

Additionally any user may supply their own hash function (required for objects that are non-integer types) by defining the macro `map_hashfunction` to the name of their own hash function of type `int64_t (*)(type)`.

11. Discussion

De novo assemblies of highly variable deep sequencing data such as populations of viral genomes or metagenomics remain a substantial challenge. We have assessed the theoretical frameworks of modern NGS short read assemblers and evaluated several implementations of those on virus population data. Having established that the current methods have inherent shortcomings due to the theoretical design, we have designed a new approach (multi dimensional *de Brujin* graphs) to tackle this problem.

The current theoretical framework for assembling genomes *de novo* from short reads have all been primarily designed with single individual (typically diploid) targets in mind. While the tools implementing those frameworks perform well on their intended targets, the need for assemblies of more diverse sequencing samples has emerged. Several attempts have already been made in either adapting the existing theoretical frameworks for the new type of data (e.g. PRICE assembler) or creating *Frankenstein* assemblers by pooling together pieces of sequences generated by different assemblers in an attempt to get a better joined assembly (e.g. Trinity assembler [Gra+11]).

Although all those approaches are justified, we have come to the conclusion that a new theoretical framework is needed, which specifically targets the shortcomings of existing ones. Especially useful in this case is retention of as much information as the individual reads can provide for the purpose of assembly which allows for more diverse datasets to be efficiently dealt with. The StarK assembler is a solid extension of existing *de Brujin* graph based approaches, but allows for additional flexibility for the assembler, which was not possible with previous theoretical frameworks at the same time complexity.

Our prototype implementation of the StarK assembly framework performs well on our training sets and has outperformed any other assembler that we tried (Table 6.5). The implementation is fully parallelised for over 90% of the program's runtime and scales to additional CPUs better then any other assembler that we tried. We have also attempted to assemble other genomes *de novo* with varying success. We obtained a decent assembly of yeast with contigs, that were short (2 - 5kb), but aligned against the reference and a not very useful assembly of *E.coli*. The latter was unsuccessful because StarK was not designed to handle the 32bp reads of the SRR001665 *E.coli* read set, but intended for use with read lengths of 80bp and longer.

The design goal behind both the StarK theoretical framework and the implementation was to achieve maximum

- accuracy in constructing contigs,
- speed when deployed on parallel computing architecture.

Both of those goals have been achieved at the cost of high main memory (RAM) consumption. Assembly of our IAV training set required 56GiB of RAM. Although modern hardware can easily handle this, the RAM consumption scales poorly to larger read sets. The compressed data structure (as discussed in section 9.8) was designed in an effort to address this issue. Preliminary results are promising, but it requires more testing and integration into our software beyond the first stage of the algorithm. We have also considered other approaches to reduce memory usage like partitioning of the data or reducing the number of stored StarK nodes at lower depths. Both of those approaches need further exploration.

In its current state the prototype implementation assembles useful contigs, runs very fast and scales well to more hardware. More work is still needed as

- contig haplotyping/mapping of variant contigs onto each other is not yet implemented,
- the former would benefit from output in FASTG file format (`http://fastg.sourceforge.net`)

- heuristics for creation of longer contigs need to be improved as we are confident that our fear of creating chimeric contigs led to a too conservative algorithm when it comes to merging sub-contigs,
- in its current state paired-end read information is not used yet and long mate pairs cannot be reliably assembled,
- support for reference/long read assisted assemblies is planned and will increase assembly accuracy significantly if added,
- the compressed data structure has only been implemented for the first stage of the assembly (importing reads) and still needs to be ported to the rest of the algorithm in order to begin making use of it's five fold memory conservation,
- explore the possibilities of partitioning the data in order to reduce memory usage.

The StarK theoretical framework provides a solid and powerful foundation for general and high variation population genome assembly. The prototype implementation has already demonstrated the ability to generate good assemblies efficiently. We are confident that with more work (listed above) we can turn our prototype implementation into a very powerful genome assembler, not only for viral, but also general purpose and possibly even metagenomics assemblies.

