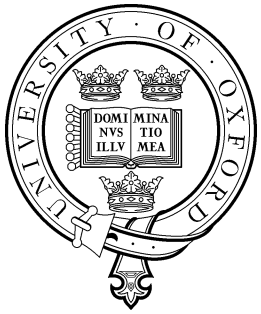
Discovering Genetic Variation in Populations using Next Generation Sequencing and De Novo Assembly



Isaac H. C. Turner
Wolfson College, University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Trinity Term 2018

Abstract

The de Bruijn Graph is a simple and efficient data structure that is used in many areas of sequence analysis including genome assembly, sequencing error correction, sequencing read compression and variant discovery. The data structure has a single parameter k, is straightforward to implement and is tractable for large genomes with high sequencing depth. It also enables representation of multiple samples simultaneously to facilitate comparison. However, unlike the string graph, a de Bruijn Graph does not retain long range information that is inherent in the read data. For this reason, applications that rely on de Bruijn Graphs can produce sub-optimal results given their input.

We present a novel assembly graph data structure: the Linked de Bruijn Graph. Constructed by adding annotations on top of a de Bruijn Graph, it stores long range connectivity information through the graph. We show that with error-free data it is possible to losslessly store and recover sequence from a Linked de Bruijn Graph. Through simulation we show that links improve performance and reduce the sensitivity to the parameter k. Many algorithms can be built on top of Linked de Bruijn Graphs, which we illustrate by implementing read error correction, variant calling, genotyping and assembly algorithms in a software package called 'McCortex'.

With assembly simulations we demonstrate that the Linked de Bruijn Graph data structure outperforms both the de Bruijn Graph and the String Graph Assembler (SGA). Using human whole genome sequence, we show that Linked de Bruijn Graphs scale up to mammalian genomes. Finally we apply McCortex to *Klebsiella pneumoniae* short read data to call SNPs, indels and large events, which we validate using PacBio sequencing data. Although $k$mer-based methods have reduced sensitivity for calling SNPs, McCortex finds 3x more polymorphic bases due to small events in *K. pneumoniae* versus mapping-based approaches and 48% more than existing de Bruijn Graph *de novo* assembly methods. We also find multiple 10kbp events that cannot be resolved with a plain de Bruijn Graph.

In conclusion the Linked de Bruijn Graph is a versatile data structure with utility in multi-sample variant calling of indels and large events, in both *de novo* and reference-aware scenarios.

Acknowledgements

I am grateful to my examiners, Pall Melsted and Gerton Lunter, who took the time to review my work and asked great challenging questions. This work was funded by the EPSRC.

I would like to say thank you to my supervisor Gil McVean, and to Zam Iqbal who acted as an advisor throughout my PhD. Without their ideas, suggestions and patience this work wouldn't have happened. Working with them has been an honour.

The McVean group has been a source of ideas, support and friendship which enriched this work. Academic highlights include collaborating with Kiran, Duncan and Oliver, as well as stimulating discussions with Alex, Luke, Ben, Jerome, Iain, Phelim and Denise.

Prof John Moriarty inspired me to work on de Bruijn graphs when he gave me my first opportunity to work on computational biology problems in Manchester. I'm also grateful for my colleagues at Shield Diagnostics who supported my final effort to finish this thesis.

I couldn't have finished this work without the support of my family and friends including those at the Oxford Doctoral Training Centre and Wolfson College. I met many kind and interesting people along the way who made my time at Oxford memorable and a joy. I'm thankful to them all but most of all to Lydia who I met at Oxford, who went through it all with me and stuck around to proofread this thesis.

Contents

Abstract **3**

Acknowledgements **6**

Contents **10**

List of Figures **12**

List of Tables **13**

List of Algorithms **14**

Abbreviations **16**

Nomenclature **16**

1 Introduction **17**
1.1 DNA sequencing . . . 18
1.2 Read mapping . . . 20
1.3 Genome assembly . . . 21
1.3.1 Contig scaffolding . . . 25
1.4 De Bruijn graphs . . . 25
1.4.1 De Bruijn graph assembly . . . 28
1.4.2 *k*mer counting . . . 29
1.4.3 Low memory de Bruijn Graph representations . . . 29
1.4.4 Suffix trees, BWT and FM-index . . . 33
1.5 Genome assemblers . . . 37
1.5.1 Other assemblers . . . 48
1.5.2 Long read assemblers . . . 48
1.5.3 Diploid aware assembly . . . 49
1.5.4 Comparing assemblers . . . 50
1.6 Variant calling . . . 52
1.6.1 Variant calling quality . . . 54
1.6.2 Reference bias . . . 55
1.7 Mapping reads to graphs . . . 56
1.8 Sequencing error correction . . . 60
1.9 Visualisation tools . . . 62
1.10 Summary . . . 63

2 Linked de Bruijn Graphs **65**
2.1 Linked de Bruijn Graph as a string graph . . . 65
2.1.1 Definitions . . . 65
2.1.2 de Bruijn Graph limitations . . . 66
2.1.3 Initial approach: shading . . . 67
2.1.4 Adopted approach: links . . . 68

2.2 Graph construction . . . 69
2.2.1 de Bruijn Graph construction . . . 69
2.2.2 de Bruijn Graph cleaning . . . 69
2.2.3 Read threading . . . 70
2.2.4 Adding graph links . . . 72
2.3 Graph traversal . . . 73
2.4 Links in loops . . . 76
2.5 Theoretical proofs . . . 76
2.5.1 Assumption of complete information . . . 76
2.5.2 Notation . . . 77
2.5.3 Repeats . . . 78
2.5.4 Sequence traversal . . . 80
2.5.5 Vertex traversal . . . 82
2.5.6 Implementation limitations . . . 86
2.5.7 Testing theory with simulation . . . 87
2.5.8 Extracting contigs . . . 89
2.5.9 Lossless storage . . . 89
2.5.10 Link coverage estimation . . . 90
2.6 Traversal confidence . . . 90
2.7 Reducing sensitivity to parameter k . . . 93
2.8 Link pruning and memory . . . 98
2.9 Discussion . . . 98

3 Implementation **101**
3.1 Implementation . . . 101
3.1.1 Commands . . . 101
3.1.2 McCortex sequence input . . . 102
3.1.3 Graph files . . . 104
3.1.4 de Bruijn Graph representation . . . 104
3.1.5 Multithreading . . . 106
3.1.6 Memory allocation . . . 109
3.1.7 JSON file headers . . . 109
3.2 Details of graph operations . . . 110
3.2.1 PCR duplicate removal . . . 110
3.2.2 Infer edges . . . 110
3.2.3 $k$mer error cleaning . . . 111
3.2.4 Link cleaning . . . 112
3.2.5 Graph walking . . . 113
3.3 McCortex tools . . . 114
3.3.1 McCortex commands . . . 114
3.3.2 Visualisation . . . 115
3.3.3 Running McCortex pipeline . . . 116
3.4 HTTP API webserver . . . 118
3.5 Development . . . 119
3.5.1 Libraries . . . 120
3.5.2 Third party software . . . 121
3.5.3 Good software practices . . . 121
3.6 Discussion . . . 122

4 Applications **125**
4.1 Simulations in this chapter . . . 125
4.2 Genome Assembly with a linked de Bruijn Graph . . . 126
4.2.1 Multithreading . . . 128
4.2.2 Assembling diploid genomes . . . 128
4.2.3 Bubble popping . . . 129
4.2.4 Haploid simulation . . . 130

4.2.5 Diploid simulation . . . 133
4.3 Variant calling . . . 135
4.3.1 Graph based calling . . . 135
4.4 Breakpoint Caller . . . 136
4.4.1 Simulation . . . 140
4.4.2 Potential improvements . . . 141
4.5 Bubble Caller . . . 142
4.5.1 Multithreading . . . 145
4.5.2 Filtering paralogous sequences . . . 145
4.5.3 Placing variants . . . 146
4.6 Genotyping a VCF against the graph . . . 147
4.7 Variant calling simulation . . . 150
4.7.1 Single diploid human 1Mbp . . . 150
4.7.2 Ten *E. coli* Simulation . . . 151
4.7.3 Summary . . . 152
4.8 Error correction . . . 152
4.8.1 Simulation . . . 153
4.8.2 Summary . . . 155
4.9 Discussion . . . 156

5 Results **159**
5.1 Human: NA12878 . . . 159
5.1.1 Calling with McCortex . . . 159
5.1.2 Results . . . 163
5.1.3 Low level link cleaning . . . 165
5.1.4 Expected power . . . 165
5.1.5 Genotyping against the graph . . . 167
5.1.6 Human genome assembly . . . 169
5.2 Bacteria: *Klebsiella pneumoniae* . . . 171
5.2.1 Building the graph with McCortex . . . 171
5.2.2 De novo assembly . . . 172
5.2.3 Calling small events . . . 174
5.2.4 Large events . . . 179
5.3 Discussion . . . 180

6 Discussion **183**
6.1 Link annotations . . . 183
6.1.1 Links vs. paired *k*mers . . . 184
6.1.2 Encoding links as succinct trees . . . 184
6.1.3 Storing links as ranked paths . . . 186
6.1.4 Alternative link encodings . . . 187
6.2 Variant calling . . . 188
6.2.1 Variant calling development . . . 189
6.3 Error correction . . . 190
6.4 Implementation . . . 190
6.5 Assembly . . . 191
6.5.1 Competing assembly approaches: Faucet . . . 191
6.6 Conclusion . . . 192

Appendices **193**

A List of publications **194**

B Haploid assembly simulation **195**

C Coverage derivation **196**

D Greedy maximal substring algorithm **199**

Bibliography **201**

List of Figures

1.1 RAM cost . . . 19
1.2 DNA sequencing cost . . . 20
1.3 Completely connected de Bruijn Graph . . . 26
1.4 De Bruijn graph $k = 5$ . . . 27
1.5 Suffix tree example . . . 33
1.6 Suffix array example . . . 34
1.7 BWT example . . . 35
1.8 FM-index example . . . 35
1.9 Velvet Tour Bus algorithm . . . 42
1.10 Velvet Breadcrumb algorithm . . . 42
1.11 Velvet Pebble algorithm . . . 43
1.12 Velvet Rockband algorithm . . . 43
1.13 Minia assembler illustration . . . 47
1.14 Inversion in a de Bruijn Graph . . . 60

2.1 Unitigs in a de Bruijn Graph . . . 66
2.2 Graph 'tangle' due to low complexity . . . 67
2.3 Error correction on a read against a graph . . . 71
2.4 Mate pair-read gap inference by traversal . . . 71
2.5 Making links from reads . . . 73
2.6 Missing link information check . . . 74
2.7 Graph traversal flowchart . . . 75
2.8 Loops in a Linked de Bruijn Graph . . . 76
2.9 Infinite loops at sequence ends . . . 77
2.10 A simple de Bruijn Graph ($k = 5$) . . . 77
2.11 Repeats in the graph . . . 79
2.12 Repeat sets of points s_w, s_x, s_y and s_z in Figure 2.11 . . . 79
2.13 Last vertices function . . . 82
2.14 Loop traversal in a three-copy repeat . . . 83
2.15 Contig assembly depends on seed $k$mer choice . . . 89
2.16 Links lost to due stochastic coverage . . . 91
2.17 Calculating contig confidence cartoon . . . 91
2.18 Plot of contig confidence varying parameters . . . 92
2.19 Simulations varying k . . . 95
2.20 Number of $k$mers with links in simulations varying k . . . 97

3.1 Infinite loop in a unitig graph with $k = 5$ . . . 113
3.2 de Bruijn Graph visualisation at $k = 15$ . . . 115
3.3 Unitig visualisation at $k = 15$ . . . 116

4.1 Diploid genome assembly . . . 129
4.2 Effect of bubble popping on links . . . 130
4.3 Haploid assembly simulation with varying k . . . 132
4.4 Breakpoint caller for different events . . . 136

4.5 Breakpoint caller algorithm . . . 137
4.6 Example Circos plot . . . 139
4.7 Breakpoint caller using private *k*mers . . . 141
4.8 SNP in a de Bruijn Graph . . . 142
4.9 McCortex complex-bubble calling algorithm in three diploid samples . . . 143
4.10 Bubble calling in adjacent sites in multiple diploid samples . . . 145
4.11 VCF coverage from the graph . . . 148
4.12 Graph structure around clustered variants . . . 149

5.1 NA12878 *k*mer coverage . . . 161
5.2 Link coverage on sample NA12878 . . . 162
5.3 *K. pneumoniae* assembly with/without links . . . 174
5.4 Circos plot of *Klebsiella pneumoniae* breakpoint calls . . . 176
5.5 Indel calls in *Klebsiella pneumoniae* . . . 177
5.6 *K. pneumoniae* large events . . . 179

6.1 Binary encoding of the link-tree . . . 185
6.2 Storing links with ranked paths . . . 186
6.3 Unrecoverable variant . . . 188
6.4 Link-*k*mer pairs vs Linked de Bruijn Graph . . . 191

B.1 Haploid assembly simulation with varying k . . . 195

List of Tables

1.1 *rank* and *select* operations . . . 32

2.1 Simulation to test traversal conjecture . . . 88
2.2 Simulation to test traversal conjecture without context . . . 88

3.1 Buffered vs. unbuffered file reading . . . 103
3.2 *McCortex* vs. *jelly-hash* hash table comparison . . . 107

4.1 McCortex diploid assembly simulation . . . 134
4.2 McCortex haploid/diploid assembly comparison . . . 134
4.3 Haploid and diploid breakpoint simulations . . . 140
4.4 Diploid variant calling simulations . . . 150
4.5 Ten haploid samples variant calling simulation . . . 151
4.6 Error correcting reads for diploid human simulation . . . 154
4.7 Error correcting reads from *E. coli* simulation . . . 155

5.1 Single human run times and memory . . . 160
5.2 McCortex on-disk file sizes . . . 161
5.3 McCortex variant calling in NA12878 . . . 163
5.4 NA12878 variant calling by class . . . 164
5.5 NA12878 variant calling limits at $k = 31$ . . . 166
5.6 NA12878 McCortex sensitivity . . . 167
5.7 McCortex/GIAB and Cortex/GIAB genotype concordance . . . 168
5.8 McCortex/Cortex/GIAB genotype concordance . . . 168
5.9 Human whole genome assembly metrics . . . 169
5.10 Human genome assembly metrics after bubble popping . . . 170
5.11 *K. pneumoniae* graph and link cleaning . . . 172
5.12 *K. pneumoniae* McCortex assembly metrics . . . 172
5.13 *K. pneumoniae* McCortex longest contigs assembly metrics . . . 173
5.14 MUMMER SNP calling . . . 175
5.15 *K. pneumoniae* variant calling Power / FP . . . 178
5.16 *K. pneumoniae* validated calls . . . 178
5.17 *K. pneumoniae* validated large events that could be found only with links . . . 180

List of Algorithms

1 O(N) Algorithm to add links to the graph from a single read segment 72
2 Assemble contigs in a given colour 127
3 Finding structural variation via reference comparison 138
4 Algorithm to find complex bubbles 144
5 O(N^3) algorithm to find all maximal substrings 199

Abbreviations

1000G 1000 Genomes Project

BAC Bacterial Artificial Chromosome

bp base pairs

BWT Burrows-Wheeler Transform

Coloured de Bruijn Graph (CdBG) A de Bruijn Graph, where $k$mers store which samples (*colours*) they were seen in

dBG de Bruijn Graph

DFS Depth First Search

DNA deoxyribonucleic acid

DRAM Dynamic Random Access Memory

FM-index Full-text index in Minute space

FN False Negative

FNR False Negative Rate

FP False Positive

FPR False Positive Rate

GiB gibibyte (2^{30} bytes)

GWAS Genome-Wide Association Study

KiB kibibyte (2^{10} bytes)

LdBG Linked de Bruijn Graph

MAF Minor Allele Frequency

MHC Major Histocompatability Complex

MiB mebibyte (2^{20} bytes)

MNP Multiple Nucleotide Polymorphism

MSA Multiple Sequence Alignment

NAHR Non-allelic homologous recombination

NGS Next Generation Sequencing

OLC Overlap-Layout-Consensus

PCR Polymerase Chain Reaction

PE Paired-end (when referring to sequencing reads)

RNA ribonucleic acid

SE Single-end (when referring to sequencing reads)

SNP Single Nucleotide Polymorphism

SNV Single Nucleotide Variant

TiB tebibyte (2^{40} bytes)

TP True Positive

TPR True Positive Rate

WGS Whole Genome Sequencing

Nomenclature

$k$mer sequence of length k over the alphabet $\{A, C, G, T\}$.

ALT Non-reference allele(s) of a variant.

BAM Binary Alignment Map – a binary file format for storing sequencing data aligned to a reference.

contig Contiguous sequence – a string of characters without gaps.

CRAM File format for storing sequencing data aligned to a reference.

de Bruijn Graph (dBG) A set of vertices ($k$mers), with directed edges between those that share $k-1$ overlap.

read A string of letters produced by a DNA sequencing machine.

Eulerian path A path through a graph that visits every edge exactly once. Can be found in O(|E|).

FASTA File format for storing sequencing data.

FASTQ File format for storing sequencing data with per-base quality scores.

Hamiltonian path A path through a graph that visits every vertex exactly once. Finding one is an NP-complete problem.

homopolymer Repeated base e.g. `'AAAA'`.

indel Insertion or deletion of DNA into a genome.

MAPQ PHRED-scored mapping quality of a read to a reference genome.

N50 Assembly metric – contig length such that over half of the bases are in contigs at least as long as the given value.

NG50 Assembly metric – contig length such that at least half of the genome length can be made up of contigs at least as long as the given value.

NP-complete Set of computational problems for which no polynomial solution is known but solutions for which can be verified in polynomial time. All problems with time complexity NP (Nondeterministic Polynomial time) can be reduced to NP-complete problems in polynomial time, making problems in NP-complete at least as hard as any problem in NP.

NP-hard Computational problems that are at least as hard as NP-complete, some of which have solutions which cannot be verified in polynominal time. It is not known if a polynomial solution for any NP-hard problem exists.

PHRED-scoring $-10log_{10}P$, where P is the estimated probability of an error.

REF Reference allele of a variant.

SAM Sequence Alignement/Map format – text based file format for storing sequencing data aligned to a reference.

unitig Sequence represented by a maximal run of vertices (e.g. $k$mers) with in-degree and out-degree between them exactly one.

VCF Reference-based file format for storing genetic variants and sample genotypes.

Chapter 1

Introduction

DNA sequencing – the process by which a genome is read – has developed faster than anyone could predict. Despite this, sequencing is still limited in quality and speed. DNA reads are short and error prone. Key analysis steps – piecing together these reads and finding differences between sequences – are complicated and have known limitations. Mapping reads to a reference genome has provided a cheap and fast method of analysing the first few thousand human genomes sequenced (McVean *et al.*, 2012). It has also been successfully used to analyse many species for which a good reference genome has been assembled. Traditional analysis of sequence data via mapping samples one at a time to a reference are not fully exploiting the data and power at hand. To take advantage of the large number of genomes being sequenced today and the newly available computer capacity, new computational approaches are required. New analysis pipelines need to be straightforward to use and must run on the heterogeneous computer clusters available in bioinformatics centres around the world.

In this thesis, we explore an alternative approach to variant calling by read mapping – *de novo* assembly for variant calling. Current *de novo* variant calling implementations (Iqbal *et al.*, 2012) do not take advantage of all the data available and suffer from low sensitivity. We propose a new data structure, a *Linked de Bruijn Graph*, along with linear time algorithms to construct and use it to explore genomes. Linked De Bruijn graphs offer a richer representation of the sequencing data, which can be exploited to make better variant calls. It is flexible, can be extended to facilitate new analysis algorithms and can be used with many different sequencing platforms. We have published this work in Turner *et al.* (2018).

Genome comparison underpins research in many areas of biology and medicine. From research into cancer, the origins and dispersal of species, the evolution of bacteria, viruses and parasites and the causes

of antibiotic resistance – finding genetic differences is a crucial step in furthering our understanding. New genome analysis methods help drive research in biology.

1.1 DNA sequencing

In the last decade DNA sequencing has drastically dropped in price – a trend that shows no sign of slowing (see Figure 1.2). At the same time, computational power has greatly increased in terms of CPU, RAM, storage space and network speeds available per dollar (see Figure 1.1). The increase in computational power allows us to analyse more genomes faster and with more computationally intensive methods.

Early genome sequencing used an approach known as *Directed sequencing*, which required each chromosome to be broken into ordered (and perhaps overlapping) DNA sequences in individual clones. Each clone is then separately sequenced. This approach was used by the International Human Genome Sequencing Consortium for their 13 year Human Genome Project, which published the draft genome in Lander *et al.* (2001).

Approaches then shifted to *Random sequencing* – which involved taking random reads from throughout the whole genome, skipping the initial step of breaking it up into ordered clones. Primarily called *shotgun sequencing* or *Whole Genome Sequencing*, it was used by Venter *et al.* (2001) in their effort to sequence the human genome.

Arduous laboratory methods are required for directed sequencing, making it desirable to shift the work to a (bigger) computational problem by using random sequencing. This has resulted in a drop in price and an increase in speed and accuracy of whole genome sequencing (see Figure 1.2), thanks in part to Moore's Law of increasing computational power. This trend is also the result of new high throughput technologies that abandon the original chain-termination Sanger and Maxam-Gilbert methods. Modern high throughput sequencing is predominantly carried out on the Illumina platform. Originally read lengths were around ~ 20 bp (base pairs), some can now go up to ~ 300bp.

Mate paired reads, originally called double-barrelled or clone end reads, are reads generated by sequencing both ends of a long clone. It was first proposed by Edwards (1990) when it was used to assemble 60kb of the human HPRT locus. Mate pairing gives a pair of short reads, separated by a gap of unknown sequence. Mate pair library preparation Nextera from Illumina prepares the sample by ligating adapters to fragments of DNA which are then circularised and sheared before being sequenced resulting in 2x250bp reads separated by up to 12kbp [1]. Mate pair information is used in read mapping and in genome assembly

[1] See: `https://www.illumina.com/products/by-type/sequencing-kits/library-prep-kits/nextera-mate-pair.html`

usually as a second step to scaffold contigs (also called "gap closing").

More recently, longer read length sequencing platforms have been developed such as *Oxford Nanopore* (Clarke *et al.*, 2009, Mikheyev and Tin, 2014) and *Pacific Biosciences'* Single Molecule Real-Time (*SMRT*) sequencing platform (Eid *et al.*, 2009). Long read platforms require more DNA, provide lower sequencing depth, have higher error rates and cost significantly more than Illumina short read sequencing. However they can generate high quality finished assemblies of bacterial genomes, often returning a handful of contigs after assembly (Loman *et al.*, 2015). A hybrid approach of error correcting long reads with Illumina sequencing has been shown to be effective (Koren *et al.*, 2012). Recently a human genome was assembled using Oxford Nanopore sequencing (Jain *et al.*, 2018)

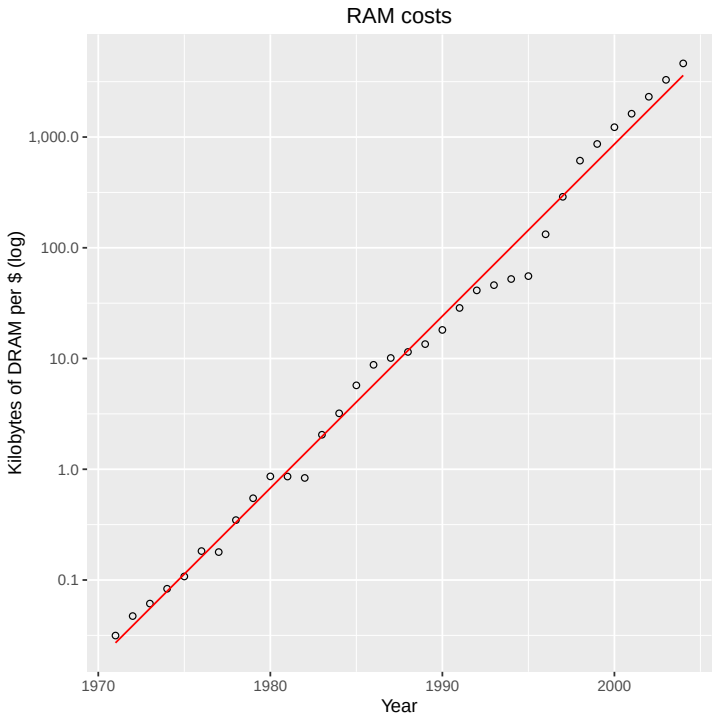


Figure 1.1: Kilobytes of RAM that can be bought for a dollar. Red line is a fitted exponential with a doubling time of ∼1.9 years. Source data from *VLSI Research Inc.* and *ITRS*, collated at `www.singularity.com/charts/page58.html`.

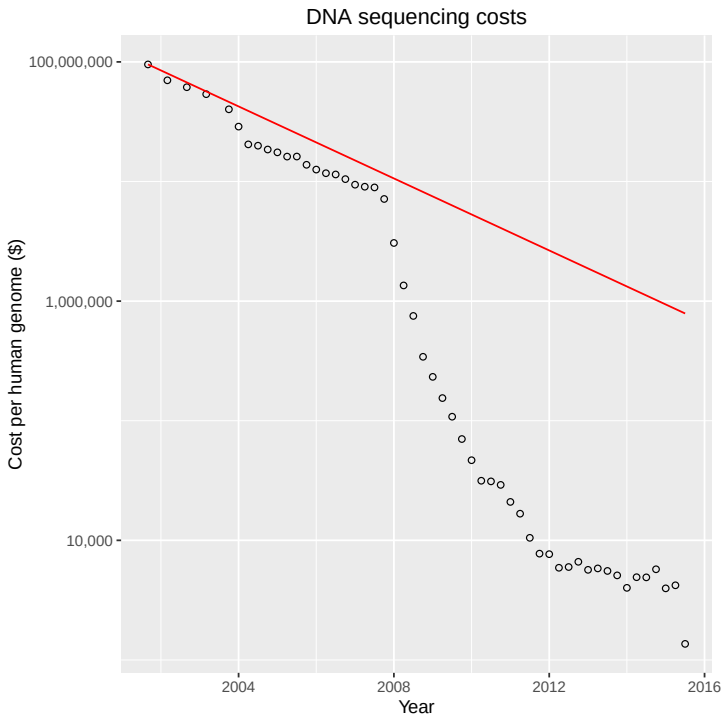


Figure 1.2: Cost of DNA sequencing a human genome with the equivalent of 30X Illumina short reads (source data from Wetterstrand (2016)). Red line shows a halving in price every two years (Moore's Law).

The falling cost of sequencing along with the amount of data produced by high coverage short read sequencing projects is causing sequencing data storage requirements to grow much faster than storage costs are falling. Data repositories will need to develop guidelines of when to keep data and how much lossy compression should be used – some guidelines have already been suggested (Cochrane *et al.*, 2012).

1.2 Read mapping

The aim of read mapping is to place each DNA sequencing read against the reference at the location from which they are most likely to have originated, and to align the read to maximise its alignment score at that location (Li and Homer, 2010). Read mappers usually provide a *mapping quality* which represents the confidence in the read placement (Li *et al.*, 2008). Reads may be marked as unmapped if there is no location where they map confidently or if there are many locations where the reads matches equally. Read alignments are stored in the SAM text format or BAM binary format (Li *et al.*, 2009). Recently a new format CRAM (Bonfield, 2014) has been developed based on reference-based compression (Hsi-Yang Fritz *et al.*, 2011), which offers lower file sizes and therefore faster read and write speeds for equivalent data. Mapped reads are stored in order of their occurrence along the genome and can be accessed on-disk

quickly to view the reads mapping to a particular genomic region (Li *et al.*, 2009).

Several read mapping algorithms and implementations exist including Bfast (Homer *et al.*, 2009), MAQ (Li *et al.*, 2008) 2BWT (Lam *et al.*, 2009), BWA (Li, 2013, Li and Durbin, 2009, 2010), Stampy (Lunter and Goodson, 2011), Novoalign [2], Bowtie (Langmead *et al.*, 2009) and Bowtie2 (Langmead and Salzberg, 2012). GPU optimised mappers have been developed (Luo *et al.*, 2013) which offer $2-10\times$ speed up in mapping and can explore lower similarity matches. Read mappers have been developed for newer long read sequencing platforms such as PacBio's SMRT (Liu *et al.*, 2015) including *Minimap* (Li, 2016) and *Minimap2* (Li, 2018).

RNA-Seq reads present a slightly different problem in that the data have introns removed. Such reads can be mapped to a reference with purpose built mappers such TopHat (Kim *et al.*, 2013, Trapnell *et al.*, 2009), STAR (Dobin *et al.*, 2013), HISAT (Kim *et al.*, 2015) and HISAT2 (based on mapping with a BWT of a graph (Sirén *et al.*, 2014)). HISAT builds an FM-index (see Section 1.4.4) of the whole genome and additional indexes representing smaller regions. For the human genome, 48,000 FM-indexes are used with each representing a regions of ∼68,000bp. HISAT2 extends this by using graph-based FM-indexes which they call *Hierarchical Graph FM-index*. HISAT2 is claimed to be the first implementation of mapping to a graph instead of linear reference (Sirén *et al.*, 2014).

1.3 Genome assembly

Genome assembly is attempting to answer the question: "given a set of short overlapping and error prone reads, what was the complete sequence of the genome that was measured?". Short read assemblers take in sequence data which includes quality scores for each base and if used, read pair information such as insert length distributions. A base quality score or q-score (usually assigned using *PHRED* scoring (Ewing *et al.*, 1998)) of q corresponds to a probability of $10^{-q/10}$ that the base is correct (Batzoglou *et al.*, 2002). An assembler's job is to output one or more possible assemblies, perhaps with likelihood or quality scores. Assemblies usually consist of multiple contigs, grouped together in one or more scaffolds (or supercontigs). A scaffold often places an ordering over the contigs it contains, as well as distance estimates between the contigs. Scaffolds can then be ordered and assigned to chromosomes through linkage studies.

Resulting assemblies often contain gaps (represented by stretches of N bases), where coverage was too low, polymorphic sequence prevented consensus, or repeating (low complexity) sequences made resolving the sequence impossible. Additionally it may not be possible to assign some short contigs to a chromosome due to gaps or contamination.

[2]Novoalign is proprietary, unpublished and developed by Novocraft

The assembly problem was originally framed as that of the Shortest Superstring Problem (SSP). This problem states, given a set of strings $S = \{s_1, ..., s_n\}$, find the shortest string T, such that each s_i appears as a substring of T. This formalism is exactly the question addressed by de Bruijn (1946), leading such strings to be known as *de Bruijn Sequences*. The Shortest Superstring Problem is known to be NP-complete (Gallant, 1980). Heuristics for DNA assembly have been developed, but due to the NP-complete nature of the problem, they have been unable to handle sequence lengths beyond a BAC-sized[3] genome. It is therefore not realistic to aim to achieve *optimal* global assembly. This is similar to the problem of multiple alignment, which has many parallels to assembly.

Overlap-Layout-Consensus

The search for sub-optimal solutions to the assembly problem led to the development of the *overlap-layout-consensus* (OLC) method. Overlap-layout-consensus approaches work in three steps:

Overlap: Find overlaps between reads – overlaps are where the end of one read matches the start of another.

Layout: By representing each read as a vertex and the overlaps as edges between overlapping reads, create a read-overlap graph.

Consensus: Given a read-overlap graph, merge overlapping reads to create consensus sequences, outputting the merged reads and possibly the remaining unresolved edges.

Overlap detection was sometimes seeded with a matching $k$mer approach and extended with an alignment approach. An OLC approach was used by the Celera assembler (see Section 1.5) to assemble the Drosophila genome in Myers *et al.* (2000). The string graph (Myers, 2005) implementation in SGA (Simpson and Durbin, 2012) is an efficient implementation of the OLC approach.

de Bruijn Graph

Using a de Bruijn Graph (described in depth in Section 1.4) for assembly was first proposed by Pevzner (1989) as an alternative to the OLC approach. De Bruijn graphs required much more memory than the OLC approach as reads had to be broken into all their $k$mers. However it enabled the use of a new short read sequencing technology that had just been proposed: *sequencing by hybridisation* (Iu *et al.*, 1988, Roberts, 1988). The idea was to assemble the genome from its 'l-tuples' (subwords of length l, referred to as $k$mers in the later literature). Pevzner used N. G. de Bruijn's work on finding Eulerian paths

[3]BAC: Bacterial Artificial Chromosome

(Aardenne-Ehrenfest and de Bruijn, 1951) to calculate the length of the l-tuples required to restrict the total number of possible assemblies.

Although sequencing by hybridisation is not used today to sequence DNA, the approaches originally developed to handle short reads apply equally well to short reads from the sequencing by synthesis methods currently used for high-throughput sequencing.

Pevzner and Tang (2001), Pevzner *et al.* (2001a) went on to reframe the problem of fragment assembly as an Eulerian cycle through a De Bruin graph rather than the Hamiltonian cycle required to solve the Shortest Superstring Problem. This made the problem tractable.

Assembly graphs

Read overlap and de Bruijn Graphs are two examples of *assembly graphs*. An assembly graph is defined with vertices representing sequences and edges between sequences that were seen adjacent in the sequence data. Assembly graphs provide an intermediate data structure between raw input data and a genome assembly. They facilitate operations to combine reads and remove errors and variants through bubble-popping (Myers, 2005). A run of vertices with in-degree and out-degree between them of one is known as a *unitig*. These are graph substructures that can be unambiguously merged together. When applied to de Bruijn Graphs, this creates what is known as a *compact de Bruijn Graph* (Chikhi *et al.*, 2016).

Ideal assembly

The ideal genome assembly can be thought of as a set of sequences that are mostly likely to have resulted in the reads given. We can formulate this as a Hamiltonian path through the read overlap graph, having estimated the copy number of vertices in the graph and duplicated those with copy number greater than one. A Hamiltonian path visits every vertex in the graph exactly once. However the Hamiltonian path problem has been shown to be NP-complete and therefore there exists no known polynomial time algorithm that can solve it.

Eulerian path assembly

The problem of visiting every vertex exactly once can be reformatted instead as traversing each edge exactly once – known as the Eulerian path problem. There may exist zero, one or many Eulerian path solutions for a given path. We can ensure there is at least one solution by adding edges to the graph to ensure all vertices have in-degree equal to their out-degree. This approach was first described and

implemented by the EULER assembler (Pevzner *et al.*, 2001b). The complexity of finding a Eulerian path is $O(N)$ where N is the number of vertices in the graph. The vertices in the read-overlap graph are reads or merged contiguous overlapping reads. Therefore this solution scales linearly with the number of input reads. The output is one or more likely paths for each connected component in the graph. The number of valid Eulerian paths through a connected subgraph increases very quickly with the number of reads. An algorithm for counting the number of such paths is the *BEST theorem* of which *N. G. de Bruijn* is an author (Aardenne-Ehrenfest and de Bruijn, 1951, Tutte and Smith, 1941).

String graph

At the *DIMACS Workshop on Combinatorial Methods for DNA Mapping and Sequencing* in 1994, E. Myers proposed the fragment overlap graph (now the read overlap graph) and Idury and Waterman presented the de Bruijn Graph – each proposed for the first time as different graphical approaches to genome assembly. Both approaches were published back to back in 1995 (Idury and Waterman, 1995, Myers, 1995). Pevzner later implemented the de Bruijn Graph for assembly in 2001 as the EULER assembler (Pevzner *et al.*, 2001a).

Myers later defined the String Graph – a theoretical construction – produced from taking an entire genome and collapsing it down at repeats greater than the read length, then removing transitive edges (Myers, 2005). Such a graph contains all the information in the reads. A string graph is *read coherent*, meaning any valid path through it is a valid assembly of the reads. Additionally the true genome itself is a valid path through the graph. In the same paper Myers also provided a linear time algorithm for constructing a String Graph from a read overlap graph (see Overlap Layout Consensus approach in Section 1.3).

In order to generate a read-overlap graph, we must find all overlaps between reads. Given N reads, there are $\frac{1}{2}N^2$ possible overlaps. Testing all possible overlaps is too computationally expensive for modern sequencing projects. The issue was overcome by *SGA* (Simpson and Durbin, 2012) using the FM-index (see Section 1.4.4), which enables searching for a subsequence of length m in $O(m)$ time, therefore finding all overlaps takes $O(N \times M)$ where N is the number of reads and M the maximum overlap. A graph can be constructed with reads as vertices and edges as overlaps. Simplification of the graphs using basic operations to merge contiguous reads and pop small graph bubbles results in a *String Graph*. Extracting the sequence associated with each graph vertex gives a set of assembled contigs. The read-overlap / String Graph approach very naturally uses all the connectivity information in individual reads, as we are not breaking them up into independent $k$mers (as done in de Bruijn Graphs). It can also exploit base quality scores in the read overlap detection step. The inclusion of mate pair information is not straightforward, as which reads are involved in each collapsed vertex is not usually recorded to reduce

memory usage. SGA can assemble a single human genome with peak memory usage of 54GB (Simpson and Durbin, 2012). As previously mentioned, this string graph memory requirement increases linearly with the number of reads.

1.3.1 Contig scaffolding

Once a set of contigs have been generated, the scaffolding step groups contigs into chromosomes and orders them. Scaffolding contigs is NP-complete (Pop *et al.*, 2004) requiring heuristics for a practical solution. Implementations include *Bambus* (Pop *et al.*, 2004), *SSPACE* (Boetzer and Pirovano, 2014), BBEST (Sahlin *et al.*, 2014), *WiseScaffolder* (Farrant *et al.*, 2015) and MeDuSa (Bosi *et al.*, 2015).

Linkage of variants in F_2 crosses can be used to group and order contigs (Mascher *et al.*, 2013). Linkage Disequilibrium (LD) can be used to place contigs, as was done with 4Mbp in the human genome (Genovese *et al.*, 2013). However, LD decays quickly and population structure and admixture can lead to spurious high LD, as seen in confounded GWAS studies (Aissani, 2014).

A similar approach is used with Hi-C (Belton *et al.*, 2012, Lieberman-Aiden *et al.*, 2009, Putnam *et al.*, 2016) chromosome proximity data to assign contigs to chromosome and order them (Burton *et al.*, 2013). Hi-C is based on Chromosome Conformation Capture (3C) whereby chromosomes are cross-linked with formaldehyde to physically-nearby chromosome. Sequences adjacent to cross-links are then sequenced.

A set of contigs do not necessarily represent a single fragmentation of the genome. Contigs may have multiple copy number which would cause the overall total length of contigs to be less than the expected genome size. Contigs can also have long overlaps with each other – caused by extending outwards into repeats which cannot be resolved unambiguously. Therefore contigs from high coverage data are likely to start and end with copies of repeats or errors (Simpson and Durbin, 2012).

1.4 De Bruijn graphs

Nicolaas Govert de Bruijn's first work on what are now called de Bruijn Graphs (dBG) was published in de Bruijn (1946), where he described a graph for creating de Bruijn Sequences. Given a set of words W, a de Bruijn Sequence for the set W is a sequence that has as substrings each of the elements of W. The de Bruijn Sequence problem is to find the shortest sequence that contains all words as substrings. This is very close to the assembly problem of finding the shortest sequence (assembly) that contains all of the DNA reads as substrings. De Bruijn's initial work was limited to a binary alphabet. He expanded his work to

n-letter graphs in de Bruijn (1975), in which he also showed that the question of de Bruijn Sequences had already been answered in 1894 by C. Flye Sainte-Marie. This work also led into calculating the number of Eulerian cycles in a graph in an algorithm called BEST (Aardenne-Ehrenfest and de Bruijn, 1951, Tutte and Smith, 1941), named after the authors: de Bruijn, van Aardenne-Ehrenfest, Smith and Tutte. A Eulerian cycle on a de Bruijn Graph is the solution to the problem of finding a minimal de Bruijn Sequence. This algorithm is used in Eulerian path approaches to sequence assembly (as in EULER, see Section 1.5).

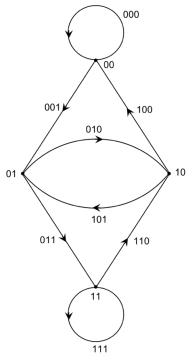


Figure 1.3: Completely connected de Bruijn Graph, as described in de Bruijn (1946). The first de Bruijn Graph showing vertex length $k = 2$ over the alphabet {0,1}

Although N. G. de Bruijn has not worked on sequence assembly, his mathematical work has has laid the foundations for it. De Bruijn Graphs were first used for fragment assembly in Pevzner (1989), work that was developed further by Idury and Waterman (1995). Also called $k$mer graphs or l-tuples, they are 4-letter (bases A,C,G,T) de Bruijn Graphs with read coverage information. This means that they lacked vertices and edges where there was no sequence covering them, and therefore were not complete de Bruijn Graphs as initially defined. The de Bruijn Graph was also given a formal mathematical grounding by Alekseyev and Pevzner (2007) when looking at whole-genome duplication events.

Formally, a de Bruijn Graph is a set of vertices and edges with a single parameter: 'word length' k. Each vertex has a unique label of length k, which is a word over the given alphabet. Given vertices (v,w), there exists an edge $v \to w$ if the last $k-1$ characters of v equal the first $k-1$ character of w. An example is shown in Figure 1.3 for $k = 2$ over the alphabet {0,1}.

De Bruijn graphs are commonly implemented on computers using hash tables to map a fixed length sequence (a $k$mer) to a vertex in a graph. Each vertex has directed edges that can be followed in a path giving a possible assembly. Alternatively it can be implemented as a set of edges of length $k+1$, with adjacent edges being identified by those sharing the first or last k bases. Since a de Bruijn Graph of $k+1$ is the *line graph* of the de Bruijn Graph of k, these two implementations – k vertex based and $k+1$ edge

based – are equivalent and are therefore just an implementation detail[4].

When implemented for sequence assembly, the set of vertices is the set of all subwords of length k in the read data. In some implementations (Iqbal *et al.*, 2012), an edge $v \to w$ has the additional constraint of only existing if $v + w[k]$ is in the read data, where $+$ is the concatenation of strings and $w[k]$ is the last character of vertex w. An example of a de Bruijn Graph as used in sequence assembly is shown in Figure 1.4.



Figure 1.4: De Bruijn graph with $k = 5$. Region in red matches between reads, causing the graph to collapse down at these positions. This collapsing down is called a 'tangle'.

With short reads, the high density of read starts means it is now possible to estimate how often a $k$mer appears in the genome given its $k$mer coverage. Assuming uniform sampling of reads across the genome and that most $k$mers in the genome are unique, mean $k$mer coverage (the number of times a $k$mer is seen in the sequencing data) can be estimated from sequencing depth and mean read length (Iqbal *et al.*, 2012):

$$k\text{mer coverage} = \frac{D \times (R - k + 1)}{R} \tag{1.1}$$

where D is the per-base sequencing depth, R is read length and k is $k$mer size.

The log-odds or A-statistic from Myers (2005) (first used in Celera assembler in 2000) can be used to label an edge as traversed <1, $= 1$ or >1 times. This was less reliable when using fewer long reads with more variable length.

De Bruijn graphs were extended to incorporate multiple alignment data as in A-Bruijn graphs (Zhi *et al.*, 2006), used in versions of EULER (see Section 1.5). Multiple alignment is computationally intensive, and the resulting graphs require larger amounts of RAM. However they have been shown to be useful for examining repeat domains and *de novo* generated repeat libraries (Zhi *et al.*, 2006).

[4]As pointed out by Heng Li `http://lh3.github.io/2017/11/15/on-assembly-de-bruijn-graphs`

Optimal k in a de Bruijn Graph

Selecting the best value for k depends on genome repeat content, sequencing error and coverage. It is not always possible to know the best value of k to use before constructing a de Bruijn Graph, although some heuristics exist. Increasing k reduces the number of false edges between $k$mers. Decreasing k increases $k$mer coverage (improving the distinction between erroneous and read $k$mers) and captures shorter overlaps between reads. k should be shorter than the distance between errors and ideally just over half a read length, to increase the probability that a sequencing error will create a disconnected component in the graph rather than a bubble or false path between existing $k$mers. k can be empirically determined by building graphs with varying k and selecting the one that optimises some metric (e.g. unitig N50) such as the VelvetOptimizer[5] or the Cortex pipeline (Iqbal *et al.*, 2013). Alternatively, optimal k can be estimated by quickly generating approximate $k$mer coverage histograms for various values of k as with *KmerGenie* (Chikhi and Medvedev, 2014); or with similar projects[6].

In long reads with high error (so-called third generation sequencing data), the two heuristics given directly contradict each other. In order to tolerate the high error rates much of the connectivity information in the read must be thrown away.

Coloured de Bruijn Graph

A coloured de Bruijn Graph (CdBG) (Iqbal *et al.*, 2012) is a de Bruijn Graph that represents multiple samples. It can be thought of as a de Bruijn Graph for each sample, loaded into memory in parallel. Coloured de Bruijn Graphs allow us to compare the de Bruijn Graphs of different samples.

In its simplest implementation, a coloured de Bruijn Graph with parameter k need only answer the query "*does sample x contain the kmer S?*". There are many possible ways to implement a coloured de Bruijn Graph that trade off memory usage, build time and disk space (Almodaresi *et al.*, 2017, Iqbal *et al.*, 2012, Minkin *et al.*, 2017, Muggli *et al.*, 2017). Some applications such as genotyping require the de Bruijn Graph to return the coverage per sample rather than just presence/absence.

1.4.1 De Bruijn graph assembly

De Bruijn graphs can be used to assemble a set of contigs by constructing a de Bruijn Graph from the reads, cleaning off the low coverage unitigs (run of connected $k$mers with in-degree and out-degree between them of 1) and outputting the remaining unitigs. This approach is simple but naive implementations

[5] Velvet Optimiser – http://bioinformatics.net.au/software.velvetoptimiser.shtml
[6] `optimal_k`: `https://github.com/ksahlin/optimal_k`

use a significant amount of memory (e.g. $\sim$ 60 GB for a single human genome with Cortex (Iqbal *et al.*, 2012)) and give very short contigs. This approach is also sensitive to the value of k used, which has to be optimised through repeated runs of the assembly. A large k gives longer unitigs as it resolves longer repeats, however increasing k also decreases effective $k$mer coverage, reducing the effectiveness of the error cleaning and introducing gaps in coverage.

The memory required for a de Bruijn Graph increases linearly with the number of errors, after coverage is sufficient to sample the entire genome. Since error rates are of the order $\sim$1%, de Bruijn Graphs are better suited to handling very high coverage samples than string graphs (e.g. above 100X).

1.4.2 $k$mer counting

The task of $k$mer counting is to take in sequencing reads and output a list of the $k$mers seen and how many times they were seen. This then facilitates the removal of low count $k$mers (which are likely due to errors) while streaming over the input reads (without loading all the reads into memory). The distribution of $k$mer counts indicates the coverage, repeat content and error rate of the sequencing data. $k$mer counting is often the first step in error correcting reads. Although $k$mer counting is simple, implementations can become complex to achieve speed and low memory and disk usage (Pérez *et al.*, 2016). Implementations include Tallymer (Kurtz *et al.*, 2008), BFCounter (Melsted and Pritchard, 2011), Jellyfish (Marçais and Kingsford, 2011), DSK (Rizk *et al.*, 2013), KMC (Deorowicz *et al.*, 2013), KMC2 (Deorowicz *et al.*, 2015) and khmer (Crusoe *et al.*, 2015, Zhang *et al.*, 2014b).

1.4.3 Low memory de Bruijn Graph representations

Using a 64 bit word to store a 32 base $k$mer (2 bits per base), storing the list of $k$mers in the human genome requires $\sim 2.6 \times 10^9 \times 8$ bytes = 19 GiB. Storing edges in a single byte per $k$mer (4 bits for forward edges $\{A, C, G, T\}$, 4 for reverse edges) adds an additional 2.4GiB. If the $k$mers are stored in a hash table with a load of 0.75 (25% of the hash table memory is empty), we require 29 GiB of RAM. If we store coverage for each $k$mer in a 32 bit unsigned integer ($k$mer coverage up to $\sim$ 4 billion), we require a total of 42GiB of RAM. Sequencing error generates additional $k$mers – assuming 100 base reads, 30X coverage and a per-base error rate of 0.01 we get 2.4×10^8 additional $k$mers for a total memory consumption of 46GiB. We may require additional memory for storing graph annotation information or loading multiple samples at once etc. Such high memory requirements need specialized hardware which causes issues – high memory machines are expensive and are often not found in biology labs.

To compound matters, we have only looked at the human genome (∼3 billion bases) – larger genomes exist:

1. Metagenome samples (e.g. human gut) can contain up to 7 Gbp (Qin *et al.*, 2010)
2. *Pinus taeda* (loblolly pine): 22 Gbp – largest sequenced genome (Neale *et al.*, 2014)
3. *Protopterus aethiopicus* (marbled lungfish): 133 Gbp – largest vertebrate genome (Leitch, 2007)
4. *Paris japonica* (Japanese flower) 150 Gbp – largest known eukaryotic genome (Pellicer *et al.*, 2010)

For these reasons it would be useful to have a lower memory representation of the de Bruijn Graph. It may be worth making speed / memory trade-offs to reduce the peak memory required to assemble and analyse a genome. Possible motivations include: handling metagenomic data and larger genomes, running on off-the-shelf clusters without custom high memory machines, using whole genome graphs in analyses on ones laptop, loading multiple sample graphs at once to compare, using pan-genome reference graphs to analyse samples and saving memory for additional graph annotations / data structures.

A de Bruijn Graph data structure G is equivalently a set of $k$mers, with an edge between a pair of $k$mers if they overlap by $k-1$ bases. The operations required are:

1. $dBExist(kmer, G)$ returns true/false if $k$mer exists
2. $dBNext(kmer, G)$ returns set of next $k$mers
3. Optionally: ability to iterate over the entire set of $k$mers

$dBNext(kmer, G)$ can be implemented by looking up each of the next four possible $k$mers. Storing edges is not required and is usually done only as an optimisation to speed up graph traversal – looking up the next $k$mer requires 1/4 of the lookups if we know which $k$mer is next (only one $k$mer is next for the majority of $k$mers).

Iterating over $k$mers is important for many graph operations such as variant calling and error cleaning. We can overcome the lack of an iteration function if we instead store enough seed $k$mers such that every $k$mer in the graph can be reached from a seed $k$mer using the $dBNext(kmer)$ function repeatedly. Below we review some of the data structures used to represent de Bruijn Graphs alongside a discussion of the trade-offs made.

Reversible hash functions

The naïve approach already discussed is storing a (possibly ordered) list or hash table of $k$mers, using 2 bits per base. Hash table memory can be reduced by using an approach first used by JELLYFISH

(Marçais and Kingsford, 2011) $k$mer-counting software. The hash of a object determines its position in the hash table, therefore the position of an object in the hash table gives you some information about the object itself. By using this information, we do not need to store the whole object itself in the hash table. This requires a *reversible hash function*, that is a hash function $f(x)$ for which we have a reverse function: $f'(f(x)) = x$ e.g. $f(x) = x \times 3$ and $f'(x) = x/3$ (for x modulus 2^N – i.e. x stored as a binary number).

If b is the number of items per bucket in our hash table, 2^l is the number of buckets, m is the number of bits per item and 2^r is the rehash limit, a hash table of size $b \times 2^l$ entries requires $(m - l + r)$ bits per entry. If $m \leq l$ you do not need a hash table as there are no collisions – use a bit array, one bit per entry. A bit array is viable for $k = 15$ which uses 1 GiB or $k = 17$ which uses 16 GiB. Beyond that it becomes unmanageable. With 30 items per bucket and 2^{27} buckets, we can store 3.01 billion items with load factor 0.75. This requires $62 - 27 + 4 = 39$ bits per item, a reduction of ∼40%. Total memory is 18 GiB vs. 30 GiB for a simple hash table using 64 bit words for $k = 31$ $k$mers.

Sparse bitsets

Conway and Bromage (2011) used information theory to calculate the minimum theoretical amount of memory required to store a set of $k$mers. For a given k, we have 4^k possible $k$mers, of which we have N in our de Bruijn Graph. If we list the $k$mers using 2 bits per base we have $2kN$ bits. If we know we have N $k$mers, there are $\binom{2^k}{N}$ possible $k$mer sets that we have in memory. Therefore $log_2\binom{2^k}{N}$ bits are required to record the set. For large N and small k (common for genome analysis), this is less than storing the explicit list. For human at $k = 26$, they calculate this theoretical limit at 12GiB.

A data structure that stores a list of zeros and ones is known as a *bit array* or *bit set*. If very few bits are set, it is called *sparse*. A data structure that uses close to the theoretical limit of memory required to represent the information held is called a *succinct* data structure.

We can store our set of $k$mers with a *succinct sparse bitset*, of which there are a few implementations. They store a compressed representation of 4^k bits, one bit representing each possible $k$mer, set to one if present in the set, zero if absent. In order to query the bitset, *rank* and *select* operations are implemented. The rank operation reports for a given position how many positions before it were non-empty. The select operation reports the position of the i-th set bit – this is the reverse of the rank operation. An example of this is given in Table 1.1.

Stored bits:	0	1	1	0	0	1	0	1	0	1
Index i:	0	1	2	3	4	5	6	7	8	9
$rank(i)$:	0	1	2	2	2	3	3	4	4	5

(a) Rank operation: returns how many of the stored bits $(0..i)$ are set.

j:	0	1	2	3	4
$select(j)$:	1	2	5	7	9

(b) Select operation: returns the index of the j-th stored bit

Table 1.1: Rank and Select operations for an example bit array

Using a sparse bitset, we can implement $dBExist(x) = rank(x) - rank(x-1)$. We can iterate over $k$mers with $x = select(i)\ for\ i\ in\ 1..N$.

Conway and Bromage used an entropy-compressed rank/select dictionary from Okanohara and Sadakane (2007) and achieved 16GB for the graph after cleaning. This data structure is static – it cannot be modified and adding / removing edges requires constructing a new graph. $k$mers are sorted on disk before the graph is constructed in memory.

Bloom filters

A Bloom filter is a probabilistic data structure for storing set membership, much like a hash set (Bloom, 1970). However when testing if a given element is in the set, it may return false positives but never false negatives. In other words the Bloom filter may say an element is in the set when it isn't, but will not say an element is not in the set when it is. This limitation allows for a vast reduction in the amount of memory required compared to a full hash table. This makes Bloom filters very useful if rare false positives can be tolerated.

Bloom filters are implemented as a bit array of M bits, with H independent hash functions. To add an element x to the Bloom filter, we generate a hash value with each of the hash functions $(h_0(x), ..., h_H(x))$ and set the corresponding bits in the bit array. To test if an element x is in the set, we generate the H hash values of x, and return `true` (the item *is probably* be in the set) if the all the bits are set, `false` (the item is *not* in the set) otherwise.

Given N elements, M bits and H hash functions the false positive rate can be calculated with Equation (1.2) and therefore the optimum number of hash functions can be calculated using Equation (1.3), both of which can be derived through basic probability (Mitzenmacher and Upfal, 2017, p. 116-118).

$$\text{Bloom false positive rate} = \left(1 - e^{\frac{-HN}{M}}\right)^H \tag{1.2}$$

$$\text{Bloom optimal number of hash functions } = \frac{M}{N} \ln 2 \tag{1.3}$$

Chikhi and Rizk (2013) implement low memory genome assember *Minia* using a Bloom filter with a hash table of false positives that can be hit during traversal (see Section 1.5). Salikhov *et al.* (2013) show that this can be improved upon with cascading Bloom filters. Holley *et al.* (2016) show how a Bloom Filter Trie can be used to quickly construct a low memory coloured de Bruijn Graph. Pellow *et al.* (2016) provide a review of different *k*mer Bloom filter approaches and their various trade-offs in speed and *k*mer retrieval false positive rate. *LightAssembler* (El-Metwally *et al.*, 2016) uses a pair of Bloom filters and achieves genome assembly with less than a quarter the memory requirement of *Minia*. Many Bloom filter approaches beat the lower bound on memory calculated by Conway and Bromage (2011) – see Section 1.5 for an explanation.

Hierarchical tree-like Bloom filters can be used to quickly query terabytes of sequencing data to search for subsequences (Solomon and Kingsford, 2016). For example, searching 2.7TB of gzipped fasta for a 1000 base sequence using a 40GiB index took less than four minutes on a single threaded machine (Solomon and Kingsford, 2018).

Bloom filters can also be used in error cleaning to test if a *k*mer is in the set of non-error *k*mers, where a small number of errors is acceptable (see Section 1.8).

1.4.4 Suffix trees, BWT and FM-index

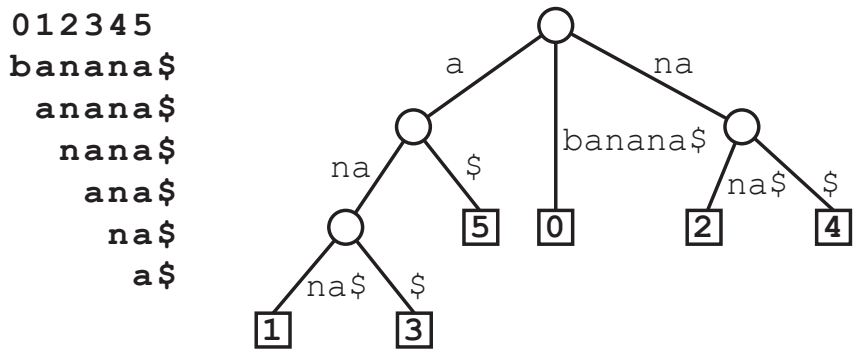


Figure 1.5: Suffix tree example for the word `banana`. The dollar symbol $ represents the end of the input. On the left are all non-empty suffixes of the input string. On the right, those suffixes are collapsed into a tree. Numbers in leaf nodes indicate the starting index of the suffix that ends at that leaf.

The suffix tree is a common data structure in computer science for string processing. An example, using the word 'banana' is shown in Figure 1.5. Suffix trees can be constructed in linear time with respect to input length using algorithms such as Ukkonen (1995), Weiner (1973) or Mccreight (1976). Once constructed, searching for an arbitrary string s in tree T takes $O(|s|)$, where $|s|$ is the length of s. The difficulties of using suffix trees come from the memory requirements and complexities in updating them. Compressing suffix trees against a reference suffix tree (Gagie *et al.*, 2015) shows promise in using suffix trees to represent multiple sample genomes.

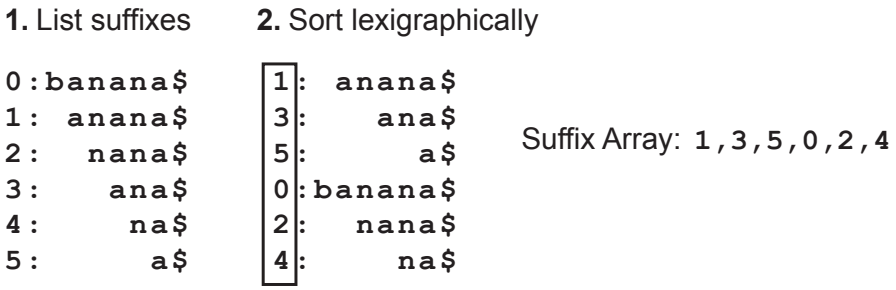


Figure 1.6: Suffix array example for the word `banana`. In the first step (left) we list all suffixes of the input string. In the second step we sort into Lexicographical order. The final Suffix Array (SA) is the list of starting indices for each suffix.

A suffix array is equivalent to a suffix tree stored as an array – it has linear space complexity. An example is given in Figure 1.6. Every substring of T is also the beginning of a suffix of T, therefore we can search for a substring S of T using a binary search over the suffix array in $O(|S| \log |T|)$ where $|S|$ is the length of S and $|T|$ is the length of T. Naive construction of a suffix array requires sorting all suffixes which takes $O(N \log N)$ time and requires storing the start position of each substring which requires $O(N)$ space where N is the length of the input string.

Suffix arrays can be sorted very quickly with 3-way radix quicksort (Bentley and Sedgewick, 1997), (also known as multikey quicksort), which sorts a large list of strings in asymptotically optimal time [7]. They can also be constructed by doing a lexicographic depth-first traversal of a suffix tree. Given that suffix trees can be constructed in linear time, this results in a linear time construction.

Multiple words can be stored in a *generalised suffix array*. Words w_i ($i \in 1 \ldots N$) are ended by a separator $\$_i$, which are given an ordering $\$_i < \$_j$ if and only if $i < j$. This is useful for storing a set of strings instead of a single string – for instance separate chromosomes in a reference genome.

[7] For large inputs, no worse than a constant multiplier slower the optimal solution

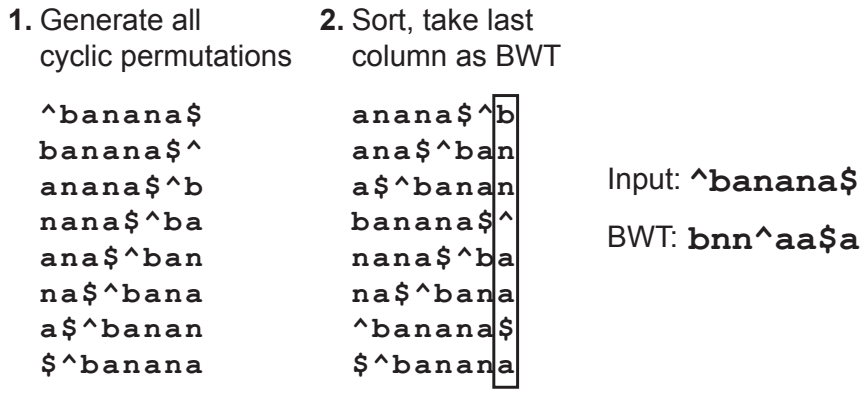


Figure 1.7: Burrows-Wheeler Transform (BWT) example for the word `banana`. In the first step (left) we shift the input string one character at a time to generate all the cyclic permutations of the input string. In the second step we sort all the cyclic permutation into Lexicographical order. The final Burrows-Wheeler Transform is a concatenation of the last character of each permutation.

Cazaux *et al.* (2014a) present a linear time algorithms to convert suffix arrays to de Bruijn graphs and also to contracted de Bruijn graphs (also called unitig graphs or compact de Bruijn graphs). Further they argue that constructing a de Bruijn graph from an underlying suffix array allows dynamically updating the order k of the de Bruijn graph.

The Burrows-Wheeler transform was proposed by Burrows and Wheeler (1994) as a means to compress data. There is a transform from a piece of text `T` to its transform `T'` (shown in Figure 1.7), and a reference transform from `T'` back to `T`. The `T'` has many matching characters grouped together, making compression on `T'` much more efficient than compression on `T`, using methods such as run length, move-to-front and Huffman encoding.



Figure 1.8: FM-index example for the word `banana`. Symbols $ and ^ mark the start and end (respectively) of the input string. After generating all cyclic permutations of the input string, they are lexicographically sorted. The vectors F and L – concatenations of the first and last characters of the permutations – are the resulting FM-index.

The FM-index proposed by Ferragina and Manzini (2000) is based on the Burrows-Wheeler Transform, originally developed for use in text compression. Once constructed, an FM-index can be searched for a sequence match of length N in $O(N)$ time complexity. Heavily used in read mapping (such as BWA (Li and Durbin, 2009)) and in finding read overlaps in the construction of string graphs (e.g. SGA (Simpson and Durbin, 2012)), the BWT/FM-index construction takes $O(L)$ time where L is the sum of the lengths of the sequences, when radix sort is used. Radix sort is good for small alphabets, otherwise sorting takes $O(N \log N)$.

Li (2014a) published *ropebwt* – a fast FM-index construction implementation suitable for long reads. Having the ability to store long reads also enables storing a unitig graph as an FM-index (with unitigs instead of reads as input) which facilitates searching for $k$mers of any k. The ability to map to a BWT representing an acylic graph was illustrated by GCSA (Sirén *et al.*, 2014).

The MEGAHIT assembler combines a FM-index with a sparse bit vector using rank/select. It is GPU accelerated and builds a succinct de Bruijn Graph at multiple $k$mer sizes. A 4Gbp soil sample was assembled in 44 hours using both the CPU and GPU, and in 99 hours using only the GPU, and required 345GB peak memory usage.

Minimizers

The de Bruijn Graph is a simple data structure that stores $k$mers, from which longer unitigs can be extracted. A graph of connected unitigs (also called a compact de Bruijn Graph) can be represented with much less memory than a de Bruijn Graph and enable traversal but does not have the ability to query a particular $k$mer. This is the same restriction as seen in Minia's Bloom filter backed de Bruijn Graph implementation (Chikhi and Rizk, 2013) (see Section 1.5). Unitig graphs still represent a genome assembly and can be used for variant calling. Once constructed a unitig graph can be stored in an FM-index, which allows querying individual $k$mers (Chikhi *et al.*, 2014a).

Minimizers enable the construction of unitigs from $k$mers without requiring that all $k$mers be loaded into memory first. Unitig computation with minimizers was first shown with *BCALM* (Chikhi *et al.*, 2014a). *BCALM* requires as input a set of $k$mers, so $k$mer counting must be performed first (see Section 1.4.2).

Minimizers work by partitioning $k$mers into groups such that $k$mers in a group are likely to be adjacent. For a given $k$mer we must select a single l-mer to represent it and group it with all other $k$mers that also are represented by that l-mer. We can then assemble these groups iteratively, merging them together. By dealing with only small numbers of $k$mers at any one time, we can assemble with much lower memory.

BCALM assembles a human genome in 2GiB of RAM. The algorithm is as follows:

1. Input: set of $k$mers; Choose some l where $0 < l < k$
2. Count all l-mers in input $k$mers:
   (a) 4^l total l-mers, for each one store 32 bit count (4 bytes) (e.g. $l = 10$, $4^{10} \times 4\text{bytes} = 4MB$ RAM required)
3. Iterate over $k$mers, for each one:
   (a) find l-mer L in the $k$mer with the lowest count
   (b) write the $k$mer to a file corresponding to l-mer L (e.g. $l = 10$, $4^{10} = \sim$ a million files required)
4. For each l-mer file:
   (a) load file contents into memory and merge $k$mers/ unitigs into longer unitigs
   (b) for each unitig, use the next lowest l-mer count and write to the corresponding file

As with picking k, l requires optimising for speed, memory and disk usage for a given genome. Unlike k however, the choice of l does not affect the assembly result. This approach requires a file for each l-mer. Modern file systems should cope with a million files, although many require spreading files between nested directories to avoid hitting limits on the number of entries in a single directory (e.g. ext3 limits 32,000 files per directory). Implementations may choose to optimise disk access by concatenating some files together (as BCALM does).

Step 4a requires hashing only the first and last $k - 1$ bases of each sequence. This enables us to find sequences that overlap by $k - 1$ bases indicating that they should be merged. If two or more sequences match one end of a sequence, we have hit the end of a unitig indicated by a fork (or bifurcation) in the graph.

Li (2016) showed how minimizers could be used to align long reads (**Minimap**) and then assemble them (**Miniasm**) (see 1.5.2).

1.5 Genome assemblers

In the last 20 years, sequencing technologies, computer power, algorithmic techniques and software engineering have developed dramatically. Over time various genome assemblers have been developed to exploit the latest advances. We review a few of them to cover the various techniques that have been used.

Celera

The Celera assembler was used to assemble the whole genome of *Drosophila* (Myers *et al.*, 2000). Based on ideas from Myers (1995), the Celera assembler used the overlap-layout-consensus approach in two stages. First it masked repeats using predefined repeat libraries and the log-odds ratio described earlier (see Section 1.4), then constructed contigs. This created two types of contigs: G-Contigs were restricted in length by low coverage either side; R-contigs were restricted by repeating regions. These contigs were then assembled into scaffolds using mate pair information.

Eugene Myers, the first author on the Celera paper, went on to work on the String Graph (Myers, 2005). The String Graph had grounds in the unitig approach of Celera, but offered a graph method that was argued to be more efficient then the de Bruijn Graph assemblers that succeeded Celera. It avoided the decomposition into $k$mers that causes high memory use in de Bruijn Graphs, and Myers presented linear algorithms for its construction. It has been implemented in the SGA assembler (Simpson and Durbin, 2012).

ARACHNE

ARACHNE was published in Batzoglou *et al.* (2002) and at the time the only other assembler intended for large complex eukaryotic genomes was Celera (see Section 1.5). ARACHNE borrowed several ideas from Celera, such as merging reads up to repeats (first described in Myers (1995)). It also uses the same overlap-layout-consensus approach.

First, read trimming based on quality scores is performed. Then overlaps are detected using a sorted $k$mer list – similar to a simplified implementation of the de Bruijn Graph. It works by taking all substrings of length k from all reads and storing them in a sorted list. Each $k$mer in the list is annotated with the read and positions in which it occurs. For each substring of length k in a read, a binary search algorithm takes $O(NlogN)$ time to find all reads that share the substring. This approach was first described in Batzoglou's PhD thesis (Batzoglou, 2000).

For each overlap, multiple alignment is performed for all reads involved (for which they created their own multiple aligner for efficiency using dynamic programming). This step also includes error correction of reads that have a mismatched base with a low quality score. It then disregards poor overlaps and chimeric reads. A mate pair of reads is merged with another pair if both forward reads and both reverse reads overlap giving reasonable insert lengths. This merging of reads is repeated to create contigs. Contigs are stopped before they cross repeat boundaries. These are known as *unique contigs*, or *unitigs* as called in Celera. Repeat boundaries are identified firstly by high $k$mer coverage (log-odds is used from

Myers (1995), see Section 1.4); secondly from contigs converging or diverging. Contigs are then joined into *supercontigs* (also known as *scaffolds*) by using mate pairing between contigs. At least two mate pairs must join contigs, and priority is given to joins that have the most pairs linking them. Gaps in supercontigs are then filled by one or more repeating contigs, using multiple alignment. Gaps will be left where there is insufficient read coverage.

The use of quality scores (q-scores) and multiple alignment throughout improves the chance of getting high quality alignment that is robust to sequencing errors. Assemblers Celera (Myers *et al.*, 2000), Velvet (Zerbino and Birney, 2008) and EULER (Pevzner *et al.*, 2001a) do not use q-scores after read trimming. ARACHNE achieves this without a large computational overhead. An assembly of *Drosophila* (175Mbp) took only 21 hours on a single 667MHz processor and used 8.4GB of memory (Batzoglou *et al.*, 2002).

Adaptations to make ARACHNE 2 were described in Jaffe *et al.* (2003) enabling the assembly of the mouse genome ($\sim$ 2.5 Gbp) (Chinwalla *et al.*, 2002) – a project involving 41 million reads. Three adjustments were included:

1. Breaking and shuffling of the scaffold using mate pairing brought a 64-fold increase in N50 length and appeared to eliminate misjoins
2. Mate pairing information was used to fill gaps between contigs bringing an additional N50 increase of 50%
3. Memory use was reduced fourfold and performance increased through software engineering improvements

The mouse genome was assembled using only 32GB of RAM and was completed in 15 days.

EULER

First published in Pevzner and Tang (2001), Pevzner *et al.* (2001a), the assembly problem was framed as that of finding a Eulerian cycle of a de Bruijn Graph (Eulerian method was shown to be practical by Myers (1995)).

EULER first uses an error correction 'Spectral Alignment Search' to correct errors in $k$mers, mapping them on to local $k$mers with higher coverage. To do this, each pair of $k$mers or $k$mer paths with one base different are compared, and merged if one path's coverage suggests it is the result of sequencing error. This was shown to halve the number of vertices ($k = 20$) on the *Neisseria Meningitidis* Genome from 9 million to 4 million (Pevzner *et al.*, 2001a).

EULER converts mate paired reads into 'mate-reads' by finding a path through the de Bruijn Graph between the two reads with a path length that satisfies the expected insert length. Chaisson *et al.* (2009) argue that this makes the read length irrelevant. This graph search can be expensive computationally and fails if more than one path exists with suitable length. Repeats in the graph (or tangles, see Figure 1.4) are resolved by matching entrance and exit paths using single reads and mate reads (Bresler *et al.*, 2013, Pevzner and Tang, 2001).

The problem of visiting every edge of a graph is known as the Chinese Postman Problem whose optimal solution is the Eulerian path when one exits. The problem of visiting every edge *exactly once* is known as the Eulerian Path Problem. Not all graphs have a Eulerian path – all vertices must have *in-degree* equal to *out-degree* and all vertices must be connected. Therefore multiplicities of the de Bruijn Graph edges must be set (how often it can be traversed – effectively duplicating an edge) to ensure at least one Eulerian path exists. This is done using the log-odds ratio described earlier (Section 1.4 and used in Celera (Myers, 2005, Myers *et al.*, 2000)). Eulerian paths can then be found in linear time (see Section 1.3).

In 2004, the 'RepeatGluer' algorithm for handling repeats and 'FragmentGluer' assembler algorithms were included into EULER v2.0 (Pevzner *et al.*, 2004). These algorithms combine a de Bruijn Graph with multiple alignment creating a so-called *Repeat* or *A-Bruijn* graph. This graph is constructed by doing pairwise local alignments between all reads and 'gluing' together aligned vertices. Inconsistencies in the alignments create short cycles in the A-Bruijn graph – classified as either 'whirls' (a cycle – edges $x \to y$ and $y \to x$) or 'bulges' (edges $x \to y \to z$ and an edge $x \to z$). These are subsequently cleaned up.

The pairwise alignment on large genomes was found to be computationally intractable in Zhi *et al.* (2006). The A-Bruijn Aligner was adapted and used to identify repeat domains and analyse *de novo* repeat libraries found through other means Zhi *et al.* (2006). A-Bruijn graphs have moved from assembly to repeat domain analysis where they can be used to explain biological development and implication of repeat structure. An online A-Bruijn alignment service was published by Jones *et al.* (2006) to provide multiple alignment of sequences to show domain rearranging, useful in analysing repeats and domain shuffling in proteins.

Finishing a genome with long reads can be done with EULER-PCR (Mulyukov and Pevzner, 2002). It handles gaps in coverage and pseudogaps (breaks in a contig due to a repeat).

Velvet

Velvet is the result of Zerbino's PhD thesis "Genome assembly and comparison using de Bruijn Graphs" (Zerbino, 2009). The initial algorithm and accompanying code were published in Zerbino and Birney (2008).

The novel features of Velvet included only using odd numbered k values in the de Bruijn Graph, in order to prevent $k$mers being their own reverse complement. It also used an intricate de Bruijn Graph that included reads glued to their reverse complement. This is a novel solution to not knowing which of the paired DNA strands a read is from and restricts graph searches to valid paths. It is a common problem in assembly that can significantly increase the computational load on an assembler, as considering each read twice doubles the input set.

Velvet includes error correction on its graph similar to other de Bruijn Graph approaches. The error correction step handles three classes of sequencing error as they appear in the graph:

Tip: a chain of vertices that is disconnected at one end in the graph. These are usually created when overlaps are lost because of errors near the ends of reads. Tips can also be caused by gaps in coverage and therefore Velvet only removes them if they are shorter than $2 \times k$ (where k is the word length of the graph).

Bulges: called "bubbles" elsewhere in the literature. These are paths in the graph that split and rejoin. They are caused by errors in the middle of reads or by repeats with SNPs on some copies. For this reason *bulges* are only compressed to the consensus sequence when all but one of the paths has low coverage.

Erroneous connections: where two or more reads converge in the graph without any further overlaps in the given direction. This can be due to cloning errors.

'Tour Bus' is an algorithm within Velvet for removing *bulges* from the de Bruijn Graph that have remained (due to high coverage or length) after the error correction step (see Figure 1.9). It follows paths through the graph to discover bubbles (a linear time algorithm). Upon discovering a bubble the two paths are aligned and compared, and if similar are merged together.

Velvet's use of mate pair information is all done by the 'Breadcrumb' algorithm (see Figure 1.10). When multiple paths exist between a couple of long contigs, mate pairs of reads on the long contigs are used to choose one path between them.

The Velvet algorithm as initially published was simplistic. The Tour Bus algorithm caused repeat copies (which may only differ by a few SNPs) to be merged, thereby masking repeats. No attempt is made by

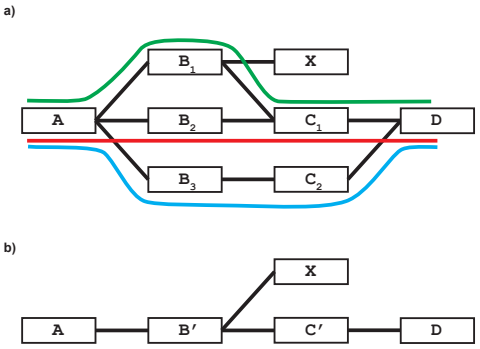


Figure 1.9: An example of the Velvet algorithm *Tour Bus*. a) A contig graph with a complex bubble and a tip (X). Vertices represent contigs and edges represent overlaps of $k-1$ bases. Three paths diverge through the bubble then rejoin (green, red and blue lines). b) The paths are collapsed down, merging the contigs B_1, B_2, B_3 into B' and C_1, C_2 into C'. The tip X remains.

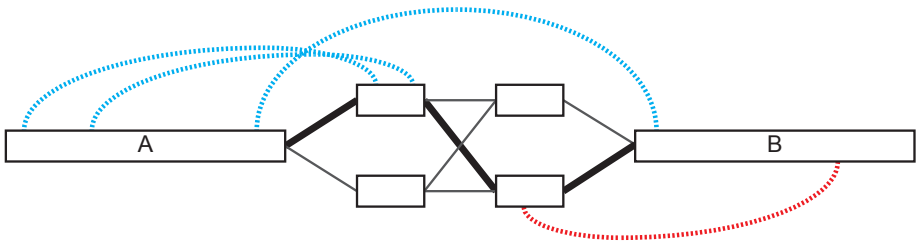


Figure 1.10: Velvet algorithm *Breadcrumb* for resolving the path between long contigs using mate pair information. Vertices represent contigs, and edges overlaps of $k-1$ bases. Dashed lines (red/blue) indicate read pairs of reads used to construct contigs A and B. Read pairs select the path shown in bold.

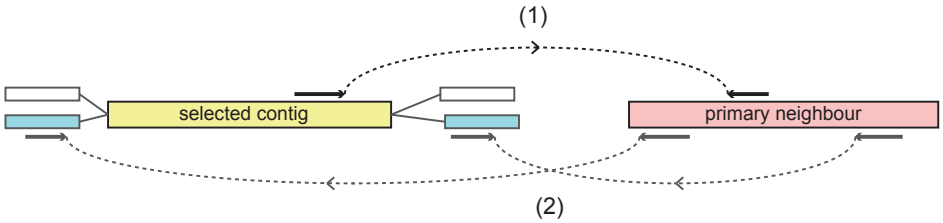


Figure 1.11: Velvet's *Pebble* algorithm in two steps. Vertices are contigs, shown as rectangles. Short arrows represent reads, and dashed arcs represent mate-pairings between reads. Step 1: read pairs on the selected contig are used to identify a 'primary neighbour' contig. Step 2: read pairs on the 'primary neighbour' are used to find contigs adjacent to the initial contig. The initial contig is then merged with the contigs selected by the second step (shown in blue).

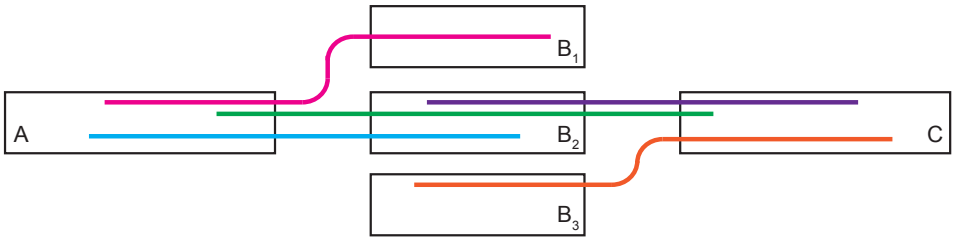


Figure 1.12: Velvet's *Rockband* algorithm uses long reads to identify contigs to be merged. Rectangles represent contigs, coloured lines represent long reads. The green contig indicates that contigs A, B_2, C can be merged into a single contig.

the algorithm to recover this information. Mate pair information was only used as an afterthought and not in the construction of contigs. Alterations to Velvet were published in Zerbino *et al.* (2009). This replaced Breadcrumb with a new algorithm ('Pebble') and added a new algorithm ('Rockband') for using long reads.

In order to extend a contig, Pebble finds a contig that it mate pairs forward to ('primary neighbour', see Figure 1.11). Then it finds all reads that that contig mate pairs back to. Due to the length of the contigs and variation of insert sizes, many of these reads will lie either side of the original contig. The original contig is then extended to include these new reads. In assembly of simulated data with 10% insert size variation, the ideal length for insert sizes was shown to be around 6,000 bases for this method.

The 'Rockband' algorithm uses long reads over short contigs to order them and fill in the gaps (see Figure 1.12). It is unclear how Rockband's use of long contigs adds much to Velvet's capability that is not provided by paired ends. It requires coverage by long reads, which make the short contigs they are binding almost redundant. What it achieves through resolving 'tangles' between contigs has been achieved in EULER with coverage metrics, network flow analysis and mate pair information. The use of long reads requires expensive additional laboratory techniques to finish genomes. EULER has also published an extension to use long reads in this manner (Mulyukov and Pevzner, 2002).

SPAdes Assembler

The Paired de Bruijn Graph was proposed as a de Bruijn based data structure to exploit mate pair reads (Medvedev *et al.*, 2011). This followed the observation that as the distance between $k$mers grows, the correlation between them drops. Additionally, the length of reads required to uniquely probe the genome drops dramatically when read pairs are available (Chikhi and Lavenier, 2009). Medvedev *et al.* (2011) proposed a new data structure (Paired De Bruijn Graph) where $k$mers are replaced by a pair of $k$mers $k_a|k_b$ where k_a is the i-th $k$mer from the first read and k_b is the i-th $k$mer from the second read. With error free reads and perfect coverage, they showed that a read length of 20bp and fragment size of 1000bp was enough to perfectly assemble *E. coli* and a read length of 300 bp with a fragment size of 1,300 bp was adequate to perfectly assemble human chromosome 22. This is promising as it is much easier to vary the insert size than to increase read lengths. To adjust to the variation in insert size, $k$mer pairs are merged if they have the same left hand $k$mer and their right hand $k$mer is within Δ bases within the underlying de Bruijn Graph. However even in the error free simulation, variance in the insert size of $\pm$ 10bp around a mean of 1,000bp caused the largest 100 contigs to drop from covering $\sim$ 100 % of chromosome 22 to covering only $\sim$ 50%. They point out that some researchers had managed to generate sequencing data with an insert size of 208 bp with a standard deviation of 13 bp (Bentley *et al.*, 2008). The distance between $k$mers in a $k$mer pair can be arbitrarily sized without changing the memory required.

The idea of Paired de Bruijn Graphs was extended by the *SPAdes* assembler (Bankevich *et al.*, 2012). SPAdes uses a multi-k de Bruijn Graph, allowing it to use large values of k to resolve low complexity regions, but then drop to using a lower k in regions of low coverage where connectivity would otherwise be lost. SPAdes uses read pairs to estimate the distance between pairs of unitigs, by using the Fourier transform on the distribution of distance estimates given by the reads. They then explore the graph separating the pair of unitigs to find paths that approximate the expected lengths. Given corrected distances between unitigs, a paired de Bruijn Graph can be constructed as previously described.

ExSPAnder (Prjibelski *et al.*, 2014), implemented as part of SPAdes and built on top of the idea of a *Rectangle Graph* (Vyahhi *et al.*, 2012), is a universal repeat resolver for genome assembly. It traverses up to a fork and chooses which edge to take based on the mapping of read pairs on the traversed contig and the proposed fork edges. In a way this is scaffolding whilst assembling.

ABySS

ABySS (Assembly By Short Sequences) (Birol *et al.*, 2009, Robertson *et al.*, 2010, Simpson *et al.*, 2009) is a distributed de Bruijn Graph assembler – it shares the work of assembly across multiple machines.

Distributed architectures can make use of much more memory allowing them to store more information about the input data. However distributed architectures are often slower than single machine architectures due to the slow cross talk between machines across the network. $k$mer memory access patterns are relatively random, making de Bruijn Graph assembly a less than ideal problem to solve with distributed computing. ABySS maps $k$mers to machines randomly, meaning when traversing the graph the next $k$mer has a $1/N$ probability of being on the same machine where N is the number of machines. Due to the slow nature of communication between networked machines, ABySS uses an asynchronous communication model and batches messages together to reduce the number of short packets sent across the network.

Performance of distributed de Bruijn Graphs can be improved by clustering adjacent $k$mers onto the same machine. One possible improvement would be to cluster similar $k$mers using Minimizers (see Section 1.4.4). First all machines in the cluster would agree a mapping $M(y)$ which maps each l-mer to a unique integer, where $l < k$. $M(x)$ could be a table of random numbers assigned to each l-mer, or a hash function that takes an l-mer and returns an integer. A $k$mer x is mapped to a machine (numbered $0 \dots N-1$) by finding the l-mer x which minimises $M(x)$. Since $k$mers share l-mers (except the first or last) with adjacent $k$mers, runs of adjacent $k$mers in the graph would be mapped onto the same machine.

Once the de Bruijn Graph has been constructed, ABySS carries out iterative cleaning to remove tips and pop bubbles. Unlike other de Bruijn Graph assemblers (Iqbal *et al.*, 2012), they do not remove low coverage unitigs. They assemble contigs by walking the graph until they hit a $k$mer with out-degree not equal to one (dead-end or fork). The last step uses read pair information to merge contigs. To do this the reads are mapped onto the contigs and contigs are considered paired if they are linked by $\geq p$ congruently mapped read pairs. Linked contigs are merged if there is a unique path between them whose length matches the expected gap between the read pairs. The graph search is an exhaustive strategy with heuristic optimisations.

ABySS2 has recently replaced the underlying de Bruijn Graph with a Paired de Bruijn Graph similar to *SPAdes* and implemented a Bloom filter to reduce memory usage caused by sequencing errors (Jackman *et al.*, 2017).

SGA

Overlap graphs must find all read overlaps longer than parameter τ, the minimum overlap permitted. These overlaps create a graph where vertices are the reads and edges are the overlaps. The graph is simplified by a transitive reduction step to remove redundant edges and contained reads. Given edges

$A \to B \to C$ and $A \to C$, the latter is a transitive edge to be removed. A contained read is one that is a substring of a longer read.

Edena (Hernandez *et al.*, 2008) was the first assembler to use a suffix array to find exact overlaps between reads in order to construct a string graph. Finding exact overlaps takes $O(N)$ time where N is the total number of input bases. Removing transitive edges and contained reads took $O(M^2)$ time where M is the number of reads.

The maximal overlap problem – finding only the overlaps that will not be removed by transitive reduction – can be solved optimally with suffix trees (Gusfield, 1997). However suffix trees require a large amount of memory making them unsuitable for mammalian genome assembly. Early string graph approaches required too much memory and were too slow to scale to deep sequencing of mammalian genomes with short reads. This led researchers to focus on de Bruijn Graph assemblers. However Simpson and Durbin (2010) developed an algorithm using the FM-index in place of a suffix tree to find read overlaps and transitively reduce the resulting graph at the same time with time complexity $O(N)$. Suffix array SA was constructed on a tuple of $\langle read, offset \rangle$, from which a BWT and FM-index are created. FM indices are created for the forward and reverse complement set of input reads. An array mapping read index to lexicographic index is created from the suffix array.

SGA (Simpson and Durbin, 2012) uses an overlap-graph approach to build a string graph as originally described by Myers (2005). First they use an FM-index to find all overlaps between reads with time and space complexity linear with the total length of the input sequences. From the overlaps they construct a graph with a vertex for each read and edges for the read overlaps. The graph is simplified in a series of steps by merging overlapping reads and removing vertices contained within other vertices. The output is the sequence associated with the remaining vertices.

Minia

De Bruijn graphs can be implemented using Bloom filters (see Section 1.4.3) as demonstrated by Chikhi and Rizk (2013) with their Minia assembler. They stored all $k$mers in a Bloom filter and used an additional hash table to store false positive $k$mers that were adjacent to $k$mers in the graph (see Figure 1.13). $dBNext(x, G)$ for $k$mer x and graph G, is implemented by testing for the next four possible $k$mers in the Bloom filter and keeping only the $k$mers that are in the Bloom filter but not in the hash table. $dBExists(x, G)$ cannot be implemented using this data structure, therefore it may be thought of as a traversal de Bruijn Graph instead of a true set of $k$mers.

By only storing critical false positives, Minia reduces the amount of memory required but also limits itself

to traversal rather than being a fully featured set of $k$mers with the ability to query if a particular $k$mer exists.

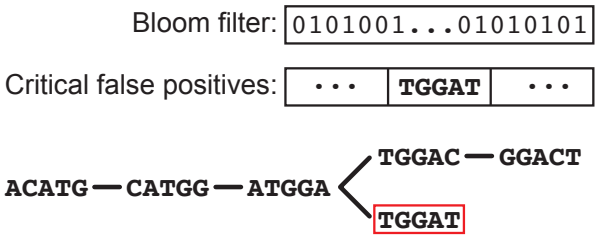


Figure 1.13: The Minia assembler uses a Bloom filter and a hash table of 'critical false positives' to store and traverse a de Bruijn Graph. Starting at seed $k$mer `ACATG` on the left, we check which of the next four possible $k$mers exist in the Bloom filter. The Bloom filter may return false positives, as in the case with $k$mer `TGGAT`. False positive $k$mers that branch off from true $k$mers are stored in the 'critical false positives' hash table. By checking if a $k$mer is a known false positive, we avoid traversing away from real $k$mers.

Minia requires only 5.7GB to assemble a real human genome sample (NA18507, SRA:SRX016231, 47X coverage) after discarding low count $k$mers (count < 3) leaving 2,712,827,800 $k$mers. An additional 78,762,871 $k$mers (2.9%) were stored as critical false positives. This is below Conway and Bromage's estimate of the theoretical minimum memory required (12GB) and their implementation (16GB) (Okanohara and Sadakane, 2007). Their theoretical minimum was calculated for storing set membership and had not taken into account that most $k$mers overlap by $k-1$ bases with at least two other $k$mers and therefore share a lot of redundant information.

Minia has been run on a RaspberryPi – a low power computer the size of a credit card, costing just £25. It has 512MB of RAM, 4 USB ports, 1 ethernet port, an SD card slot and has such low power requirements it can be run on small batteries. *Minia* was used on the RaspberryPi to assemble the Red Raspberry genome (400Mbp)[8].

Bloom filter are a form of *approximate membership query* data structures, others include the cuckoo and quotient filter. Pandey *et al.* (2017b) proposed a new *Counting Quotient Filter* to store approximate counts in a multiset, which was used to implement a low memory *weighted de Bruijn Graph* – one that stores $k$mer counts – called *deBGR* (Pandey *et al.*, 2017a). Similar to *Minia*, *deBGR* stores topological structure exactly but unlike *Minia*, it also stores approximate $k$mer counts.

[8] http://gatb.inria.fr/assembling-raspberry-on-raspberry-pi/

1.5.1 Other assemblers

Usually assembly is followed by a read mapping step to use mate pair information to scaffold contigs (see Section 1.3.1). Chikhi and Lavenier (2011) proposed a paired-string graph that could incorporate read pair information into the contig assembly step and implemented the idea in the *Monument* assembler. From a given starting vertex and using a local greedy algorithm it can assemble across small ambiguities such as bubbles and gaps. Longer jump libraries can still be used to scaffold the output contigs. The idea of incorporating mate pair information into contig assembly is also implemented in SPAdes (Medvedev *et al.*, 2011) Paired de Bruijn Graph (see Section 1.5).

The *StriDe* assembler (Huang and Liao, 2016) uses both string graph and de Bruijn Graph methods to assemble bacterial genomes. They show very good performance on GAGE-B bacterial genome comparison, beating existing genome assemblers. However it appears to be optimised for assembling bacterial genomes – it uses 2GiB for a 5Mbp genome – so is unlikely to scale to mammalian genomes. It extends the SGA code base, using the FM-index as a de Bruijn Graph – very similar to the proposal by Chikhi *et al.* (2014b). *StriDe* adds dynamic programming error correction and gap closing between paired end reads by a breadth-first search of the de Bruijn Graph. $k$mer statistics are used to detect errors in the string graph and reads are broken at suspected errors. Coverage of adjacent $k$mers is compared with a hard threshold to find errors. Overlap detection and graph simplification are carried out as in SGA, with additional steps, including the removal of graph motifs introduced by chimeric reads. The distribution of overlap sizes is examined to choose a threshold at which to discard certain read overlaps. Repeated bubble popping and tip-removal are carried out.

Reference-guided assembly, rather than *de novo*, uses existing sequences in the assembly of a new genome. The reference fills gaps in coverage and provides long range structure. This approach performed better than de novo assembly in a human resequencing project (Bailey *et al.*, 2001).

1.5.2 Long read assemblers

Long reads have a high error, low coverage and large overlaps, which present new challenges in genome assembly. They also provide long range connectivity information that can be used to assemble across repeats that were previously unresolvable with short read data. Recent approaches using long reads for assembly include **Minimap** and **Minasm** (Li, 2016).

Minimap (Li, 2016) finds overlaps between long reads by exploiting long overlaps. Minimizers (see 1.4.4) are used to summarise stretches of sequence (of length w) with a few representative $k$mers. Each long

read is then reduced to a list of summary $k$mers. These lists are effectively concatenated into a single list l and the result aligned to itself in a matrix M where $M[i, j] = 1$ if $l[i] = l[j]$. A *Hough Transform* inspired approach is used to find lines within the matrix M, and the results are mapped back to read overlaps. Miniasm is very fast – it was able to map 1.6Gbp of human SMRT sequence data in under three minutes. However it will not find short overlaps and therefore cannot be used with short reads.