The current source code for our prototype implementation is available at `https://github.com/sergeylamzin/stark` and is licensed under GNU Public Licence (GPL) version 3. We welcome external contributions of forks of the program.

Bibliography

[Alt+90] Stephen F Altschul et al. "Basic local alignment search tool". In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.

[Bak+13] Kate S Baker et al. "Metagenomic study of the viruses of African straw-coloured fruit bats: detection of a chiropteran poxvirus and isolation of a novel adenovirus". In: *Virology* 441.2 (2013), pp. 95–106.

[Ban+12] Anton Bankevich et al. "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing". In: *Journal of Computational Biology* 19.5 (2012), pp. 455–477.

[Bar05] John M Barry. *The great influenza: The story of the deadliest pandemic in history*. Penguin, 2005.

[Bat+02] Serafim Batzoglou et al. "ARACHNE: a whole-genome shotgun assembler". In: *Genome research* 12.1 (2002), pp. 177–189.

[Ben+05] Simon T Bennett et al. "Toward the 1000*humangenome*". In: (2005).

[Ben06] David R Bentley. "Whole-genome re-sequencing". In: *Current opinion in genetics & development* 16.6 (2006), pp. 545–552.

[BP08] Nicole M Bouvier and Peter Palese. "The biology of influenza viruses". In: *Vaccine* 26 (2008), pp. D49–D53.

[Bra+13] Keith R Bradnam et al. "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species". In: *GigaScience* 2.1 (2013), pp. 1–31.

[Bus02] SA Bustin. "Quantification of mRNA using real-time reverse transcription PCR (RT-PCR): trends and problems". In: *Journal of molecular endocrinology* 29.1 (2002), pp. 23–39.

[CB13] Ayling S. Caccamo M Clavijo B. Mapleson D. "KAT—Kmer Analysis Tool". Manuscript submitted for publication. 2013.

Available online at: `http://www.tgac.ac.uk/kat`.

[CC76] Louise Clarke and John Carbon. "A colony bank containing synthetic CoI {EI} hybrid plasmids representative of the entire E. coli genome". In: *Cell* 9.1 (1976), pp. 91 –99. ISSN: 0092-8674. DOI: `http://dx.doi.org/10.1016/0092-8674(76)90055-6`. URL: `http://www.sciencedirect.com/science/article/pii/0092867476900556`.

One of the first publications to use modern DNA Sequencing theory.

[CR+12] Rayan Chikhi, Guillaume Rizk, et al. "Space-efficient and exact de Bruijn graph representation based on a Bloom filter." In: *WABI*. 2012, pp. 236–248.

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[DSH08] Siobain Duffy, Laura A Shackelton, and Edward C Holmes. "Rates of evolutionary change in viruses: patterns and determinants". In: *Nature Reviews Genetics* 9.4 (2008), pp. 267–276.

[Ear+11] Dent Earl et al. "Assemblathon 1: A competitive assessment of de novo short read assembly methods". In: *Genome research* 21.12 (2011), pp. 2224–2241.

[Eid+09] John Eid et al. "Real-time DNA sequencing from single polymerase molecules". In: *Science* 323.5910 (2009), pp. 133–138.

[Eis12] Michael Eisenstein. "Oxford Nanopore announcement sets sequencing sector abuzz". In: *Nature biotechnology* 30.4 (2012), pp. 295–296.

[Gar+09] Rebecca J Garten et al. "Antigenic and genetic characteristics of swine-origin 2009 A (H1N1) influenza viruses circulating in humans". In: *science* 325.5937 (2009), pp. 197–201.

[Gat09] Derek Gatherer. "The 2009 H1N1 influenza outbreak in its historical context". In: *Journal of Clinical Virology* 45.3 (2009), pp. 174–178.