Overlap detection is the first step of generating a string graph, and in the same paper an assembler is presented – **Miniasm** – which uses the overlaps found by Miniasm to construct a string graph, pop bubbles and generate an assembly using common graph simplification techniques. Miniasm uses a topological sort derived from Kahn (1962) to find bubbles. Similar bubble finding approaches are taken by Brankovic *et al.* (2016) and Onodera *et al.* (2013).

Unlike many assemblers, Miniasm does not attempt error correction. Miniasm can assemble 160X *E. coli* SMRT long read data in two minutes. Several bacterial genomes were assembled into a single contig per chromosome. Miniasm assemblies are likely to contain many sequencing errors and paralogous sequences will likely be collapsed down.

1.5.3 Diploid aware assembly

Bubble popping is an operation performed on a graph to identify graph paths that diverge and rejoin, and replace them with a single sequence. Bubbles may be caused by sequencing error, variation in a diploid sample or paralogous sequences (small differences between repeat copies). In error free simulations of human data with the ABySS assembler, it was shown that roughly 5% of contigs contained mismatches due to bubble popping ($k = 27$) (Simpson *et al.*, 2009).

Given a diploid sample with a high rate of polymorphisms, large structural events or bubbles missed by the bubble popping step, two paths may be needed to cover the assembly graph. In these situations, many duplicate contigs will be output by a haploid assembler, with only small variants differentiating them. The relation between these contigs is lost in the set of contigs. The total length of the contigs no longer approaches the genome length and is instead between $1 - 2\times$ the genome length. Duplicate contigs present issues during the scaffolding step as reads will not map well when presented with multiple valid mapping positions. One solution to this is to build diploid aware assemblers. The steps to adapting to diploid assembly are in theory not onerous – walk the graph with two 'walkers' in parallel that can diverge as long as they meet up again within a short distance. One example is the diploid-aware version of SPAdes, DiSPAdes (Safonova *et al.*, 2014). Similarly for PacBio SMRT long read sequencing data, the polyploid aware assembler *FALCON* (Chin *et al.*, 2016) has been developed.

Assembling from a haploid genome can result in much better N50 than assemblers trying to generate haploid assemblies using diploid genetic material (Steinberg *et al.*, 2014). Steinberg *et al.* (2014) assembled the haploid CHM1 human cell line and found 549 genes not present in the human reference genome, including some that are known to be clinically relevant. By working from a haploid genome broken into BAC libraries they were better able to identify segmental duplications – heterozygous regions in the graph can not be caused by allelic variation. They found 4% of the genome contains segmental duplication, of which 63% is intrachromosomal; 93% of segmental duplications are >10kbp with >94% identity.

1.5.4 Comparing assemblers

There are several issues with assembler comparisons: assemblers each make different trade-offs; published evaluations mix statistics from simulated vs real genomes, bacterial vs mammalian genomes; finding optimal parameters for assemblers in different settings is non-trivial; different sequencing technologies are used and coverage isn't always the same. Assembly techniques honed to specific platforms (or error models) are at a disadvantage when assembling reads from an alternative sequencing technology. This overfitting gives improvements in results in the short term, but is a disadvantage over time as sequence technologies develop.

Assemblers EULER, PHRAP, CAP and TIGR are briefly compared for assembling *Neisseria meningitidis* in (Pevzner and Tang, 2001, Pevzner *et al.*, 2001a). EULER was shown to perform best, achieving 99.1% of the genome covered with 149 contigs. The *Neisseria meningitidis* genome is 2.3Mbp long (Tettelin *et al.*, 2000).

Velvet, EDENA, SSAKE and VCAKE are compared for assembling *Pseudomonas syringae pv. syringae* B728a in Farrer *et al.* (2009). In the latter project, EULER-SR and SHARCGS were also attempted, but both consistently ran out of memory (32GB maximum was available). They concluded that the use of paired end reads by Velvet gave by far the best assemblies – longest and most accurate. SSAKE was the only other assembler that took mate pair information. They also showed that given 30× coverage and paired end reads it was possible to assemble *de novo* contigs spanning at least 96% of the genome. They also showed that the remaining unassembled regions were rich in noncoding RNA sequences and mobile genetic elements rather than protein coding genes.

When validating an assembly (disjoint set of contigs), it is treated as a hypothesis of the truth which can be tested experimentally (Bradnam *et al.*, 2013, Magoc *et al.*, 2013). In this framework reads are mapped back onto the assembly to assess its quality. This can identify mapping irregularities caused by misassemblies / misjoins; collapsed regions with unexpected mapping coverage; unmappable reads whose

sequence is missing from the assembly; and regions of the assembly with no read support.

N50 is an metric of contig lengths. It is calculated by sorting the contigs by length, and taking the N longest contigs such the sum of their lengths is equal to or just greater than the total sum of the contigs lengths – the N50 is the length of the N1st contig. A slightly different metric is that of NG50 which uses half the genome size instead of half the sum of all contigs. If you were to drop into the genome randomly, you have a 50% probability of being in a contig longer than the NG50.

Assemblathon I (Earl *et al.*, 2011) sought to compare assemblers through assembly of reads sampled from a simulated diploid genome ($\sim$ 100Mbp). Assemblies were aligned to the two haplotypes and simulated contamination to create a Multiple Sequence Alignment (MSA) graph called an *adjacency graph*. The authors presented several useful assembly metrics:

- For a pair of $k$mers on a true haplotype, the $k$mers are *correctly contiguous* in an assembly if they appear in the same alignment columns in the assembly against the MSA graph. The distance such that 50% of $k$mer pairs are correctly contiguous is referred to as **CC50**.
- Genome coverage is the fraction of the genome which is correctly assembled.
- Transcript coverage is the fraction of genes that are correctly assembled.
- Genic correctness is the fraction of bases in spliced transcripts that BLAST with 95% coverage in the assembly.

They showed that for the assemblies compared, some common assembly metrics correlated (e.g. scaffold NG50 and contig NG50, NG50 and genic correctness) whilst others did not (e.g. contig NG50 and genome coverage, substitution errors and NG50). This suggests N50/NG50 could be reliably used to compare assemblers. Of the 41 assemblies from 17 research groups, no assembler came out better than the others. Instead each assembler excelled only for a few metrics. This is concordant with a trade-off in assembler design between accuracy and contiguity. The extrapolation of the conclusions of Assemblathon I to real genomes is limited by the simulated genome being an order of magnitude smaller than mammalian genomes and having a much lower repeat content.

Assemblathon II (Bradnam *et al.*, 2013) used real short read sequence data from a bird (budgerigar), a fish (Lake Malawi cichlid) and a snake (boa constrictor) to compare 43 assemblies from 21 research teams. They suggested that assembling genes into a single contig is very important and therefore calculated for each assembly the proportion of genome assembled into contigs $\geq 25Kbp$ (the average length of a vertebrate gene). They found that this did not correlate with NG50 – for example the Ray Assembler (Boisvert *et al.*, 2010) bird assembly had the 9th longest NG50 scaffold length but the 2nd highest

proportion of genome assembled into scaffolds $\geq 25Kbp$ (out of 11 assemblies). Assemblathon II also looked for the fraction of 458 core eukaryotic genes that were each contained within a single scaffold which gives a similar metric indicating how genes have been assembled in an assembly. By calculating z-scores across all metrics Assemblathon II ranks assemblies for each species which they find is highly correlated to N50 but they conclude:

> These analyses reveal that – at least in this competition – it is very hard to make an assembly that performs consistently when assessed by different metrics within a species, or when assessed by the same metrics in different species.

Overall the best assemblies in Assemblathon II were for the snake genome despite having the largest genome and half the sequencing coverage of the bird genome. The authors suggest this is likely due to the relatively low repeat content and low heterozygosity.

Comparison of *k*mer based transcription assemblers is presented in Rana *et al.* (2016). Additional tools for comparing genome assemblies include CheckM (Parks *et al.*, 2015), REAPR (Hunt *et al.*, 2013) and QUAST (Gurevich *et al.*, 2013)

1.6 Variant calling

Variant discovery is the problem of finding genetic differences caused by mutations. These differences may exist within a single individual or between multiple individuals. Variant discovery is a fundamental step in genetics and underpins the study of diseases (such as cancer), biology and evolution. Genetic differences can be classified as Single Nucleotide Polymorphisms (SNPs), Multiple Nucleotide Polymorphisms (MNPs), insertions and deletions (indels), inversions, translocations and copy number variants (CNVs). Variants larger than SNPs are known as Structural Variation (SVs) (Feuk *et al.*, 2006).

The predominant strategy for finding variants is known as *'resequencing'* – sequencing samples with short reads and mapping these reads to a previously assembled reference genome (Korbel *et al.*, 2009, Li *et al.*, 2009). Reference genomes exist for a fraction of the extant species, but the number is growing. Reference genomes have the advantage of providing a shared coordinate system and being annotated (with genes, enhancer and promoter regions, CpG islands, repeats, duplications, known variants etc.) and allow the comparison of genotypes across samples. Mapping short reads to a genome limits the variants that can be discovered to those that occur on a background that has been assembled and can be mapped to reliably.

Mapped coverage can be used to indicate copy number variation of genes or entire chromosomes (Bailey

et al., 2001, Zare *et al.*, 2017). Events larger than the read length can be discovered using mate-pair reads and by looking at mapped coverage. Read pairs whose insert gap when mapped is either larger or smaller than the expected fragment size indicate insertion or deletion has taken place versus the reference. A marked increase or decrease in mapped coverage may similarly indicate that an insertion or deletion has taken place (Hajirasouliha *et al.*, 2010, Lee *et al.*, 2009). However these approaches do not resolve to base pair resolution, instead indicating only the loci affected.

Reads with split mappings can be used to call indels (Zhang *et al.*, 2011). *BreakDancer* (Chen *et al.*, 2009) can predict indels, inversions, and translocations up to 1Mbp in size through *anomalous read pairs*. It finds 'breakpoints' and does not resolve the sequence between the breakpoints. Ye *et al.* (2009) showed detection of small indels ($1 - 20$bp) within mapped reads as well as the breakpoints of larger indels ($1 - 10$kbp). LUMPY (Layer *et al.*, 2014) uses read-pair, split-read, read-depth and known variants to call breakpoints in multiple samples at once.

Reads (or pairs of reads) are mapped independently of each other, meaning the optimal alignment of reads to a region often does not yield the optimal multiple sequence alignment of those reads (Li, 2014b). In order to leverage the information of locally mapped reads, many variant callers include a realignment step that performs a multiple sequence alignment of reads. GATK performs a realignment step and Platypus has an equivalent local assembly step. Freebayes (Garrison and Marth, 2012) compares haplotypes (spanning multiple variants) from reads rather than their precise mapping, effectively equivalent to doing realignment.

Events close to (and beyond) the read length can be resolved if local assembly is used to reconstruct the alleles. Local assembly is a more constrained computational task than whole genome assembly that scales much more easily to large high coverage genomes. Mapping based callers with local assembly include GATK (DePristo *et al.*, 2011), Freebayes (Garrison and Marth, 2012), TIGRA (Chen *et al.*, 2014), Platypus (Rimmer *et al.*, 2014), Scalpel (Narzisi *et al.*, 2014) and MetaSV (Mohiyuddin *et al.*, 2015). Dindel (Albers *et al.*, 2011, Fang *et al.*, 2014) finds candidate indels from which it generates alternative haplotypes against which it realigns reads. These haplotypes are constructed from SNPs and indels indicated by the mapped reads as well as known variants. It can call insertions and deletions up to 50bp long with 100bp reads. SOAPIndel (Li *et al.*, 2013) uses unmapped reads with a mapped partner and local assembly with exhaustive traversal of the de Bruijn Graph to call indels. *MindTheGap* (Rizk *et al.*, 2014) uses a de Bruijn Graph approach to locally assemble insertions against the reference – it was able to assemble an insertion >1kbp from NA12878 human data in under 14GiB RAM (using the *Minia* assembler (Chikhi and Rizk, 2013)).

Discovar (Weisenfeld *et al.*, 2014) locally reassembles reads after mapping. The local assembly approach

matches read pairs up with supporting read pairs and merges them, similar to the approach used by Batzoglou *et al.* (2002). These resulting contigs are then joined at overlaps creating an acyclic graph which is anchored against the reference. Discovar was designed to call variants in the last 10% of human genome that existing variant callers struggle with, where they miss up to 25% of variants. It is built to work specifically with $\sim$ 60X 2x250bp reads from a PCR-free library and fragment sizes of $\sim$ 450bp. In validating their approach, they report that 30% of variation happens in 10% of genome which is made up of low complexity sequence and segmental duplications (Weisenfeld *et al.*, 2014).

GATK provides a machine learning filtering process called *Variant Quality Score Recalibration* (*VQSR*). It takes as input a set of putative variants and a set of highly trusted variants. The trusted set is usually the intersection of multiple callers from large sequencing projects such as *1000 Genomes*, masked to exclude low-complexity genome regions which typically have a higher rates of false calls (Li, 2014b). Using sites in the intersection of the call sets, VQSR learns the annotations of these calls in the putative set. Annotations include sequence context, base quality scores and allele read counts. This annotation profile is then used to assign quality scores (*variant quality score log-odds (VQSLOD)*) to all calls in the putative set. Calls are then filtered with a threshold chosen to have a given sensitivity to calls in the trusted call set. VQSR could be applied to calls generated by other methods as long as there are suitable annotations in the VCF. There have been reports that VQSR does not work well with single samples (Li, 2015b).

1.6.1 Variant calling quality

SMaSH (Talwalkar *et al.*, 2014) sets out a data set and methodology with which to compare SNP and indel callers for specificity and sensitivity on synthetic mammalian data. O'Rawe *et al.* (2013) compared variant calling approaches for human exome sequencing data. They used five pipelines: SOAPaligner+SOAPsnp, BWA+GATK, BWA+SNVer, GNUMAP and BWA+samtools. SNPs were validated by amplicon sequencing on the MiSeq platform. SNP calling sensitivity ranged from 86.60% (GNUMAP) to 95.34% (GATK1.5) and specificity from 99.59% (SAMtools) to 99.79% (SOAPsnp). For indel calling, SOAPindel, SAMtools and GATK had overall concordance of only 26.8%.

Li (2014b) used the CHM1hTERT haploid cell line (Jacobs *et al.*, 1980) and diploid NA12878 whole genome sequence to estimate variant calling error rates and sensitivity with several mapping-based variant callers: FreeBayes, SAMtools, GATK-UnifiedGenotyper, GATK-HaplotypeCaller and Platypus. Heterozygous calls for the haploid cell line (H_h) can be considered erroneous calls and compared to NA12878 heterozygous calls (N_d). The false positive call rates can then be estimated as N_h/N_d and sensitivity

estimated as $N_d - N_h$. They found the false positive rate of indel calling to be 10-40% in low complexity regions of the genome and 1-8% outside of low complexity regions. For the definition used, low complexity regions make up 2% of the genome but were responsible for 80-90% of indel and 60% of SNP false positives. The authors attribute this to PCR errors around homopolymers and poor read realignment. They propose a set of simple filters that can reduce the false call rate by a factor of ten. Homopolymers longer than 10bp are highly mutagenic, and allele frequencies do not show expected $1/f$ distribution (Albers *et al.*, 2011).

Some errors in variant calling can be caused by DNA damage during sample preparation (Chen *et al.*, 2017), errors incorporated during PCR (Ewing and Green, 1998) (more likely to cause indels (Li, 2014b) than SNPs) and errors during sequencing (such as around `GG` motifs on Illumina machines (Schirmer *et al.*, 2016)).

1.6.2 Reference bias

Mapping to a reference introduces biases for SNPs and indels (Lunter and Goodson, 2011, Stevenson *et al.*, 2013). Preference for mapping to the reference SNP allele has been demonstrated (Brandt *et al.*, 2015, Degner *et al.*, 2009). Deletions are more likely to be called than insertions (Zhang *et al.*, 2011). Reference genomes are usually constructed from a small group of individuals that are not necessarily a good genetic representation of the entire species (Dilthey *et al.*, 2015). Incomplete references (or references missing regions that are present in the sample) can result in more false positives for a given sample as reads from unassembled regions are mapped to assembled regions (Li, 2014b). The human reference genome may be missing up to 19-40 Mbp of sequence that is present in some human populations (Li *et al.*, 2010a). Assembly of 1000 Genomes low coverage data showed 3Mbp of sequence missing from the human reference (Iqbal *et al.*, 2012). Another study found 7Mbp of sequence including six predicted genes in an Asian individual not present in the reference genome (Cao *et al.*, 2015). Allelic sequences and segmental duplications are often confused both during assembly and read mapping, resulting in false positive variants caused by off-target alignment (Steinberg *et al.*, 2014).

One proposal to overcome reference bias is to augment the reference genome with known alternative alleles. The first such human reference was GRCh37 (Church *et al.*, 2011) where including alternative alleles was shown to reduce off-target sequence alignment (Church *et al.*, 2015). With synthetic data this has been shown to reduce sites with allelic bias by 63% and with RNA-seq data it was shown that this can increase the number of reads that map by 15% (Satya *et al.*, 2012).

Church *et al.* (2015) show the effect of alternative alleles in reducing mis-mapping and argue that we

should move to a graph-based reference, similar to the models used in genome assembly. A reference genome representing multiple samples or populations is known as a *pan-genome* (Nguyen *et al.*, 2015).

Another solution is to construct personal genomes for each individual before mapping (Chen *et al.*, 2016) – this requires known or estimated sample genotypes. This was done by Maretty *et al.* (2017) who *de novo* assembled 150 samples and compared the assemblies rather than map reads to a reference genome. If each sample is being assembled, one can instead compare the assembled genome to the reference (Nattestad and Schatz, 2016).

Dilthey *et al.* (2015) proposed a *population reference graph* – a reference augmented with alternative alleles for SNPs, indels and larger structural variants. Reads can be mapped to the resulting acyclic $k$mer graph using a hidden Markov model. The authors demonstrated the concept by showing how augmenting the 8.5Mbp highly variable MHC region with 8 assembled haplotypes and 87,640 known SNPs improved read mapping and genome inference.

Introducing variants into the reference introduces reference ambiguity and increases computational costs for mapping (Pritt and Langmead, 2018) which in turn harms the ability to map reads well. The authors argue this can be controlled by being selective about which variants are included in a graph and present a model for doing so: *FORGe*. They obtain the best results using 6-30% of variants from the 1000 Genomes project and discarding most variants with minor allele frequency <5%, as also done in Maciuca *et al.* (2016).

1.7 Mapping reads to graphs

Mapping to multiple genomes simultaneously was first proposed in Schneeberger *et al.* (2009). There are several solutions for mapping to a pan-genome represented by a linear reference augmented by alternative linear alleles using $k$mer based approaches (Schneeberger *et al.*, 2009) or BWT (Huang *et al.*, 2013). *BWBBLE* (Huang *et al.*, 2013) constructs a linear-reference multi-genome by appending to the reference alternative alleles padded by N flanking bases where N is the expected read length. The resulting alignment is slower but, the authors argue, much faster than aligning to all references in a collection one-at-a-time – especially as the number of alternative alleles increases slower than the number of genomes in a collection. Other proposals for encoding variation in a BWT, allowing mapping reads to a graph, have been made (Maciuca *et al.*, 2016).

When mapping reads to a flat sequence, reads that appear within a repeat cannot be placed canonically

as there are many equally valid positions they could be mapped to. Reads that overlap the end of one contig and the start of another are difficult to place. Repeats in the contigs that are artefacts of assembly rather than actual repeats in the genome (e.g. long unitig overlaps between a pair of unitigs from a de Bruijn Graph assembly caused by sequencing errors or variation that was confused for a repeat (Steinberg *et al.*, 2014)) will frustrate attempts to map reads to the contigs. Instead of mapping reads to the contigs of an assembly as a later step, reads could be mapped onto the de Bruijn Graph used to make the contigs.

Read mapping to de Bruijn Graphs (Liu *et al.*, 2016) was shown to be NP-complete (Limasset *et al.*, 2016). Limasset *et al.* (2016) proposed a greedy alignment algorithm that runs ∼38× faster than an exhaustive approach with real data from *E. coli*, with roughly equivalent results. The mapper they developed – a fast and low memory one – called BGREAT, requires only 10GiB of RAM for human WGS data. As input it takes a set of reads and a compact de Bruijn Graph (a unitig graph with $k-1$ bp overlaps). They construct an index of the k-1 bp overlaps between unitigs. Reads are scanned for $k$mers that are in the index – reads to be mapped must match perfectly with at least one of these overlaps. Once the first overlap has been identified, BGREAT proceeds greedily by finding the next overlap such that the sequence between them on the read aligns to the unitig on the graph with fewer than t mismatches. Only reads which span two or more unitigs will be mapped by BGREAT. The authors propose a pipeline (GGMAP) which takes as input a set of reads which it assembles into a compact de Bruijn Graph using the low memory unitig assembler *BCALM2* (Chikhi *et al.*, 2016). Next reads are mapped to this unitig graph using BGREAT. Finally reads that do not map are mapped to the unitigs with Bowtie2 (Langmead and Salzberg, 2012) (this step maps reads that are contained within unitigs). They found that mapping to a reference graph instead of flat contigs led to a 22% increase in the number of reads that map uniquely (for a highly fragmented assembly).

In a different approach to mapping reads to graphs, *GCSA*, an extension of BWT to acyclic labelled graphs has been proposed (Sirén *et al.*, 2014), which turns substring search queries into path queries of a graph. They apply this to a pan-genome of an assembled reference genome combined with known variants. They also demonstrate its utility in mapping to splice graphs with applications in RNA-seq. This approach is being built on by HISAT2[9] for mapping RNA-seq reads, and extended into VG (Garrison *et al.*, 2017). *VG* (Garrison *et al.*, 2018) provides a toolset for constructing, manipulating and mapping reads to variant graphs.

As well as being potentially useful in better scaffolding, mapping reads to a de Bruijn Graph also provides us with the first step in annotating the de Bruijn Graph with information potentially useful in traversal

[9] `https://github.com/infphilo/hisat2`

and assembly.

Paten *et al.* (2014) proposed representing both samples and reference genomes by a *sequence graph* in which each base is a node with a unique identifier. Graph mappings then relate one graph to another. A hierarchy of *reference structures* allows the reference to be updated without re-mapping all samples.

GraphMap(Sović *et al.*, 2016) maps Oxford Nanopore long reads from MinION to a graph structure. When compared to existing long- and short- read mappers, the authors found that it had an increased mapping sensitivity of 10–80%.

Reference-free and assembly based variant calling

Some variant calling approaches avoid reference dependence and bias by avoiding the use of one altogether.

mInDel (Lv *et al.*, 2016) finds amplicons containing indels from genome sequence without using a reference. First it assembles all samples using SOAPdenovo2 (Luo *et al.*, 2012) (for Illumina data) or Newbler (from 454 Life Sciences (Margulies *et al.*, 2005)). For one of the samples, overlapping amplicons and corresponding primers are selected that cover the entire genome. These primer pairs are mapped (using Bowtie (Langmead and Salzberg, 2012)) to each sample assembly in turn. Primer pairs that map at most once to each sample are used to infer approximate indel size and location. They validate the process using a sample of the indels called in maize, finding $\sim 80\%$ of the amplicons can be validated by PCR and gel-electrophoresis.

Identifying variants during genome assembly was proposed by Fasulo *et al.* (2002), by looking for bubbles in the fragment assembly graph. Bubble calling in acyclic graphs can be achieved with $O(m\ log\ n)$ time (Sung *et al.*, 2015) and more recently in $O(m + n)$ (Birmelé *et al.*, 2012, Brankovic *et al.*, 2016). Of course this does not apply to a graph such as the de Bruijn Graph where there may be cycles but it can be applied to assembly graphs such as those used by Dilthey *et al.* (2015) and Discovar (Weisenfeld *et al.*, 2014). Such acyclic variant graphs are constructed from a reference genome, known variation and mapped reads.

Cortex (Iqbal *et al.*, 2012, 2013) uses a coloured de Bruijn Graph data structure to represent multiple samples including reference genomes (if available). The de Bruijn Graph is cleaned of low coverage $k$mers (and short tips), and simple bubble structures are identified as putative variants. A coverage model and Hardy-Weinberg filter are used to remove bubbles likely to be caused by repeats and sequencing errors. If a reference is available, bubble flanking sequence is mapped to it to place variants on a common coordinate system. Cortex has been used to call variants in ten chimpanzee genomes at once (each sample a separate

colour) and 164 low coverage human genomes from the 1000 genome project (The 1000 Genomes Project Consortium, 2010) when merged into a single colour (Iqbal *et al.*, 2012). The Cortex pipeline attempts variant calling with multiple values of k and merges the result (Iqbal *et al.*, 2013).

Fermi (Li, 2012) assembles a unitig graph from a string overlap graph with a similar approach to SGA. They use a novel single bi-directional DNA FM-index (FMD) rather than two separate indices of the reads in their forward and reverse complement orientations. This speeds up queries by 80%. Error correction is done for single base changes by constructing an index of all 23-mers in the reads along with the number of A, C, G and T bases that follow each of its occurrences. For each base in the read they use the base quality and flanking 23-mers to decide if the base is an error and which base is most likely correct. The *FMD-index* is used to find the longest overlaps between reads to construct a string graph. This is then simplified into a unitig graph by merging reads with one-to-one edges. Reads are mapped back to unitigs to get read counts on the unitigs. A topological sort is used to find and collapse down complex bubbles – two or more paths through the graph are merged into one if they end at the same vertex and only one has high coverage. Short fragment read pairs are used to merge and break contigs. If five or more read pairs link a pair of unitigs they are merged and overlapping reads are used to estimate the gap between the contigs. Contigs are broken at regions with no bridging reading pairs to correct assembly errors. Finally unitigs are mapped to a reference genome and variants are called from the gaps and mismatches in the alignments. The authors find that the sensitivity and specificity of this SNP calling is roughly equivalent to read mapping approaches and their indel calling is much better. Unlike from de Bruijn Graphs, unitig graphs created via overlap graphs do not lose information from the reads. Li makes the argument that such a unitig graph can therefore be considered a compression of the reads.

Fermi has been incorporated into the *FermiKit* (Li, 2015b) de novo variant calling pipeline. *FermiKit* uses *BFC* (Li, 2015a) for error correction, ropeBWT2 (Li, 2014a) to construct the BWT and BWA-MEM (Li, 2013) to map unitigs.

Graphtyper discovers new variants and genotype samples by mapping short-reads to a variant graph (Eggertsson *et al.*, 2017). Maretty *et al.* (2017) used short read sequencing and *ALLPATHS-LG* to assembly 150 genomes from Danes. The assembled genomes were each aligned to the human reference to call variants resulting in a very high quality variant set including novel sequence not present in the reference genome.

There are several reference-free variant callers and sample classifiers designed for metagenomic data including compareads (Maillet *et al.*, 2012), crAss (Dutilh *et al.*, 2012), MaryGold (Nijkamp *et al.*, 2013), Bubbleparse (Leggett *et al.*, 2013), DiscoSNP (Uricaru *et al.*, 2015), Commet (Maillet *et al.*, 2014), and FOCUS (Silva *et al.*, 2014), metafast (Ulyantsev *et al.*, 2016) and LueVari (Alipanahi *et al.*, 2017).

Inversions

Inversions can be found between short pairs of sequences through augmenting the traditional Smith-Waterman pairwise alignment algorithm (Schöniger and Waterman, 1992).

Inversion calling using a de Bruijn Graph was implemented in *TakeABreak* (Lemaitre *et al.*, 2014). The idea is to look for the graph motif induced by clean inversions (see Figure 1.14) and filter to remove sequences that are not inversions and those that are flanked by low complexity sequence. Inversions create bubbles in de Bruijn Graphs of $k-1$ $k$mers either side of the inverted sequence, with the alleles traversing the middle allele in opposite directions. *TakeABreak* is implemented on top of the low memory de Bruijn Graph developed by Minia (Chikhi and Rizk, 2013) and implemented in GATB (Drezen *et al.*, 2014). Their reported results for a simulation on human chromosome 22 (using $k = 31$) are a discovery rate of 87.6% with a false discovery rate of 7.5%. Inversions were simulated with length sampled from a uniform distribution 0 . . . 1,000 bases and sequence was simulated with 100bp single-end reads with 1% error.



Figure 1.14: Illustration of an inversion in a de Bruijn Graph. a) In red the original sequence and blue its reverse complement. b) The de Bruijn Graph ($k = 5$) of the inversion shown in (a), coloured edges indicate the path through the graph for each of the reads. $k$mer reverse complement sequence is shown in grey under the $k$mer sequence for the middle $k$mers only. For simplicity $k$mers (rather than $k$mer-keys) are shown and reverse complement sequences are only shown for some $k$mers (underneath them in grey). $k$mer-keys are sequences that represent both a $k$mer and its reverse-complement and are defined as the lexicographically lower of the two.

1.8 Sequencing error correction

DNA sequencing is prone to making mistakes, known as sequencing errors. Common errors include incorrect base incorporation (causing a single base change), polymerase slippage (causing insertions or deletions) and false mate pairing. The types, distribution in reads and rate of sequencing errors are platform dependent. Sequencing errors such as slippage are usually sequence dependent, with low complexity repetitive sequence more likely to cause mistakes. For example on the Illumina Genome Analyzer (GA) platform *GGC* sequences and inverted repeats longer than 8bp are associated with at least ∼80%

of errors (Nakamura *et al.*, 2011). On the more recent Illumina MiSeq 2x300bp platform, indel errors happen at one hundredth of the rate of substitution errors and 95% of them occur in the first 10bp of reads (Schirmer *et al.*, 2015). Illumina data show increasing error rates towards the end of reads, and higher error rates on the second read of a pair (Nakamura *et al.*, 2011, Schirmer *et al.*, 2015).

Sequencing errors can cause ambiguities that disrupt genome assemblies resulting in shortened contigs, incorrect bases and mis-joined contigs. Sequencing errors can also result in false positive variant calls and low frequency somatic mutations that cannot be differentiated from errors.

Error correction is the process of correcting sequencing errors in reads using the consensus of overlapping reads – first called the *spectrum alignment problem* by Pevzner *et al.* (2001a). By doing this we can hopefully reduce the number of false positive variant calls and improve our genome assemblies. Many assemblers include their own error correction steps that output corrected reads (e.g. SGA (Simpson and Durbin, 2012), Fermi (Li, 2012)). They usually focus on error correcting single base errors. Several standalone error correction programs have also been developed. *Quake* (Kelley *et al.*, 2010) is an existing tool that uses *Jellyfish* (Marçais and Kingsford, 2011) $k$mer counter to correct single base substitution sequencing errors. *Quake* is designed for use with Illumina reads. It uses read quality scores and learns nucleotide to nucleotide error rates.

BFC (Li, 2015a) uses heuristics to constrain an exhaustive search of possible edits to a read to error correct short Illumina reads. It takes a parameter k, a set of trusted $k$mers (those that appear > 3 times) along with base quality scores in a read to correct single base change errors. Trusted $k$mers are found with $k$mer counting software KMC2 (Deorowicz *et al.*, 2015) and are stored in a Bloom filter. To correct a read, the longest stretch of trusted $k$mers is identified, or if not found, the first k bases are exhaustively searched for a trusted $k$mer one-mismatch away. The algorithm uses a dynamic programming approach, optimised to use a priority queue initially containing just the first trusted $k$mer. It takes the highest scoring solution from the queue, extends it by one base for each of $\{A, C, G, T\}$ and adds each to the queue. BFC does not attempt to error correct bases with quality score ≥ 20 if it's preceded by a trusted $k$mer or if too many corrections have been made in the last 10 bases. If too many corrections are suggested *BFC* labels the read *'uncorrectable'*. Error correcting $\sim 40X$ 2x150bp reads from human sample NA12878 took 23 GiB of RAM and $\sim$5 hours. A random subset of 2M reads were mapped to the reference – after correction $\sim$20% of reads mapped better and 0.2% mapped with a lower mapping score. Indel error correction has not been implemented.

Musket uses multi-$k$mer spectrum to error correct both substitutions and indel errors (Liu *et al.*, 2013). *Lighter* (Song *et al.*, 2014) error corrects without $k$mer counting, instead using a pair of Bloom filters. *LoRDEC* (Salmela and Rivals, 2014) long read error correction focuses on Pacific Biosciences SMRT

sequencing platform ('PacBio'), which produces long reads. Lee *et al.* (2014) use Illumina short reads to error correct longer PacBio reads. Correction and lossy compression is provided by Diginorm as part of the *Khmer* (Brown *et al.*, 2012) software package. *Khmer* uses a *k*mer based approach designed for large metagenomic samples. In this scenario, error correction and data reduction usefully reduce data storage requirements. BayesHammer (Nikolenko *et al.*, 2013) uses Hamming graphs and Bayesian subclustering, with the aim of error correcting single cell sequencing data but which can also be applied to other use cases.

1.9 Visualisation tools

Visualisation is an important tool in understanding the structure of an assembly and identifying possible errors. Historically there has been a shortage of suitable visualisation tools in genome assembly and graph-based variant visualisation due to the volume of the data and the diversity of data structures being used. Graphical Fragment Assembly (GFA)(Li, 2016)[10] is a multi-purpose graph format for storing and exchanging graph information. Bandage (Wick *et al.*, 2015) is a powerful interactive desktop application that visualises GFA files. It loads the entire GFA into memory, so subgraphs of an assembly graph need to be generated before they can be visualised. Unlike with sequencing data, on-disk index and on-disk random access schemes have not been developed for graph-based formats.

ABySS-Explorer (Nielsen *et al.*, 2009) is an interactive assembly graph visualisation software designed for viewing unitig structure, rather than base pair resolution. This is useful for debugging assembly errors produced by de Bruijn Graphs, however it can only handle sub graphs rather than a full assembly graph. It takes as input DOT formatted files, a format commonly used by the graph visualisation software Graphviz (Ellson *et al.*, 2003).

For other visualisation needs, there are existing tools that are built on flat reference genome coordinate systems. Tools such as the Ensembl (Hubbard *et al.*, 2002) and UCSC (Kent *et al.*, 2002) genome browsers provide a web based interface displaying a flat reference view of a genome. Users can zoom into regions and show annotation tracks containing (for example) repeat annotations or common variants. IGV (Robinson *et al.*, 2011) is software for viewing sequencing data aligned to a reference genome. Specifically designed as a lightweight desktop application, it can fetch data from disk or a remote server and provides an interactive nucleotide-level view of sequencing data. IGV is very useful for inspecting sequencing read mapping quality, coverage, per-base quality scores and read pairing around a putative variant in multiple samples. Just as assembly graph visualisation helps identify assembly errors, visualisation of

[10]GFA spec: `http://gfa-spec.github.io/GFA-spec/`

the sequencing data that contributed to a variant call can help identify error modes and missed calls in variant callers.

1.10 Summary

Mapping based variant calling is limited by requiring a reference, suffers from reference bias and has low sensitivity for large events. Graph-based variant calling approaches could overcome these limitations and take advantage of known variation through a population graph instead of comparing against a flat reference. Current graph data structures are optimised for single genome assembly but could be leveraged for variant discovery. The best suited graph structure, the de Bruijn Graph, lacks some of the information stored in a String graph. In this thesis we propose an augmented graph data structure that possesses the strengths of a string graph whilst still behaving like a de Bruijn Graph. We describe its construction and demonstrate its properties; develop error correction, assembly, variant calling and visualisation algorithms for it and finally demonstrate these uses.

Chapter 2

Linked de Bruijn Graphs

2.1 Linked de Bruijn Graph as a string graph

In this chapter we propose a new data structure, built on top of a de Bruijn Graph, called a Linked De Bruijn Graph. It satisfies Myers' definition of a String Graph (Myers, 2005) and we present a linear time algorithm for its construction from read data. Our work is published in Turner *et al.* (2018). The implementation is available at `https://github.com/mcveanlab/mccortex`.

2.1.1 Definitions

A $k$mer is a string of k characters over the alphabet of $\{A, C, G, T\}$. A $k$mer-key is the lexicographically lower of a $k$mer and its reverse complement. Each vertex in a de Bruijn Graph represents a $k$mer *and* its reverse complement and is stored as its $k$mer-key. k is always odd, in order to ensure that a $k$mer is never its own reverse complement.

Vertices in a de Bruijn Graph can be traversed forwards or backwards. We call this the *orientation* of the traversal. Traversing a vertex forward gives its $k$mer-key and in reverse gives the reverse complement sequence. *In-degree* is the number of edges going into a vertex (assuming a given orientation), *out-degree* is the number of edges leaving a vertex. A *fork*, *bifurcation* or *junction* vertex is a vertex with out-degree greater than one. Traversing past a vertex with $outdegree(n) > 1$ requires making a junction choice.

In our diagrams of graphs, if you enter a vertex from the left you must leave by an edge to the right and vice versa. For simplicity traversal will be assumed to be left to right: in-degree will refer to edges on

the left and out-degree to edges on the right.

A unitig is a maximal path of connected vertices in a de Bruijn Graph, such that all but the first and last vertex have in-degree and out-degree of exactly one (see Figure 2.1). Unitigs must not visit the same vertex twice. In the case of cycles a simple rule can be used to determine the start vertex, such as the lexicographically lowest $k$mer-key. Our implementation (McCortex, see Chapter 3) defines the unitig start vertex in cycles as the $k$mer-key with the lowest index in the hash table. Therefore, for any de Bruijn Graph there is exactly one set of unitigs and it covers all vertices exactly once.

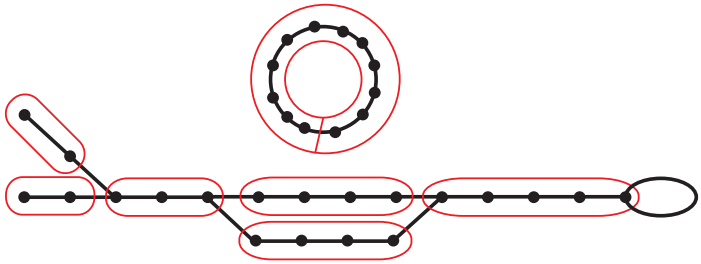


Figure 2.1: $k$mer vertices in a de Bruijn Graph are grouped into unitigs (red lines). Each unitig has a start and end, assigned based on the which $k$mer occurs first in the hash table. In-edges are those to the left of a vertex, out-edges are those to the right. Entering a vertex from one side requires exiting from the other.

Multiple samples can be loaded into a de Bruijn Graph in parallel. Doing this overlays the de Bruijn Graphs on top of each other in order to facilitate direct comparison between them. Each sample is loaded into a graph *colour*. A graph with multiple samples loaded is referred to as a *multicolour graph* (Iqbal *et al.*, 2012).

2.1.2 de Bruijn Graph limitations

De Bruijn graphs are limited in the information they can store. They lose all connectivity information at or beyond $k+2$ bases (k bases is within a $k$mer; $k+1$ base connectivity is stored in edges between $k$mers).

Resolving repeats longer than k is possible by estimating $k$mer copy number and enumerating Eulerian paths through the entire graph (Pevzner *et al.*, 2001b). Ukkonen's condition (Pevzner, 1995, Ukkonen, 1992) requires all interleaved and triple repeats be bridged in order to resolve a sequence graph into a single Eulerian cycle. A interleaved or triple repeat is bridged if the shortest repeat is covered by a read that extends before and after it by at least one base. Bresler *et al.* (2013) show that this gives a lower bound for required sequencing coverage and present an approach (MULTIBRIDGING) that modifies the graph similarly to EULER (Pevzner *et al.*, 2001b). They approach the lower coverage limit derived

from Ukkonen's condition for assembly. These algorithms don't scale to large mammalian genomes and cannot be used to locally resolve small subgraphs, which is a requirement for graph based variant calling. Additionally, by editing the graph, they make multi-sample comparison impossible by breaking the underlying structure of the de Bruijn graph. Copy number estimation can also be error prone due to amplification bias and large variances in short segments.

If a repeat of length k occurs within input genomes, local traversal of the de Bruijn Graph will not be able to assemble past it. Base connectivity information from input reads of length $R+2$ bp is required (where R is the repeat length) in order to match up one base either side of the repeat to form a contiguous path through it. This important information from the reads is currently not used by some de Bruijn Graph implementations such as Cortex (Iqbal *et al.*, 2012). This severely limits its power as an assembler, which in turn reduces its power as a variant caller.

Repeats appear in the graph as 'tangles' where the graph appears to collapse down due to duplicated sequence or low sequence complexity (see Figure 2.2). In order to assemble across tangles we must match each in-edge with an out-edge. If the repeat length is shorter than the read length we can do this by looking for reads that cross the entire repeat, thereby pairing in- and out- edges.

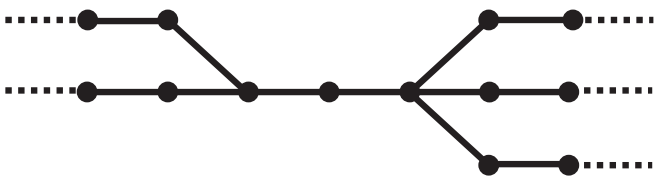


Figure 2.2: 'Tangle' in a de Bruijn Graph. Vertices are $k$mers, dashed lines are edges to the rest of the graph. Tangles are the result of low complexity sequence – in this example three adjacent $k$mers occur at multiple places in the genome). In order to traverse past the tangle we must match up in- and out-edges.

In order to store and exploit this information from the reads, we must augment our de Bruijn Graph with connectivity annotations.

2.1.3 Initial approach: shading

Our initial approach was to build a lossy representation of the reads. This was implemented on top of the de Bruijn Graph implemented in Cortex. To do this we added one byte (8 bits) per $k$mer per sample/colour. These $k$mer bits were referred to as 'shades'. Reads that crossed tangles were used and their $k$mers were all given the same *shade* - meaning they all had the same bit set. Information was lost in low complexity regions of the graph where all bits were set. This had the useful property of masking

out very difficult regions of the graph. Obviously this also limited the usefulness of the approach. An approach similar to this has been pursued by Alipanahi *et al.* (2017) with a concept called *read coloured de Bruijn Graph*, where each read has a different colour.

We discovered that this approach does not scale well. Each sample stores their connectivity information completely separately from the rest of the population. Memory required to store this data is fixed at $N_s \times N_k$ bytes, where N_s is the number of samples and N_k is the number of $k$mers in the graph. Fixing the memory required is a nice property since it simplifies the implementation and makes running the software easier. It also means we can handle an overload of connectivity data and contamination gracefully, instead of running out of RAM and crashing. However, since connectivity is likely to be shared across samples in a population, sub linear memory growth should be achievable for some populations. Linear memory growth with such a large constant, for a lossy data storage data structure, is clearly undesirable.

We saw an improvement in assembly and better rates of variant discovery in simulations on *Staphylococcus aureus* (data not shown). However we opted for a lossless approach which stored connectivity across samples. This new approach does not offer a predefined memory limit. However it stores connectivity information losslessly for multiple samples together, reducing memory usage for related samples. This also enables us to compare different samples' connectivity at different places in the graph.

2.1.4 Adopted approach: links

We have augmented our de Bruijn Graph by adding connectivity annotations to the graph which we call 'links'. A *Linked de Bruijn Graph* is constructed with a given k as before and then annotated with reads in a second pass. Links can be thought of as matching in-edges to out-edges in a tangle.

Vertices in the graph are given a list of links that start there. A link consists of an orientation from which it leaves the vertex (forwards or reverse) and a series of junction choices. Each link also has a count of how many times it is seen in each sample, which can be loaded as a binary presence/absence when count is not required. During traversal of the graph, every time we walk over a vertex, we pick up any links it has facing the orientation of traversal – if we traverse a vertex forward, we pick up all of its forward links and vice versa. As we walk we accumulate links, which are used to make traversal decisions at junctions (when we reach vertices with out-degree greater than one).

2.2 Graph construction

An annotated de Bruijn Graph is constructed in several steps and two passes of the read data. To start we build a multicolour graph and clean it, as described in Iqbal *et al.* (2012). This involves building a de Bruijn Graph for each sample from the sequence reads. Next error cleaning is performed on the de Bruijn Graphs. In a second pass of the read data, reads are threaded through the graph and links are added. Finally, links with low counts (the result of sequencing error) are discarded.

2.2.1 de Bruijn Graph construction

A de Bruijn Graph is constructed by loading reads and breaking them at low quality bases (a user parameter) and at uncalled bases ('N's). Reads are then split into overlapping $k$mers (one starting at every base). A read of length R will give at most $R - k + 1$ $k$mers. $k$mers are added to the graph and edges are added between adjacent $k$mers (those that overlap by $k - 1$ bases). As we are adding $k$mers we keep track of how many times they have been seen. We do not keep counts on edges.

2.2.2 de Bruijn Graph cleaning

Graph cleaning removes $k$mers created by sequencing error. We do this by removing short 'tips' and low coverage unitigs. A graph 'tip' is a unitig with in-degree + out-degree less than two. In other words they connect to the graph with at most one edge. Since they do not connect two parts of the graph together they are not useful to assembly. Tips are usually the result of sequencing error at the start or end of reads, but they can also be caused by gaps in coverage. To be removed a tip must be shorter than a user supplied threshold – the default is k vertices since this is the maximum number of $k$mers produced by a single-base sequencing error near the end of a read.

Unitigs are considered low coverage if the mean coverage of their $k$mers is below a certain threshold. Unitig coverage forms a bimodal distribution with a peak of low coverage unitigs caused by error and a higher coverage peak resulting from faithfully sequencing the input genomes. Unitigs caused by errors should have a geometric coverage distribution, whereas error-free unitigs should have Poisson coverage. We automatically pick a cleaning threshold between these two distributions and remove unitigs with mean coverage below the threshold (for detail see Appendix C).

In the case of many low coverage samples of the same species, cleaning per individual is less effective. If the distributions of errors and true sequences overlap too much, cleaning will remove too much real sequence without removing many of the errors. Instead we can pool samples into a single colour and

clean this pooled population in one step. Individuals are then intersected with the cleaned population to remove unitigs private to them. This approach loses coverage on sequence that is private to an individual, but will still keep sequence and variants that are shared by multiple individuals.

2.2.3 Read threading

Once a cleaned de Bruijn Graph has been constructed, the next step is to add the connectivity information from the reads that has been lost. Essentially we are aligning reads to the graph, correcting sequencing errors and recording the junction choices they make. From this we add links that hold the connectivity information from the reads. Links have counts indicating how many times they were seen in each sample. We exhaustively add links to all vertices where they are informative. This gives the graph the property of complete local connectivity information - you can drop into the graph and start traversing with all the relevant read information available.

Aligning reads to the graph

To align the reads to the graph we look up their constituent $k$mers. The reads being aligned may not have been those used to build the graph. Additionally the graph may have been cleaned of sequencing errors. Therefore some $k$mers from the reads will be missing from the graph. This results in reads having gaps in their alignment to the graph. Gaps are also introduced in alignments at uncalled and low quality bases in the reads. Additionally, insert gaps in paired-end reads are treated as gaps in the alignment. We do not attempt to connect paired end reads that overlap by more than $k-1$ bases – they are treated as single end reads. Such read pairs should be merged into a single contig first using a third party package (e.g. FLASH (Magoč and Salzberg, 2011), COPE (Liu *et al.*, 2012), PEAR (Zhang *et al.*, 2014a) or PANDASeq (Masella *et al.*, 2012)).

We attempt to resolve these gaps by traversing the graph to cross them (see Figures 2.3 and 2.4). This is equivalent to the conversion of mate pairs to 'mate-reads' in *EULER* (Pevzner and Tang, 2001, Pevzner *et al.*, 2001a). We attempt to cross one way (read 1 $\rightarrow$ read 2) and if that fails attempt the reverse direction (read 2 $\rightarrow$ read 1) – see Section 2.3 for the linear time graph traversal algorithm. Traversing the graph infers the $k$mers that should fill it. As we have a rough idea of the gap length, we check that the inferred path is close in length to the expected gap size. In effect we are error correcting reads against the graph. This is done without needing to know about the sequencing platform used or the expected error mode. This approach can correct small insertions, deletions and single and multiple base substitutions. We use a similar approach later to error correct and return the reads themselves (see Chapter 4).

If link information is available, we use it when attempting to traverse sequencing gaps. If we can traverse across a gap, we check that any link information picked up is consistent with the sequence either side of the gap. In this way, we can use one set of reads to improve error correction of a second set of reads.

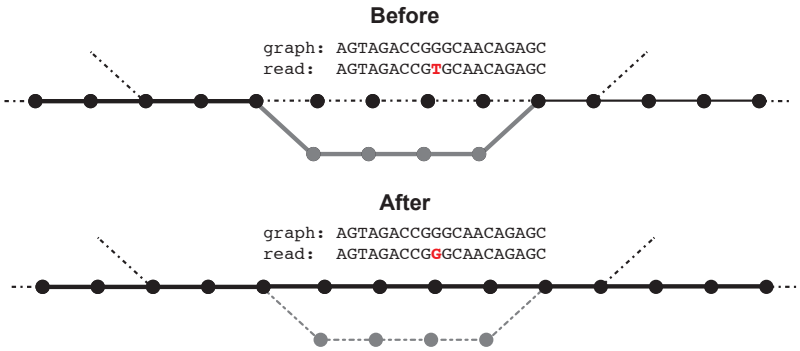


Figure 2.3: Error correction on a read against a de Bruijn Graph. Solid lines indicate *k*mers in the read. Gray vertices and edges are in the read but not in the graph. Before error correction, the read diverges from the *k*mers in the graph. By traversing this gap in the graph, the read is corrected (`G>T` error).

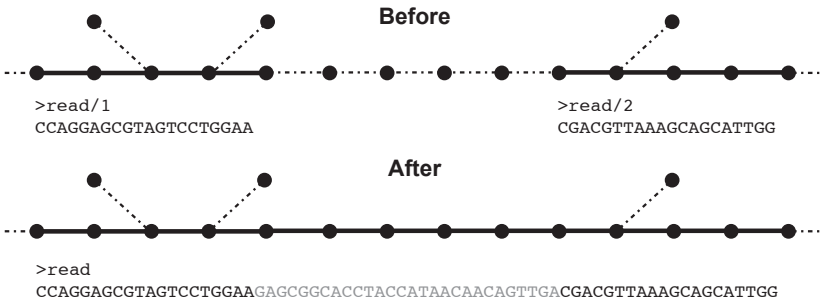


Figure 2.4: Mate pair-read gap inference by traversal. Vertices are *k*mers in a de Bruijn Graph. Lines represent edges between *k*mers. For a given pair of reads, we can locate their *k*mers in the graph and traverse to fill in the gap between them, giving us a single contig with more bases than the original read-pair. Solid lines show edges between *k*mers in the reads before gap filling and between inferred *k*mers after gap filling. All other edges are dashed.

We do not compare the **sequence** of the gap against the expected sequence. A pairwise alignment of the two sequences could be used to inform the decision of whether or not to accept the filled gap. However this would be an expensive step, and our results (shown in Chapter 4) suggest the length comparison step is sufficient. Plausible corrections are also dependent on the sequencing platform used.

The result of aligning a read or read pair to the graph is a series of read segments that perfectly match the graph, with gaps between them that could not be traversed.

2.2.4 Adding graph links

After aligning a read or read pair to the graph, we get stretches of sequences that perfectly match the graph called a 'read segment'. Each of these read segments describes a path through the graph. If that path is informative about connectivity, we add link information to the graph. A path is informative if it starts before and finishes after the graph collapses into a tangle. For all vertices in the path directly preceding a tangle, we add a link to the last vertex in the sequence that is directly after a tangle (see Algorithm (1) and Figure (2.5)). We do the same in the opposite direction – all vertices directly after a tangle are given a link to the first vertex before a tangle.

Algorithm 1 O(N) Algorithm to add links to the graph from a single read segment

▷ A read segment can be represented by a path through the graph
▷ A path through the graph is just a list of vertices with orientations
▷ A vertex is a $k$mer-key in the graph

vertices ← array of N vertices ▷ (indexed $\{0 \dots N-1\}$) representing the read segment

procedure REVERSE($\{v'_0, \dots, v'_m\}$)
    Returns $\{\widehat{v'_m}, \dots, \widehat{v'_0}\}$ ▷ Returns the reverse of a path through the graph
procedure INDEGREE(v) ▷ Returns the number of edges entering vertex v
procedure OUTDEGREE(v) ▷ Returns the number of edges leaving vertex v
procedure ADDLINK(v,*path*) ▷ Add or update link at vertex v
    if vertex v already has a link matching *path* **then**
        Increment link count
    else
        Add a new link to vertex v, covering *path*, with *count* = 1

if (INDEGREE(i) $\leq 1 \, \forall i \in \{1..N-2\}$) OR (OUTDEGREE($i$) $\leq 1 \, \forall i \in \{1..N-2\}$) **then**
    Give up ▷ Read Segment doesn't contain any connectivity information

$first \leftarrow$ lowest index i such that INDEGREE($vertices[i]$) > 1 AND $i > 0$
$last \leftarrow$ highest index i such that OUTDEGREE($vertices[i]$) > 1 AND $i < N-1$

for i in $\{1 \dots N-2\}$ **do** ▷ Iterate over *vertices* skipping first and last vertex
    if INDEGREE($vertices[i]$) > 1 **then**
        ADDLINK($vertices[i-1], vertices[i \dots last+1]$)
    if OUTDEGREE($vertices[i]$) > 1 **then**
        ADDLINK(REVERSE($vertices$[i+1]), REVERSE($vertices$[first-1 ... i]))

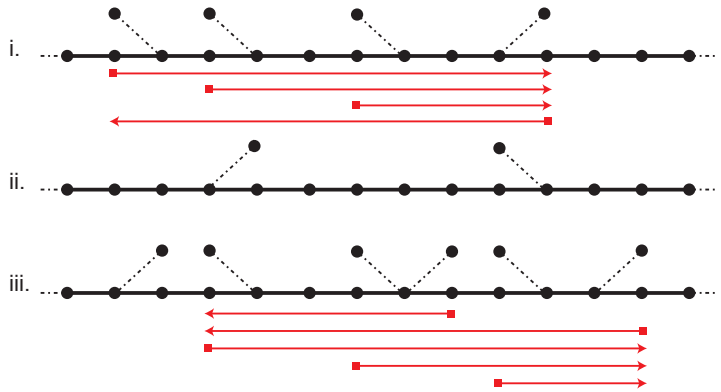


Figure 2.5: Adding complete local connectivity information, in the form of links, to a de Bruijn Graph for three separate reads (i, ii, iii). Vertices are $k$mers, solid edges are edges in the read, other edges in the graph are dashed lines. Red lines represent links to be added for each read and the choice of junctions they make. Square ends indicate the vertex that links are added to, arrows indicate the direction of travel for which they are used.

By exhaustively adding links to all vertices preceding a tangle, we give the graph the property of having 'complete local connectivity information'. This means you can drop into the graph anywhere and start traversing with complete connectivity information. An example tangle illustrated in Figure 2.2.

Links are cleaned by looking at coverage in the same manner as unitig graph cleaning (see Section 2.2.2).

The de Bruijn Graph construction and graph annotation algorithms are both linear in time complexity with respect to the total number of input bases. This is the same time complexity as construction of a string graph from an overlap graph. Construction of an overlap graph is an extra step that is also linear for small overlaps and a given maximal error rate.

Augmenting with link information does not require comparing reads to each other. Instead each read is processed separately against a static de Bruijn Graph. We then sum across the contribution of all reads. This avoids higher time complexity algorithms that would become computationally challenging in low complexity sequence regions, where many reads interact with each other.

2.3 Graph traversal

Graph traversal, which we refer to as 'walking' the graph, can start at any $k$mer. We can drop into the graph anywhere and begin walking. As we walk we pick up graph links just before the graph collapses into a tangle. These links help resolve junctions when we leave the tangle. For each of the links picked up by graph walking, we record how many vertices ago it was picked up. This is referred to as link

'age'. When we reach a junction we follow the oldest link. This is because the oldest link gives the most context of where on the genome we actually are. In other words the oldest link gives the lowest number of possible places we are currently at – it is the most specific about where we are.

During traversal we maintain two lists of links. *Primary* links are those picked up from vertices we visit. *Secondary* links are those picked up from vertices that join vertices we visit. When we reach a vertex with in-degree greater than one, we pick up the links held by the vertices before it and add them to the 'secondary' collection. Links in our primary collection are used for making junction choices, those in our secondary collection are only used for checking that all edges at a junction have links on them. When we reach a junction we check that we have links (either primary or secondary) that go to all the next vertices. If we do not, we have identified missing connectivity information in the graph and halt traversal.

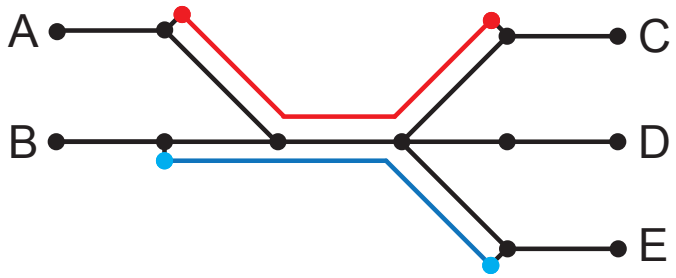


Figure 2.6: Traversal of a Linked de Bruijn Graph is halted when we reach a junction with an edge down which no links go. Links are shown in red and blue. When traversing from B from left to right, we reach the junction $C/D/E$. No links from A or B take the edge towards D. The existence of the edge to D indicates that we are missing at least one link at this junction, which could be $A \rightsquigarrow D$ or $B \rightsquigarrow D$. Since there is uncertainly traversal stops at the junction $C/D/E$.

In Figure 2.6 we illustrate the missing information check. Walking left to right from vertex B, we pick up the blue link and add it to our primary collection. When we reach the vertex with in-degree greater than one (A and B collapse down into a repeat), we add the red path to our secondary collection. Continuing forward, we reach the junction at $C/D/E$. We use the union of primary and secondary links to check each edge has a link on it. The edge to D does not have a link on it but we know it must be connected to A or B. If B connects to D we ought to stop traversal at the junction. If A connects to D we can safely carry on traversal (to E). Since we lack information about the truth, we halt traversal to avoid making an assembly error.

A repeat that has no missing links is equivalent to a fully-bridged repeat as described by Bresler *et al.* (2013).

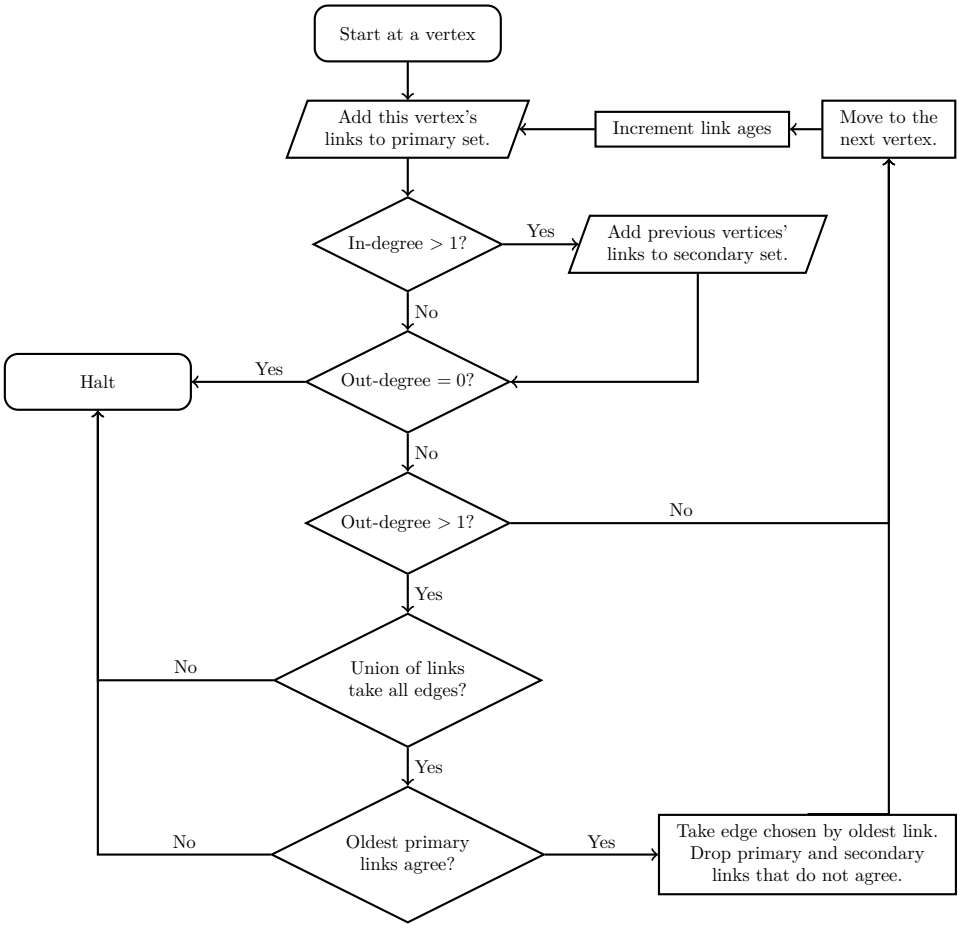


Figure 2.7: Decision flowchart for walking through a linked de Bruijn Graph. At each $k$mer in the Linked de Bruijn Graph we follow this flow chart to traverse whilst maintaining two set of links: the primary set for making junction choices, the secondary set for identifying junctions with missing links. Each link in the primary set has an 'age' – how long we have held it for.

After we have determined we have sufficient links to resolve a junction, we select the edge taken by the oldest primary link. If we have multiple oldest links that disagree we abandon traversal. Our oldest link selects the next vertex to move to and we drop all links that disagree with this choice (in both the primary and secondary lists). This is illustrated in the graph walking flowchart in Figure 2.7.

2.4 Links in loops

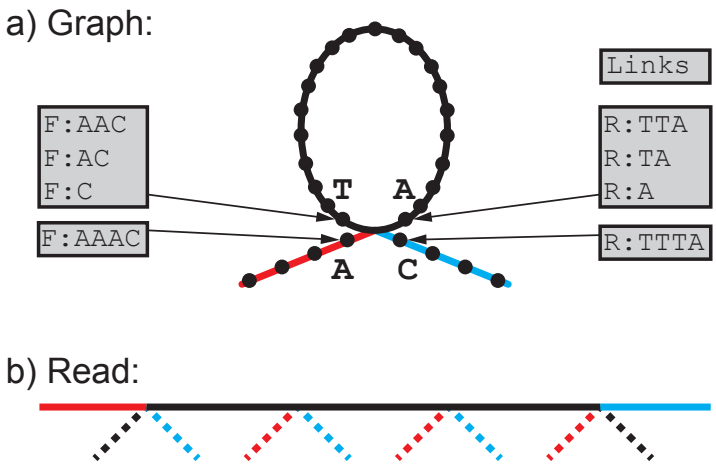


Figure 2.8: A loop in a Linked de Bruijn Graph caused by three adjacent identical copies of a repeat. (a) Linked de Bruijn Graph, with links associated with each $k$mer listed in grey boxes. Link `F:AAAC` is picked up when traversing forward (`F`) through the last $k$mer before the repeat. Link `F:AAAC` means when traversing forwards (`F`; left-to-right), take the junction choices A, A, A, then `C` – in other words go around the loop three times. If we visited the first $k$mer after the loop backwards (i.e. reverse `R`) we would pick up the `R:TTTA` link, which would lead us to traverse the loop clockwise three times, expressing the reverse complement of the $k$mers and therefore the repeat. Red and blue sections indicate unique sequences on the left and right sides of the repeat, which is shown in **black**. Junction choice bases are labelled at the junctions and correspond to the link junction choices. (b) The relevant genome sequence annotated with local graph structure (dashed lines show edges not taken). The repeat is in **black**, unique flanking sequences are shown in red and blue.

Two or more tandem repeats with repeat elements longer than $k-1$ cause loops structures in the graph. Loops are just a subset of graph tangles and Linked de Bruijn Graphs resolve them in the same manner (see Figure 2.8).

2.5 Theoretical proofs

In this section we prove some properties of Linked de Bruijn Graphs.

2.5.1 Assumption of complete information

Sequencing genomes involves breaking up the genome into small pieces which are then read by a sequencing machine, which introduces sequencing errors. In the following section we make the assumption that the graph is constructed from the sequences of complete chromosomes without reading errors – that is, the graph is built with *complete information* of its true underlying sequences. In human genome

assembly this is analogous to creating a graph with 46 perfect sequences (one per chromosome). This is obviously an unrealistic assumption for genome assembly which we will relax later, but it simplifies the initial discussion.

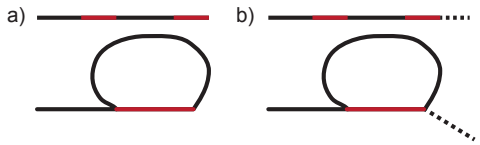


Figure 2.9: Simplified representation of a de Bruijn Graph, with vertices omitted. A repeat is shown in red. (a) a sequence that ends in a repeat (top) and the resulting de Bruijn Graph (bottom). (b) Adding a unique sequence to the end of the input (top, dashed lines) ensures infinite loops do not exist in the resulting de Bruijn Graph (bottom, added $k$mers and edges shown by dashed line).

In addition to assuming the input sequences are complete chromosomes, we also assume that chromosomes start and end with unique $k$mers. An undesired edge effect can appear if sequences end with $k$mers that appear elsewhere in the graph, resulting in loops at the start or end of the graph representation of the sequences with in-degree greater than out-degree (see Figure 2.9). One can force this to be true by explicitly adding a unique $k$mer to the start/end of sequences. In the case of high coverage genome assembly this edge case is rare.

2.5.2 Notation

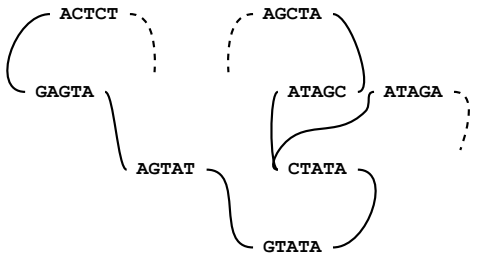


Figure 2.10: A portion of a de Bruijn Graph with $k = 5$. Dashed lines indicate edges to the rest of the graph (omitted). If we traverse left-to-right we emit the $k$mer-key, otherwise we emit the reverse-complement of the $k$mer-key. A path from `AGCTA` $\rightarrow$ `ATAGC` $\rightarrow$ `CTATA` represents the sequence `AGCTAT` – we start with `ATAGC` and append the reverse-complemented first base of `ATAGC` since we traversed it backwards, then append the last base of `CTATA` as we traversed it forwards).

The Linked de Bruijn Graph is defined as $LG(k) = (V, E, L)$ where k is the de Bruijn Graph parameter, and V and E are defined as in a de Bruijn Graph. Vertices V is the set of $k$mer-keys. A $k$mer is represented by a $\langle vertex, orientation \rangle$-tuple called an *oriented-vertex*. Oriented-vertex v' is vertex $v \in V$ with an orientation – either forwards $\overrightarrow{v}$ or backwards $\overleftarrow{v}$. $\widehat{v'}$ is the opposite orientation of v'. Therefore $V_O = (\overrightarrow{v}\ \forall v \in V) \cup (\overleftarrow{v}\ \forall v \in V)$ is the set of all oriented vertices. E is the set of directed edges between

oriented-vertices. $L(v')$ is a set of valid paths through the graph (*links*) that start at oriented vertex $v' \in V_O$.

Traversal means moving through a path of oriented vertices and along edges between vertices, using link information. $v'_a \rightsquigarrow v'_b \rightsquigarrow v'_c$ is a path through the graph starting at v'_a, finishing at v'_c and containing v'_b. It is the result of starting walking at vertex v'_a, passing through some vertices, reaching vertex v'_b then continuing traversal through 0 or more vertices to reach vertex v'_c, each with an unambiguous route. In other words if we start at v'_a there is only one valid path to follow and it reaches v'_b and so on to v'_c. $v'_v \not\rightsquigarrow v'_y$ means we cannot traverse unambiguously to vertex v'_y if we start at v'_x. We can therefore make the following statements:

$$v'_x \rightsquigarrow v'_y \rightsquigarrow v'_z \implies (v'_x \rightsquigarrow v'_y) \wedge (v'_x \rightsquigarrow v'_z) \tag{2.1}$$

$$v'_x \rightsquigarrow v'_y \not\Rightarrow v'_y \rightsquigarrow v'_x \tag{2.2}$$

In Figure 2.10, we can see that it is possible to traverse unambiguously from $k$mer $AGCTA$ to $GAGTA$ ($\overrightarrow{AGCTA} \rightsquigarrow \overleftarrow{GAGTA}$) but not in the reverse direction ($\overrightarrow{GAGTA} \not\rightsquigarrow \overleftarrow{AGCTA}$). This is because we hit a fork (bifurcation) in the graph that cannot be resolved.

An oriented vertex in the graph represents a $k$mer which may occur in one or more locations in the input genome. For a given position in the input genome s_x, we make the following definitions:

1. $V(s_x)$ is the oriented vertex representing the $k$mer starting at position s_x in the input genome; this is a many-to-one mapping.
2. $\widehat{V(s_x)}$ is $V(s_x)$ in the opposite orientation.
3. $S(v'_x)$ is the set of sequence positions represented by oriented vertex v'_x. This is the reverse of $V(s_x)$ and a one-to-many mapping.

2.5.3 Repeats

A repeated substring is a substring that occurs more than once in the whole string. Substrings may be overlapping and may be reverse-complemented. A maximal repeated substring is a substring such that adding a single character to the head or tail would decrease the number of occurrences of it (Masada *et al.*, 2012). A greedy algorithm for finding the set of maximal repeated substrings for any string is shown in Algorithm (5) (Appendix D). The greedy approach has time complexity O(N^3) but with dynamic

programming O(N^2) solutions are possible[1]. As an example, for the string *abababa* the set of maximal repeated substrings is $\{abababa, ababa, aba, a\}$ with occurrence counts $\{2, 3, 4, 5\}$ respectively.

We can paint repeats onto the input genome by exhaustively finding all maximal substrings that appear at one or more other positions in the input genome. Repeats must be of length $\geq k$, where k is the parameter of the de Bruijn Graph. Graph construction is then equivalent to gluing all repeats together (see Figure 2.11).

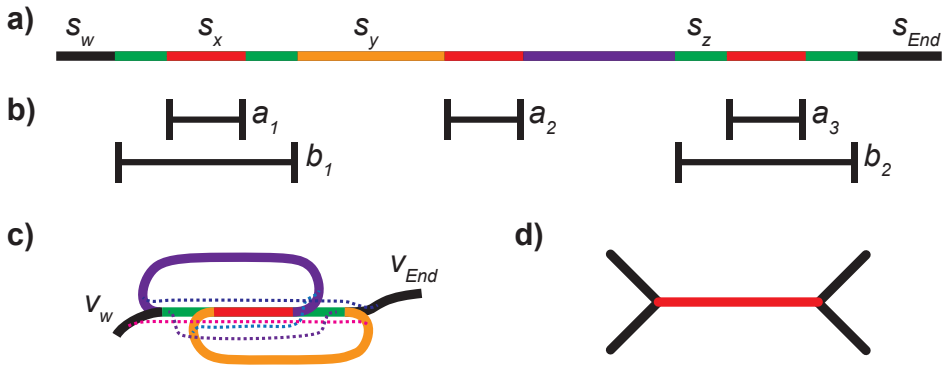


Figure 2.11: Repeats in an input genome. (a) input sequence, repeats each shown in a separate colour. Points s_w, s_x, s_y and s_z labelled. (b) Repeats labelled. (c) graph structure with links added (dashed lines). (d) De Bruijn graph structure of all repeats in the graph, with the repeat shown in red – additional edges (creating junctions) in the repeat from contained repeats are omitted.

All repeats now take the general form shown in 2.11.(d) – at the start the graph collapses down from two or more vertices and at the end forks into two or more vertices. Between the start and end of the repeat the sequences that carry the repeat do not separate. We can now see that every fork in the graph is the ending of a repeat, and conversely that every repeat ends at a fork. If we are traversing a path through the graph and hit a fork, either we started in the repeat or we passed the start of the repeat that ends at this fork.

$$
\begin{aligned}
REPEATS_S(s_w) &= \varnothing & REPEATS_V(V(s_w)) &= \varnothing \\
REPEATS_S(s_x) &= (a_1, b_1) & REPEATS_V(V(s_x)) &= (a_1, a_2, a_3, b_1, b_2) \\
REPEATS_S(s_y) &= \varnothing & REPEATS_V(V(s_y)) &= \varnothing \\
REPEATS_S(s_z) &= (b_2) & REPEATS_V(V(s_z)) &= (b_1, b_2)
\end{aligned}
$$

Figure 2.12: Repeat sets of points s_w, s_x, s_y and s_z in Figure 2.11

We define:

[1] O(N^3) Perl implementation is in *scripts/find_maximal_substrings.pl*; An O(N^2) C implementation is in *libs/maximal_substrs/* directory of McCortex

1. $REPEATS_S(s_x)$ as the set of repeats in which sequence position s_x is contained by.

2. $REPEATS_V(v'_x)$ as the union of $REPEATS_S(X)$ for all positions X in the sequence where the $k$mer associated with vertex v'_x appears:

$$REPEATS_V(v'_x) = \bigcup_{s \in S(v'_x)} REPEATS_S(s) \tag{2.3}$$

An example is shown in Figure 2.12. Note that vertex orientation has no effect on the set of repeats a vertex is in:

$$REPEATS_V(v'_x) = REPEATS_V(\widehat{v'_x}) \tag{2.4}$$

Entering repeats presents no issue in assembly. Leaving repeats requires some information about current location(s) in the underlying sequence. This information must be stored *before* you enter a repeat. Picture starting graph traversal from the middle of the red repeat in Figure 2.11.(c): once you reach the end of the red repeat, you cannot make a decision about where to go. If you were to start at v'_w, you can see that just before you enter each repeat, you pick up an annotation which enables you to resolve it.

Starting traversal from within a repeat is equivalent to walking multiple places in the input genome at once, and only tracing the consensus sequence of the repeats (stopping at the end of the repeat).

2.5.4 Sequence traversal

Assuming a Linked de Bruijn Graph constructed with complete information (input is entire chromosomes with no error), we can prove the following rules about traversal:

Proposition 2.5.1 (Sequence Traversal)**.** *Let s_x, s_y be points on the same input genome, where s_y follows s_x. Traversal from the vertex representing s_x to the vertex representing s_y is possible iff the set of maximal repeated substrings of s_x is a subset of s_y i.e.*

$$V(s_x) \rightsquigarrow V(s_y) \iff REPEATS_S(s_x) \subseteq REPEATS_S(s_y)$$

Proof. In order to traverse from one vertex to another we must not leave any of the repeats we were

originally within, as doing so would mean we hit a fork we could not resolve, therefore:

$$V(s_x) \rightsquigarrow V(s_y) \implies REPEATS_S(s_x) \subseteq REPEATS_S(s_y) \tag{2.5}$$

If we enter repeats that start after s_x then we pick up annotations just before they start which are used to resolve them. The only graph features that will stop traversal between two connected vertices are forks representing the end of repeats that we started within. These cannot be resolved unambiguously. Therefore if we do not leave repeats we started in, we can traverse connected vertices:

$$REPEATS_S(s_x) \subseteq REPEATS_S(s_y) \implies V(s_x) \rightsquigarrow V(s_y) \tag{2.6}$$

We can see that Proposition (2.5.1) follows from equations (2.5) and (2.6). □

Proposition 2.5.2 (Sequence Transitivity)**.** *Let s_x, s_y, s_z be points that appear in that order on the same input genome. If we can start at $V(s_x)$ and reach $V(s_y)$ and start at $V(s_y)$ and reach $V(s_z)$, then if we start at $V(s_x)$ we can reach $V(s_z)$ i.e.*

$$(V(s_x) \rightsquigarrow V(s_y)) \wedge (V(s_y) \rightsquigarrow V(s_z)) \implies V(s_x) \rightsquigarrow V(s_z) \tag{2.7}$$

Proof.

$$V(s_x) \rightsquigarrow V(s_y) \implies REPEATS_S(s_x) \subseteq REPEATS_S(s_y) \quad \text{using (2.5.1)} \tag{2.8}$$

$$V(s_y) \rightsquigarrow V(s_z) \implies REPEATS_S(s_y) \subseteq REPEATS_S(s_z) \quad \text{using (2.5.1)} \tag{2.9}$$

$$\text{Thus by (2.8) and (2.9)} \implies REPEATS_S(s_x) \subseteq REPEATS_S(s_z)$$

$$\implies V(s_x) \rightsquigarrow V(s_z)$$

□

Proposition 2.5.3 (Sequence Context)**.** *Let s_x, s_y, s_z be points that appear in that order on the same input genome. If we can start at $\widehat{V(s_y)}$ and reach $\widehat{V(s_x)}$ and start at $V(s_x)$ and (through $V(s_y)$) reach $V(s_z)$, then if we start at $V(s_y)$ we can reach $V(s_z)$ i.e.*

$$(\widehat{V(s_y)} \rightsquigarrow \widehat{V(s_x)}) \wedge (V(s_x) \rightsquigarrow V(s_y) \rightsquigarrow V(s_z)) \implies V(s_y) \rightsquigarrow V(s_z) \tag{2.10}$$

Proof.

$$\widehat{V(s_y)} \rightsquigarrow \widehat{V(s_x)} \implies REPEATS_S(s_y) \subseteq REPEATS_S(s_x) \quad \text{using (2.5.1)} \tag{2.11}$$

$$\begin{aligned} V(s_x) \rightsquigarrow V(s_y) \rightsquigarrow V(s_z) &\implies (V(s_x) \rightsquigarrow V(s_y)) \text{ and } (V(s_x) \rightsquigarrow V(s_z)) && \text{using (2.1)} \\ &\implies REPEATS_S(s_x) \subseteq REPEATS_S(s_y) && \text{using (2.5.1)} \\ &\text{and } REPEATS_S(s_x) \subseteq REPEATS_S(s_z) && (2.12) \\ \text{(2.11) and (2.12)} &\implies REPEATS_S(s_x) = REPEATS_S(s_y) \\ &\text{and } REPEATS_S(s_x) \subseteq REPEATS_S(s_z) \\ &\implies REPEATS_S(s_y) \subseteq REPEATS_S(s_z) \\ &\implies V(s_y) \rightsquigarrow V(s_z) && \text{using (2.5.1)} \end{aligned}$$

□

2.5.5 Vertex traversal

We define $REPEND_V(v'_x)$ to be the set of last oriented-vertices of repeats that vertex v'_x is in (see example in Figure 2.13). The last vertex of a repeat is dependent on vertex orientation: $REPEND_V(v'_x)$ is not necessarily equal to $REPEND_V(\widehat{v'_x})$.

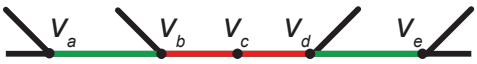


Figure 2.13: Illustration of $REPEND_V(\overrightarrow{v_x})$ function. De Bruijn graph edges are shown but only some $k$mer vertices are drawn. A repeat is shown in red and is a substring of another repeat shown in green. $REPEND_V$ maps a $k$mer v to the $k$mers that mark the end of repeats that $k$mer v belongs to. $REPEND_V(\overrightarrow{v_c}) = (\overrightarrow{v_d}, \overrightarrow{v_e})$; $REPEND_V(\overleftarrow{v_c}) = (\overleftarrow{v_a}, \overleftarrow{v_b})$.

$REPEND_V(v'_x)$ is a set of vertices that mark forks in the graph. The set of all fork vertices in the graph is:

$$\bigcup_{v \in \mathcal{V}} \left(REPEND_V(\overrightarrow{v}) \cup REPEND_V(\overleftarrow{v}) \right) \tag{2.13}$$

where $\mathcal{V}$ is the collection of all vertices in the graph.

If you start traversal in a repeat it is not possible to leave it. That means if you start traversal at some vertex v'_x, you cannot traverse unambiguously past any vertex in $REPEND_V(v'_x)$.

Proposition 2.5.4 (Vertex Traversal)**.** *Let v'_1 and v'_n be vertices with orientations and $\{v'_1, \ldots, v'_n\}$ be a connected path through the graph. We can traverse starting at vertex v'_1 and reach vertex v'_n if and only*

if none of the vertices $v'_1 ... v'_{n-1}$ are the last vertex of a member of $REPEATS_V(v'_1)$ i.e.

$$v'_1 \rightsquigarrow v'_n \iff v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{1, 2, \ldots, n-1\}$$

Proof. By the same logic as Proposition (2.5.1). The only graph features that will stop traversal between two connected vertices are forks representing the end of repeats that we started within. If we can traverse from vertex v'_1 to vertex v'_n then repeats that started in v'_1 do not end before v'_n:

$$v'_1 \rightsquigarrow v'_n \implies v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{1, 2, \ldots, n-1\} \tag{2.14}$$

If a repeat that we started in does not end before v'_n, then the only forks we encounter are from repeats that start and end between v'_1 and v'_n. For these repeats we have an opportunity to pick up annotations that will resolve them, therefore traversal will succeed:

$$v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{1, 2, \ldots, n-1\} \implies v'_1 \rightsquigarrow v'_n \tag{2.15}$$

We cannot traverse past the last vertex of a repeat that we were already in when we started traversal of the graph. If we hit a vertex that is in $REPEND_V(v'_x)$, it marks a fork in the graph that we cannot resolve. We can see that Proposition (2.5.4) follows from equations (2.14) and (2.15). □

Corollary 2.5.4.1. *Since we cannot traverse past the end of a repeat, it follows that:*

$$v'_1 \rightsquigarrow v'_n \implies REPEATS_V(v'_1) \subseteq REPEATS_V(v'_n) \tag{2.16}$$

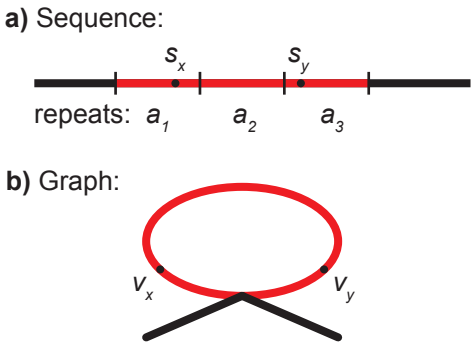


Figure 2.14: (a) Sequence with three-copy repeat (in red), with copies labelled $\{a_1, a_2, a_3\}$ and flanked by unique sequence shown in **black**. (b) The resulting de Bruijn Graph of the sequence from (a) (vertices omitted for simplicity). The position in the input sequence S_x maps to the $k$mer V_x and sequence position S_y maps to the $k$mer V_y.

Unlike traversing between sequence points (see proposition (2.5.1)), Corollary (2.5.4.1) is not bi-directional: $REPEATS_V(v'_1) \subseteq REPEATS_V(v'_n) \implies v'_1 \rightsquigarrow v'_n$. For an example of why this is not the case, see Figure 2.14. Vertices v'_x and v'_y are both in the same set of repeats ($REPEATS_V(v'_x) = REPEATS_V(v'_y)$), but traversing from v'_x to the right hits the last vertex of repeats a_1, a_2 and a_3 (which we cannot traverse past) before we reach v'_y.

Corollary 2.5.4.2. *Conversely, if the set of repeats at vertex v'_1 is not the same as, or a subset of, vertices at v'_n, then in order to traverse from v'_1 to v'_n we must exit a repeat. The repeat(s) we must exit must already have started at or before v'_1, therefore we cannot uniquely and unambiguously escape from them. Therefore it follows that:*

$$REPEATS_V(v'_1) \not\subseteq REPEATS_V(v'_n) \implies v'_1 \not\rightsquigarrow v'_n \tag{2.17}$$

Proposition 2.5.5 (Vertex Transitivity)**.** *Let v'_1, v'_n and v'_m be vertices and $\{v'_1, \ldots, v'_n, \ldots, v'_m\}$ be a connected path through the graph. If we can start traversal at v'_1 and reach v'_n and start traversal at v'_n and reach v'_m then we can start traversal at v'_1 and reach v'_m:*

$$(v'_1 \rightsquigarrow v'_n) \wedge (v'_n \rightsquigarrow v'_m) \implies v'_1 \rightsquigarrow v'_n \rightsquigarrow v'_m \tag{2.18}$$

Proof.

$$v'_1 \rightsquigarrow v'_n \iff v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{1, 2, \ldots, n-1\} \qquad \text{using (2.5.4)} \tag{2.19}$$

$$v'_1 \rightsquigarrow v'_n \implies REPEATS_V(v'_1) \subseteq REPEATS_V(v'_n) \qquad \text{using (2.5.4.1)} \tag{2.20}$$

$$v'_n \rightsquigarrow v'_m \iff v'_i \notin REPEND_V(v'_n) \quad \forall i \in \{n, \ldots, m-1\} \qquad \text{using (2.5.4)} \tag{2.21}$$

$$(2.20) \wedge (2.21) \implies v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{n, \ldots, m-1\} \tag{2.22}$$

$$(2.19) \wedge (2.22) \implies v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{1, \ldots, n, \ldots, m-1\}$$
$$\implies v'_1 \rightsquigarrow v'_n \rightsquigarrow v'_m \qquad \text{using (2.5.4)}$$

□

Proposition 2.5.6 (Vertex Partial Traversal)**.**

$$v'_x \rightsquigarrow v'_y \rightsquigarrow v'_z \implies v'_y \rightsquigarrow v'_z \tag{2.23}$$

Proof. If v'_y is in a repeat that starts after v'_x and ends before v'_z, it could be resolved by links to enable traversal $v'_x \rightsquigarrow v'_y \rightsquigarrow v'_z$ (following (2.5.4)). However in this scenario, if we start at v'_y we cannot traverse

to v'_z ($v'_y \not\rightsquigarrow v'_z$) as we cannot exit the repeat in which we initiated traversal (following (2.5.4.2)). □

Proposition 2.5.7 (Vertex Context)**.** *Let v'_1, v'_n and v'_m be vertices with orientations and $\{v'_1, \ldots, v'_n, \ldots, v'_m\}$ be a connected path through the graph. If we can traverse backwards from $\widehat{v'_n}$ to $\widehat{v'_1}$ and traverse from v'_1 to v'_m, then we can traverse v'_n to v'_m:*

$$(v'_1 \rightsquigarrow v'_n \rightsquigarrow v'_m) \wedge (\widehat{v'_n} \rightsquigarrow \widehat{v'_1}) \implies v'_n \rightsquigarrow v'_m \tag{2.24}$$

Proof.

$$\widehat{v'_n} \rightsquigarrow \widehat{v'_1} \implies REPEATS_V(v'_n) \subseteq REPEATS_V(v'_1) \quad \text{using (2.4), (2.5.4.1)} \tag{2.25}$$

$$v'_1 \rightsquigarrow v'_n \rightsquigarrow v'_m \iff v'_i \notin REPEND_V(v'_1) \quad \forall i \in \{1, \ldots, n, \ldots, m-1\} \quad \text{using (2.5.4)} \tag{2.26}$$

$$\begin{aligned} (2.25) \wedge (2.26) &\implies v'_i \notin REPEND_V(v'_n) \quad \forall i \in \{1, \ldots, n, \ldots, m-1\} \\ &\implies v'_n \rightsquigarrow v'_m \quad \text{using (2.5.4)} \end{aligned}$$

□

Proposition 2.5.8. *When assembling contigs, we only need to seed with kmers that have not themselves already been used in a contig.*

Proof. Let v'_1, v'_n and v'_m be vertices with orientations and $\{v'_1, \ldots, v'_n, \ldots, v'_m\}$ be a connected path through the graph that we have traversed from $v'_1 \rightsquigarrow v'_n \rightsquigarrow v'_m$. If we start traversal from vertex v'_n and continue to reach vertex v'_i, then we can start traversal at v'_1 and reach vertex v'_i:

$$\begin{aligned} \left(v'_1 \rightsquigarrow v'_n\right) \wedge \left(v'_n \rightsquigarrow v'_i\right) &\implies v'_1 \rightsquigarrow v'_n \rightsquigarrow v'_i && \text{using Proposition (2.5.5)} \\ &\implies v'_1 \rightsquigarrow v'_i && \text{using Equation (2.1)} \end{aligned}$$

If we start traversal in the opposite direction $\widehat{v'_n} \rightsquigarrow \widehat{v'_1}$ and continue to reach a vertex $\widehat{v'_j}$, we can show that we can reach $\widehat{v'_j}$ if we start traversal from $\widehat{v'_1}$ (mapping $\{\widehat{v'_n} \ldots \widehat{v'_1} \ldots \widehat{v'_j}\}$ to $\{v'_1 \ldots v'_n \ldots v'_m\}$):

$$\left(\widehat{v'_n} \rightsquigarrow \widehat{v'_1} \rightsquigarrow \widehat{v'_j}\right) \wedge \left(v'_1 \rightsquigarrow v'_n\right) \implies \widehat{v'_1} \rightsquigarrow \widehat{v'_j} \quad \text{using (2.5.7)}$$

Since we cannot reach a vertex that we could not reach from our seed vertex, and we have traversed as far as possible from our seed, re-using a $k$mer already in a contig would not give us a contig that is not

contained within $\{v'_1, \ldots, v'_n, \ldots, v'_m\}$. □

Using context in assembly

To assemble a contig we pick a random vertex (the *seed*) in the graph and traverse from it as far as we can backwards and then forwards. Assume the random seed vertex is v'_y. Traversing backwards gives $\widehat{v'_y} \rightsquigarrow \widehat{v'_x}$. Is this contig informative when traversing forwards from v'_y? We can show that it is not. In order to be informative, a repeat must start after v'_x and before v'_y, and end after v'_y. If it does, then $REPEATS_V(v'_x) \subset REPEATS_V(v'_y)$. If this is the case then following Corollary (2.5.4.2), we could not have traversed $\widehat{v'_y} \rightsquigarrow \widehat{v'_x}$.

2.5.6 Implementation limitations

Our implementation is limited by two things:

1. We add additional edges to the graph, between all $k$mers that overlap by $k-1$ bases. Using definitions from Cazaux *et al.* (2016), this converts the graph from a DBG_k^+ (edges are supported by sequence data) to DBG_k^- (edges exist between all $k$mers that share a $k-1$ overlap). These are either: edges not seen due to low sequence coverage, or false edges. As these edges are never seen in the sequence data we have no link information that uses them. Traversal up to junctions introduced by inferring edges will not be able to traverse past them due to our missing information check.

2. Probabilistic cycle detection is used to check if we are stuck in an infinite loop. This can happen due to graph structures like that shown in Figure 2.9. We have implemented cycle detection by hashing the walker state when we visit a vertex we have already visited. Hash values are then stored in a Bloom filter, a probabilistic data structure which stores the values and returns whether or not they have been seen before. Bloom filters can return a false positive of a matching entry. It does not return false negatives. For this reason they are a fast and low memory solution to avoid the assembler being stuck in an infinite loop.

Our input data is missing information for the following reasons:

1. We are missing information due to stochastic coverage, which loses $k$mers and edges thereby leaving gaps in assemblies.

2. There is missing information due to short reads which lose connectivity. This means some junctions will not be resolvable in the annotated graph.

3. Sequencing error introduces false graph connections between otherwise unconnected sections of the graph. This adds junctions which may trigger the 'missing information' check or that we cannot unambiguously resolve, both of which halt traversal.

4. Genome ploidy, when greater than one, is equivalent to duplicating all input sequences and adding some variation between them. This results in very long repeats ($\sim$ 1,000 bp between variants in human genomes). As these 'repeats' are likely to be longer than the input reads, most of them will not be resolvable.

2.5.7 Testing theory with simulation

DNA sequencing samples the true sequence with short reads, which loses long range connectivity information. Therefore assumptions about complete information in our graph are broken. Here we use simulations to measure how often proposition (2.5.7) and conclusions on assembly in Section 2.5.5 hold.

Given v'_b as our starting vertex, we traverse backwards ($\overleftarrow{v_b}$) for up to 500 vertices. The vertex we reach is labelled v'_a. We then traverse $v'_a \rightsquigarrow \overrightarrow{v_b}$ and continue on as far as we can. The vertex we reach is labelled v'_c. Finally, we start traversal from $\overrightarrow{v_b}$ and test if we can reach v'_c without using $v'_a \dots v'_b$ to prime traversal.

We limit the distance $\overleftarrow{v_b} \rightsquigarrow \widehat{v'_a}$ to a maximum of 500 $k$mers. If we do not set a limit and instead walk until we cannot walk any further, we regularly fail to traverse $v'_a \rightsquigarrow v'_b$ (for this simulation). This is likely because contigs end in regions that are hard to assemble and are poor choices to start assembly from.

We use 1Mbp of chromosome 22 (chr22:17,000,000-17,999,999) from the *hg19* reference assembly. From this haploid reference 100bp single-ended reads were simulated without error and with both perfect and stochastic coverage. Perfect coverage means a read starting at every base. Stochastic coverage means we randomly sample reads from across the genome. Both perfect and stochastic sampling result in 100X coverage. Graphs were built and annotated at $k = 31$ for both perfect and stochastic coverage.

Test 1	Perfect Coverage	Stochastic Coverage
Number of runs	10,000	10,000
$v'_a \rightsquigarrow v'_b$ wrong	0	1
$v'_a \rightsquigarrow v'_b$ failed	554	601
Bad seed $k$mer	47	37
Number of tests	9,399	9,361
$v'_b \rightsquigarrow v'_c$ wrong	0	0
$v'_b \rightsquigarrow v'_c$ failed	0	0
$v'_b \rightsquigarrow v'_c$ overshot	0	0
Success	9,399	9,361
Conjecture holds true	100.00%	100.00%

Table 2.1: Traversing from $k$mer vertex $\overrightarrow{v_b}$ and testing if we can reach $k$mer vertex v'_c, which can be reached if we start traversal at $v'_a \rightsquigarrow v'_b \rightsquigarrow v'_c$. *Bad seed kmers* are starting $k$mers from which no traversal is possible. Data are simulated 100X coverage with 100bp single-ended error free reads taken from 1Mbp of hg19 chr22. Perfect coverage is a read starting at every base, stochastic coverage means we uniformly sample read starts from across the 1Mbp source region.