[Gee+10] Lewis Y. Geer et al. "The NCBI BioSystems database". In: *Nucleic Acids Research* 38.suppl 1 (2010), pp. D492–D496. DOI: 10.1093/nar/gkp858. eprint: http://nar.oxfordjournals.org/content/38/suppl_1/D492.full.pdf+html. URL: http://nar.oxfordjournals.org/content/38/suppl_1/D492.abstract.

[Gra+11] Manfred G Grabherr et al. "Full-length transcriptome assembly from RNA-Seq data without a reference genome". In: *Nature biotechnology* 29.7 (2011), pp. 644–652.

[Ham50] Richard W Hamming. "Error detecting and error correcting codes". In: *Bell System technical journal* 29.2 (1950), pp. 147–160. URL: http://www.caip.rutgers.edu/~bushnell/dsdwebsite/hamming.pdf.

[HG09] Edward C Holmes and Bryan T Grenfell. "Discovering the phylodynamics of RNA viruses". In: *PLoS computational biology* 5.10 (2009), e1000505.

[IW95] Ramana M Idury and Michael S Waterman. "A new algorithm for DNA sequence assembly". In: *Journal of computational biology* 2.2 (1995), pp. 291–306.

[JC69] Thomas H Jukes and Charles R Cantor. "Evolution of protein molecules". In: *Mammalian protein metabolism* 3 (1969), pp. 21–132.

[Joh67] Stephen C Johnson. "Hierarchical clustering schemes". In: *Psychometrika* 32.3 (1967), pp. 241–254.

[Knu73] Donald E Knuth. *Sorting and Searching (The Art of Computer Programming volume 3)*. 1973.

[Koz+09] Iwanka Kozarewa et al. “Amplification-free Illumina sequencing-library preparation facilitates improved mapping and assembly of (G+ C)-biased genomes”. In: *Nature methods* 6.4 (2009), pp. 291–295.

[KRE88] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*. Vol. 2. prentice-Hall Englewood Cliffs, 1988.

[Lan+01] E S Lander et al. “Initial sequencing and analysis of the human genome”. In: *Nature* 409.6822 (Feb. 2001), pp. 860–921. DOI: 10.1038/35057062.

Human genome publication.

[Lan+09] Ben Langmead et al. “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”. In: *Genome Biol* 10.3 (2009), R25.

[LD09] Heng Li and Richard Durbin. “Fast and accurate short read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760.

[Li+09] Heng Li et al. “The sequence alignment/map format and SAMtools”. In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.

[Maa00] Moritz G Maaß. “Linear bidirectional on-line construction of affix trees”. In: *Combinatorial Pattern Matching*. Springer. 2000, pp. 320–334.

[Mar+05] Marcel Margulies et al. “Genome sequencing in microfabricated high-density picolitre reactors”. In: *Nature* 437.7057 (2005), pp. 376–380.

[Mar08] Elaine R Mardis. “Next-generation DNA sequencing methods”. In: *Annu. Rev. Genomics Hum. Genet.* 9 (2008), pp. 387–402.

[McC10] Alice McCarthy. “Third generation DNA sequencing: pacific biosciences’ single molecule real time technology”. In: *Chemistry & biology* 17.7 (2010), pp. 675–676.

[McK+11] Trevelyan J McKinley et al. “A Bayesian approach to analyse genetic variation within RNA viral populations”. In: *PLoS computational biology* 7.3 (2011), e1002027.

[Mul+87] Kary B Mullis et al. *One of the first Polymerase Chain Reaction (PCR) patents*. US Patent 4,683,195. 1987.

[Mur+10] Pablo R Murcia et al. "Intra-and interhost evolutionary dynamics of equine influenza virus". In: *Journal of virology* 84.14 (2010), pp. 6943–6954.

[Mye+00] Eugene W Myers et al. "A whole-genome assembly of Drosophila". In: *Science* 287.5461 (2000), pp. 2196–2204.