Results are shown in Table 2.1. In this simulation, the traversal $v'_a \rightsquigarrow v'_b$ is not required to find vertex v'_c, since we already know the path between vertices v'_a and v'_b. Additionally traversing $v'_a \rightsquigarrow v'_b$ often fails. Instead of traversing $v'_a \rightsquigarrow v'_b$ we can use this path to prime traversal by taking the path $v'_a \ldots \overrightarrow{v_b}$ as truth and continue from $\overrightarrow{v_b}$ for as far as we can. The vertex we reach is then labelled v'_c. We then repeat the test above to see if we can reach vertex v'_c from $\overrightarrow{v_b}$ without the context of $v'_a \ldots v'_b$. These results are shown in Table 2.2.

Test 2	Perfect Coverage	Stochastic Coverage
Number of runs	10,000	10,000
Bad seed $k$mer	56	48
Number of tests	9,944	9,952
$v'_b \rightsquigarrow v'_c$ wrong	0	0
$v'_b \rightsquigarrow v'_c$ failed	0	0
$v'_b \rightsquigarrow v'_c$ overshot	0	0
Success	9,944	9,952
Conjecture true	100.00%	100.00%

Table 2.2: Traversing from $\overrightarrow{v_{v_b}}$ and testing if we can reach v'_c, which can be reached if we use $v'_a \ldots v'_b$. *Bad seed kmers* are $k$mers where no traversal is possible. 100X coverage with 100bp single-ended error free reads taken from 1Mbp of hg19 chr22.

In both simulations, and with perfect and stochastic coverage, we can see that the contig $v'_a \ldots v'_b$ does not improve traversal from $\overrightarrow{v_b}$. Following on from these simulations, when assembling contigs we need only traverse in each direction from a $k$mer, without using either side to inform the assembly of the other. Following from Proposition (2.5.8), we only need to seed with $k$mers that have not themselves already been used in a contig.

2.5.8 Extracting contigs

Seed	Contig
b, c or d	bcd
a, e or f	$abcdebcdf$

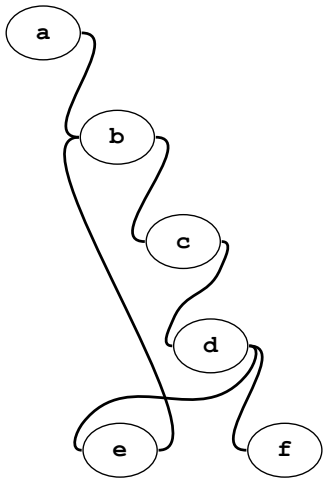


Figure 2.15: Assembling contigs from a sequence graph of the input: $abcdebcdf$. Right: simplified sequence graph of the input sequence. Left: Table showing, for each vertex, what the resulting assembled contig would be if it was used to seed an assembly. Contigs are assembled (as in a de Bruijn Graph) by traversing out from a seed $k$mer in both directions, stopping when hitting a vertex with out-degree not equal to one. Some seeds give contigs that contain other contigs. Using vertices that occur in the input exactly once as seed $k$mers results in a final contig that matches the input sequence.

Contigs are assembled by picking a seed $k$mer in the graph and traversing away from it to the left and then to the right. Assembling to the right uses no information from the traversal to the left. $k$mers used in this contig are then marked as having been used and are no longer used as seed $k$mers. Picking $k$mers in repeats means only the paralogous sequence is pulled out (see Figure 2.15). Many of these repeat contigs will be substrings of larger contigs that assemble across the repeat. We assemble contigs until all $k$mers are marked as used. At this point we remove substrings from the set of contigs to remove the repeat contigs.

If a sequence is stored in the graph with no $k$mer unique to it, it will not be possible to assemble it. Instead it will be assembled as many collapsed down repeat contigs.

2.5.9 Lossless storage

If we construct a Linked de Bruijn Graph from a single sequence that starts and ends with a unique $k$mer, following from Proposition (2.5.4), we should be able to recover it in its entirety from the graph. We have tested this by generating a 10kbp random sequence (over $\{A, C, G, T\}$) and adding unique 7-mers to the start and end. Using this sequence we build and annotate a graph at $k = 7$, then assemble contigs and remove substrings. In our tests we always get exactly one contig out of this process, and when compared

to the original sequence it always matches[2].

To test the lossless recovery of a genome sampled with short reads from a Linked de Bruijn Graph, we repeated the first test taking as input short reads instead of the entire 10Kb genome. We generated a random 10kbp haploid genome as above and identified the length of longest repeat (LR) in our genome by implementing a dynamic programming solution to the *longest common substring problem*. We generated perfect error-free coverage of the genome with a read length of $LR + 2$ starting at every base. We then built a Linked de Bruijn Graph ($k = 7$) from the reads, assembled contigs and removed contained contigs (those that were substrings of other contigs). After checking that we were left with a single contig, we compared it for an exact match to the original genome. This simulation was run 100 times without fail[3]. With $k = 7$, there are only $4^7 = 16,384$ possible $k$mers, so a random 10kbp genome will be quite repetitive.

As demonstrated, a linked de Bruijn Graph satisfies the requirements of a *string graph*: that every valid path is a valid sequence and that the original input sequence is itself a valid path. Given sufficient read data, it traverses all repeats shorter than the read length.

2.5.10 Link coverage estimation

Given sequencing depth and read length, we can work out the length of links with a given mean coverage by rearranging Equation (1.1) with the following expression:

$$L_l = R - \frac{D_l.R}{D} + 1 \tag{2.27}$$

where L_l is link length in bases, D_l is the mean coverage on the links of length l, R is read length and D is per-base sequencing depth. This is useful for estimating the length of the longest links kept after using a given cleaning threshold.

2.6 Traversal confidence

Traversal of a linked de Bruijn Graph constructed from error-free stochastic coverage can result in assembly errors, due to links being missed stochastically. By link length we mean the number of vertices in the path described by a link – from the vertex it starts on to the vertex its last junction choice points to. Assuming relatively high coverage sequencing, we can assume the following:

[2]This test can be re-run in the *tests/lossless* directory of McCortex
[3]This test can be re-run in the *tests/lossless_reads* directory of McCortex

1. Links of length greater than the read length will never be added to the graph.
2. Links much shorter than the read length should reliably be added.
3. Links close to but less than the read length will be added sporadically due to the stochastic coverage.

Repeats most likely to cause a misassembly are those of length $read - length - 2$, where some but not all valid paths across them will be annotated with links.

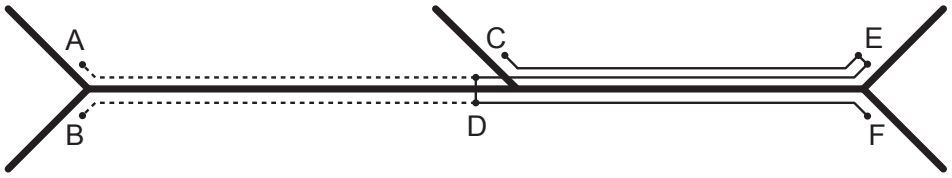
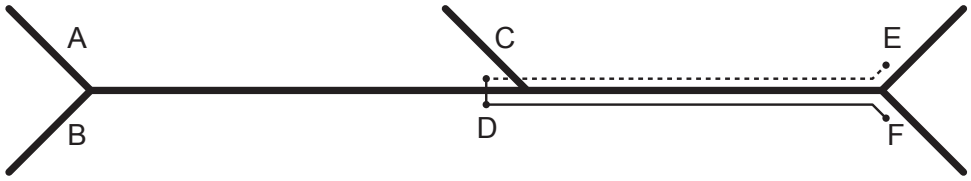
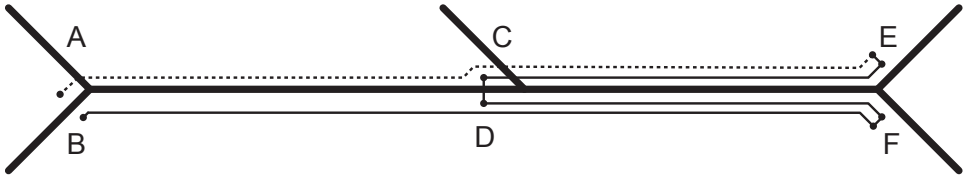


Figure 2.16: As the distance between junctions C/D and E/F approaches the read length, links across the tangle will be dropped as a result of stochastic coverage. The true structure is shown with dashed lines – these links are not present as they are longer than the read length and therefore never seen.

In Figure 2.16, we illustrate a tangle in the graph, where the true sequence is $A \rightsquigarrow E$, $B \rightsquigarrow F$, $C \rightsquigarrow E$. We assume the length of the repeat between junctions C/D and E/F is close to the read length. Links from A and B are never added since they are longer than the read length. The links we can expect to see are $C \rightsquigarrow E$, $D \rightsquigarrow E$ and $D \rightsquigarrow F$. However, if we have stochastic coverage (i.e. we are not guaranteed an error free read starting at every base) then we may see only some of these links. If just the $D \rightsquigarrow E$ link is lost, when we come from A we will end up at F instead of E – resulting in a misassembly stitching two unrelated sequences together at a repeat. If we missed both $D \rightsquigarrow E$ and $D \rightsquigarrow F$ links, we would halt traversal at the E/F junction and no misassembly would occur.



(a) In order to resolve the junction E/F, we are relying on a link from D to F, therefore confidence is calculated on missing a contradicting link from D to E.



(b) In order to resolve the junction E/F, we are relying on a link from B to F, therefore confidence is calculated on missing a contradicting link from B to E.

Figure 2.17: Calculating contig confidence at the junction E/F, in different scenarios. Proposed contradicting links that are not in the graph are shown as dashed lines.

Each time we resolve a junction using link information we can calculate our confidence in the chosen edge. To do this we calculate the probability of missing a contradicting link assuming one existed. The missing contradicting link is one that would disagree with our youngest un-contradicted link. This is illustrated in two scenarios in Figure 2.17. Once we have found the shortest contradicting link that would have to exist to change our behaviour (i.e. halt because our oldest links would disagree), we calculate the probability of missing said link. For this we adapted the equation used by Cortex (Iqbal *et al.*, 2012) to calculate expected power for seeing a given $k$mer:

$$\lambda = Coverage/ReadLength$$

$$R_d = ReadLength - JunctionDistance + 1$$

$$confidence = \left(1 - e^{-\lambda R_d}\right) e^{-\lambda e^{-\lambda R_d}}$$

This assumes $ReadLength \ll GenomeSize$. Figure 2.18 shows a plot of confidence calculations given different read length, coverage and gap distance values.

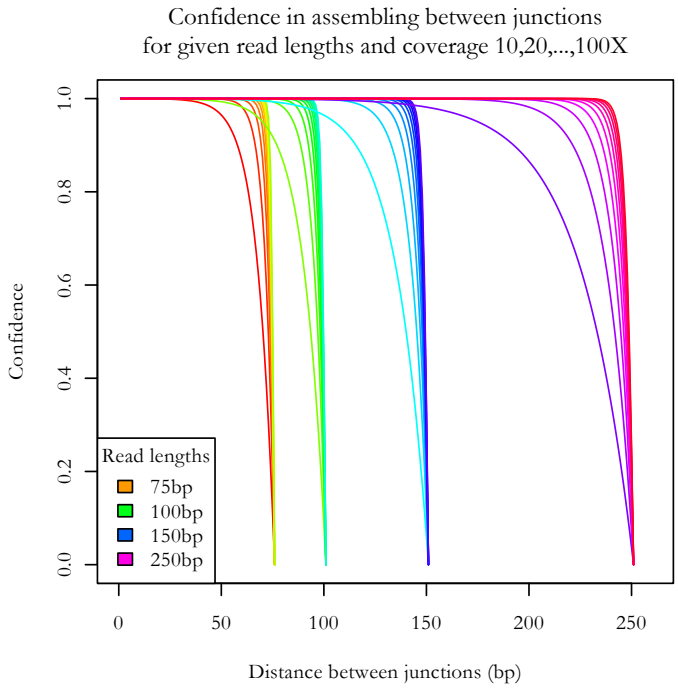


Figure 2.18: Calculating contig confidence given gap length between junctions. Shown for read lengths 75, 100, 150 and 250bp, each for $10, 20, \ldots, 100X$ coverage.

As we walk the graph, we keep track of the largest contradicting gap seen as well as multiplying together

junction confidences to give an overall assembly certainty for the contig. This is a conservative measure as at each junction we are assuming an alternative connection exists and we have missed it. However such certainty scores should give an ordering over which contigs are to be trusted and which are more likely to contain misassemblies.

During the read-threading step we record the size of reconstructed *read segments* (these are stretches of reads that match $k$mers in the graph). This histogram is then loaded by the assembly command which uses the contig length distribution and genome size in calculating junction confidences. Using a histogram gives a more informative distribution of actual read lengths than a single read length. It also allows the calculation of confidences when assembling with a mix of read lengths.

2.7 Reducing sensitivity to parameter k

As discussed in Chapter 1 (see Section 1.4), the single parameter of a de Bruijn Graph (k), can have a large effect on the outcome of an assembly. Optimal k depends on genome size and content, sample heterogeneity and population variation (π statistic), read length and sequencing error rate. In order to get the best results many popular assemblers today assemble at multiple values of k and combine results or choose the best output (e.g. velvet optimiser, IDBA(Peng *et al.*, 2010), SPAdes(Bankevich *et al.*, 2012), MEGAHIT (Li *et al.*, 2014), cortex_var pipeline (Iqbal *et al.*, 2013)). As 'best' can be hard to define for an assembly, results may be biased toward giving a large N50 but not necessarily a more convincing assembly given the data.

Genome assembly with linked de Bruijn Graphs is implemented by picking random $k$mers and walking as far as we can in the graph in both directions. This repeats until all $k$mers and links have been visited. Contigs that are substrings of other contigs are removed. More details on assembly are discussed in Section 4.2.

We can show that assembly results from a Linked de Bruijn Graph are not only better than a plain de Bruijn Graph, but are also less dependent on the value of parameter k [4]. Figure 2.19 shows the results from an assembly simulation on 1 Mbp of the human genome (GRCh37 chr22:17,000,000-17,999,999). Results are shown for three different 100bp single-ended simulated read data sets:

1. Perfect coverage means an error free read starting at every base (effectively 100X coverage).

2. Stochastic data are 100X coverage, with read starts sampled uniformly across the 1Mbp region.

[4] Code to run this analysis is in `results/kmer_size_experiment` in the McCortex repository

3. Stochastic data with error is 100X coverage, with read starts sampled uniformly and with sequencing base error of 0.5%.

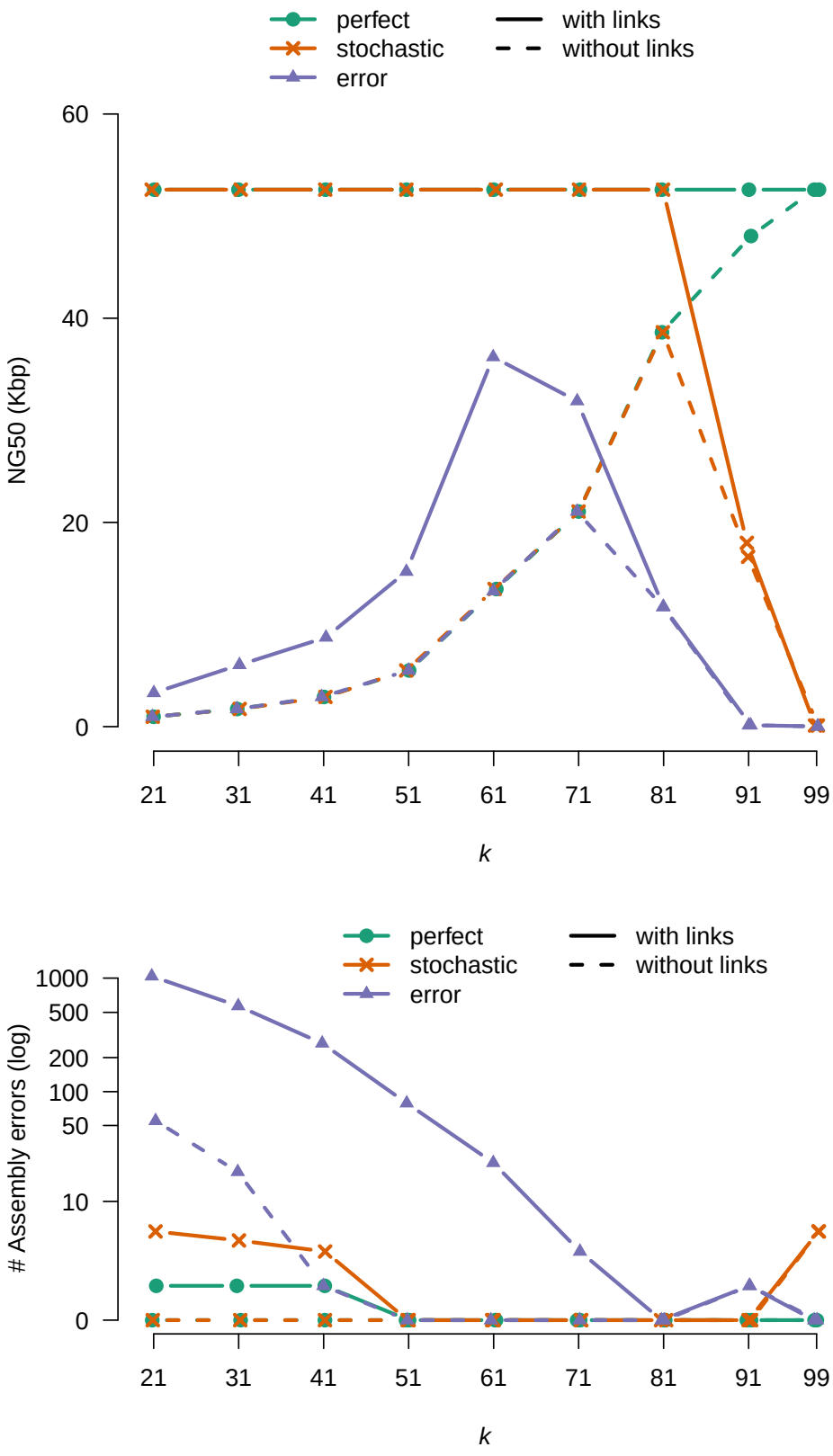


Figure 2.19: De Bruijn graph (with and without links) genome assembly simulations with varied k from simulated 100X coverage with 100bp single-ended reads. Source sequence is haploid 1Mbp from GRCh37 chr22:17,000,000-17,999,999. 'Perfect' coverage (green) is an error free read starting at every base. 'Stochastic' coverage (orange) means read starts sampled uniformly. 'Error' coverage (purple) is uniformly sample read starts and 0.5% sequencing base error. *Top:* NG50 after breaking contigs at assembly errors. Solid lines show results when using a Linked de Bruijn Graph, dashed lines represent performance with a plain de Bruijn Graph. *Bottom:* Number of assembly errors, found by comparing contigs to the true sequence.

The three sets of single-ended simulated reads were assembled with McCortex both with and without links. Assemblies were assessed with a script we wrote [5]. For each query string (assembled contig), we report all maximal substring alignments between it and the source genome (true assembly). Alignments are of length $\geq k$ and can include single base mismatches as long as they are flanked by k matching bases. From these alignments broken at assembly errors, we calculate an assembly NG50 and number of assembly errors. NG50 assembly metric and assembly errors for the three data sets are shown in Figure 2.19.

For *perfect* coverage, the traditional de Bruijn Graph (green dashed line) is very sensitive to k and improves up until there are only 2 $k$mers per read. The Linked de Bruijn Graph (green solid line) is insensitive to k, generating the same optimal NG50 for all values of k. With perfect coverage the Linked de Bruijn Graph makes no more than one assembly error, due to a loop created at the end of the genome when $k \leq 41$.

With the introduction of stochastic coverage, coverage gaps appear when $k$mer coverage decreases (i.e. at values of k that approach the read length). Coverage gaps cause NG50 to drop off at $k = 91, 99$. Additionally there is a slight increase in assembly errors caused by edges being lost that turn bifurcations into unambiguous forward edges resulting in overconfidence in traversal instead of stopping due to ambiguity.

When sequencing error is introduced we see a decrease in the assembly quality measured by NG50 and a large increase in the number of assembly errors. Linked de Bruijn Graphs still outperform on NG50 but with a large increase in assembly errors. It is clear that de Bruijn Graphs with or without links are very sensitive to sequencing error (even at low levels) and that our error correction is not doing a good enough job of correcting errors. McCortex cleans and assembles with the same value of k – error cleaning with a lower k may improve assembly. Alternatively using specialised error correction programs first should have a large effect on McCortex performance.

[5] `scripts/python/break-contigs-vs-truth.py` in the McCortex repository

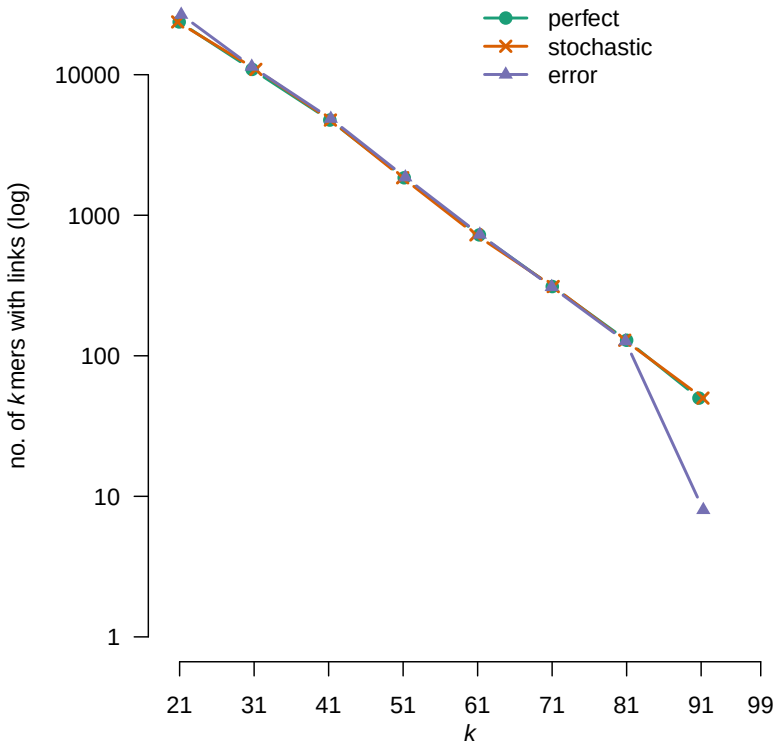


Figure 2.20: Number of $k$mers with links, generated from 100bp single-ended read simulations with varied $k$mer-size. Source sequence is haploid 1Mbp from GRCh37 chr22:17,000,000-17,999,999. 'Perfect' coverage (green) is an error free read starting at every base. 'Stochastic' coverage (orange) means read starts sampled uniformly. 'Error' coverage (purple) is uniformly sampled read starts and 0.5% sequencing base error.

The numbers of links generated for each assembly are shown in Figure 2.20. As expected there is an exponential drop off of $k$mers with links as k increases. If we had a randomly generated genome (with all four bases represented equally), and we had N copies of repeats of length exactly M, we would expect $4N$ copies of repeats of length $M-1$. Given the genome is not random we don't see such an aggressive drop off of repeats – in this 1Mbp of the human reference we see on average 2.4 fewer $k$mers adjacent to a repeat every 10 bases k increases.

In the absence of sequencing errors, links remove the sensitivity to k but at the cost of higher memory consumption to store exponentially more links at lower k.

2.8 Link pruning and memory

Once a set of reads have been threaded through the graph to add links, $k$mers will have lists of links associated with them. Some link lists will contain links that are prefixes of other links, e.g. $k$mer `ATCTAGCTAGTGCCGACTCTG` with links with junction choices $\{A, AC, ACT, ACTC, G, GC, GCG\}$. The longest link is given by the read that went the furthest after passing this $k$mer. Shorter links come from reads that ended shortly after this $k$mer. This set of links gives the same connectivity information as the set $\{ACTC, GCG\}$. Once we have amassed a set of links we wish to keep, we can reduce the set of links by removing contained links from link lists. This significantly reduces the memory requirement without impacting assembly, as shown with human data in Chapter 4.

Links represent connectivity in the graph and can be added to the graph from multiple sources not used in the construction of the original de Bruijn Graph. For example, graphs could be constructed from high coverage Illumina sequence and links added using PacBio long reads.

Sequencing errors can create reads that do not represent the sample they are from but that do describe a valid path through the sample graph. An single base error at the site of a SNP is one example. Another example is a base change at a junction in the graph. These reads perfectly match the graph but result in false connections between parts of the graph. In high coverage data these become more likely. Such false links will have low counts therefore we remove links if their count is below a threshold. The link cleaning threshold is calculated in the same way as the $k$mer cleaning threshold, using as input the distribution of link counts for links of length $k + 2$ only.

Links do not need to extend beyond unique $k$mers ($k$mers that only occur once in a sample). A unique $k$mer describes a single position in the input genome – we know exactly where in the genome we are and there is only one way to traverse forward. For this reason, if given very long contigs (e.g. a whole chromosome from a reference genome), memory usage can be improved by breaking input contigs at unique $k$mers (leaving a copy of the unique $k$mer at either side of the break). Given high coverage data, unique $k$mers could be conservatively estimated from coverage. The memory reduction for storing links would be most pronounced for long read data.

2.9 Discussion

We have presented a new data structure, called a linked de Bruijn Graph, that has the same properties of a string graph constructed from an overlap graph. We aimed to unify two opposing graphical approaches first proposed twenty years ago. In doing so we use the complete read information from a string graph

and combine it with the power of a de Bruijn Graph in variant calling and multi-sample comparison. Our linear time algorithm scales to multiple human genomes on current hardware (see Chapter 4) and makes no assumptions about sequencing platforms used. Sequence data from multiple platforms can be combined. Sensitivity to the only parameter (k) is reduced. The memory requirement for links increases linearly as read length increases relative to k. This is a property of having more information about connectivity. The underlying error-cleaned de Bruijn Graph remains roughly constant in memory as coverage and read length increases.

Construction works with long reads in the same manner as short reads. This is not the case for read-overlap approaches where longer reads give longer overlaps which accumulate errors. Graph annotation includes an intrinsic error correction step, reducing the need for quality control before beginning assembly. Obviously third party error correction steps can still be used (e.g. Diginorm (Brown *et al.*, 2012)).

Whereas the EULER assembler (Pevzner *et al.*, 2001b) and MULTIBRIDGING (Bresler *et al.*, 2013) changed the structure of the de Bruijn Graph, we achieve the string graph properties with an immutable de Bruijn Graph. Mutating the de Bruijn Graph brings up memory allocation issues and breaks the ability to compare multiple samples. Instead we can load up multi-colour linked de Bruijn Graphs to compare. By annotating the de Bruijn Graph after it has been constructed in a two-pass pipeline, we add the option of adding annotations to the graph later as more sequencing is done and long read data becomes available. It also allows using different annotations for different purposes e.g. conservative annotations could be used for variant calling, more liberal annotations could be used for assembly where errors can be identified and fixed downstream and contig length is more important.

The EULER assembler was the first to use copy number to aid in resolving repeats – copy number must be estimated to generate an accurate eulerian path. Linked de Bruijn Graphs rely only on connectivity information in reads, not coverage, to resolve repeats. This enables their use with low-read-count long read technologies as well as high-throughput short read technologies.

An advantage to a de Bruijn Graph rather than read-overlap approach is the ability to drop into the graph anywhere in a multi-colour graph and start walking. This is useful for variant calling amongst other things. As we are walking we can identify low confidence steps that may be missed in read-overlap approaches, which would fail to notice pathological graph motifs. By looking at local graph structure during assembly, we calculate confidence scores across contigs and for overall assemblies.

We avoid storing redundant read data, whilst still making all useful read information available to downstream analysis. This enables working with the read data in a compressed and error corrected form. In Chapter 3 we discuss implementation details and in Chapter 4 show several such applications. We notice

that the idea of threading reads through a de Bruijn Graph has been previously suggested as a method of compression (Conway and Bromage, 2011), and using a unitig graph as a reduced compressed version of the read data has also been raised (Li, 2015b). In the next chapter we bring these ideas together in an implementation addressing de novo assembly for variant calling.

Chapter 3

Implementation

3.1 Implementation

We have implemented the *de novo* assembler and variant caller *McCortex* in *C* as a command line application and released it with the MIT license. As input it will read sequence in multiple formats including FASTA, FASTQ, SAM, BAM or plain (one sequence per line). The program is multi-threaded to take advantage of modern multiprocessor machines. McCortex runs on Mac and Linux, and the code and all dependencies are open source. McCortex generates de Bruijn Graph files and then re-annotates these in a second pass of the reads to generate link files that contain long distance connectivity information. Errors in graph and link files are removed in cleaning steps. After generating both *k*mer and link files, they are then used together as a Linked de Bruijn Graph. Linked de Bruijn Graphs can be used for several algorithms including those that assemble the genome, correct reads, find polymorphisms or identify large structural events. McCortex includes a script for simplifying common analysis tasks (*McCortex Pipeline*). McCortex is freely available at `https://github.com/mcveanlab/mccortex`.

3.1.1 Commands

We have aimed to provide a library of code that can be used to manipulate multi-sample (coloured) and link-annotated graphs. McCortex is a modular set of commands within a single executable. Adding additional commands is designed to be straightforward. We have implemented the following commands:

```
usage: mccortex31 <command> [options]
commands:
  breakpoints  use a trusted assembled genome to call large events
  bubbles      find bubbles in graph which are potential variants
```

```
build        construct cortex graph from FASTA/FASTQ/BAM
calls2vcf    reduce set of strings to remove substrings
check        load and check graph (.ctx) and link (.ctp) files
clean        clean errors from a graph
contigs      assemble contigs for a sample
correct      error correct reads
coverage     print contig coverage
index        index a sorted cortex graph file
inferedges   infer graph edges between kmers before calling 'thread'
join         combine graphs, filter graph intersections
links        clean and plot link files
pjoin        merge link files (.ctp)
popbubbles   Pop bubbles in the population graph
pview        text view of a McCortex link file (.ctp)
reads        filter reads against a graph
rmsubstr     reduce set of strings to remove substrings (post-assembly processing)
server       interactively query the graph
sort         sort the kmers in a graph file
subgraph     filter a subgraph using seed kmers
thread       thread reads through cleaned graph to make links
uniqkmers    generate random unique kmers
unitigs      pull out unitigs in FASTA, DOT or GFA format
vcfcov       coverage of a VCF against cortex graphs
vcfgeno      genotype a VCF after running vcfcov
view         text view of a cortex graph file (.ctx)
```

Previously, Cortex (Iqbal *et al.*, 2012) required recompiling for different numbers of samples. We have done away with this requirement. However you still need to compile for the $k$mer size range to be used: `mccortex31` handles k from $1 \ldots 31$, `mccortex63` handles k from $33 \ldots 63$ and so on.

3.1.2 McCortex sequence input

In order for McCortex to support multiple input formats that are commonly used in the bioinformatics community, we have implemented and shared a C library to read multiple file formats. Our library `seq_file`[1] provides a simple C API to read sequence files and has the following features:

1. Reads SAM, BAM, FASTA, FASTQ, gzipped FASTA/FASTQ and plain one-per-line sequences
2. Auto-detects file formats based on content rather than file name
3. Auto-detects quality score offset used
4. Buffers input to speed up reading, especially with gzip versions before 1.2.3.5 (Jan 2010)

Table 3.1 shows the effect of buffering on loading a single chromosome (`chr22.fa`) with `seq_file`. Buffered reading loads large chunks of the file into a buffer before parsing it, where as unbuffered processing reads a character or one line at a time from the file. The effect on networked file systems is more

[1] `seq_file`: `https://github.com/noporpoise/seq_file`

pronounced (data not shown), where a system call to read from a file has high latency. The test was run 5 times with and without buffering and the mean and standard deviations reported. The test was run once without recording the times to avoid caching issues (on disk, in RAM and by the operating system). Buffered reading appears to speed up loading times by a factor of three. We expect this effect to be even larger on hard drives and networked file systems.

Reading	Mean Loading Time (s) [σ]
Un-buffered `chr22.fa`	0.319 [0.0077]
Buffered `chr22.fa`	0.098 [0.0023]

Table 3.1: Time to load an uncompressed FASTA file from disk with the `seq_file` C library. The file contains a single chromosome (`chr22.fa`). Mean and standard deviation are for 5 runs on a Macbook Pro (13-inch, 2010); 2.66 GHz Intel Core 2 Duo; 4 GB 1067 MHz DDR3; Mac OS X (10.10.3); using an internal solid state disk (SSD). Buffer size used 2^{20} bytes ($1MiB$).

`seq_file` supports SAM and BAM formats through the `htslib`[2] library. This is optional, but compiling without it results in an error message when a SAM/BAM file is passed in.

All McCortex commands can take input from a unix pipe (streamed from another command) and output can be written to a file or to standard out (STDOUT). This allows McCortex to be quickly integrated into a shell script or command pipeline without having to write intermediate files to disk. An example of how this might be useful is during conversion from an unsupported file format without creating intermediate files e.g. `samtools bam2fq in.cram | mccortex31 build -k 31 -sample NA12878 out.ctx`. This is also useful for filtering large datasets. Instead of duplicating your data on disk to remove certain reads, you can filter the input file as it is streamed into McCortex. Enabling the use of piping prohibits multi-pass processing on files but enables commands to be quickly integrated into simple command pipelines. Here is an example command that converts and samples data, constructs a graph, plots it and finally outputs a pdf:

```
samtools bam2fq nin.cram | head -1000 | mccortex31 build -q -k 31 --sample NA12878 - | \
  scripts/cortex_to_graphviz.pl -k 31 - | dot -Tpdf > NA12878.pdf
```

When reading from multiple files and using multiple threads, we use a library we developed called *msg-pool* which operates like a lockless many-to-many consumer-producer queue. It is not really a queue since it does not preserve the order of the inputs, hence it is a 'pool' rather than a 'queue'. It is implemented by creating an array of objects over which producers perform a linear search for an empty spot to enter a new input. Consumers perform a similar linear search to find items that are waiting to be processed. The implementation avoids the use of locks by instead using compare-and-swap (CAS) instructions.

Using a message pool makes reading from multiple files at the same time simple in the code and enables us

[2] `htslib`: `https://github.com/samtools/htslib`

to parallelise operations on input data. The number of threads consuming input sequence data can be set with the '`-t,-flags <N>`' flag for McCortex or with '`NTHREADS=<N>`' for the McCortex pipeline.

3.1.3 Graph files

Specific samples can be loaded from a graph file, by using the pattern `<in>:<file>:<colours>` instead of just giving the path the graph file. As an example, if we have a graph file of a hundred individuals and want to use only two of them to do bubble calling:

```
mccortex31 bubbles -o bubble.ben.jerry.txt.gz 0,1:pop.ctx:3,42
```

This syntax means we load colours 3 and 42 into graph colours 0 and 1. Having this ability greatly improves the flexibility of the McCortex programs and makes working with multiple samples (multi-colour graphs) much easier. This syntax can be used with any McCortex command that reads graph files. Additionally, most commands will accept multiple graph files to avoid having to merge files on disk beforehand. Both these features were added after experience with Cortex, where they were absent. These features also work with link files.

3.1.4 de Bruijn Graph representation

De Bruijn graphs can be stored in memory in several ways, the most obvious being a hash table of $k$mers. This was the approach used in Cortex, with each hash entry containing a $k$mer and all associated data (e.g edges and coverage in each sample). Instead we have split out associated data and only store $k$mers in the hash table. Depending on the command being run, we allocate separate arrays for each field required (e.g. coverage per sample). For many purposes this significantly reduces memory usage. Many commands need only to know if a sample is in a specific colour – for this we use a single bit per $k$mer per colour to indicate a $k$mer was seen in a sample. An additional benefit of allocating separate arrays at runtime rather than at compile time is we no longer need to compile a separate executable for different population sizes.

The hash table approach used by Cortex and McCortex requires 80GB of RAM for a human sample. There are alternative approaches to storing a de Bruijn Graph and many of these make meaningful reductions in the memory required (see Section 1.4.3). Conway and Bromage (2011) calculated a lower bound on memory required to store the human genome de Bruijn Graph at 12GB, if we assume $\sim$3 billion random $k$mers in the graph. An implementation they provided using rank and select operations on a bitvector achieved 16GB. Minia (Chikhi and Rizk, 2013) managed to achieve peak RAM usage of 5.7GB

for human sample by using a Bloom filter approach. One downside to this approach is the use of disk access to remove false positive $k$mers which slows down processing of large amounts of data. Additionally their data structure is unable to tell if a given $k$mer is in the graph or not. Instead querying of the graph may only be done by traversing through it one $k$mer at a time to adjacent $k$mers. For most assembly operations this is a sufficient requirement.

To investigate one low memory approach, we have implemented a low memory hash table inspired by Jellyfish (Marçais and Kingsford, 2011). This approach uses the fact that a hash value, and therefore where in a hash table an item is placed, tells us something about the item itself. This technique is called "quotienting" or *abbreviated keys* and proposed by Knuth (Knuth, 1973, Vol 3, Section 6.5, Exercise 13) was first used in a hash table with open addressing by Clerry (1984). By using a hash function with a known inverse function this information can be recovered, thereby reducing the amount of memory per item that we need to store. To conserve additional space the hash table is stored as a bit-packed array. In Jellyfish, multiplication by a random matrix is used as the hash function and multiplication by the inverse matrix as the reverse hash function. The random matrix and its inverse are generated each time the program is run. Instead of matrix multiplication, we have derived the inverse function for a commonly available hash function known as `twang32` which uses multiplication and bit shifting. Optimised versions have been made available by Facebook Inc.[3].

In our implementation (*jelly-hash*), element memory usage is $(m - l + 5)$ bits with hash table capacity of $b \times 2^l$, where m is the number of bits per entry, 2^l is the number of buckets, b is the bucket size. The hash of an element gives l bits that specify which bucket it is in. The remaining $k - l$ bits are stored in the bucket. Each element has 5 housekeeping bits: one is a read/write lock and the remaining 4 indicate how many times it has been hashed due to collisions ($1..2^5 - 1 \Rightarrow$ up to 31 times). Given a hash table position that is occupied by an entry that was hashed three times, we must concatenate the entry with its bucket number and call the inverse hash function three times to get the original item before it was added to the hash table. *jelly-hash* is multithreaded, mutex-free (using compare-and-swap (CAS) instructions) and allows items of any number of bits in size. Since the hash table uses a bit-packed array, hash table entries may span computer-word boundaries. Writing an entry therefore may require multiple CAS operations. This requires us to use a lock bit to indicate that an entry is being written so other threads do not read half-written entries. We could have used bucket locks instead of item level locking.

In Table 3.2 we compare the memory and time taken to fill each of these two hash table implementations. Although memory usage was reduced by ∼30%, speed reduced by a similar amount. Given a human sample uses 80GB for the hash table and we are using machines with terabytes of RAM, we decided that

[3]Reversible hash functions: `https://github.com/facebook/folly/blob/master/folly/Hash.h`

the trade-off was not worthwhile. Iterating over items in *jelly-hash* is also much slower since we must call the inverse hash function multiple times per entry. The inverse hash function is much slower than its counterpart.

The hash table in McCortex is allocated to a fixed size when a command is run. The size is calculated based on the size of input data and an upper bound specified by the user. Alternatively the user can require the hash table to be of a given capacity. If the table fills up, we give an error and exit. We do not attempt to resize the hash table. This has the advantage of using all available memory and saving time reallocating, but requires the user to give a meaningful upper bound on memory usage when the command is run and re-run if the command fails due to exhausting its memory limit.

3.1.5 Multithreading

Cortex was entirely single threaded. The memory required by assemblers is large – 350GB for variant calling with 10 humans in Cortex. High RAM servers usually have 12-48 CPU cores. Using only one core and all the memory leaves much of the paid-for resources (CPU time) unused, slows down the job and holds up others who need computer resources.

Therefore, we have worked to make McCortex highly multithreaded. Loading sequence data is multi-thread, as are all commands' operations on the graph. The number of threads used can be set at run time for all commands with the `-t,-threads <N>` flag. Where possible we have avoided the use of mutex locks which can slow down multithreaded programs, instead using low level Compare-And-Swap and atomic ADD, AND, OR and XOR-bitwise instructions (known as 'intrinsics'). For example, the atomic ADD instruction allows us to increment the coverage on a given $k$mer without worrying about whether other threads are accessing the same $k$mer.

In order to simplify multithreading, we wrote a multithreaded hash table that uses locks to access each bucket (as discussed in the previous section). The number of buckets and bucket size are decided and fixed when a command is run so we do not have to worry about resizing the hash table. This greatly simplifies and speeds up hash table operations. Each bucket lock is represented by a single bit. This is achieved by writing our own lock implementation based on using the compare-and-swap instruction [4]. Several commands that carry out parallel operations on the whole graph also use a lock per $k$mer (each one bit) to ensure that only one thread operates on a any given $k$mer at a time. For a human sample, a bit lock per $k$mer uses 0.5GB, and the hash table uses 12MB for bucket locks.

Failing to acquire a lock causes us to interrupt the thread, wait until the operating system schedules the

[4]Single bit locks (BitArray): `https://github.com/noporpoise/BitArray`

Platform	Table	Bucket size	# Buckets	Table size	# Elements	Memory	Bits / element	Mean time (s) [σ]	Time ratio	Memory ratio
mac	*jelly-hash*	25	2^{22}	104,857,600	80,000,000 (76%)	562.5MiB	59	58 [2.1]		
mac	*McCortex*	25	2^{22}	104,857,600	80,000,000 (76%)	808.0MiB	85	40 [0.5]	1.45	0.70
linux	*jelly-hash*	25	2^{22}	104,857,600	80,000,000 (76%)	562.5MiB	59	42 [0.1]		
linux	*McCortex*	25	2^{22}	104,857,600	80,000,000 (76%)	808.0MiB	85	19 [2.5]	2.18	0.64
linux	*jelly-hash*	32	2^{25}	1,073,741,824	800,000,000 (75%)	5.2GiB	56	492 [0.7]		
linux	*McCortex*	32	2^{25}	1,073,741,824	800,000,000 (75%)	8.1GiB	87	216 [16.8]	2.28	0.64

Table 3.2: Comparing hash tables: *jelly-hash* vs. *McCortex*. We filled a hash table with entries $0 \ldots N$ where N is the number of items inserted into the hash table. Hash entries were 64 and 62 bits for *McCortex* and *jelly-hash*, representing storing 31−mers. Jobs were run five times and the median and standard deviation of the times in seconds reported. The last two columns show the relative time and memory used for *jelly-hash* vs. the *McCortex* hash table. The linux machine used GCC 4.4.7 and has 8 × Intel(R) Xeon(R) 8 core CPU running at 2.70GHz, with 2TB of RAM running CentOS release 6.5 and linux kernel version 2.6.32. The mac used clang Apple LLVM version 6.1.0 and is a MacBook Pro (13-inch, Mid 2010); 2.66 GHz Intel Core 2 Duo; 4 GB 1067 MHz DDR3; Mac OS X (10.10.3)

thread again and retry acquiring the lock. Since we are using bit-packed locks and the compare-and-swap intrinsic, we will fail to acquire the lock if another thread attempts to acquire a lock in the same byte as the one we are attempting to lock. This failure rate can be calculated in the same way as the birthday problem, where T is the number of threads.

$$\text{bytes used } B = \left\lceil \frac{\text{Number of locks}}{8} \right\rceil$$

$$p(2+ \text{ accesses to same byte}|B,T) = 1 - \prod_{1}^{T} \frac{B-i+1}{B}$$

When using 48 threads on a human genome sample where we expect ~ 4 billion hash table entries, using $k$mer single bit locks will give a collision rate of 2×10^{-6}. Using buckets locks (expect $\sim$100 million buckets, each of $\sim$40 $k$mers) will give a collision rate of 9×10^{-5}. The probabilities of two threads attempting to lock the same $k$mer/bucket is even lower. Therefore acquiring a lock for a $k$mer or hash table bucket before operating on it adds at most a couple of processor operations. Bit packed locks appear to be a good choice when the number of items to be protected is orders of magnitude larger than the number of threads.

Simple parallel operations on each $k$mer do not require locks if we can share out $k$mers between threads. Underlying the hash table is an array of $k$mers, which can be divided into T sections, each to be processed by one of T threads. In order to perform parallel operations on unitigs, we have T threads looping over $k$mers, finding $k$mers that have not been visited before, pulling out their unitig and marking the member $k$mers as visited. $k$mers are marked as visited using intrinsics (atomic instructions), whose return value is used to determine if another thread visited a $k$mer whilst we were trying to mark it. Using this approach we provide a simple API to allow parallel operations to loop over unitigs. This is used by graph cleaning, bubble popping and the extract unitigs commands.

Loading of input sequence is also done in parallel by a process of asynchronous input as described in Section 3.1.2. Each file has a thread reading from it, parsing input and putting read sequences or pairs of sequences into a 'message pool'. The message pool supports multiple-producer, multiple-consumer threads. It is similar to a queue, however the pool does not preserve order of items as they are added/removed. Relaxing this requirement allows an improvement in performance by avoiding the need to use mutex locks. This is fine for our usage, as the order of input sequences does not matter. This is the reason the error correction command re-orders reads.

3.1.6 Memory allocation

When running McCortex we take an argument for the maximum memory to use. This is used as a limit on memory allocation of large data structures (hash table and link storage). This allows the user to use the tool without needing to do calculations first or know the fine details of McCortex data structures. If the user specifies too little memory, we exit with an estimation of the required memory for the task.

Since we know our memory limit when we start, the hash table is of fixed size and cannot be resized. Previously Cortex required the user to specify hash table parameters (bucket size and number of buckets). Instead, we use the overall memory limit to automatically calculate good values for these parameters.

As C requires manual management of our dynamic memory usage, we have written wrapper functions for common memory allocation functions to protect against common allocation bugs: `malloc(2)`, `calloc(3)`, `realloc(2)` and `free(1)`. The wrappers on allocation functions check for allocation failure and exit with an error. These wrappers also increment a counter on each call to allocate memory and de-increment on calls to `free()`. On a successful exit we report an error if the number of un-freed allocations is not zero. This helps us catch memory leaks during development and catch out-of-memory errors during running. An additional function `reallocarray(3)` enables us to rigorously check for overflows in multiplication before allocation. An example of such a bug is trying to allocate -1×2 byte elements (without a check, this would attempt to request $2^{64} - 2$ bytes).

3.1.7 JSON file headers

Graph links are stored in link files with the `.ctp` file extension, or `.ctp.gz` when compressed with `gzip`. Graph link files, bubble and breakpoint call files all start with a JSON header. From `json.org`:

> JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999.

The advantage of using JSON is that most programming languages have libraries to read JSON and the format can natively represent common data types (integers, reals, booleans, arrays and strings). It is also a formal standard and human readable. We believe this is preferable to using the bespoke formats created for VCF or SAM files or inventing our own.

Our headers include a tree of commands used to generate them. Each command has the username of the user who ran the job, machine host name, creation time and version of the software used. We record all

of the info we can to improve reproducibility and accountability. We have found this to be very useful and encourage others to do the same.

3.2 Details of graph operations

3.2.1 PCR duplicate removal

PCR duplicate removal identifies reads that represent the same DNA fragment as other reads in the input. It is necessary to remove localised peaks in coverage. Such peaks adversely affect error cleaning and variant genotyping.

PCR duplicate removal is carried out in a similar way to Cortex, with a few improvements in the implementation. In Cortex, if PCR duplicate removal was specified, the $k$mer-key at the start of each read was marked as visited. If a read started at a $k$mer-key already marked it was dropped. For paired end reads both reads had to start at an already marked $k$mer-key.

In McCortex we mark the first $k$mer as seen in the given orientation – this reduces collisions of reads starting at the same $k$mer-key and going in opposite orientation. By reducing such false positives we can reduce lost reads and improve coverage. $k$mer marking is done by storing a bit array of size $2N$ bits where N is the number of $k$mers in the hash table. This is a four-fold reduction in memory on the one byte per $k$mer used by Cortex.

Additionally, we now allow the user to specify read pair orientation used by the library (i.e. FF (forward-forward), FR (forward-reverse), RF and RR). Therefore we can identify the first $k$mer of each read, rather than just assuming reads are in FF orientation. It is important to use the first $k$mer rather than the last as duplicated templates from PCR are likely to start from the same base, but reads from them could end in different places – giving multiple different end $k$mers.

Finally, reads from SAM/BAM files can be filtered if they are already marked as duplicates.

3.2.2 Infer edges

In Cortex (Iqbal *et al.*, 2012), edges between $k$mers in the de Bruijn Graph were added only if the two $k$mers were seen overlapping in the input sequences. This is the same as saying that an edge implies a string of length $k+1$ was seen in the input sequences. This is the same as the definition of DBG_k^+ given by Cazaux *et al.* (2016).

In McCortex, we reduce the information stored in a $k$mer by adding all edges between two $k$mers in the graph if they share an overlap of $k-1$ bases. We call this the *infer edges* step. It makes the information given by edges redundant given the set of $k$mers in the graph. The resulting de Bruijn graph is equivalent to the DBG_k^- definition given by Cazaux *et al.* (2016). Inferring edges has the effect of reducing the effective $k$mer-size by 1 base. It adds edges that have been missed in the graph because each of the two $k$mers were only seen at the ends of reads, therefore they were never seen adjacent to each other. This step may also add invalid edges, however these invalid edges would also exist in a traditional graph of $k-2$ of the input sequences. Inferring edges allows us to merge samples without needing to store the edges for each sample. Now we only need to store the merged population edges and rely on the $k$mer colours for sample information only. To test if a colour has an edge between two $k$mers, all we need is the union of all samples' edges and information about which sample each $k$mer is in.

Without inferring edges we would need to store a byte (8 edges) per colour per $k$mer. For 10 human samples, the RAM required works out to be: 10 samples $\times 3$ billion $k$mers $\times 1$ byte $= 30$GB. Instead we use a single bit per $k$mer per colour and 1 byte per $k$mer for the union of edges. For 100 human samples the RAM required is: 10 samples $\times 3$ billion $k$mers $\times \frac{1}{8}$ bytes $+ 3$ billion $k$mers $\times 1$ byte $= 6.75$GB.

The ABySS de Bruijn Graph assembler (Simpson *et al.*, 2009) carries out a similar operation, adding edges between all $k$mers that share $k-1$ bases of overlap.

3.2.3 $k$mer error cleaning

We model coverage on erroneous $k$mers with a Poisson whose mean is given by a Gamma distribution. Our rationale is that erroneous $k$mers are generated from real $k$mers with errors introduced by sequencing. The coverage on the real $k$mers is assumed to be Poisson. Some erroneous $k$mers will arise from mistakes on different backgrounds and therefore be more common. For instance erroneous $k$mers generated from $k$mers in repeats will have much higher mean coverage than erroneous $k$mers from elsewhere in the genome. It follows that the expected coverage on erroneous $k$mers is Poisson with a mean drawn from some unknown distribution. We model the distribution of mean error coverage as a gamma and estimate the parameters α and β.

The probability of seeing an erroneous $k$mer with coverage x is given by a Poisson with mean given by a gamma distribution (see derivation in Appendix C):

$$p(x) = \int \frac{\beta^\alpha}{\Gamma\alpha}\mu^{\alpha-1}e^{-\beta\mu}.\frac{e^{-\mu}\mu^x}{x!}d\mu \tag{3.1}$$

$$= \frac{\beta^\alpha}{\Gamma(\alpha)}\frac{1}{x!}\int \mu^{\alpha+x-1}e^{-\mu(1+\beta)}d\mu \tag{3.2}$$

$$= \frac{\beta^\alpha}{\Gamma(\alpha)}\frac{1}{x!}\frac{\Gamma(\alpha+x)}{(1+\beta)^{(\alpha+x)}} \tag{3.3}$$

We then pick a cleaning threshold by choosing a cut-off T (i.e. $T = 1/100$ $k$mers due to error) and find the lowest coverage c such that $errcovg(c)/covg(c) < T$. Unitigs without a $k$mer with $coverage \geq c$ are considered the result of error and are removed.

Different tasks may have different optimal cleaning thresholds. Each error remaining in the graph will likely stop graph traversal (ambiguous fork). Variant calling seeds in many places, so variants may still be recovered. Successful variant calling requires short contigs, and it is not important if we have to walk the path through the graph forward or in reverse. Assembly is more sensitive to errors, which break contigs and reduce N50. Many of these gaps can be recovered through mapping paired end and long reads back onto the contigs in the scaffolding step.

3.2.4 Link cleaning

In order to remove errors in links due to sequencing errors, we iterate over each $k$mer, and for each $k$mer construct a tree of the links leaving it in a given orientation. Each branch in the tree is a junction choice made in the graph by one or more reads. Branches are labelled with their read count. True coverage will drop off down the tree, as will coverage due to errors, as that is also a function of true coverage. This tree is pruned of branches with counts below a given threshold. Each leaf of the tree then emits a link. This greatly reduces the number of links by collapsing down links which are substrings of longer links as well as removing many links due to error. The threshold is chosen by sampling the first 1000 $k$mers with links and taking the distribution of counts of branches one $k$mer away from the root $k$mer. We fit our $k$mer coverage model to this distribution and pick a threshold that will give us fewer than one in a thousand false links. This is repeated for the distribution of counts 2...5 $k$mers away from the root $k$mer and the median threshold is used. A length-dependent link cleaning threshold could also be used, and may improve performance.

3.2.5 Graph walking

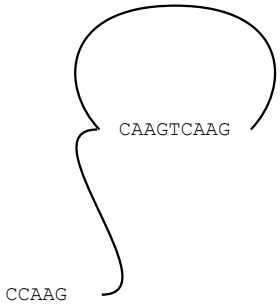


Figure 3.1: Infinite loop in a compacted de Bruijn Graph with $k = 5$. If we were to walk forwards from `CCAAG`, we would be stuck forever in the loop `CAAGTCAAG`. In order to avoid this we must detect that we are stuck in a loop and halt walking.

During graph walking, it is possible to wander into a loop with no edges leaving it (see Figure 3.1). To avoid being stuck forever in the loop, we must detect that we are stuck in a loop and halt traversal. Graph walking uses a bitvector to track which $k$mers have been visited (1-bit per $k$mer) and a Bloom filter to record the state of the walker every time it revisits a $k$mer. A Bloom filter is a probabilistic space-efficient data structure which returns whether or not an element is in a set, with zero false negatives but a low number of false positives (see Section 1.4.3). As discussed in previously, assuming the hash functions are independent and have good distribution, the false positive rate of a Bloom filter is (Mitzenmacher and Upfal, 2017, p. 116-118):

$$\left(1 - e^{\frac{-jn}{m}}\right)^j \tag{3.4}$$

where m is the number of bits, n is the number of elements and j is the number of hash functions. The optimal number of hash functions can be approximated by (Mitzenmacher and Upfal, 2017, p. 118):

$$\text{optimal number of hash functions } = \frac{m}{n} \ln 2 \tag{3.5}$$

An Alu element is a ∼300bp repeat that occurs only in primates. A human genomes contains approximately one million copies of it (Szmulewicz *et al.*, 1998). We use 4MB (33.6 million bits) per walker instance (usually one per thread) and three hash functions which each return 22 bit output. Fast hash calculation is achieved by creating one 64 bit hash of the walker instance state and splitting it into three 22 bit hashes. Assuming we walk through 100 copies of an Alu (∼ 300 *k*mers long) in a contig, we have a false collision rate for the last *k*mer of the last Alu to be 1.9×10^{-8}, and a total failure rate of 0.01%[5]:

$$\prod_{m=1}^{30,000} \left(1 - \left(1 - e^{\frac{-3\times 2^{22}\times 8}{m}}\right)^3\right) = 0.01\% \tag{3.6}$$

In other words, we stop prematurely after 100,000 repeated *k*mers only 0.01% of the time. Assuming 1 million Alus evenly distributed in the human genome, we expect to cover 100Mbp before we reach the 0.01% premature assembly halt. Increasing the Bloom filter size reduces false positives but increases the memory we must zero each time we restart walking (e.g. once for each contig assembled).

3.3 McCortex tools

3.3.1 McCortex commands

McCortex provides several tools for manipulating population graphs.

`check` the validity of graph and link files. For graph files it checks that all edges point to *k*mers that are in the graph and that all edges are reciprocated. Reciprocated edges means that if there is an edge from $vertex1 \rightarrow vertex2$, there is also an edge $vertex2 \rightarrow vertex1$. This ensures all edges are bidirectional. For link files it checks that all links are valid paths through the graph. These commands are useful when developing programs which create and manipulate graphs and links.

`coverage` command can be used to pull down *k*mer coverage and edges for given input sequences.

The `join` command combines multiple sample graphs into a single multicolour output graph. It can also filter graph intersections. For very large populations and, it has the ability to do a low memory merge. All samples are loaded into a single colour. This is then written to a memory mapped file. We then load a few samples, storing their coverage and edges this time, and write their sample values to the file. We loop over all samples until the output file has individual *k*mer coverages and edges for each sample. The

[5] see `scripts/calculations/bloom-filter-fpr.pl` in the McCortex repo

limit on graph file size is then the number of $k$mers in the pooled population, where memory usage is $8N$ bytes where N is the number of $k$mers, and each $k$mer takes 8 bytes (when $k < 32$). If the `join` command is filtering a single file it does this as a stream so memory usage is very low.

The `subgraph` can retrieve a graph subset using seed $k$mers given with sequence files and a Depth First Search (DFS) up to a given distance specified by the user.

The `unitigs` command prints the *unitigs* that make up a graph. *Unitigs* are maximal runs of $k$mers with in-degree and out-degree of one. As defined in Section 2.1.1, McCortex forces a strict one-to-one mapping of a $k$mer graph to a set of unitigs.

A set of reads can be filtered against the graph using the `reads` command, which only prints out read pairs that touched a $k$mer in the graph. It was envisioned that this would be used to do local re-assembly of genome regions of interest. In this situation a genome region would be provided to filter a small set of reads from one or more samples, which would then be used for aggressive re-assembly with lower memory.

Text representation of binary graph files can be generated with the `view` command. This is used in plotting scripts, as well as by users who may wish to interrogate the graph for $k$mers, coverage and edges themselves.

The `sort` command sorts the order of $k$mers in a graph file alphabetically. The `index` command can then create an index of the sorted file for fast disk-based access. This is for external programs to take advantage of, as McCortex does not use disk access to run.

3.3.2 Visualisation

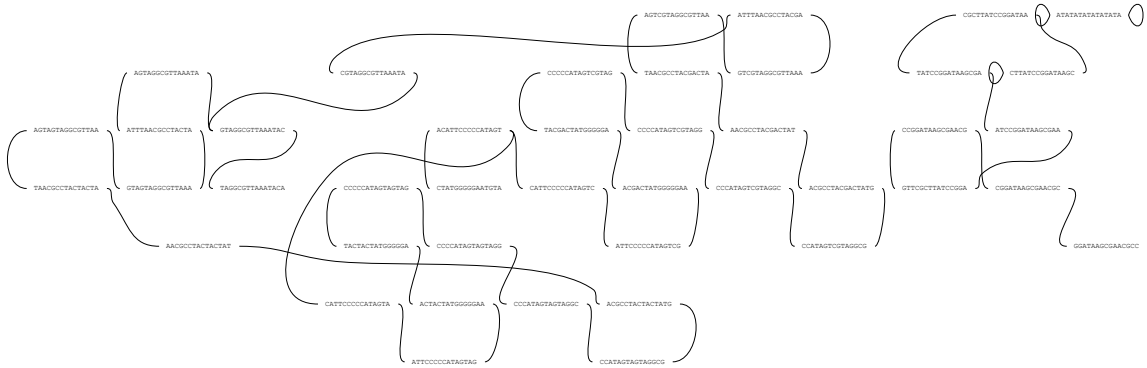


Figure 3.2: An example de Bruijn Graph visualisation created by McCortex at $k = 15$. Edges are represented by black lines leaving one side of a $k$mer and entering another $k$mer. Entering a $k$mer from the left means you read the $k$mer as written and exit from the right side. Entering a $k$mer from the right means you read the reverse complement of the $k$mer and exit from the left.

We provide two methods to simplify and visualise de Bruijn Graphs. To visualise a small region of a larger graph, the `subgraph` command is used to pull out surrounding vertices. For example a $k$mer adjacent to a variant could be used to pull out the subgraph representing the variant in human sample NA12878. We visualise de Bruijn Graphs by generating a text file in the *Dot Language* which we render with the GraphViz tool (Ellson *et al.*, 2003). An example of a $k$mer graph is shown in Figure 3.2.

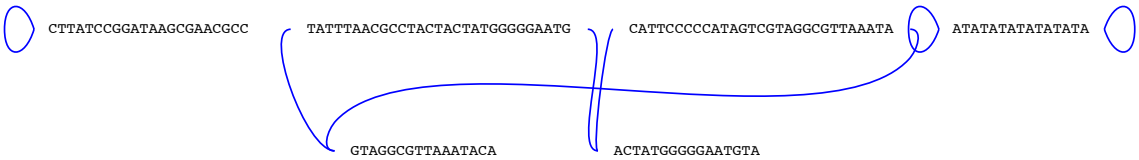


Figure 3.3: An example unitig graph visualisation created by McCortex at $k = 15$. Edges are represented by blue lines leaving one side of a unitig and entering another unitig. Entering a unitig from the left means you read the unitig as written and exit from the right side. Entering a unitig from the right means you read the reverse complement of the unitig and exit from the left.

To plot the graph with a vertex representing each unitig (see Figure 3.3), we use the `unitigs` command and specify that we want output in dot format:

```
./bin/mccortex31 unitigs --dot graph.ctx > graph.unitigs.dot
```

Otherwise we can use a perl script to wrap the `view` command and plot the graph with a vertex representing each $k$mer:

```
./scripts/cortex_to_graphviz.pl graph.ctx > graph.kmers.dot
```

Dot files are plotted as PDFs using Graphviz with `dot -Tpdf in.dot > out.pdf`.

Kiran Garimella (McVean group, personal communication) has also used McCortex as a backend to power an interactive web application to visualise and explore genome graphs. He has been applying this to the assessment of genome assembly quality.

3.3.3 Running McCortex pipeline

To be widely used, software needs to be straightforward to use. Biologists and bioinformatics users may be new to unix, WGS, genome assembly or computing on a cluster. In order to simplify usage, we have written a wrapper script (*scripts/make-pipeline.pl*) to automate running McCortex. It takes as input a file listing the samples and their input sequence files and generates a Makefile.

Makefiles are recipes for generating target files. They have become a staple in software development and in bioinformatics. Each target file lists its prerequisites. If a target file is missing or a prerequisite file has

been updated recently, this triggers Make to regenerate the target file. This means if the job is stopped or stops due to an error, running the same Make command again will restart the pipeline from where it was interrupted.

There are a few advantages to using a Makefile instead of a script that runs all commands directly. Firstly, we can list all commands as they would be run, without actually running. This can be used to list all the commands remaining (*–dry-run*) or list all commands in the pipeline (*–dry-run –always*). Secondly if an error occurs, Make will delete the failed output file. Once the user fixes the issue and re-runs Make, it will resume the pipeline from the failed command. Thirdly, Make can run tasks in parallel (using *-j N*), running up to N jobs at once, as long as their pre-requisites are satisfied. This allows us to run sample graph construction in parallel if we have free memory and unused CPU cores. Make is a common unix tool, supported on many platforms. Reproducing its features would be possible in a script but would likely be error prone.

Below is an example of running the McCortex pipeline for two samples with a reference, going from sequence to variant calls using multiple $k$mer sizes (31 and 51):

```
# Generate the makefile
echo "NA12878 NA12878.bam\nNA12800 NA12800.bam\n" > samples.txt
./scripts/make-pipeline.txt -r hg19.fa 31,51 NA12878_NA12800 samples.txt > job.mk
# Print all the commands to be run
make -f job.mk --dry-run MEM=200G CTXDIR=~/bin/mccortex > job.sh
# Run the job
make -f job.mk MEM=200G CTXDIR=~/bin/mccortex
# Print all the commands
make -f job.mk --dry-run --always MEM=200G CTXDIR=~/bin/mccortex > job.sh
```

The steps in the pipeline for each sample, at each $k$mer-size are:

1. Build each sample graph
2. Clean each sample graph ($k$mer threshold automatically chosen)
3. Thread the reads against the cleaned sample graph to generate single-ended links
4. Clean the single-ended links (link threshold automatically chosen)
5. If paired end reads are available:
   (a) Thread paired end reads against the cleaned sample graph, using cleaned single-ended links to aid assembly across the insert gap
   (b) Clean the paired-end links (link threshold automatically chosen)

Once sample graph and link files have been constructed and cleaned we run bubble calling and breakpoint

calling (see Section 4.3) to find variants (also run at multiple kmer sizes).

1. Run bubble calling
2. Extract and map bubble flanks to the reference using BWA (Li and Durbin, 2009)
3. Align bubble calls to the reference to decompose into a VCF
4. Run breakpoint calling against the reference
5. Align breakpoint calls to the reference to decompose into a VCF

Bubble and breakpoint calls made at various *k*mer-sizes are then merged into a single VCF, which is normalised, compressed and indexed using *bcftools*[6].

The generated Makefile generates output at the *k*mer sizes specified. It works with and without a reference. If a reference is given, we can generate a single VCF for all bubble calls and a single VCF for all breakpoint calls. If a reference isn't available, the script will stop after building each sample graph and generating variant bubble calls.

3.4 HTTP API webserver

We implemented a multithreaded HTTP webserver in Python (`mccortex-server.py`)[7] which runs a persistent McCortex process. Once McCortex has loaded the graphs and link files, the Python script listens on a TCP socket for incoming HTTP requests. It supports three commands: *k*mer look up; selecting a random *k*mer; and providing summary graph and server information. Responses from the web server are in JSON single line format. To request a *k*mer: `curl localhost:2306/GTCCACACCAC`, random *k*mer: `curl localhost:2306/random` and request graph and server information: `curl localhost:2306/info`.

The intent is to provide a standardised interface that other programs can plug into to exploit Linked de Bruijn Graphs without having to implement file formats or deal with efficiently storing large hash tables. By using standard and common HTTP requests with JSON responses, we avoid the need to write wrappers for many different languages. HTTP/JSON are well supported in most languages. The server instantly responds to kmer queries without having to read from disk as the graph is loaded into memory. It is hoped that this will be useful for developing web applications to visualise and explore genome graphs. As it also includes *k*mer counts, it could also be used to genotype variants.

[6] *bcftools*: `https://github.com/htslib/bcftools`
[7] Thank you to Jerome Kelleher for discussions in developing the web server.

Hosting the webserver on a powerful server enables exploration of large genome graphs on many smaller computers across the internet.

3.5 Development

We use *git* version control, which allows us to save snapshots of our work. We host our project on GitHub[8], along with our wiki[9] to document how to use McCortex, how it works and how to submit changes. We review and accept pull requests (suggested changes) from people outside our group.

The robustness and reliability of bioinformatics software is important. McCortex is tested with both unit and integration tests. Components from libraries we have split out are also covered by unit tests.

Continuous Integration is done using the free service for open source programs from Travis CI[10]. Each commit has both unit and integration tests run against it on a linux machine using both *gcc* and *clang* compilers. Compilation is done with most warning flags turned on. Each pull request is also tested. We receive an email alert if tests fail.

Static code analysis is the analysis of source code by tools that find potential issues in the code. Static refers to the fact it is done on the code rather than on the running program – the latter is known as *dynamic analysis*. We regularly run static code analysis with Coverity[11]. Pushing code to the `coverity_scan` branch on GitHub triggers the code to be submitted to Coverity for analysis. We receive an email alert if new defects are detected. Coverity has been used to identify and fix 59 defects in the code, with an additional 20 false positives, in ∼50,000 lines of C code. These have included accessing invalid memory and using variables before they are initialised. Occasionally we manually run *clang*'s static code analyser, which has also caught issues.

Both the status of CoverityScan and Travis CI are shown with icons on our GitHub page. By making continuous testing automated and passive, it is impossible for us to forget to test, or not realize that tests are failing. It also tells us immediately if a proposed change from someone outside the group is causing tests to fail.

All simulation results in this thesis can be regenerated; steps for doing so are in the McCortex repository. To update McCortex, we provide a script which downloads the latest version and updates all third party libraries: `./scripts/update.sh`.

[8] McCortex repository: `https://github.com/mcveanlab/mccortex`
[9] McCortex wiki: `https://github.com/mcveanlab/mccortex/wiki`
[10] Travis CI: `travis-ci.org`
[11] McCortex on CoverityScan `https://scan.coverity.com/projects/2329`

So far ten users (from outside our lab) have provided input on software components developed for McCortex. This has included bug reports, benchmarking, errors spotted in the code, suggestions for features and adding new features themselves. This was without soliciting input – we simply open-sourced the code. Pre-publication, our software had already had many eyes look over it, and we suggest this is preferable to releasing code only when we publish as we can find and resolve more issues before finalising results.

3.5.1 Libraries

Several components of McCortex have been developed as independent libraries.

1. seq_file – buffered sequence file reading with *dnacat*, a tool named after the unix cat command, in C

2. seq-align – Needleman-Wunsch and Smith-Waterman alignment in C

3. bit_array – bit array implementation in C. Bit arrays are used throughout McCortex as condensed storage of a list of boolean values. This library also includes bit level locking – an array of bits is used as an array of mutex locks, each only 1 bit in size. We implemented these using compare-and-swap (CAS) intrinsics which are very fast. On failure to get the lock they yield the processor and attempt again. This is used to lock buckets in the threadsafe hash table. We have not seen this done elsewhere.

4. msg-pool – pass data between threads locklessly. This is similar to a multiple-producer multiple-consumer queue, however input ordering is not conserved. Dropping the constraint on ordering enables us to avoid using locks, which should improve performance.

5. string_buffer – strings for C. This includes buffered line-by-line reading of gzipped files. It also includes fast methods for converting integers into their string representations that is faster than using `sprintf(3)`.

6. sort_r – a cross platform thread safe qsort_r wrapper.

Libraries used in testing and for toy examples:

1. bioinf-perl – a collection of bioinformatics perl scripts. Perl scripts are only used for testing and plotting small results.

2. readsim – simulate reads with error

3.5.2 Third party software

We have attempted to limit our external dependencies to ensure our software works across most computing environments. Our dependencies are a C compiler, zlib and perl for some functions.

Fast non-cryptographic hash functions are used for *k*mer hash table and for hashing the graph walker state to avoid getting stuck walking in a loop:

1. xxHash[12]
2. lookup3[13] by Bob Jenkins
3. CityHash[14] by Google

The hash function used for the hash table can be decided at compile time with `make HASH=CITY`, `XXHASH` or `LOOKUP3`. We do not find that changing the hash function significantly speeds up run time. All hash functions appear to have good performance.

We use the following third-party libraries:

1. htslib – reading and writing SAM, BAM and VCF files
2. BWA (Li and Durbin, 2009) – mapping bubble flanks to generate a VCF (can use e.g. *stampy* (Lunter and Goodson, 2011) instead)
3. bcftools – for VCF normalisation, indexing
4. BFC (Li, 2015a) – for optional error correction before running McCortex

3.5.3 Good software practices

McCortex is a command line tool that often handles large and expensive datasets. The computational resources required to use it are valuable and a limited resource to research groups on a budget, as is the time of the user. Therefore our software follows some simple rules to ensure correct results. These have been influenced in part by recent discussion of best practices (Seemann, 2013).

- We try to operate on the principle of least astonishment – that is the behaviour should be what the user expected: we should not surprise them. One aspect of this is we try to comply with common conventions for unix command line programs. We use the common flags `-h,-help`, `-v,-version` and `-f,-force` as expected.

[12]Yann Collet's xxHash: `https://github.com/Cyan4973/xxHash.git` by Yann Collet
[13]Bob Jenkins' Lookup3: `burtleburtle.net/bob/c/lookup3.c`
[14]Google's CityHash: `https://code.google.com/p/cityhash/`

- A command should always warn before overwriting files. We exit if an output option already exists, and provide the `-f-force` to override this behaviour.
- We validate our inputs, and report an error if arguments are given multiple times as this may indicate an mistake on the user's side.
- We check we can create output files and read input files before running long jobs. This saves CPU and researcher time, for instance by avoiding having a two day long job fail because it tried to save its end results to an incorrectly typed output path.
- When run with no options or wrong options print help. This, along with the `-h,-help` flag, helps users get running with the correct arguments quickly
- Describe in command help which output files will be created. This allows commands to be used in scripts without having a human find the results before writing the next step of a pipeline.

3.6 Discussion

We have implemented a flexible set of modular tools for operations on our Linked de Bruijn Graph. These tools are written in C to be fast, and all support multithreading. In the design we have opted for speed, simplicity and suitability for multithreading over optimising for memory use.

To carry out common tasks we have added a script (McCortex Pipeline) to automate common tasks starting from raw sequence reads – such as variant calling with multiple values of k and merging the results.

Recently several novel low memory approaches for de Bruijn Graph construction and simplification to unitigs have been proposed using Bloom filters, FM-index and other succinct data structures (Boucher *et al.*, 2014, Bowe *et al.*, 2012, Cazaux *et al.*, 2014b, Conway and Bromage, 2011, Deorowicz *et al.*, 2015, Li *et al.*, 2014, Melsted and Pritchard, 2011, Rizk *et al.*, 2013). Since constructing a de Bruijn Graph is the first step of running McCortex these approaches and implementations could be adapted to replace McCortex's native graph construction.

Accumulating link information currently causes peak memory to be very high even for single individuals (350GB for NA12878). Better data structures for link information would likely improve this greatly (e.g. FM-index). The data we are storing is just DNA strings of `<kmer><junction-choices>`, along with a count. They could also be represented in memory as tree-like structures with a root at each $k$mer in the graph. Another option to reduce peak memory usage is to write to disk, sort and then merge repeated

links.

Chapter 4

Applications

We have already shown that the Linked de Bruijn Graph, a data structure equivalent to the String Graph, can be used to compress and store read data in a graph structure from which we can construct contigs through graph traversal. In this chapter we develop algorithms that use our data structure. They are: genome assembly, two approaches to variant calling, variant genotyping and read error correction. Although they are designed to exploit links, these algorithms work with or without links on a de Bruijn Graph. Comparisons are made to Cortex (Iqbal *et al.*, 2012), McCortex's predecessor from the same research group.

4.1 Simulations in this chapter

Our simulations use 1Mbp from human reference GRCh37 (hg19; chr22:17,000,000-17,999,999) as our reference. Chromosome 22 was selected as it is the smallest autosome to download individually. 17,000,000-17,999,999 is the first megabase without any 'N' bases. In the next chapter we apply these algorithms to real data.

To simulation mutations (in diploid and multi-sample simulation), we wrote a simple perl script[1] to sample random positions to insert SNPs and indels. SNPs alternative base is chosen with equal probability over the three other bases. Variants were not placed overlapping or within 1 bp of a previous mutation. Mutations were randomly applied to individuals, with individual $i \in 1..N$ receiving the mutation with probability $\frac{1}{N}$ (individuals shuffled for each variant). Indels were simulated by applying only deletions and inverting the probabilities in half of cases, creating equal expectation of insertions and deletions.

[1] Mutation simulation: `https://github.com/noporpoise/bioinf-perl/blob/master/sim_mutations/sim_mutations.pl`

Deletion size was sampled from geometric distribution with mean of 2 bases.

4.2 Genome Assembly with a linked de Bruijn Graph

Exhaustive contig construction from a Linked de Bruijn Graph starts by iterating over $k$mers and using them to 'seed' a contig. A $k$mer is marked once it is used as a seed or in a contig. Additionally, if we traverse a link to its end we mark it as used. Only unmarked $k$mers are used to seed contigs. Once all $k$mers have been marked, links that are not part of any contig are used to seed contigs. This is done by iterating over all $k$mers and then iterating over their links. By sorting links and removing redundant ones before we use them, we reduce the number of redundant contigs (contigs that are substrings of other contigs). This algorithm is shown in Algorithm 2. The commands required to construct a Linked de Bruijn Graph are shown in Listing 4.1 and those required to assemble a sample are shown in Listing 4.2. The commands to use McCortex Pipeline to generate an assembly are shown in Listing 4.3.

Listing 4.1: FASTQ to a cleaned graph and links with $k = 31$

```
# Construct de Bruijn Graph
mccortex31 build -m 80G -k 31 --sample NA12878 --seq NA12878.fq graph.raw.ctx
# Clean graph kmers
mccortex31 clean -m 80G -o graph.ctx graph.raw.ctx
# Generate links
mccortex31 thread -m 250G -o graph.raw.ctp.gz --seq NA12878.fq graph.ctx
# Pick link cleaning threshold, and use it to clean
mccortex31 links -T 0.001 -L 1000 graph.raw.ctp.gz 2> links.log
LINK_THRESH=`grep 'suggested_cutoff=' links.log | grep -oE '[0-9,]+$'`
mccortex31 links --clean $LINK_THRESH --out graph.ctp.gz graph.raw.ctp.gz
```

Listing 4.2: Generating an assembly from a Linked de Bruijn Graph. Scaffolding is not included. Assumes steps in Listing 4.1 have already been run.

```
# Assemble contigs
mccortex31 contigs -m 80G -o contigs.fa -p graph.ctp.gz graph.ctx
# Remove redundant contigs
mccortex31 rmsubstr -m 20G -o contigs.rmdup.fa contigs.fa
```

Listing 4.3: Steps to go from FASTQ to assembled contigs using the McCortex pipeline.

```
echo "NA12878 NA12878.bam seq.1.fq.gz:seq.2.fq.gz" > samples.txt
scripts/make-pipeline.pl 31 NA12878 samples.txt > job.mk
make -f job.mk CTXDIR=~/bin/mccortex MEM=250G NTHREADS=2 contigs
```

Algorithm 2 Algorithm to assemble contigs in a given colour. The population is used to fill in low coverage regions.

```
1: procedure AssembleFromKmer(graph,colour,kmer)
                          ▷ While walking, if we reach the end of a link we are using, Mark the link
2:     A ← Walk from kmer backwards
3:     B ← Walk from kmer forwards
4:     contig ← reverse(A) + B                    ▷ contig is a list of k-mer-keys with orientations
5:     return contig

6: procedure AssembleFromLink(graph,colour,kmer,link)
                          ▷ While walking, if we reach the end of a link we are using, Mark the link
7:     B ← Walk from kmer along link
8:     A ← Extend B backwards
9:     C ← Extend B forwards
10:    contig ← reverse(A) + B + C                ▷ contig is a list of k-mer-keys with orientations
11:    return contig

12: procedure AssembleContigsFromKmers(graph,colour)
13:    for kmer in kmers(graph) do
14:        if kmer is not marked then
15:            contig ← Assemble contig both directions from kmer
16:            Emit contig
17:            for k in kmers(contig) do
18:                Mark k

19: procedure AssembleContigsFromLinks(graph,colour)
20:    for kmer in kmers(graph) do
21:        links ← links starting at kmer
22:        links ← sort links longest to shortest
23:        for link in links do
24:            if link is not marked then
25:                AssembleContigFromLink(graph,colour,kmer,link)

26: procedure AssembleContigs(graph,colour)
                                   ▷ As contigs are emitted they are written to the disk
27:    AssembleContigsFromKmers(graph,colour)
                                                   ▷ By now all k-mers are marked
                                   ▷ Some links may not have been used
28:    AssembleContigsFromLinks(graph,colour)
29:    contigs ← load all contigs from disk
30:    contigs ← remove substrings from contigs
31:    return contigs
```

4.2.1 Multithreading

McCortex assembly is multithreaded – the command to extract contigs splits up the hash table of $k$mers into t equal segments, where t is the number of threads. Each thread gets a subset of the hash table and therefore $k$mers to use as contig *seeds*. For assembly the graph is read-only, however each $k$mer and each link has a single bit that is set if it has been used in a contig. These are shared across threads and are updated synchronously using atomic bitwise-OR operations. Each thread uses its $k$mers to seed contigs unless they have already been used in a contig (by any thread). This approach scales well with the number of threads used since the mammalian genome is large; contigs are comparatively short and so the probability of two threads trying to assemble the same contigs is low.

Since assembly is seeded from unused $k$mers in a random order, some contigs that are assembled are substrings of larger contigs. This is caused by seeding a contig in a repeat, then seeding with a $k$mer outside the repeat to assemble a contig that spans the repeat. To deal with these substrings we provide a command `mccortex31 rmsubstr` which reads in a FASTA file and prints out only contigs that are not substrings of other contigs.

4.2.2 Assembling diploid genomes

Issues remain with assembling a haploid sequence from a diploid or polyploid genome, as discussed in Section 1.5.3. Haploid assemblies lose information about the genome being studied, such as chromosome copy number and allele phasing. However they are easier to use for read mapping, fixing a coordinate system and identifying genes, amongst other things. In order to construct a haploid genome assembly from diploid data it is necessary to make choices about which allele to take at heterozygous sites. This is often done in a step known as 'bubble-popping' where bubble structures are identified in the graph and collapsed down to one of the alleles. By doing this independently per site, phasing information across sites is lost. Each final assembled chromosome does not represent either sequenced chromosome but instead a mix of each pair of autosomal chromosomes. Additionally, some sequences can be incorrectly collapsed down to match similar (paralogous) sequences in the genome.

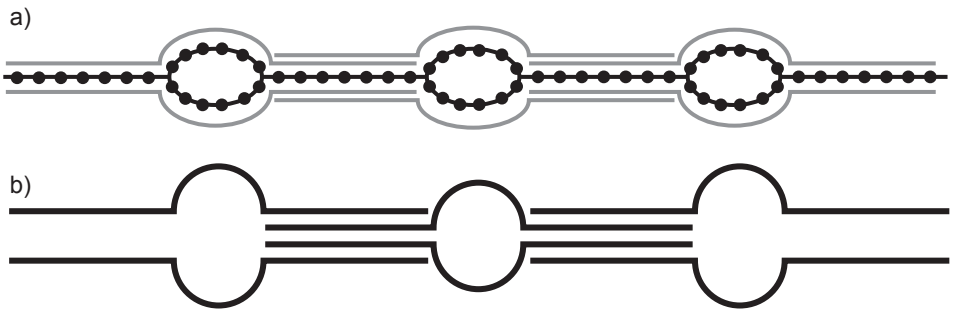


Figure 4.1: Diploid genome assembly with unpopped bubbles. (a) A de Bruijn Graph with $k$mers shown as dots with solid black edge between them. Contigs that can be assembled from the graph are overlaid as grey lines. (b) Showing only the contigs from (a) without the de Bruijn Graph. The effect of remaining graph bubbles at assembly time is nearly identical/redundant contigs and contigs with large overlaps.

To adapt McCortex to assemble haploid sequences from diploid genomes there are three possible approaches. Firstly, the graph cleaning step could include a bubble popping step. This would involve identifying bubbles in the graph where coverage is roughly half expected coverage on each allele. An allele would be chosen at random and removed. In the later read threading step, reads would be automatically corrected against the graph to use the remaining allele.

A second approach is to assemble contigs as normal and add a post-processing step to identify contigs that represent an allelic pair. This can be done by aligning the contigs or mapping them to one another. The affect of assembling contigs in this manner is illustrated in Figure 4.1. The advantage of such an approach is that the assembly represents the diploid individual and retains phasing information. However in a species such as humans where the expected gap between heterozygous sites is longer than the sequencing fragment lengths, the haplotype contigs are short and the resulting assembly is fragmented.

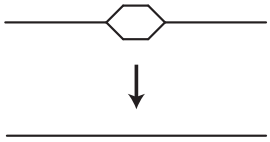
Lastly an approach similar to the PacBio assembler *FALCON* (Chin *et al.*, 2016) could be used to assemble allelic haploid contigs in tandem (using two graph walkers together). The result is a diploid a-cyclic assembly graph which can be collapsed down to a haploid assembly later.

We have chosen to implement the bubble popping approach, although other approaches would also be viable with a Linked de Bruijn Graph.

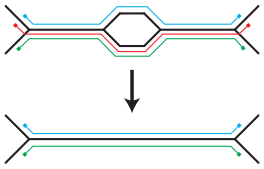
4.2.3 Bubble popping

Bubble popping reduces graph complexity. This is useful for assembly when the bubble is caused by a polymorphism or sequencing error on a unique background (sequence that occurs only once in the genome). However bubble popping in repeats makes similar repeats identical in the graph. This has a negative effect on assembly in the presence of links as illustrated in Figure 4.2. Popping small bubbles

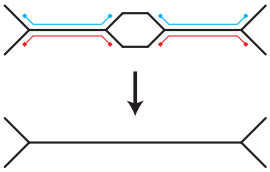
that represent real sequence increases the distance between forks in the graph, reducing the physical coverage crossing the collapsed repeats and therefore reducing the number of links.



(a) Popping bubbles caused by sequencing error or small scale variation (SNPs, small indels). Bubble popping improves assembly when the bubbles are isolated between two long unitigs.



(b) Bubbles caused by sequencing error would have halted traversal since the links diverge. Popping the bubble collapses the red and blue links together.



(c) Popping the small bubble increases the distance between junctions, reducing the number of links that span the region to zero. This makes the region un-traversable.

Figure 4.2: Illustration of a case where popping small bubbles improves assembly ((a) and (b)) and where bubble popping reduces our ability to assemble (c). Bi-directional pairs of links are shown as coloured lines with squares indicating the $k$mers to which each link is attached. Some links are omitted for clarity.

Bubble popping for assembly could be improved by only removing alleles whose coverage indicated it was due to sequencing error or where the background (flanking) sequence's coverage indicated copy number equal to the sample ploidy (i.e. we are not in a repeat). Alternatively bubbles could be popped only if they are flanked by long unitigs.

Since links are compressed using the graph, the graph structure cannot be changed after links have been added. If links were already on the graph we can make more informed decisions about which bubbles to pop to improve contig assembly distance.

4.2.4 Haploid simulation

To assess the ability of McCortex to assemble a haploid genome, we ran a haploid assembly simulation using reads sampled from 1 Mbp of the human genome[2]. We then compared the final assembly to the original genome section. We used 1Mbp of GRCh37 reference assembly chromosome 22 (hg19; chr22:17,000,000-17,999,999) as the source haploid genome. From this haploid reference 100bp paired-end reads with 200bp insert gap were sampled with 100X coverage and with 0.5% per base sequencing error. Reads were error corrected with *bfc* (with parameter $k = 33$). The data were assembled as single

[2]Code to reproduce this simulation can be found in the McCortex repository at `results/kmer_size_experiment/`

and paired-end using McCortex with $k = 21, 31, \ldots, 81, 91$. We also assembled the single end reads with SGA (Simpson and Durbin, 2012) (SGA does not use paired reads in the assembly process, only in the scaffolding step). SGA commands were adapted from the *E. coli* example given in SGA[3].

Assemblies were assessed with a script we wrote [4]. For each query string (assembled contig), we report all maximal substring alignments between it and the source genome (true assembly). Alignments are of length $\geq k$ and can include single base mismatches as long as they are flanked by k matching bases. From these alignments broken at assembly errors, we calculate an assembly NG50 and number of assembly errors.

We attempted to also run Velvet (Zerbino and Birney, 2008), but after breaking contigs on each mismatch with the original genome and removing resulting substring contigs, the output contigs had a sum length less than half the size of the input genome – ranging from 161kbp to 448kbp for $k = 21, 31, \ldots, 91$. Velvet reports many N bases in its contigs, giving long contigs with regions of uncertainty. This may be one reason that our rather strict NG50 metric calculation penalised Velvet. Since Velvet didn't manage to assemble half the genome once broken at errors greater than a single base, we couldn't generate an NG50 to give a meaningful comparison to McCortex and SGA. Overall the Velvet assembly error rate at all k values was higher than any seen in McCortex.

[3] `src/examples/sga-ecoli-miseq.sh` in the SGA repository
[4] `scripts/python/break-contigs-vs-truth.py` in the McCortex repository

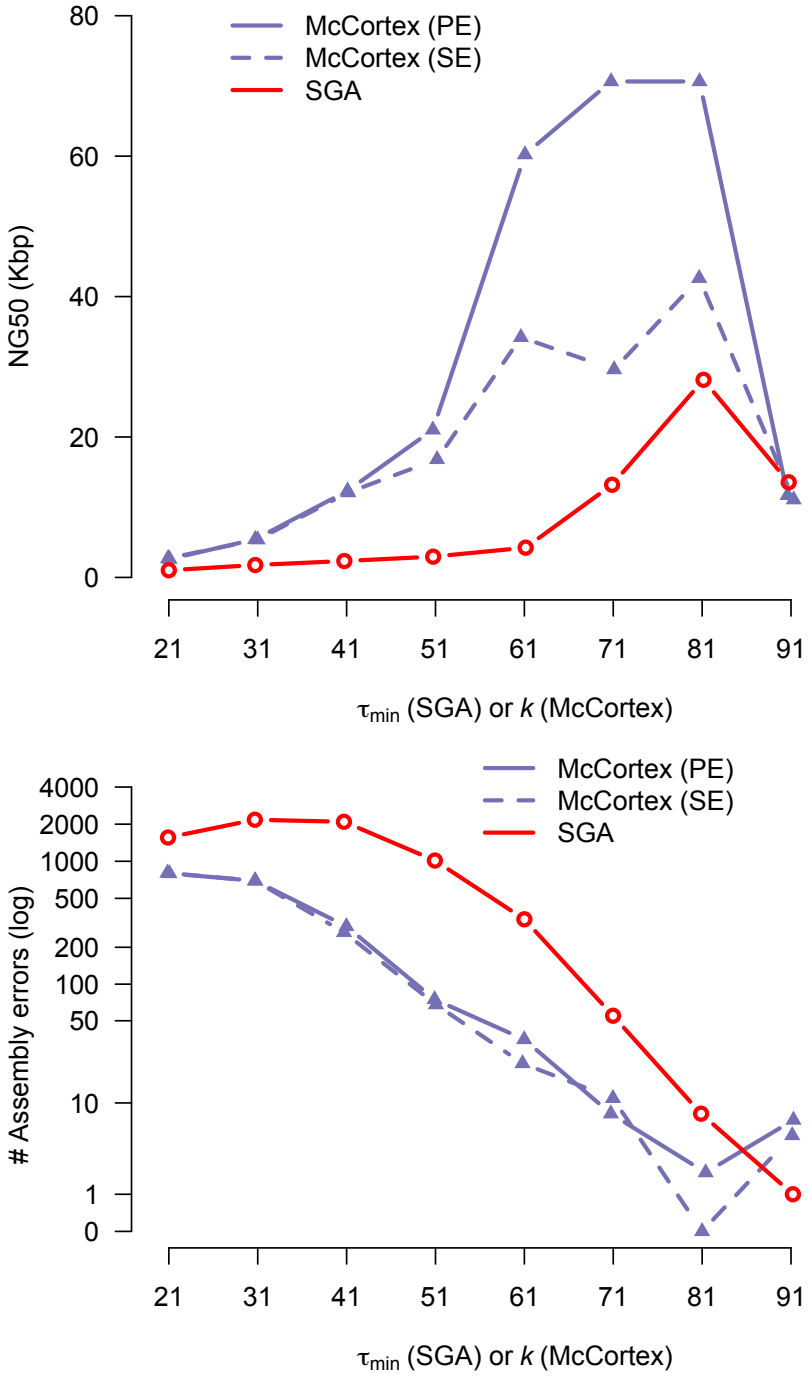


Figure 4.3: Assembling simulated reads from a haploid genome with varied $k$mer-size. Source genome is haploid 1Mbp from GRCh37 chr22:17,000,000-17,999,999. Reads are 100X coverage with 100bp paired-end from a 400bp fragment size and 0.5% sequence base error. Reads were error corrected with *bfc* first (with parameter $k = 33$). McCortex is shown in purple – solid line is using paired-end reads (PE), dashed line is using single reads only (SE). SGA results are shown in red.

Results from SGA (using single reads) and McCortex (single reads and paired-end reads) are shown in Figure 4.3. McCortex outperforms SGA with single-end reads on both NG50 and the number of assembly

errors. In McCortex, both metrics improve as k increases, suggesting a value of k greater than half the read length performs best for de Bruijn Graph assembly, as previously discussed (see Section 1.4). Paired end reads seem to provide a large improvement to NG50 over this limit ($k > 51$) without showing much effect on the error rate. NG50 performance drops off as k approaches the read length and as $k$mer coverage drops off.

McCortex and SGA results without performing *bfc* error correction first are shown in Appendix B. Without error cleaning McCortex's NG50 is degraded at high values of k (underperforms SGA at $k = 81, 91$) and the number of assembly errors at lower k is greater than SGA (underperforms SGA at $k = 21$).

In Turner *et al.* (2018), we showed the ability of McCortex to assemble haploid genomes by using the E.coli MG1655 dataset provided on the Illumina website (single-end 150bp reads giving 365X coverage of the 4.6 Mbp genome)[5]. We compared McCortex, SPAdes, SGA and Velvet. We also included an iterative McCortex assembly approach whereby contigs from McCortex were trimmed by 50bp and used as input into McCortex a second time – the rationale is that errors usually halt assembly and occur at the end of contigs (Simpson and Durbin, 2012). QUAST's NG50 metric (Gurevich *et al.*, 2013) was used to assess performance. Across all k values, iterative McCortex had better NG50 values than Velvet or SGA and was almost comparable to SPAdes.