[Mye05] Eugene W. Myers. "The fragment assembly string graph". In: *Bioinformatics* 21.suppl 2 (2005), pp. ii79–ii85. DOI: 10.1093/bioinformatics/bti1114. eprint: http://bioinformatics.oxfordjournals.org/content/21/suppl_2/ii79.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/21/suppl_2/ii79.abstract.

[NP13] Niranjan Nagarajan and Mihai Pop. "Sequence assembly demystified". In: *Nature Reviews Genetics* 14.3 (2013), pp. 157–167.

[Pen+12] Yu Peng et al. "IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth". In: *Bioinformatics* 28.11 (2012), pp. 1420–1428.

[Pla83] Robin L Plackett. "Karl Pearson and the chi-squared test". In: *International Statistical Review/Revue Internationale de Statistique* (1983), pp. 59–72.

[Pot01] Christopher W Potter. "A history of influenza". In: *Journal of applied microbiology* 91.4 (2001), pp. 572–579.

[PTW01] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. "An Eulerian path approach to DNA fragment assembly". In: *Proceedings of the National Academy of Sciences* 98.17 (2001), pp. 9748–9753.

[Qua+12] Michael A Quail et al. "A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers". In: *BMC genomics* 13.1 (2012), p. 341.

[RBD13] J Graham Ruby, Priya Bellare, and Joseph L DeRisi. "PRICE: software for the targeted assembly of components of (meta) genomic sequence data". In: *G3: Genes| Genomes| Genetics* 3.5 (2013), pp. 865–880.

[Rob+11] Kim L Roberts et al. "Lack of transmission of a human influenza virus with avian receptor specificity between ferrets is not due to decreased virus shedding but rather a lower infectivity in vivo". In: *Journal of General Virology* 92.8 (2011), pp. 1822–1831.

[Rus+08] Colin A Russell et al. "The global circulation of seasonal influenza A (H3N2) viruses". In: *Science* 320.5874 (2008), pp. 340–346.

[SD10] Jared T Simpson and Richard Durbin. "Efficient construction of an assembly string graph using the FM-index". In: *Bioinformatics* 26.12 (2010), pp. i367–i373.

[SD12] Jared T Simpson and Richard Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome research* 22.3 (2012), pp. 549–556.

[Sha48] Claude Shannon. *A mathematical theory of communication*. First Edition. Vol. 27. Bell Systems, 1948.

[Sim+09] Jared T Simpson et al. "ABySS: a parallel assembler for short read sequence data". In: *Genome research* 19.6 (2009), pp. 1117–1123.

The most widely used parallel MPI based short-read assembler.

[Smi+09] Gavin JD Smith et al. "Origins and evolutionary genomics of the 2009 swine-origin H1N1 influenza A epidemic". In: *Nature* 459.7250 (2009), pp. 1122–1125.

[Smi+85] Lloyd M Smith et al. "Fluorescence detection in automated DNA sequence analysis." In: *Nature* 321.6071 (1985), pp. 674–679.

[SNC77] Frederick Sanger, Steven Nicklen, and Alan R Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences* 74.12 (1977), pp. 5463–5467.

[Sta+12] J Conrad Stack et al. "Inferring the inter-host transmission of influenza A virus using patterns of intra-host genetic variation". In: *Proceedings of the Royal Society B: Biological Sciences* (2012), rspb20122173.

[Str13] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[SW81] Temple F Smith and Michael S Waterman. "Identification of common molecular subsequences". In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

[Utt+14] Sagar M Utturkar et al. "Evaluation and validation of de novo and hybrid assembly techniques to derive high-quality genome sequences". In: *Bioinformatics* 30.19 (2014), pp. 2709–2716.

[Wil38] Samuel S Wilks. "The large-sample distribution of the likelihood ratio for testing composite hypotheses". In: *The Annals of Mathematical Statistics* 9.1 (1938), pp. 60–62.