4.2.5 Diploid simulation

We generated a diploid genome using 1Mbp of GRCh37 reference assembly chromosome 22 (hg19; chr22:17,000,000-17,999,999) and adding 1,000 SNPs, 100 small deletions (56 from one chromosome, 44 from the other), using the mutation simulation script described in Section 4.1. Reads were sampled with 60X coverage (30X per chromosome), 1% sequencing error rate and 150bp paired-end reads with mean insert gap of 250bp and $\sigma = 50$. We then carried out five assembly approaches at $k = 31$. Firstly we just extracted unitigs from the de Bruijn Graph. Secondly, we assembled contigs (without links) before and after popping bubbles. Lastly, we used the Linked De Bruijn Graph to exploit read and paired-end information, before and after popping bubbles in the graph.

[5]Analysis was run by Kiran V. Garimella

		Plain		Links	
	Unitigs	Contigs	Contigs (popped)	Contigs	Contigs (popped)
Number of contigs:	11,927	7,081	5,638	4,102	2,250
Number of validated contigs:	10,768	7,164	5,691	4,711	2,872
Assembly errors:	0	118	88	644	657
Validated contigs NG50:	694	1,538	1,577	2,099	3,899

Table 4.1: McCortex diploid genome assembly simulation at $k = 31$, using 1Mbp of GRCh37 reference assembly chromosome 22 (hg19; chr22:17,000,000-17,999,999). A diploid genome was created with 1,000 SNPs and 100 deletions (56 from one chromosome, 44 from the other). Reads were sampled with 60X coverage (30X per chromosome), 1% sequencing error rate, 150bp paired-end reads with mean insert gap of 250bp, $\sigma = 50$. Simulation script is in `results/assembly/` of McCortex repository. Plain/Links indicates whether links were used in the assembly step. '(popped)' indicates that bubbles with branches shorter than 40 *k*mers were popped. Validated contigs are those that align to the truth, have been split at assembly errors and have redundant contigs removed.

As we know the true assembly, we mapped the contigs back to the truth to assess the rate of misassembly of these five approaches. Contig metrics are shown in Table 4.1. Popping bubbles gave a mild improvement in NG50 and can reduce the number of assembly errors. Assembling with links gave a large improvemnt over extracting unitigs or assembling without links, at the cost of an increase in the number of assembly errors. Popping bubbles improved NG50 and reduced the number of assembly errors.

	Unitigs	Plain	Links
Number of contigs:	8,639	5,387	2,852
Number of validated contigs:	8,601	5,356	3,317
Assembly errors:	0	0	496
Validated contigs NG50:	1,627	1,633	4,418

Table 4.2: McCortex haploid genome assembly (one haplotype from Table 4.1) with 60X coverage, 1% sequencing error rate, 150bp paired-end reads with mean insert gap of 250bp, $\sigma = 50$. 'Unitigs' is extracting all unitigs from the graph. 'Plain' is traversal of a traditional de Bruijn Graph. 'Links' is contig assembly using a Linked de Bruijn Graph. Bubble popping was not carried out. These assembly statistics can be compared to the corresponding diploid assembly simulation with the chromosome region in Table 4.1

To evaluate the effect of the diploid genome, we generated double coverage of one of the two haplotypes used and ran three haploid assembly approaches: unitigs, de Bruijn assembly without links (*plain*) and Linked de Bruijn Graph assembly. The results are shown in Table 4.2. Comparing to the diploid assembly with links (Table 4.1), our results show that diploid assembly with bubble popping approaches the performance of haploid assembly (last column of Table 4.1 vs last column of Table 4.2).

4.3 Variant calling

As discussed in Chapter 1, the traditional approach to variant calling has been to map reads to a reference and use the read pileup to identify variant sites. This has proved fast and effective at identifying SNPs in human samples where a high quality reference has been produced, germline samples are closely related to the reference (1 in 1000 bases differ on average) and the majority of mutational events are SNPs.

Even so, mapping reads does a poor job of identifying indels, resolving translocation at single base level and identifying novel sequence in an individual. Mapping paired-end reads helps identify the presence of larger events, but cannot on its own reassemble the event. Short read length gives an upper bound on the length of the events that we can resolve.

By relying on a reference to identify variants, mapping based approaches introduce *reference bias*. A example of this is the slightly higher sensitivity to deletions versus insertions. Reads from regions not represented in the reference will be forced to align elsewhere resulting in high rates of polymorphism or will be dropped altogether. In cases where a high quality reference is not available, samples are highly diverged from the reference, or if large mutation events are of interest, mapping to a reference to do variant calling results in missed mutations and biased results. For more details, see Section 1.6.

4.3.1 Graph based calling

Several-reference free and graph-based variant calling methods have been proposed (Leggett and MacLean, 2014). Some of these methods are limited to small microbe genomes, such as Pilon (Walker *et al.*, 2014) and SplitMEM (Marcus *et al.*, 2014). Microbes can have high divergence from their reference genome, often have clinical significance and can have a high number of indels and complex rearrangements. These factors make a tailored variant calling pipeline desirable.

Local re-assembly (such as GATK (DePristo *et al.*, 2011) and Platypus (Rimmer *et al.*, 2014)) have much lower memory requirements than whole genome re-assembly but have improved calling for small indels.

Assembling a sequence graph (either string or de Bruijn) of a population creates substructures in the graph around variants called *bubbles* – where a path splits and rejoins. Identifying these subgraphs provides a *de novo* method for finding polymorphic sites within an input population. Work on finding these includes detecting superbubbles in assembly graphs (Onodera *et al.*, 2013) and methods for finding bubbles quickly: 2k+2 bubble searches (Younsi and MacLean, 2015). These ideas are implemented in Cortex (Iqbal *et al.*, 2012) and DiscoVar (Weisenfeld *et al.*, 2014) which provide variant calling for

mammalian genomes as well as smaller genomes.

Superbubbles (in a directed graph (Onodera *et al.*, 2013)) and generalised to ultrabubbles (in bidirected graphs (Paten *et al.*, 2018)) are acyclic subgraphs that connect to the rest of the graph through a source and sink vertices. All vertices in the subgraph can be reached from either the source or sink vertex. Superbubbles segregate into strictly nested or disjoint subgraphs (a superbubble is fully contained by another *or* they do not share any vertices). This naturally defines a nesting of putative variants.

Initially, assembling a genome for variant calling required large amounts of RAM. More recently, low memory *de novo* variant calling methods have been developed, such as MindTheGap (Rizk *et al.*, 2014) which called variants in NA12878 with only 14GB of memory. The GATB toolbox (Drezen *et al.*, 2014) provides a library for low memory genome assembly and analysis.

4.4 Breakpoint Caller

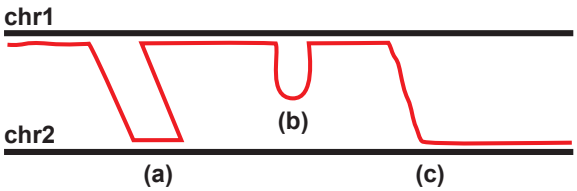


Figure 4.4: Three events that can be discovered with the breakpoint caller. The *trusted sequence* (reference) is in black (made up of two chromosomes *chr1* and *chr2*) and the sample is in red. (a) A translocation. If the translocation is shorter than the minimum required flank length it will be reported as one event. Otherwise will be reported as two separate events as in (c). We also find translocations within the same chromosome. (b) SNP, insertion or deletion, reported as a single event. (c) Non-allelic homologous recombination (NAHR) would give us a single breakpoint with flanks on different chromosomes. NAHR events on the same chromosome would be discovered in the same way.

The breakpoint caller allows us to compare a Linked de Bruijn Graph to a trusted sequence (e.g. a reference sequence or previously assembled genome). The output is a set of *breakpoint calls* – putative variants between the sample and trusted sequence. Each breakpoint call consists of a flank where our samples follow and then break from the reference, some sequence not in the reference, and then a second flank where the samples rejoin the reference. This can be used to identify small and large events as well as rearrangements as illustrated in Figure 4.4.

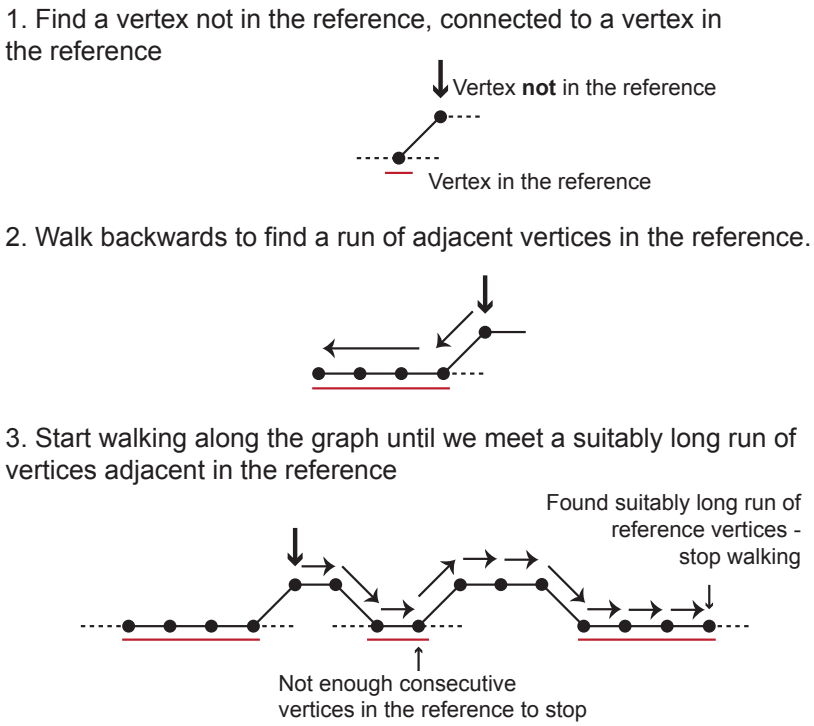


Figure 4.5: Breakpoint Caller algorithm showing large event calling in a single individual. Black vertices represent $k$mers in the Linked de Bruijn Graph, edges connect the $k$mer vertices. Dashed lines signify connections to the rest of the graph. Red underlined $k$mers are those found in the reference. For multi-coloured graphs, the process is repeated per sample. By calling in all samples at once we are able to make some optimisations since many samples are likely to walk the same graph regions.

The breakpoint caller first loads a Linked de Bruijn Graph, then reads in a 'trusted sequence'. $k$mers in the graph are labelled with the position(s) that they occur at in the reference genome. We then search for a $k$mer that is not in the reference ($k$mer A) but that has an edge to a $k$mer that is ($k$mer B). We traverse into the reference ($A \rightarrow B$) to establish a flank of at least N $k$mers in the reference (where N is a user specified value). Then we traverse $B \rightarrow A$, out of the reference until we hit the reference again or exceed a user specified maximum traversal distance. We do not consider that we have rejoined the reference until we have seen N consecutive $k$mers in the reference. An illustration is shown in Figure 4.5 and the pseudocode is listed in Algorithm 3.

Algorithm 3 Algorithm to find large events by comparing sample local assemblies to a reference

```
1: procedure KMERS(Graph)                                   ▷ Returns the set of k-mers in the graph

2: procedure ANNOTATEREFGRAPH(Graph,ReferenceSeq)
3:     RefPos{kmer} ← ∅; ∀kmer ∈ KMERS(Graph)               ▷ Map from kmer → List of ref positions
4:     for i=0; i+k ≤ length(ReferenceSeq); i++ do
5:         kmer ← ReferenceSeq[i : i + k − 1]               ▷ Substring from i to i + k − 1 inclusive
6:         if kmer ∈ KMERS(Graph) then
7:             RefPos{kmer} ← RefPos{kmer} ∪ i
8:     return RefPos

9: procedure FETCHREFRUNS(Graph,RefPos,minFlank,colour,vertex0,vertex1)
                                                            ▷ Find set of ref runs that samples have
                                  ▷ vertex0 is used to start walking but is not included in contig
10:    contigs ← {vertex1}
11:    prev ← vertex1
12:    next ← Walk in colour from vertex0...vertex1
13:    while ADJACENTINREF(RefPos,prev,next) do
14:        contig ← contig ⋃ next
15:        prev ← next
16:        next ← Walk from next
17:    if |contig| ≥ minFlank then
18:        contigs ← contigs ⋃ contig
19:    return contigs

20: procedure CALLBRKPTSFROM(Graph,RefPos,minFlank,maxAllele,colour,vertex)
21:    for next ∈ NEXTVERTICES(Graph, vertex, colour) do
22:        if RefPos{next} = ∅ then                         ▷ next is not in reference
23:            flanks5p ←FetchRefRuns(
   [7] Graph, RefPos, minFlank, colour,
   [7] REVERSE(next), REVERSE(vertex))
24:            for flank in flanks5p do
25:                contig ← REVERSE(flank) ⋃ next
26:                while (vertex ← Walk) and |contig| − |flank| < maxAllele do
27:                    contig ← contig ⋃ vertex
28:                    if ADJACENTINREF(RefPos,contig[|contig| − minFlank...|contig|]) then
29:                        Emit contig
30:                        break

31: procedure CALLBREAKPOINTS(Graph,ReferenceSeq,minFlank,maxAllele,colour)
32:    RefPos ← ANNOTATEREFGRAPH(Graph, ReferenceSeq)
33:    for kmer ∈ KMERS(Graph) do
34:        CALLBRKPTSFROM(Graph, RefPos, ⟨kmer, FORWARD⟩, minFlank, maxAllele)
35:        CALLBRKPTSFROM(Graph, RefPos, ⟨kmer, REVERSE⟩, minFlank, maxAllele)
```

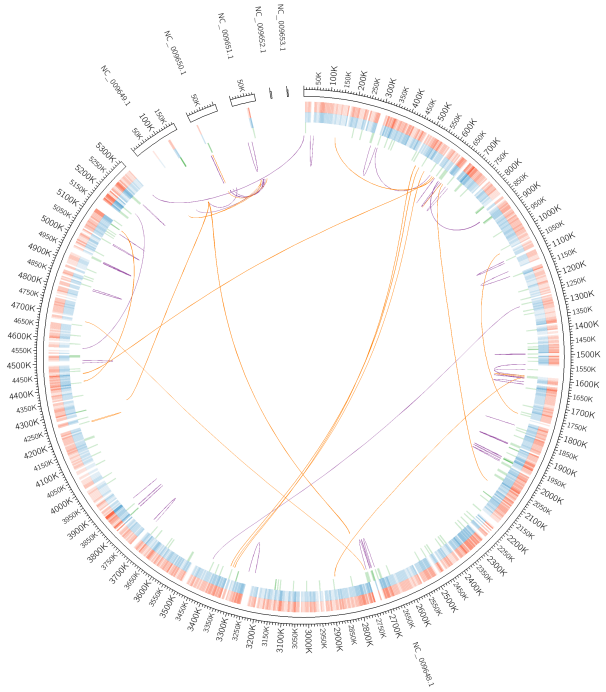


Figure 4.6: An example Circos plot generated by McCortex from Breakpoint calls on *Klebsiella pneumoniae*. Variant calling in *Klebsiella pneumoniae* is discussed in detail in Section 5.2. Genome coordinates are written on the outside. The first three tracks show events in 5kbp bins. SNPs/MNPs are shown in red. Small insertions and deletions (<200bp) are shown in blue and green respectively. Purple lines indicate breakpoint causing larger indel events. Orange lines indicate breakpoints where the opposite strands join in the sample, which indicates an inverted section was moved or copied.

Similar to the bubble caller (described in Section 4.5), after calling we decompose small events into a VCF file (Danecek *et al.*, 2011). Small events must start and end on the same chromosome and the alleles must be of similar lengths (user specified parameter). Since breakpoint calls are already placed on a genome, we do not need to map the flanks. Instead we take the longest match to the reference. Flanks can be mapped if the trusted sequence is different from the reference we are trying to place the variants on. Potential variants, that satisfy both the longest allele threshold and largest absolute allele length difference threshold, are aligned to the reference allele and a VCF is generated. We have also written a script to convert breakpoint output to a Circos plot (Krzywinski *et al.*, 2009) (see an example in Figure 4.6). McCortex steps required to run the breakpoint caller and generate a Circos plot are shown in Listing 4.4. The steps to make breakpoint calls using the McCortex Pipeline are shown in Listing 4.5.

Listing 4.4: Steps to generate breakpoint calls then produce a VCF. Breakpoint Circos created in `my-circos-plot/circos.png`. Assumes steps in Listing 4.1 have already been run.

```
mccortex31 breakpoints -m 20G -o breakpoints.txt.gz -p graph.ctp graph.ctx
mccortex31 calls2vcf -o breakpoints.vcf breakpoints.txt.gz hg19.fa
scripts/make-circos.pl my-circos-plot breakpoints.txt.gz
```

Listing 4.5: Calling breakpoints using McCortex pipeline and *k*mers 31, 51.

```
echo "NA12878 NA12878.fq" > samples.txt
scripts/make-pipeline.pl 31,51 NA12878 samples.txt > job.mk
make -f job.mk CTXDIR=~/bin/mccortex MEM=250G NTHREADS=2 \
               USE_LINKS=yes breakpoints-vcf
# result is now in NA12878/vcfs/breakpoints.joint.k31.k51.vcf.gz
```

4.4.1 Simulation

In order to assess the ability of the Breakpoint caller to detect gross rearrangements, we simulated several rearrangement events in a haploid and then a diploid genome. For this simulation we used 1Mbp of human chromosome 22 (chr22:17,000,000-17,999,999) as the 'input genome'.

For the haploid simulation, 99 breakpoints were sampled uniformly across the input genome. This created 100 genome segments which were shuffled into a random order and each one was then reverse complemented with 50% probability. All segments were then concatenated. We sampled 150bp paired-end reads with 30X depth and sequencing error rate of 1% from this haploid genome. We ran the McCortex breakpoint pipeline on these haploid data.

For the diploid simulation, we sampled another 30X paired-end reads from the input genome before it was broken up and shuffled. These data were combined with the reads from the haploid simulation. We then re-ran the breakpoint pipeline. This simulation tests the ability of McCortex to distinguish breakpoints in the presence of a diploid genome. This is important in cancer genomes where one copy of a chromosome may have undergone gross rearrangements. As we don't use ploidy in the breakpoint algorithm, it should be robust when dealing with polyploid data so long as the *k*mer and link cleaning steps are still effective.

	Haploid	Diploid
Breakpoint calls:	187	187
Breakpoints detected:	88 / 99 (88.89%)	88 / 99 (88.89%)
Calls per breakpoint:	2.00	2.00
False positives:	11	11

Table 4.3: Haploid and diploid breakpoint simulations using 1Mbp of GRCh37 reference assembly chromosome 22 (hg19; chr22:17,000,000-17,999,999). The haploid chromosome was generated by segmenting and shuffling the reference. The diploid genome was generated by combining both the shuffled chromosome and the original reference chromosome. Reads were sampled with 30X haploid sequencing depth, 1% sequencing error rate, 150bp paired-end reads with mean insert gap of 250bp, $\sigma = 50$. The number of breakpoints recovered in each simulation are shown. Simulation script is in `results/breakpoint_calling/` of the McCortex repository.

Haploid and diploid results are shown in Table 4.3. Adding a autosomal-pair chromosome has no effect on the sensitivity or false positive rate of detection (Table 4.3). We detect ~90% of breakpoint events.

Events we do not detect could be due to breakpoints creating adjacent *k*mers in the sample graph from non-adjacent reference positions. In this case the sample does not appear to diverge from the reference – this is the marker we use to start looking for a divergence and rejoining to the reference. As well as searching from graph forks where a sample leaves the reference, we could also start the search along the reference for sites where the sample joins the reference at a new position. However this may increase the false positive rate significantly. Tracing a sample through a repeat would appear as if the sample had just joined the reference at hundreds of repeat positions.

4.4.2 Potential improvements

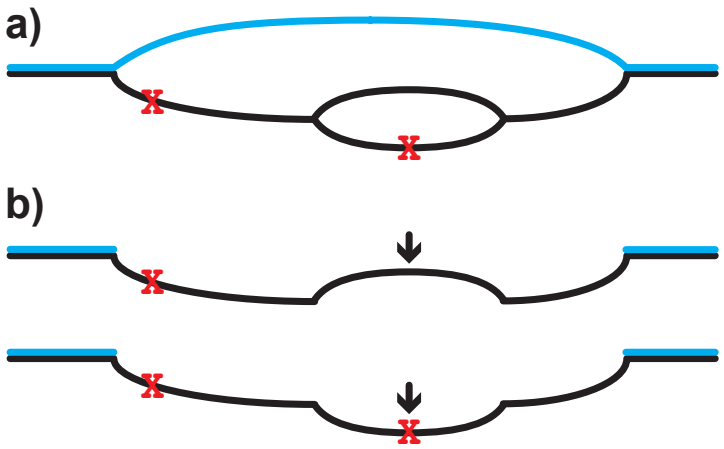


Figure 4.7: (a) An example de Bruijn Graph with a heterozygous sample (black) that differs from the reference (blue). An improved breakpoint caller would be needed to recover these polymorphisms. Red X indicates mutations that we want to recover. (b) Arrows indicate *k*mers private to the sample that could be used to seed the breakpoint assembly. Paths through the graph back to the reference could be found either by traversal with links or depth-first-search (DFS). For each of the possible seed *k*mers the resulting alleles ares shown.

The breakpoint caller could be improved by exploiting unused sample or non-reference *k*mers to make additional breakpoint calls. These novel *k*mers would help us recover heterozygous sites where neither allele path touches the reference path through the graph[6]. This is illustrated in Figure 4.7. The paths from both sides of the novel seed *k*mer to the reference could either be found by depth-first search (DFS) through the graph or else we perform graph traversal using links. DFS would require a maximum number of paths to attempt to follow before giving up (the number of paths would explode quickly in low complexity areas). Graph traversal with links would fail as soon as ambiguous regions are reached – such as other heterozygous sites.

[6]These ideas came about through discussion with Kiran Garimella.

4.5 Bubble Caller

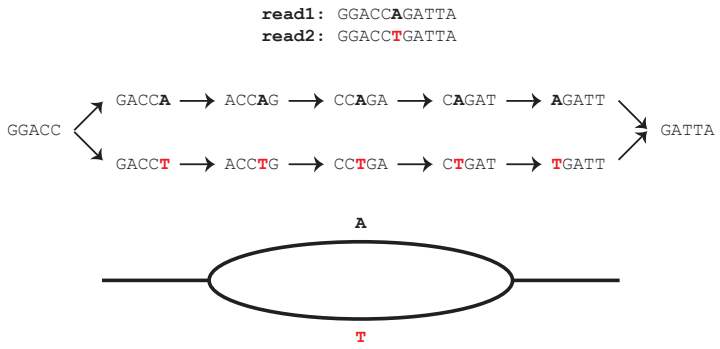


Figure 4.8: SNPs create 'bubble' structures in de Bruijn Graphs. *Top:* two reads that contain an `A/T` SNP (highlighted). *Middle:* $k = 5$ de Bruijn Graph constructed from the reads. *Bottom:* simplified view of the graph structure.

A property of the de Bruijn Graph is that polymorphisms create bubble structures in the graph (see Figure 4.8). A bubble starts with a forking vertex, from which paths leave and later rejoin with each other. The paths are often called 'branches' and represent the alleles, and the unitigs either side of the bubble provide flanking sequences. By identifying these bubble structures in the graph, we can find polymorphisms and their identifying flanking sequence, which can be used to place them on a reference genome. Bubbles can also be caused by sequencing errors and paralogous sequence (e.g. a family of repeats in a single individual).

Cortex only found bubbles with exactly two branches, both of which were made up of exactly one unitig (all vertices had in-degree and out-degree one). In order to place the variant on a reference it was required that the reference exactly matched one of the branches, and it was also required that the reference was used in calling the bubble. Since bubble branches were unitigs, there was no benefit to using link information with the same algorithm as there were no junctions to resolve. A user specified maximum allele length parameter was the only factor that determined whether a simple bubble was called. To remove duplicates that resulted from assembling a bubble from left-to-right or right-to-left, Cortex would only output bubbles if the left hand fork $k$mer was lexically lower than the right hand $k$mer.

We propose a new bubble calling algorithm that finds all complex bubbles in a graph where two or more paths diverge from the same $k$mer and later rejoin, so long as those paths can be assembled in the same direction. To do this we iterate over all $k$mers in the graph to find those that are fork vertices. From fork vertices the algorithm assembles all branches in each of the samples. Samples will likely share many of the same paths, in which case this scales better than samples $\times$ alleles, as we assemble in parallel across samples. We still use the maximum allele length parameter and assemble each branch until we

reach a junction that cannot be resolved, a gap in coverage or reach the assembly limit. As we have link information we can now assemble each branch across many unitigs. Once we have a set of paths leaving a fork vertex, we look for $k$mers where two or more paths meet. We then output the paths up to that $k$mer and the flanks either side of the bubble.

By attempting to walk from both sides of a bubble, we improve our discovery rate as some alleles can only be resolved in one direction. Duplicate variants and alleles are removed after bubbles have been decomposed against a reference.

Our bubble calling is similar to finding all superbubbles (in a directed graph (Onodera *et al.*, 2013)) or ultrabubbles (in a bidirectional graph (Paten *et al.*, 2018)), although we permit cycles in the induced subgraph, do not enforce a minimality condition and have a notion of vertex traversal orientation that requires two paths to traverse a $k$mer in the same directionn to have 'joined'. Given our relaxed definition of a superbubble, the bubbles we find do not segregate into nested or disjoint sets.

The algorithm is illustrated in Figures 4.9 and 4.10, and the pseudocode is written in Algorithm 4.

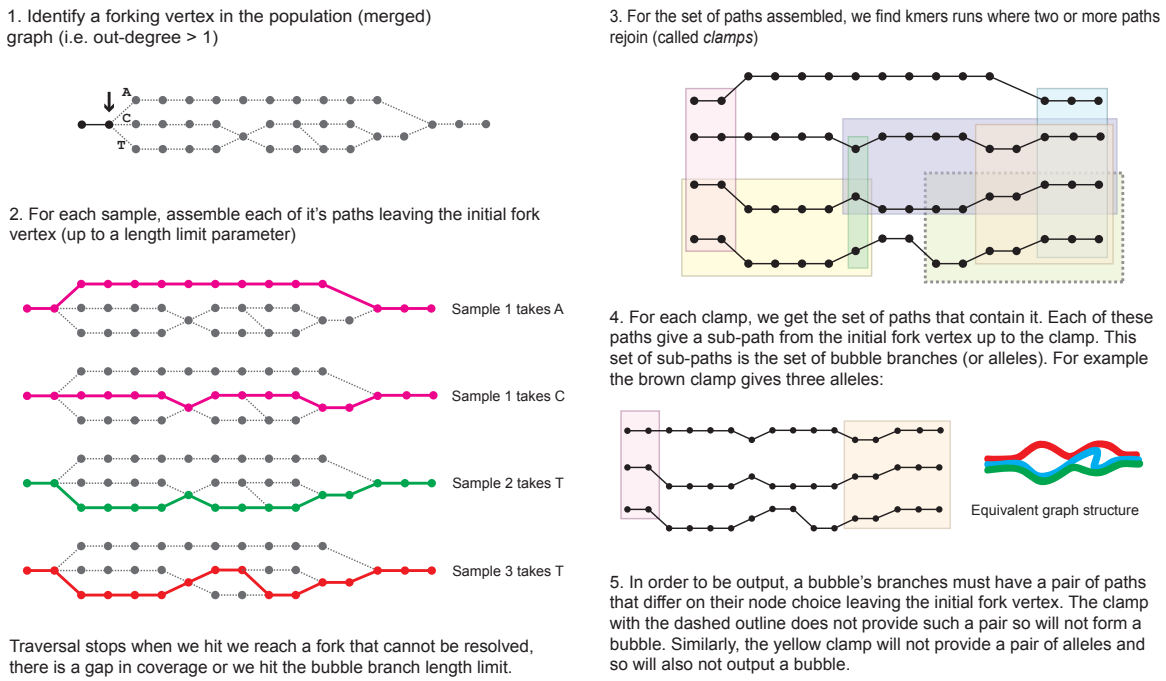


Figure 4.9: McCortex complex-bubble calling algorithm illustrated in three diploid samples. The de Bruijn Graph is represented by vertices (black dots) and edges (lines). Bubble calling iterates over all forking vertices in the graph, in this example we step through the processing of a single forking vertex with edge choices $\{A, C, T\}$. A single forking $k$mer could be in many bubbles, each with two or more branches / alleles.

Algorithm 4 Algorithm to find complex bubbles

```
1: procedure VERTEXHASCOLOUR(n,colour)
                ▷ Returns true if vertex n is in colour colour. Otherwise returns false.
2: procedure ASSEMBLE(vertices)
                ▷ Assemble from vertices onwards. Return list of unitigs visited.
3: procedure ORIENTATION(unitig)
                ▷ Return orientation of unitig (FORWARD or REVERSE)
4: procedure NEXTVERTICES(v_x)        ▷ Return the set of vertices directly after v_x in the graph
5: procedure REVERSE(v_x)                                  ▷ Return the reverse of vertex v_x
6: procedure OUTDEGREE(v_x)                                ▷ Return the out-degree of vertex v_x
7: procedure PRINTALLELES(A)
      ▷ Print out alleles if they are informative. ▷ A is a list of Alleles. ▷ Alleles are 0-based lists of unitigs.
8:     if |A| > 1 then
9:         if ∃a, b ∈ A ∧ a[1] ≠ b[1] then
10:            if ∃a, b ∈ A ∧ a[|a| − 2] ≠ b[|b| − 2] then
11:                emit A
12: procedure CALLBUBBLESFROMVERTEX(v_x,Graph,maxAllele)
13:    M{s} ← ∅; ∀s ∈ Unitigs(Graph)                          ▷ Map Unitig → Set of alleles
14:    for v_y in nextVertices(v_x) do
15:        for colour in colours do
16:            if VERTEXHASCOLOUR(v_x, colour) ∧ VERTEXHASCOLOUR(v_y, colour) then
17:                allele ← ASSEMBLE({v_x, v_y})
18:                for i in 1 ... |allele| − 1 do
19:                    M{allele[i]} ← M{allele[i]} ⋃ allele[0 ... i]
20:    for s in Unitigs(M) do                                 ▷ Iterate over unitigs visited
21:        A ← M{s}                                           ▷ Get alleles that visit this unitig
22:        A_f ← {ORIENTATION(a[|a| − 1]) = FORWARD; a ∈ A}
23:        PRINTALLELES(A_f)
24:        A_r ← {ORIENTATION(a[|a| − 1]) = REVERSE; a ∈ A}
25:        PRINTALLELES(A_r)
26: procedure CALLBUBBLES(Graph,maxAllele)
27:    for kmer in kmers(Graph) do
28:        if OUTDEGREE(kmer, FORWARD) > 1 then
29:            CALLBUBBLESFROMVERTEX(kmer, FORWARD, Graph, maxAllele)
30:        if OUTDEGREE(kmer, REVERSE) > 1 then
31:            CALLBUBBLESFROMVERTEX(kmer, REVERSE, Graph, maxAllele)
```

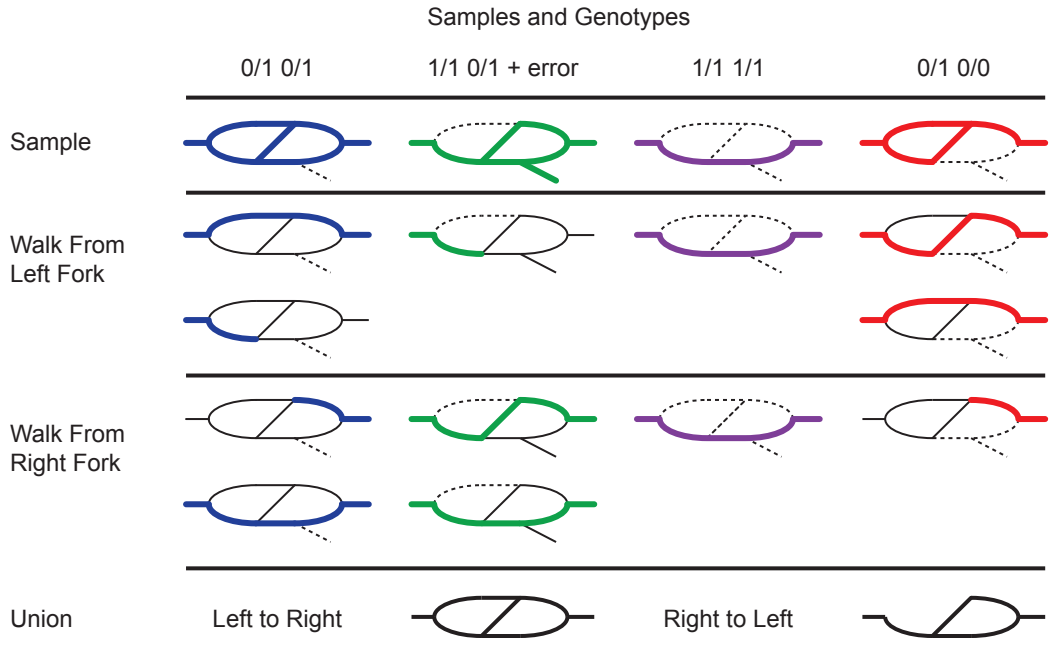


Figure 4.10: McCortex complex-bubble calling in three diploid samples at two adjacent SNP sites. McCortex attampts traversal of a bubble in both directions, thereby discovering these subgraphs. Cortex bubble-calling only attempts traversal in a single direction and only for simple SNPs, so would only discover the left hand SNP if the right flank was lexically lower than the left flank. It would not have discovered the right hand SNP, since the population contains forks it cannot traverse past. It would not have discovered the larger complex event either (two bubbles overlapping).

4.5.1 Multithreading

Multithreading for bubble calling is implemented similarly to contig construction (see Section 4.2.1). The graph is read-only so it is simple to have multiple threads reading it simultaneously without locks. The hash table of $k$mers is split into t segments, where t is the number of threads. Each thread searches its subset of $k$mers to find fork vertices. If a fork vertex is found, it is used to assemble branches which are then output. Only one mutex lock is required to protect the output file.

4.5.2 Filtering paralogous sequences

It is possible to specify which graph colours represent haploid sequence when running the bubble calling. This is useful when running with a reference sequence in one of the colours or with haploid samples. If a bubble has more than one branch with coverage in a haploid colour the bubble is assumed to represent paralogous rather than polymorphic sequences, and so is abandoned. This feature is useful for removing false positive polymorphisms caused by repeats.

4.5.3 Placing variants

Once all the bubbles have been identified in the graph and saved to a file, we have a set of putative polymorphisms. In order to make sense of these mutations and use them in downstream analyses we usually require that they be placed on a reference and decomposed into individual mutations. Polymorphic sites within k-bp of each other are merged into one bubble by the de Bruijn Graph. In order to decompose them back into individual events, we need to align them against themselves or against a reference.

We map the 5' flank using a read mapping program (e.g. Stampy (Lunter and Goodson, 2011), BWA-MEM (Li, 2013)). This is used to place the bubble on the genome. We then search locally for the 3' flank. The reference sequence between the mapped 5' flank and the aligned 3' flank gives us the reference allele. Our implementation uses two user defined thresholds: longest allele to align; and largest difference between alleles to attempt to align. We remove duplicate alleles and alleles that are too long. If we are left with two or more unique alleles, we align them one at a time with the reference allele and decompose them into separate mutation events.

Ideally we would do a multiple sequence alignment of these alleles in order to decompose them. We implemented this using the multiple sequence alignment algorithm TCOFFEE (Rausch *et al.*, 2008) from the *SeqAn* library (Döring *et al.*, 2008). However we found this was too computationally expensive and greatly slowed down variant calling. Instead we do a pairwise global alignment of each of our bubble alleles against the reference allele. These pairwise alignments are decomposed into separate mutations by breaking at matching bases. We output a VCF of sites with alleles, which is sorted, left-aligned and duplicates are removed.

Sequence alignment is implemented using the Needleman-Wunsch (Needleman and Wunsch, 1970) sequence alignment C library we developed called Seq-Align[7]. The 5' flank may map on either the plus (forward) or minus (reverse) strand, which makes the 5' flank the right or left side respectively. To search for the 3' flank, first we work out the maximum search region of the reference as $1.1 \times$maximum allele length$+$ k bases from the end of the 5' flank. If a perfect match for the first $k$mer of the 3' flank cannot be found, we do a global Needleman-Wunsch alignment of the search region of the reference and our 3' flank. The alignment is done with zero gap-open and gap-extend penalty for gaps at the start or end of the alignment to allow optimal placement. Pairwise alignments are done with penalties match +1, mismatch -2, gap-open -4, gap-extend -1. Alignments of alleles against the reference are broken at matching bases in order to decompose into smaller events.

Commands required to run the bubble caller and generate a VCF are shown in Listing 4.6 and doing the

[7]https://github.com/noporpoise/seq-align

same with the McCortex Pipeline are shown in Listing 4.7.

Listing 4.6: Steps to generate bubble calls then produce a VCF. Assumes steps in Figure 4.1 have already been run.

```
# Call bubbles
mccortex31 bubbles -m 20G -o bubbles.txt.gz -p graph.ctp graph.ctx
# Map bubble flanks
./scripts/cortex_print_flanks.sh bubbles.txt.gz > bubbles.5pflanks.fa
bwa mem hg19.fa bubbles.5pflanks.fa > bubbles.5pflanks.sam
# Decompse bubbles to create a VCF
mccortex31 calls2vcf -F bubbles.5pflanks.sam \
                -o bubbles.vcf bubbles.txt.gz hg19.fa
```

Listing 4.7: Calling bubbles using McCortex pipeline and $k$mers 31, 51.

```
echo "NA12878 NA12878.fq" > samples.txt
scripts/make-pipeline.pl 31,51 NA12878 samples.txt > job.mk
make -f job.mk CTXDIR=~/bin/mccortex MEM=250G NTHREADS=2 \
               USE_LINKS=yes bubbles-vcf
# result is now in NA12878/vcfs/bubbles.joint.links.k31.k51.vcf.gz
```

4.6 Genotyping a VCF against the graph

In Cortex, genotyping of variants was limited to those called by the bubble caller, and was limited in accuracy by looking for exact $k$mer matches to bubble branches. We have built a couple of commands to implement generalised VCF genotyping against one or more de Bruijn Graphs. The variants being genotyped do not need to have been called by McCortex. When genotyping, we do not limit ourselves to the $k$mers used to discover variants. By using a two pass streaming of a sorted VCF file we limit our memory use to a fraction of that needed to load the entire de Bruijn Graph, enabling us to genotype multiple samples at once. The McCortex command to run genotyping is: `mccortex31 vcfcov -r ref.fa.gz in.vcf graph.ctx > out.vcf`.

We use $k$mer counts to calculate the coverage on variants from a VCF file and corresponding reference genome sequence. The reference sequence is loaded and variants are streamed in. For each variant we group variants that are within k bases of it and construct all possible local haplotypes. The number of possible haplotypes increases exponentially with the number of variants within k bases and we provide a parameter to skip over variants that cluster too densely. In practise this is rarely a problem. If variants occur once every 20 bases and we are using $k = 31$, we only need three variants ($2^3 = 8$ haplotypes) to genotype the middle variant. The default cut-off is no more than 8 variants occurring within k bases.

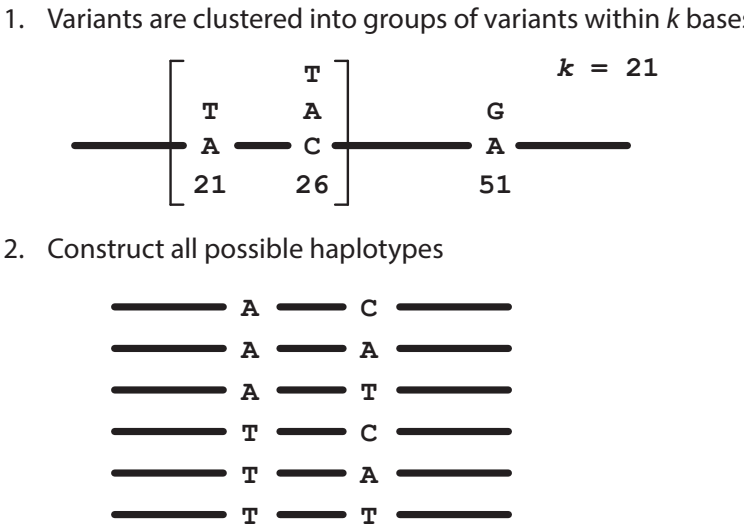


Figure 4.11: Generating VCF coverage from a de Bruijn Graph with McCortex. 1. Three consecutive variants are shown on a reference genome (**black** line) at positions 21, 26 and 51. At $k = 21$, some $k$mers cover both the first and middle variant. In order to genotype a variant (using a de Bruijn Graph with a given k), we must consider all variants within k bases of it. To genotype the middle variant (`26:C/A/T`), we must consider the first two variants together. 2. McCortex constructs all possible haplotypes for the selected variants. 3. $k$mers private to each allele are identified and the mean coverage on them is reported. Multi-allelic sites are broken down and reported as multiple biallelic sites. These $k$mer coverage are later used to genotype the sample.

From these generated haplotypes we identify $k$mers that uniquely tag a single allele and discard the other $k$mers. Once we have a set of allele-informative $k$mers, we stream in the de Bruijn Graph, loading each sample's $k$mer counts only for informative $k$mers. For each variant for each sample, we report the mean coverage on $k$mers that support an allele and those that support all other alleles ('absence' in the figure). The process is illustrated in Figure 4.11.

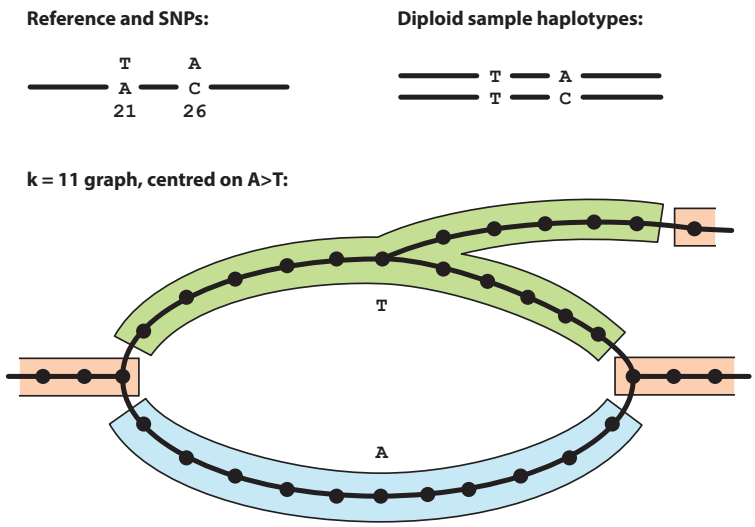


Figure 4.12: Graph structure around clustered variants. *Top-left:* Two adjacent variants shown on the reference genome (**black** line). *Top-right:* The phased haplotypes of the input sample. *Bottom:* *k*mers are coloured to show which *k*mers are used to support each of the `A/T` alleles of the `A/T` variant at position 21. The blue *k*mers support the `A` allele. The green ones support the `T` allele.

Once *k*mer counting has run, a second command (`mccortex vcfgeno`) applies the Cortex biallelic genotyping models (Iqbal *et al.*, 2012). Instead of reads arriving, we deal with *k*mers arriving (akin to setting read length to *k*mer size).

Loading only informative *k*mers makes it practical to use uncleaned graph files for genotyping – cleaning low frequency *k*mers from the graph provides no benefit to genotyping as genotyping models already account for sequencing error. Cleaning helps remove unresolved forks that would interrupt graph traversal during variant discovery, but only risk falsely removing informative *k*mers at the genotyping step.

Low coverage samples can be pooled, cleaned of errors and used for variant calling. These variants can be combined with existing known variants (e.g. 1000 Genomes in humans (McVean *et al.*, 2012)). Uncleaned individual graphs can then be used to genotype samples over the set of all variants. This approach to de novo variant calling (of sharing information across samples) can rescue polymorphisms that would otherwise be lost in low coverage individuals.

Using a VCF rather than graph call files means we can apply coverage to calls generated from various sources. Genotyping works for biallelic SNPs and indels. We can pull out *k*mer coverage for multiallelic variants but have not updated the genotyping models for them. The output VCF retains all the information from the input VCF with the addition of samples' genotypes. We mark coverage as missing where it is not possible to calculate coverage. This may be due to a dense cluster of variants or allele length exceeding the given limit (user specified parameter). This approach to genotyping relies on knowing all

the variants in the sampled population, so the haplotypes and informative *k*mers can be identified. For this reason, results will probably be better when using unfiltered variant sets.

Haplotype construction is similar in some ways to the Freebayes variant caller (Garrison and Marth, 2012) which performs realignment of mapped reads on inferred haplotypes.

4.7 Variant calling simulation

To assess the abilities of the two variant calling algorithms we ran two simulations to compare them – one single diploid individual and many haploid samples.

4.7.1 Single diploid human 1Mbp

We simulated variant calling in a diploid sample using 1Mbp of GRCh37 reference assembly chromosome 22 (hg19; chr22:17,000,000-17,999,999). The diploid genome was created with 999 SNPs and 99 deletions (54 from one chromosome, 45 from the other). Reads were sampled with 60X coverage (30X per chromosome), 1% sequencing error rate, 150bp paired-end reads with mean insert gap of 250bp with a standard deviation of 50. Variants were called with three approaches: Cortex bubble caller, McCortex bubble caller and McCortex breakpoint caller. All variant calling approaches used the reference in the graph. We compared the intersections of the resulting VCFs with a VCF representing the mutations we simulated.

Calling Method	Missed	False Positives	Found
Bubbles (links)	183 / 1100 (16.64%)	18 / 935 (1.93%)	917 / 1100 (83.36%)
Bubbles (plain)	186 / 1100 (16.91%)	18 / 932 (1.93%)	914 / 1100 (83.09%)
Breakpoints (links)	175 / 1100 (15.91%)	14 / 939 (1.49%)	925 / 1100 (84.09%)
Breakpoints (plain)	204 / 1100 (18.55%)	14 / 910 (1.54%)	896 / 1100 (81.45%)
Cortex Bubbles	213 / 1100 (19.36%)	8 / 895 (0.89%)	887 / 1100 (80.64%)

Table 4.4: Diploid variant calling simulations using 1Mbp of GRCh37 reference assembly chromosome 22 (hg19; chr22:17,000,000-17,999,999). Diploid genome was created with 999 SNPs and 99 deletions (54 from one chromosome, 45 from the other). Reads were sampled with 60X coverage (30X per chromosome), 1% sequencing error rate, 150bp paired-end reads with mean insert gap of 250bp and standard deviation = 50. All variant calling approaches used the reference. Simulation script is in `results/var_calling_diploid_chr22_1Mbp/` of McCortex repository.