[Wri+11] Caroline F Wright et al. "Beyond the consensus: dissecting within-host viral population diversity of foot-and-mouth disease virus by using next-generation genome sequencing". In: *Journal of virology* 85.5 (2011), pp. 2266–2275.

[ZB08] Daniel R. Zerbino and Ewan Birney. "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs". In: *Genome Research* 18.5 (2008), pp. 821–829. DOI: 10.1101/gr.074492.107. eprint: http://genome.cshlp.org/content/18/5/821.full.pdf+html. URL: http://genome.cshlp.org/content/18/5/821.abstract.

One of the first de Brujin-graph assemblers.

[R D10] R Development Core Team. *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria, 2010. URL: http://www.R-project.org/.

Acronyms

API Application programming interface. 172

BLAST Basic Local Alignment Search Tool [Alt+90]. 24, 79

BWA Burrows-Wheeler Aligner [LD09]. 28, 37

contig From a biological standpoint, contigs are collections of reads that clearly overlap each other and refer to the same overall sequence. 72

CPU Central Processing Unit. 78, 105, 107, 123, 134, 135, 137, 138, 145, 160

DFS Depth First Search. 128, 130, 131

DMA Direct Memory Access. 154

DNA Deoxyribonucleic acid. 38, 66, 67

gcc GNU Compiler Collection. 123

GPL GNU Public Licence. 161

GRU Global Reference Unit. 123

GUI Graphical User Interface. 62

HA Hemagglutinin. 17

HTML Hypertext Markup Language. 28, 54, 56, 62

IAV Influenza A virus. 10, 16, 17, 19, 20, 43, 61, 62, 160

KAT Kmer Analysis Tool [CB13]. 89

LSF Platform Load Sharing Facility. 30

mutex Mutual exclusion. Often used as a synonym for a mutual exclusion lock as provided by Application programming interfaces (APIs). E.g. the POSIX Pthread `pthread_mutex_t`.. 120–122

NCBI National Center for Biotechnology Information [Gee+10]. 24, 40

NGS Next Generation Sequencing. 8, 13, 18, 31, 37, 38, 50, 56–58, 61, 66, 68, 73, 94, 104, 143, 159

NUMA Non Uniform Memory Architecture. 123

OLC Overlap Layout Consensus. 67, 71–73, 75, 77, 79

OS Operating System. 121, 152

PBS Portable Batch System. 135

PCR Polymerase Chain Reaction [Mul+87]. 17, 18, 39, 71, 105

Pfu Pfu DNA polymerase is an enzyme used as part of a PCR process to aplify DNA.. 21

RAM Random Access Memory. 99, 112, 121, 138, 140, 160

RNA Ribonucleic acid. 16, 17

RT-PCR reverse transcription polymerase chain reaction [Bus02]. 17

SAM Sequence Alignment/Map [Li+09]. 37

SGI UV Supercomputer commercialised SGI. Comes in configurations of 768 - 4096 Intel® Xeon® cores. 80, 123, 143

SNP single nucleotide polymorphism. 37, 40

TCP Transport Control Protocol. 135

VM Virtual Memory. 121, 143, 151–154

x86-64 x86-64 (also known as x64, x86_64 and AMD64) is the 64-bit version of the x86 instruction set. 123

Appendices

A. Notation

Definition A.1. *Let Σ be a finite set of characters. Then we call Σ an alphabet.*

Definition A.2. *Let Σ be a finite set of characters, $n \in \mathbb{N}, a_i \in \Sigma$. We call a finite sequence $a_1 a_2 a_3 \dots a_n$ a word over Σ. We symbolise the empty sequence or empty word using the symbol λ.*

Definition A.3. *Let Σ be an alphabet and λ the empty word. We define*

$$\Sigma^k := \{a_1 \dots a_k | a_i \in \Sigma\} \tag{A.1}$$

the set of all words (k-mers) of length k,

$$\Sigma^+ := \bigcup_{k \in \mathbb{N}_{\geq 1}} \Sigma^k \tag{A.2}$$

$$\Sigma^* := \bigcup_{k \in \mathbb{N}_{\geq 1}} \Sigma^k \cup \{\lambda\} \tag{A.3}$$

the set of all (non empty) words over the alphabet Σ.

Definition A.4. *Let Σ be an alphabet. We define the concatenation function*

$$\cdot : \Sigma^* \times \Sigma^* \to \Sigma^* \tag{A.4}$$

$$: (w, v) = (w_1 \dots w_k, v_1 \dots v_l) \mapsto w_1 \dots w_k v_1 \dots v_l. \tag{A.5}$$

Instead of writing $w \cdot v := \cdot(w, v)$ we omit the invix operator and simply write wv when we mean the concatenation of two words.

Proposition A.5. *Let Σ be an alphabet. Then $(\Sigma^*, \cdot)$ is a monoid.*

Proof of the above proposition is beyond the scope of this thesis.

Definition A.6. *Let Σ be an alphabet and $w = w_1 \dots w_n \in \Sigma^+$ a word. We define*

$$`w := w_2 \dots w_n \tag{A.6}$$

$$w` := w_1 \dots w_{n-1} \tag{A.7}$$

the front and rear truncation of w respectively.

Definition A.7. *We define the genomic alphabet $\Gamma := \{A, C, G, T\}$.*

We define complement nucleotides as:

$$\widetilde{\ }: \Gamma \to \Gamma \tag{A.8}$$

$$: N \mapsto \begin{cases} T & \text{for } N = A \\ G & \text{for } N = C \\ C & \text{for } N = G \\ A & \text{for } N = A. \end{cases} \tag{A.9}$$

Definition A.8. *We define the reverse complement of a genomic sequence as:*

$$\widetilde{\ }: \Gamma^* \to \Gamma^* \tag{A.10}$$

$$: s_1 \dots s_n \mapsto \widetilde{s_n} \dots \widetilde{s_1}. \tag{A.11}$$

Definition A.9. *We define the sequence index as*

$$\iota : \Sigma^* \times \Sigma^* \to \mathcal{P}(\mathbb{Z}) \tag{A.12}$$

$$: (s, w) \mapsto \{|x| | x, y \in \Sigma^*, xwy = s\}. \tag{A.13}$$

The set of all starting positions in s where w starts and is contained as a subsequence.

Definition A.10. *When referring to the genomic alphabet we define the sequence index additionally as:*

$$\begin{aligned} \iota : \ & \Gamma^* \times \Gamma^* \to \ \mathcal{P}(\mathbb{Z}) \\ : \ & (s, w) \mapsto \ \{|x|, x \in \Sigma^* | xwy = s, y \in \Sigma^*\} \cup \\ & \qquad \{-|y|, x \in \Sigma^* | xwy = \widetilde{s}, y \in \Sigma^*, w \neq \widetilde{w}\} \end{aligned} \tag{A.14}$$

Definition A.11. *Let $f, g : \mathbb{R} \to \mathbb{R}$ be two functions. We define the big-O notation:*

$$f \in O(g) \Leftrightarrow \limsup_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| < \infty. \tag{A.15}$$

Remark A.12. *The big-O notation is often used for asymptotic estimation of resource usage. For example if the exact runtime of an algorithm is a complex function*

$$f : n \mapsto n^4 \log\log(n) + 3n \log(n) \tag{A.16}$$

(where n measures the size of the input) then this can be simplified as

$$f(n) \in O(n^4 \log(n)) \subseteq O(n^5). \tag{A.17}$$

Definition A.13. *Let $G = (V, E)$ be a graph. A path $p = (e_1, \ldots, e_n), e_i \in E$ is called an **eulerian path** if and only if each edge in E is used exactly once.*

$$\begin{aligned} \bigcup_{i=1}^{n} \{e_i\} &= E \\ \forall i \neq j : e_i &\neq e_j \end{aligned} \tag{A.18}$$