Results are shown in Table 4.4. McCortex variant calling has both better sensitivity and lower specificity than Cortex. McCortex has better sensitivity when using links, and in this simulation using links does not decrease the specificity. Using only 1Mbp of the human genome reduces the number of repeats we

expect, therefore we expect links to have a greater benefit when the whole genome is present in the graph.

4.7.2 Ten *E. coli* Simulation

Calling Method	Missed of 45,765	False Positives	Found of 45,765
Bubbles (joint, links):	3,259 (7.12%)	3,109 / 45,615 (6.82%)	42,506 (92.88%)
Bubbles (joint, plain):	3,317 (7.25%)	2,895 / 45,343 (6.38%)	42,448 (92.75%)
Bubbles (indep., links):	1,609 (3.52%)	3,962 / 48,118 (8.23%)	44,156 (96.48%)
Bubbles (indep., plain):	1,633 (3.57%)	3,152 / 47,284 (6.67%)	44,132 (96.43%)
Breakpoints (joint, links):	1,990 (4.35%)	256 / 44,031 (0.58%)	43,775 (95.65%)
Breakpoints (joint, plain):	2,038 (4.45%)	256 / 43,983 (0.58%)	43,727 (95.55%)
Breakpoints (indep., links):	2,004 (4.38%)	433 / 44,194 (0.98%)	43,761 (95.62%)
Breakpoints (indep., plain):	2,214 (4.84%)	432 / 43,983 (0.98%)	43,551 (95.16%)
Cortex (indep.):	2,353 (5.14%)	369 / 43,781 (0.84%)	43,412 (94.86%)
Cortex (joint):	18,077 (39.50%)	70 / 27,758 (0.25%)	27,688 (60.50%)

Table 4.5: McCortex variant calling in ten *E. coli* samples. The 'K12' reference sequence was used which has 4,641,652bp. We generated ten samples by simulating 45,765 biallelic SNPs with a $\frac{1}{n}$ frequency distribution. Coverage is 50X with 2x150bp PE reads with an insert gap normally distributed with $\mu = 250$bp and $\sigma = 50$bp. Reads have a uniform single base sequencing error rate of 1%. *Indep.* means the independent calling approach whereby samples are compared to the reference one-by-one. *Joint* refers to the combined sample calling approach (the opposite of independent) which loads all samples at once and borrows coverage information across the population. *Links* means we use a Linked de Bruijn Graph, *plain* means we did not use graph links. *Bubbles* is the McCortex bubble caller and *Breakpoints* is the McCortex breakpoint caller. 'Cortex' is the Cortex bubble caller. All calling approaches used the reference in the graph. In bubble calling only the reference was labelled as a haploid sample to remove ref/ref bubbles.

To evaluate McCortex variant calling performance with multiple samples, ten haploid *E. coli* genomes were simulated from the reference sequence 'K12' (4,641,652bp with no gaps / 'N' bases). We generated ten samples by simulating 45,765 biallelic SNPs with a $\frac{1}{n}$ frequency distribution. Simulated coverage was 50X with 2x150bp PE reads with an insert gap normally distributed with $\mu = 250$bp and $\sigma = 50$bp. Reads have a uniform single base sequencing error rate of 1%.

Variant calls were made with McCortex using all combinations of the bubble and breakpoint callers, using linked and plain de Bruijn Graphs and the independent vs. joint calling pipelines. *Independent calling* compares one sample at a time to the reference, *joint calling* compares all samples at once. The output VCFs were normalized with *bcftools* and an intersection of sites was taken with the normalized truth VCF file. The numbers of sites correctly called as well as the number of false calls are shown in Table 4.5. We used Cortex to do simple bubble calling on the same data using both the independent and joint pipelines. Cortex results are shown in the same table.

Overall, the McCortex bubble caller performs poorly. McCortex bubble calls have a a high false positive rate. Additionally, joint calling with the bubble caller misses many variants. Pooling samples in the joint

calling approaches has the effect of shortening the 5' flank of bubbles. This reduces the sequence available to map to the reference, which is required to place bubble calls on the reference.

In all scenarios, using links narrowly increased the number of variants we correctly called. In breakpoint calling, using a linked de Bruijn Graph only added one extra false positive out of $\sim$ 44,000 sites called. In bubble calling, using links caused a significant increase in the number of false positives.

The *E. coli* genome has low repeat content and is mostly unique, even at $k = 21$. Therefore the expected benefit of links is marginal.

4.7.3 Summary

The simulations highlighted issues with the bubble caller when doing joint calling. Decomposing bubble calls may be improved by mapping the bubble branches including the flanks on either side, and using this alignment to decompose variants into a VCF file. Modern read mappers can now handle longer sequences – a consequence of the development of long read sequencing platforms. By using the flanks on either side of the bubble, we provide more sequence context for placing variants. Hopefully this would increase sensitivity by placing more bubbles. It should also increase specificity by reducing incorrectly mapped flanks and better identifying the 3' flank.

4.8 Error correction

As we had already implemented error correction on sequence data in order to construct graph links (see Section 2.2.3), we added a command to McCortex to output the error corrected reads. This command (`mccortex31 correct`) error corrects reads against a cleaned de Bruijn Graph with or without links.

The approach used is the same as that used when threading reads to create links. Read $k$mers are found in the graph to anchor the read. Gaps between anchors (where read $k$mers are not in the graph) are filled by attempting to traverse the graph. At this traversal step, links are used if present. To adapt the error correction for sequencing reads we added the ability to traverse to the start and end of a read. When adding links from single-ended reads, recovering end $k$mers is not useful. $k$mer recovery by gap traversal is only useful when traversing between a pair of read $k$mers that are in the graph. Therefore, for error correction of a read, if the last N $k$mers from a read are not in the graph, we use the $k$mers that were found to traverse the graph and recover those N $k$mers. Corrected read bases are output as uppercase ('corrected') if they are within a $k$mer that exists in the graph, or lowercase ('uncertain') if they are not

covered by a graph $k$mer. The user may, if they wish, mask out uncertain bases after cleaning by setting lowercase bases to N. Masking data in this way avoids losing read data that we cannot correct leaving the majority of the remaining errors clearly demarcated.

Error correction of reads should use the same sample (or population) for creating the graph and adding the links. If the sample being error corrected was not represented in the cleaned graph sample, specific sequence would be cleaned off – even if it has high coverage. Reads can be cleaned against multi-coloured de Bruijn Graphs – doing so allows error correction on low coverage data, although some polymorphic sites will be lost resulting in reduced heterogeneity.

This approach to error correction is limited by only using one $k$mer size. The optimal $k$mer sizes for variant calling, assembly and read error correction may be different. This requires re-running graph construction with different values of k. Additionally, error correction will not attempt to alter a $k$mer from the read if it exists in the graph. Sequencing errors that happen to collide another part of the graph will go uncorrected.

4.8.1 Simulation

In order to assess our error correction accuracy we simulated and corrected sequencing errors with 1Mbp of diploid human sequence[8]. We used 1Mbp from GRCh37 (hg19; chr22:17,000,000-17,999,999) as our reference. In order to simulate a diploid individual we duplicated the 1Mbp reference to create a pair of chromosomes, which we mutated with 998 SNPs, 7 deletions from one chromosome ($1-3$ bp) and 3 deletions from the other ($1-4$ bp). Single-ended reads were sampled with 25X coverage uniformly across each of the two chromosome references (50X diploid coverage) with read length of 250bp. Sequencing error was applied to the reads using an empirical distributed error rate taken from the FASTQ quality scores of Illumina PhiX data. Each simulated read was paired up with an Illumina PhiX read, and the PhiX quality scores used as the error rate for a base change at that position. Uncalled `N` bases in the PhiX read were copied to the simulated data. The average sequencing error rate was 0.44% bases mutated and 0.02% bases set to 'N'.

Having simulated a diploid genome and sampled reads from it with sequencing error, we then attempted to correct them. Error correction requires three commands: 1) we built a single de Bruijn Graph of the reads at $k = 31$; 2) we cleaned the graph; 3) we error corrected the reads against the graph.

[8]Instructions for reproducing these results are in the *results/correct/* directory of McCortex

Input reads		
	(bases)	
Total match	49,768,285 / 49,999,250	(99.54%)
Total Mismatch	219,533 / 49,999,250	(0.44%)
Total Uncalled (N)	11,432 / 49,999,250	(0.02%)
Cleaned without link info		
Cleaned match	49,959,807 / 49,999,226	(99.92%)
Cleaned mismatch	37,045 / 49,999,226	(0.07%)
Cleaned Uncalled (N)	2,374 / 49,999,226	(0.00%)
Changed bp matching	195,810 / 200,261	(97.78%)
Confident total	49,708,224 / 49,999,226	(99.42%)
Confident match	49,701,347 / 49,708,224	(99.99%)
Confident mismatch	6,877 / 49,708,224	(0.01%)
Unconfident total	291,002 / 49,999,226	(0.58%)
Unconfident match	258,460 / 291,002	(88.82%)
Unconfident mismatch	30,168 / 291,002	(10.37%)
Cleaned with link info		
Cleaned match	49,978,540 / 49,999,239	(99.96%)
Cleaned mismatch	19,915 / 49,999,239	(0.04%)
Cleaned Uncalled (N)	784 / 49,999,239	(0.00%)
Changed bp matching	217,154 / 224,401	(96.77%)
Confident total	49,890,555 / 49,999,239	(99.78%)
Confident match	49,880,788 / 49,890,555	(99.98%)
Confident mismatch	9,767 / 49,890,555	(0.02%)
Unconfident total	108,684 / 49,999,239	(0.22%)
Unconfident match	97,752 / 108,684	(89.94%)
Unconfident mismatch	10,148 / 108,684	(9.34%)

Table 4.6: Error cleaning for diploid 1Mbp of *Human* chr22, 50X coverage with 250bp reads (single-ended) and 0.44% empirically distributed sequencing error. Graph cleaning removed unitigs with coverage < 7. 'Match' means bases that are correct, 'mismatch' are bases that are incorrect, 'uncalled' is the number of 'N' bases; 'Cleaned' refers to all bases after cleaning, 'confident' are bases that are reported in uppercase; 'unconfident' are bases reported in lowercase; 'changed bp matching' is the fraction of bases that error correction changed correctly. Cleaning was performed with and without links.

Results for error correction of reads, with an empirically sampled error distribution, against a graph with and without links, are shown in Table 4.6. Error cleaning reduced the number of incorrect and uncalled bases by 83% and 79% without links and 90% and 93% with links. By comparing the input and output reads, we can see that 98% and 97% of bases altered by the correction step were correct (without and with links respectively). Using links in the graph increases the number of bases changed by 12%, of which 88% were changed correctly. As we would expect, using links in the graph increases the number of bases that we change correctly but also increases the number of incorrect changes made. Finally we see that the error rate in the unconfident bases is much higher than the confident bases, with the majority of remaining incorrect bases marked as 'unconfident'.

Input reads		
	(bases)	
Total match	459,521,747 / 464,165,000	(99.00%)
Total Mismatch	4,643,253 / 464,165,000	(1.00%)
Cleaned without link info		
Cleaned match	464,109,476 / 464,165,007	(99.99%)
Cleaned mismatch	55,531 / 464,165,007	(0.01%)
Changed bp matching	4,590,581 / 4,593,471	(99.94%)
Confident total	463,946,263 / 464,165,007	(99.95%)
Confident match	463,931,898 / 463,946,263	(100.00%)
Confident mismatch	14,365 / 463,946,263	(0.00%)
Unconfident total	218,744 / 464,165,007	(0.05%)
Unconfident match	177,578 / 218,744	(81.18%)
Unconfident mismatch	41,166 / 218,744	(18.82%)
Cleaned with link info		
Cleaned match	464,135,334 / 464,165,007	(99.99%)
Cleaned mismatch	29,673 / 464,165,007	(0.01%)
Changed bp matching	4,616,743 / 4,619,946	(99.93%)
Confident total	464,067,527 / 464,165,007	(99.98%)
Confident match	464,052,848 / 464,067,527	(100.00%)
Confident mismatch	14,679 / 464,067,527	(0.00%)
Unconfident total	97,480 / 464,165,007	(0.02%)
Unconfident match	82,486 / 97,480	(84.62%)
Unconfident mismatch	14,994 / 97,480	(15.38%)

Table 4.7: Error cleaning for *E. coli* genome, 100X coverage with 250bp reads (single-ended) and 1% uniform sequencing error. Removed unitigs with max $k$mer coverage < 9. 'Match' means bases that are correct, 'mismatch' are bases that are incorrect, 'uncalled' is the number of 'N' bases; 'Cleaned' refers to all bases after cleaning, 'confident' are bases that are reported in uppercase; 'unconfident' are bases reported in lowercase; 'changed bp matching' is the fraction of bases that error correction changed correctly. Cleaning was performed with and without links.

Empirically determined PhiX error rates may be over-reliant on poorly calibrated FASTQ quality scores and are heavily distributed towards the end of reads. To test if we could handle higher error rates evenly distributed across longer reads, we ran a second simulation using a uniform 1% error rate. Using the *E. coli.* K12 assembly, we sampled (single-ended) 250bp reads with 100X coverage and 1% uniform sequencing error. The results are shown in Table 4.7 and show the same findings as in the diploid human simulation. Noticeable is the improvement in the remaining incorrect bases vs the human simulation ($p<0.01$, two-sided Fisher Exact Test) – it is not clear if this is due to using a haploid rather than diploid genome or the reference genome having higher complexity (fewer low complexity repeats).

4.8.2 Summary

An alternative (but still $k$mer-based) approach to error correction is to optimize the path through the graph described by the read. This likely requires a heuristic rather than optimal solution as the number of possible paths increases exponentially with path length. Such an approach would be able to correct

errors that created $k$mers that hit other parts of the graph. This is similar to the heuristic approach taken by *BFC* (Li, 2015a).

We did not initially set out to create a $k$mer based sequence error correction algorithm. Rather it is a side-effect of mapping reads to the graph to annotate with links, which requires a perfect match. However our implementation appears to successfully identify and correct sequencing errors and is flexible in the size of the errors (one or many bases). This is reassuring evidence that link annotation is reasonably robust to sequencing errors.

We do not anticipate this approach working for high error rate sequencing platforms, where the distance between errors can be as low as 10bp in the case of Nanopore (Ip *et al.*, 2015). To annotate a graph with such reads will require the use of third party error correction software.

4.9 Discussion

In this chapter we have described some of the algorithms we have developed to exploit the information in a Linked De Bruijn Graph data structure.

We showed that Linked de Bruijn Graph can assemble longer contigs than traditional de Bruijn Graphs, but with a higher misassembly rate. Further, the Linked de Bruijn Graph reduces the sensitivity to the only parameter of the de Bruijn Graph – k. We showed through simulation that Linked de Bruijn graphs outperform the SGA assember giving longer true contigs and fewer assembly errors.

Small event variant calling with a Linked de Bruijn Graph does not outperform a traditional de Bruijn Graph and is substantially worse than mapping based approaches for human simulations. However for large events, the Linked De Bruijn Graph shows promise in detecting insertions and deletions longer than the read length as well as chromosomal rearrangements.

Simpler variant calling approaches are also possible once an assembly has been extracted from a Linked de Bruijn Graph. One approach is to map contigs back to a reference (Li, 2015b); another is more elaborate reference-assembly comparisons (Nattestad and Schatz, 2016).

We presented a greedy algorithm to assemble haplotypes to facilitate genotyping. It should perform better (than a single biallelic site model) on adjacent sites within k bases when they are not in perfect linkage disequilibrium in the sampled individuals. Although we can count coverage of informative alleles for multiallelic sites, we do not have a model to genotype them. We are also lacking an implementation of a Hardy-Weinberg model to classify variants as repeats, errors or polymorphisms. This would help filter out false positive polymorphisms. Our genotyping implementation can be used to skip the variant

discovery step and instead directly genotype samples on known variants – without assembling them or mapping their reads.

We note that it is possible to use a *Linked de Bruijn Graph* to evaluate an assembly. An assembly is a hypothesis on the correct path through the graph. Simple heuristics could be developed to assess how likely the assembly is to be correct at various positions based on its agreement with the read information stored in the graph links.

Another application we have considered is the targeted assembly of genome regions of interest. By using the `subgraph` command to pull out the region of interest in the graph, analysis can be focussed. This was considered for sex chromosomes where $k$mers can be identified by their coverage across sample of different sexes.

In the next chapter we go beyond simulation and apply these algorithms to real sequencing data.

Chapter 5

Results

To measure the speed, resources required and accuracy of McCortex as a variant caller, we applied it to two real world data sets – a human and a bacterial sample.

5.1 Human: NA12878

NA12878 is a female sample from the 1000 Genomes project CEU population (McVean *et al.*, 2012). To call variants, we used the publicly available Platinum Genomes (Eberle *et al.*, 2017) Illumina sequencing data (2 x 100bp PE) which is PCR-free and so avoids issues around coverage variance due to PCR duplicates. We used the 1000 Genomes reference from Phase 1 (McVean *et al.*, 2012)[1] which is based on hg19 / GRCh37 coordinates. To evaluate our results, we generated variant calls with Cortex, as well as a high quality variant truth set. The Genome In A Bottle (GIAB) consortium has carried out high quality variant calling on NA12878 (Zook *et al.*, 2014) using five variant callers. They have produced a BED file describing regions where they can call variants confidently as well as a set of high confidence calls in these regions (SNPs and indels in VCF format). We use this call set to validate our calls in the high confidence regions.

5.1.1 Calling with McCortex

We called with bubble and breakpoint callers, using the reference, with and without graph linking information. *k*mer sizes of 31 and 61 were used. Graphs were built with a minimum base quality of 10 and no

[1] 1000 Genomes Phase 1 human reference genome: `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/human_g1k_v37.fasta.gz`

PCR-duplicate removal. Maximum memory permitted was specified as 400GB. Since we are running on a 1TiB RAM machine with 64 cores, a job using ∼150GiB of RAM (i.e. most McCortex steps) should use ∼10 cores to consume the same ratio of each. We specified a maximum 12 threads.

Run times and memory usage for all the steps in de novo assembly and variant calling of NA12878 are shown in Table 5.1. The most time and memory intensive step is the read threading time, which is dominated by writing from memory to disk after it finishes generating links. As it is currently implemented, this step requires traversing the links through the graph as they are written out (to slow network-attached storage in our case). Since no error cleaning has been performed at this step, there are many links (link cleaning step greatly reduces the number of links).

Step	$k = 31$		$k = 61$	
	Memory (GiB)	Time (hr:min)	Memory (GiB)	Time (hr:min)
Build ref graph	72	0:23	116	3:34
Build sample graph	50	2:40	80	2:43
Clean sample graph	52	0:44	95	0:54
Infer edges	27	1:01	57	2:49
SE read threading	400	14:16	400	8:22
SE link cleaning	<1	4:07	<1	0:13
PE read threading	400	35:14	400	6:57
PE link cleaning	<1	3:42	<1	0:13
Bubble calling (plain)	41	1:59	72	2:32
Bubble calling (links)	69	4:11	101	2:21
Breakpoint calling (plain)	55	3:22	91	3:37
Breakpoint calling (links)	93	2:10	128	2:49
Decomposing to VCF	3	<0:05	-	-
Genotyping a VCF	10	∼0:58	-	-

Table 5.1: Wall clock time (hours:minutes) and peak memory (GiB) required for various McCortex steps processing human whole genome sequencing data. Time is rounded to nearest minute and processes were run with 12 threads. The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

Total disk space for a cleaned graph and links for NA12878 with a single $k$mer is 33GiB (31 + 1.3 GiB) on disk. This is in comparison to the input data which is 157GiB if stored as a BAM or 63GiB for CRAM – see Table 5.2.

File	Size (GiB)	
	$k = 31$	$k = 61$
Input 50X as BAM	157	
Input 50X as CRAM	63	
Raw graph	38	70
Clean graph	31	55
Raw SE links	71	3.6
Clean SE links	2.7	0.7
Raw PE links	77	4.6
Clean PE links	2.2	0.6
Breakpoint calls (links)	1.5	2.2
Breakpoint calls (plain)	1.4	2.1
Bubble calls (links)	1.3	1.8
Bubble calls (plain)	1.3	1.8
Breakpoint (links) Merged VCF	0.04	
Breakpoint (plain) Merged VCF	0.04	
Bubbles (links) Merged VCF	0.04	
Bubbles (plain) Merged VCF	0.04	

Table 5.2: Source data and McCortex file sizes on disk for NA12878 built with $k = 31, 61$. The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

The $k$mer coverage distribution (for $k = 31$) is shown in Figure 5.1. From this our model (see Section 2.2.2) chose a median unitig cleaning coverage threshold of < 9. Cleaning removed 580,576,102 / 3,134,758,343 (18.5%) $k$mers. The removed unitigs consisted of 16,029,979 low coverage unitigs (295,698,498 $k$mers), 5,820 unitig tips (90,197 $k$mers) and 25,705,297 unitigs (284,787,407 $k$mers) which were both tips and had low coverage.

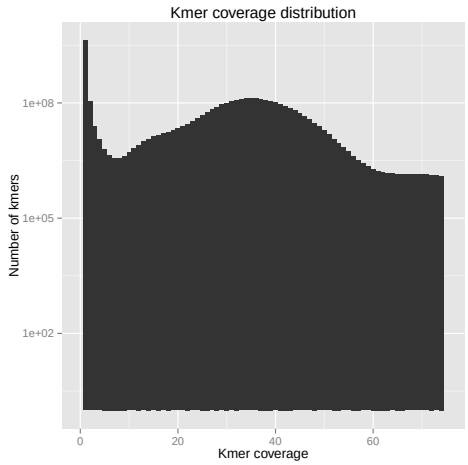


Figure 5.1: $k$mer coverage on NA12878 before cleaning for $k = 31$. From this distribution a median unitig coverage threshold of < 9 was selected for cleaning by our model (see Section 2.2.2). The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

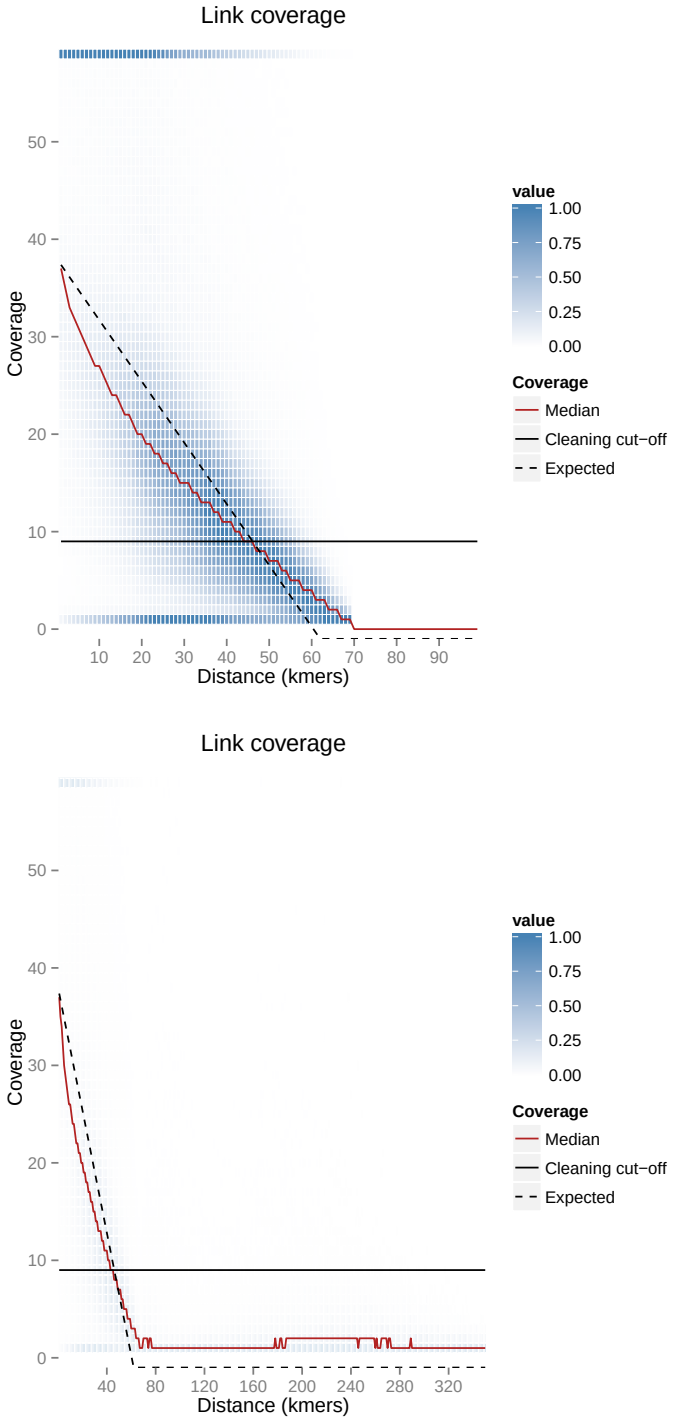


Figure 5.2: Link coverage on human sample NA12878 before cleaning: single-ended 100bp reads (top) and 2x100bp reads (bottom). Black line shows the cleaning cut-off of < 10. Within plots, counts have been normalised by column. Black dashed line is expected link coverage, given $k$mer coverage ($\sim$39) and mean read length (92bp). The input data set is $\sim$50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

When aligned to the graph (at $k = 31$), 4.3% of reads had a gap in the $k$mer path through the de Bruijn Graph. During SE read threading we were able to fill in 39.48% of these gaps, improving $k$mer coverage

from 41.96X to 42.31X. Gap filling paired end reads succeeded for 51.86% of fragments. This improved mean $k$mer coverage from 41.89X to 66.31X.

At $k = 31$, links generated from single-ended reads (SE links) were cleaned with a coverage threshold of 19 (i.e. links with count < 19 were removed). Links generated from paired end reads used a threshold of < 16. The SE link threshold was chosen using 5,000 random $k$mers which gave 75,610 links; PE link threshold was also chosen using 5,000 random $k$mers which gave 77,350 links.

We made additional calls using links cleaned with a lower threshold of < 3 to see the effect this would have on calling.

Breakpoint calling required 20 $k$mers of agreement with the reference, which is equivalent to 50bp and 80bp for $k = 31, 61$ respectively. Bubble calling required that the 5' flank mapped with a MAPQ for ≥ 30.

VCFs generated by the standard McCortex pipeline are automatically left-aligned and labelled with homologous run information to tag indels in low complexity sequence.

5.1.2 Results

		Links	Auto-cleaning		Minimal cleaning	
	$k$mers	used	Power (%)	FP (%)	Power (%)	FP (%)
Breakpoint	31, 61	Yes	81.4	0.6	82.1	0.6
Breakpoint	61	Yes	72.1	0.5	72.7	0.5
Breakpoint	31	Yes	67.1	0.4	68.3	0.4
Breakpoint	31, 61	No	80.8	0.6	-	-
Breakpoint	61	No	71.2	0.5	-	-
Breakpoint	31	No	66.3	0.4	-	-
Bubble	31, 61	Yes	82.2	1.1	82.5	1.2
Bubble	61	Yes	74.1	0.6	70.4	0.9
Bubble	31	Yes	70.0	0.8	73.0	0.6
Bubble	31, 61	No	82.0	1.0	-	-
Bubble	61	No	72.5	0.6	-	-
Bubble	31	No	69.8	0.8	-	-
Cortex	31, 61	No	78.5	0.5	-	-
Cortex	61	No	76.8	0.5	-	-
Cortex	31	No	65.9	0.3	-	-
Cortex (PG)	31, 51	No	53.1	0.2	-	-

Table 5.3: Variant calling in human sample NA12878. Truth set is Genome In A Bottle (GIAB) v2.19. FP = False Positive rate. Rows with more than one $k$mer are for the union of variant calls from both $k$mer sizes. The 'Auto-cleaning' 'Power' and 'FP' columns refer to link cleaning at < 10 (k=31,SE links), < 10 (k=31,PE), < 6 (k=61,SE links), < 7 (k=61,PE links), where SE = single-end reads, PE = paired-end reads. The last pair of columns ('Minimal cleaning') refer to link cleaning at < 3. Cortex (PG) refers to the Cortex Platinum Genomes variant call set (Eberle *et al.*, 2017). The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

Caller	k	Links	Power (%)					False Positives (%)				
			SNPs	Ins $< 10bp$	Del $< 10bp$	Ins $\geq 10bp$	Del $\geq 10bp$	SNPs	Ins $< 10bp$	Del $< 10bp$	Ins $\geq 10bp$	Del $\geq 10bp$
McCortex Breakpoints	31, 61	Yes	83.6	63.1	67.7	57.4	58.2	0.3	2.5	3.1	7.6	13.1
McCortex Breakpoints	31, 61	No	83.0	62.6	67.3	56.4	57.9	0.3	2.4	3.1	7.6	12.5
McCortex Breakpoints	31	Yes	69.7	47.5	50.4	37.4	35.2	0.2	1.7	2.1	5.3	8.3
McCortex Breakpoints	31	No	68.8	46.9	49.8	36.5	34.7	0.2	1.7	2.1	5.4	8.2
McCortex Breakpoints	61	Yes	74.1	55.3	59.1	52.6	51.9	0.2	2.1	2.6	4.9	8.0
McCortex Breakpoints	61	No	73.2	54.7	58.4	51.5	51.4	0.2	2.1	2.6	4.7	7.4
McCortex Bubbles	31, 61	Yes	84.4	64.7	68.4	58.2	53.9	0.7	3.3	3.7	6.6	10.5
McCortex Bubbles	31, 61	No	84.1	64.5	68.2	57.5	53.7	0.7	3.3	3.7	6.2	10.1
McCortex Bubbles	31	Yes	72.6	50.7	52.6	38.3	31.3	0.6	2.3	2.6	4.5	6.8
McCortex Bubbles	31	No	72.4	50.6	52.4	37.4	31.2	0.5	2.3	2.6	4.1	6.6
McCortex Bubbles	61	Yes	76.0	58.4	60.7	53.9	48.4	0.3	2.7	3.0	5.2	6.8
McCortex Bubbles	61	No	74.6	56.2	59.5	52.9	48.7	0.3	2.5	2.9	4.6	6.0
McCortex Breakpoints *	31, 61	Yes	84.3	63.6	68.2	58.2	58.8	0.3	2.5	3.2	7.5	13.9
McCortex Breakpoints *	61	Yes	74.7	55.7	59.5	53.4	52.3	0.2	2.2	2.7	4.8	8.1
McCortex Breakpoints *	31	Yes	70.8	48.3	51.4	38.6	36.2	0.2	1.8	2.2	5.2	9.2
McCortex Bubbles *	31, 61	Yes	84.7	64.9	68.5	59.1	54.2	0.8	3.3	3.8	7.3	11.7
McCortex Bubbles *	61	Yes	75.1	56.6	59.7	54.1	49.1	0.3	2.6	2.9	4.9	6.6
McCortex Bubbles *	31	Yes	73.1	51.1	52.9	39.7	31.6	0.6	2.4	2.7	5.0	7.7
Cortex	31, 61	No	80.1	66.5	67.9	58.6	52.4	0.3	2.3	2.1	3.4	2.9
Cortex	31	No	68.4	48.4	48.0	35.5	29.1	0.2	0.9	0.8	1.5	1.1
Cortex	61	No	78.5	63.5	65.6	57.1	50.8	0.3	1.9	1.8	2.7	2.3
Cortex (PG)	31, 51	No	55.7	34.4	34.2	28.5	23.3	0.1	0.3	0.2	0.8	0.7

Table 5.4: Variant calling in human sample NA12878, broken down by variant class. Truth set is Genome In A Bottle (GIAB) v2.19. FP = False Positive rate. Rows with more than one $k$mer are for the union of variant calls from both $k$mer sizes. $*$ Marked rows are results for cleaning links with a lower cleaning threshold (< 3), instead of link cleaning at < 10 (k=31,SE), < 10 (k=31,PE), < 6 (k=61,SE links), < 7 (k=61,PE links). Cortex (PG) refers to the Cortex Platinum Genomes variant call set (Eberle *et al.*, 2017). The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

We first reduced our McCortex VCFs to the regions specified by the GIAB confidence regions, then intersected with the GIAB VCFs to count how many calls are shared and how many are private to GIAB (missed by McCortex) or private to McCortex (likely false positives). The results are shown in Table 5.3 and broken down by variant class in Table 5.4. McCortex should have better sensitivity to long insertions and deletions than the variant callers used to produce the truth set, so the false positive rates may be an overestimate. Links did not appear to have a large effect on our ability to call variants and did increase the false positive rate. Competing short read NA12878 variant calling methods achieve 90-98% power and 0.08-0.31% false positive rate (Eggertsson *et al.*, 2017).

5.1.3 Low level link cleaning

Links as we were using them did not appear to improve our ability to call variants. We speculated that this may be due to the way in which they were cleaned of sequencing errors. Sequencing errors that happen to hit $k$mers elsewhere in the graph cause false links that can divert assembly down an incorrect edge or cause unresolvable regions that halt assembly. This is more likely to happen in high copy repeats, which is also where link information can be very useful. For this reason we remove links with low counts before using them. Cleaning links too aggressively leaves few and short links. Not cleaning sufficiently enough results in links that fork quickly and halt assembly by presenting ambiguity. Choosing the link cleaning threshold is discussed in Section 2.8.

We made calls using links cleaned with a threshold of < 3, instead of the thresholds chosen by our model (previously < 19 and < 16 for SE, PE (respectively) links at $k = 31$). The results of calls made with these links are shown in Table 5.3 in the last two columns. At $k = 31$ and 61 in both the bubble and breakpoint calls, a lower link cleaning threshold increases our false positive rate. With the exception of the breakpoint caller at $k = 31$, a lower link cleaning threshold also improves our discovery power.

It does not look like the efficacy of links is likely to be improved with a lower cleaning threshold.

5.1.4 Expected power

To evaluate the expected power of McCortex we calculated the number of variants from the NA12878 truth set that are discoverable with a $k = 31$ based de Bruijn Graph. We filtered the NA12878 truth set to variants that have at least 31 bases either side before the adjacent variant. This reduced our set of true variants to 2,620,040 / 3,163,065 (82.83%). We then constructed sequences of the alleles and the 31 bases of flanking reference (e.g. 31+1+31 = 63 bases for a SNP). By looking at graph structure and cleaned sample coverage in these contigs, we can count the number that should be discoverable

using the breakpoint and bubble variant callers with and without link information. Graph structure was determined with the McCortex *coverage* command which reports $k$mer coverage and edges in the graph in multiple samples for given input sequences.

By examining the graph structure locally around calls we can identify variants which have: missing $k$mer coverage after cleaning, forks that would halt assembly without links, forks that would halt assembly with links, both alleles in the reference, and those which should be resolvable. This should illustrate where variants are being lost. This does not take into account the map-ability of bubble flanks, so some calls that can be resolved in the graph will not be in the output VCF. The de Bruijn Graphs generated by McCortex and Cortex should have a similar set of $k$mers (the McCortex graph will have additional edges from the edge inference step), however Cortex chose a cleaning threshold of < 6, vs. McCortex choosing < 9.

	McCortex graph (cleaned < 9)		Cortex graph (cleaned < 6)	
ALT in the reference	141,905	(5.4%)	141,905	(5.4%)
ALT not in the NA12878 graph	108,704	(4.1%)	23,181	(0.9%)
ALT in NA12878 and not reference	2,369,431	(90.4%)	2,454,954	(93.7%)
ALT resolvable	1,924,855	(73.5%)	2,011,259	(76.8%)
ALT resolvable with **links**	125,308+1,924,855	(78.2%)	191,657+2,011,259	(84.1%)
REF & ALT resolvable	1,903,971	(72.7%)	1,969,466	(75.2%)
REF & ALT resolvable with **links**	145,220+1,903,971	(78.2%)	232,632+1,969,466	(84.0%)
Both branches single unitigs	1,802,781	(68.8%)	1,846,938	(70.5%)

Table 5.5: The number of GIAB NA12878 'truth set' variants (out of 2,620,040 sites) that we can expect to be resolved at $k = 31$ by the breakpoint or bubble calling in McCortex, with and without using links. REF/ALT refer to reference and alternative allele of a variant. 'ALT in the reference' means the alternative allele is in the reference sequence – bubbles are filtered out if the reference has coverage on both alleles, this removes repeats. Sites that have coverage in the sample and not in the reference may still be unresolvable if the graph has forks along the alleles. Some of these forks may be resolvable with links.

Results are shown in Table 5.5. The number of sites that are resolvable with link information is the upper bound as it assumes all the required read information is provided and far fewer are likely to actually be resolved with links. This is due to both sequence coverage and partly to graph structure.

A large number of events are lost due to reference coverage on both alleles. Alternatively these bubbles could be kept and repeats could be removed by masking the resulting VCF for known repeat regions. Setting a high mapping quality cutoff would also stop flanks being mapped into repeat regions. McCortex cleaned the $k$mers too aggressively; Cortex's lower cleaning threshold resulted in a higher percentage of variants where both alleles were resolvable (e.g. $75.2\% > 72.7\%$ without links).

Links rescue some variants that contain forks. Since the variant calling algorithms rely on seeding assembly with $k$mers immediately adjacent to the variant, assembly is handicapped. Seeding assembly

from novel (non-reference) $k$mers until we hit reference $k$mers may be a better approach in these low complexity cases (as proposed in Section 4.4.2).

Allele	Sensitivity (%)	Limit (%)	Fraction of limit (%)
Bubble (plain)	1,887,113 / 2,620,040 (72.2)	72.7	99.3
Bubble (links)	1,892,086 / 2,620,040 (72.4)	78.2	92.6
Breakpoints (plain)	1,804,021 / 2,620,040 (68.8)	73.5	93.6
Breakpoints (links)	1,825,935 / 2,620,040 (69.6)	78.2	89.0
Cortex	1,858,477 / 2,620,040 (70.9)	75.2	94.3

Table 5.6: The number of NA12878 variants that McCortex and Cortex actually resolved at $k = 31$ using the breakpoint or bubble caller, with links and without links (plain). Empirically defined limits are the upper bounds on sensitivity based on graph structure in both the reference and sample graphs (from Table 5.5). Limit for McCortex Bubble calling is **REF & ALT resolvable** (plain) and **REF & ALT resolvable with links** (links). Limit for McCortex Breakpoint calling is **ALT resolvable** (plain) and **ALT resolvable with links** (links). Limit for Cortex is **REF & ALT resolvable**.

We intersected our $k = 31$ McCortex and Cortex calls with the subset of sites used in the expected power estimations. The upper limit for McCortex Bubble calling sensitivity should be **REF & ALT resolvable** and **REF & ALT resolvable with links** rows in Table 5.5. The upper limit for McCortex Breakpoint calling sensitivity should be **ALT resolvable** and **ALT resolvable with links**. The upper limit for Cortex calling sensitivity is **REF & ALT resolvable**. In Table 5.6 we compare our actual variant discovery power to our estimated upper bounds. The difference between limit and achieved sensitivity is the result of missing or cleaning away required link information, failure to remove errors from the graph or variants lost because they couldn't be mapped back to the reference. Improving graph cleaning (removing false graph junctions) and link cleaning (keeping more true links, removing more false links) could help close this gap.

5.1.5 Genotyping against the graph

For genotyping, we looked at the McCortex bubble calls made with links ($k = 31, 61$). We were able to get graph coverage for 99.66% of these variants, with a maximum allele cut-off of 8 and a maximum allele length of 100. Genotyping against the uncleaned graph used 10GiB. Genotype concordance between McCortex calls and GIAB are shown in Table 5.7a. Genotype concordance between Cortex calls and GIAB are shown in Table 5.7b and for the intersection of McCortex and Cortex calls in Table 5.8. The McCortex/Cortex intersection contained 2,325,243 / 3,163,064 (73.5%) of the GIAB calls.

Truth Set \ McCortex Calls	./.	0/0	0/1	1/1	
0/1	105	858	1,494,736	8,545	99.37%
1/1	371	97	913	1,066,903	99.87%
	0.00%	0.00%	99.94%	99.21%	

(a) Overall concordance: 99.58%; 2,572,528 McCortex bubble calls with links ($k = 31, 61$)

Truth Set \ Cortex Calls	./.	0/0	0/1	1/1	
0/1	0	46	1,493,760	4,151	99.72%
1/1	0	83	8,256	976,443	99.15%
	0.00%	0.00%	99.45%	99.58%	

(b) Overall concordance: 99.49%; 2,482,739 Cortex calls ($k = 31, 61$)

Table 5.7: McCortex and Cortex genotype concordance with GIAB for all validated calls made by each. `0` is the reference allele, `1` is the alternative allele, `.` signifies missing data. `0/0` is homozygous-reference, `0/1` is heterozygous, `1/1` is homozygous-non-reference. True genotype is taken as GIAB, percentage correct is shown in last row and column. The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

Truth Set \ McCortex Calls	./.	0/0	0/1	1/1	
0/1	87	248	1,355,104	2,905	99.76%
1/1	318	44	233	966,304	99.94%
	0.00%	0.00%	99.98%	99.70%	

(a) Overall concordance: 99.84%; McCortex bubble calls with links ($k = 31, 61$)

Truth Set \ Cortex Calls	./.	0/0	0/1	1/1	
0/1	0	83	1,355,000	3,311	99.75%
1/1	0	81	5,164	961,654	99.46%
	0.00%	0.00%	99.62%	99.66%	

(b) Overall concordance: 99.63%; Cortex calls ($k = 31, 61$)

Table 5.8: McCortex and Cortex genotype concordance with GIAB for the intersection of McCortex/Cortex calls. `0` is the reference allele, `1` is the alternative allele, `.` signifies missing data. `0/0` is homozygous-reference, `0/1` is heterozygous, `1/1` is homozygous-non-reference. The McCortex/Cortex intersection contained 2,325,243 / 3,163,064 (73.5%) of the GIAB calls. True genotype is taken as GIAB, percentage correct is shown in last row and column. The input data set is ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing.

McCortex genotyping appears to be slightly better than Cortex genotyping for both the separate call sets and the intersected call sets ($p<0.01$ for both; two-sided Fisher Exact Test). It has the additional benefits that we can run on variants we have not called, we can use the uncleaned graph and memory is much lower. Additionally it takes as input any compressed or uncompressed bcf/vcf.

Genotyping uses less memory than Cortex as it only loads the required $k$mers (see Table 5.1). It also uses the uncleaned graph which removes the sensitivity of genotyping to error cleaning. The genotyping

model makes accommodations for $k$mers from sequencing error so is unaffected when using an uncleaned graph. Using the GIAB variant set as a set of known variants, we could have genotyped NA12778 sample graph against the calls to recover them without doing a variant discovery step.

5.1.6 Human genome assembly

Finally we ran assembly on the linked de Bruijn graphs generated from the NA12878 whole genome data. Results for assembly ($k = 31, 61$) with and without links, and without bubble popping are shown in Table 5.9. In Table 5.10 we show assembly metrics for $k = 61$ after doing bubble popping on the graph (with and without links).

	Unitigs (k=31)	Unitigs (k=61)	SE-Links (k=31)	SE-Links (k=61)	PE-Links (k=31)	PE-Links (k=61)
Time (hh:mm)	1:06	1:28	0:43	1:00	1:11	0:57
Memory (GiB)	30	60	85	117	58	117
No. contigs:	46,350,016	14,992,708	15,193,976	5,627,848	17,451,022	6,021,878
Sum length (Gb):	3.9	3.7	6.2	6.2	6.3	6.3
Min length:	31	61	31	61	31	61
Max length:	25,217	44,436	28,633	49,042	28,522	49,042
Mean length:	85	247	406	1,094	360	1,048
Median length:	34	85	88	501	71	433
N50:	221	940	1,331	2,365	1,354	2,410
NG50:	552	1,333	2,560	4,470	2,642	4,643

Table 5.9: De novo assembly of ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing. Wall clock time (hours:minutes) and peak memory (GiB) required for McCortex contig assembly step are shown – time is rounded to nearest minute and processes were run with 12 threads. The run times are in addition to the initial graph construction steps in Table 5.1. 'Unitigs' is de Bruijn graph unitigs, 'SE-links' is using links from single-end reads and 'PE-links' is using links from paired-end reads in the genome assembly.

	Bubbles popped	
	Plain (k=61)	PE-Links (k=61)
Time (hh:mm)	0:43	0:50
Memory (GiB)	116.6	117.2
No. contigs:	7,360,025	4,047,647
Sum length (Gb):	4.0	4.6
Min length:	61	61
Max length:	44,437	56,503
Mean length:	537	1,131
Median length:	117	258
N50:	2,624	3,661
NG50:	3,604	5,508

Table 5.10: De novo assembly of ∼50X 2x100bp PCR-free Illumina Platinum Genomes whole genome sequencing, after doing bubble popping on the graph. Wall clock time (hours:minutes) and peak memory (GiB) required for McCortex contig assembly step are shown – time is rounded to nearest minute and processes were run with 12 threads. The run times are in addition to the initial graph construction steps in Table 5.1. These assemblies are carried out after doing bubble popping on the graph before generating links. 'Plain' is de Bruijn graph assembly without links, 'SE-links' is using links from single-end reads and 'PE-links' is using links from paired-end reads in the genome assembly. Table 5.9 shows assembly metrics without bubble popping.

These assemblies represent an over-estimation of genome size and true assembly NG50, as we are assembling up until junctions which can generate contigs with large overlaps. Additionally, bubbles can create two contigs that are identical, except for the bubble allele used as the contig seed. This is an artefact of assembling a diploid genome (as discussed in Section 1.5.3), and also of errors resulting from confusing uncleaned sequencing errors and paralogous sequence. These mistakes would usually be addressed in the contig scaffolding step. To illustrate the effect of bubbles on assembly, we can look at the effect of bubble popping on the assembly at $k = 61$ using paired reads for links, without bubble popping (Table 5.9, last column), and with bubble popping (Table 5.9, last column). Bubble popping causes a reduction by a third of both the total assembled genome size (6.3Gbp to 4.6Gbp) and number of contigs (6 million to 4 million). The bubble popping step removed 1,484,423 bubbles from the graph where all branches were ≤ 71 *k*mers, involving in the removal of 62,704,368 *k*mers.

We can compare these values to those generated by other whole genome assemblers, such as SOAPdenovo (which uses a de Bruijn graph), which achieved a contig N50 of 7.4kb and 5.9kb (Li *et al.*, 2010b). Jackman *et al.* (2017) assembled human whole genome data, consisting of 250bp paired-end reads, with various assemblers and reported their NG50 values as: ABySS 1.0 (Simpson *et al.*, 2009): 30kb, ABySS 2.0 (Jackman *et al.*, 2017): 21kb, DISCOVAR de novo [2]: 82kb, BCALM2 (Chikhi *et al.*, 2016): 1.2kb, MEGAHIT (Li *et al.*, 2015): 8.2kb, Minia (Chikhi and Rizk, 2013): 4.8kb, SGA (Simpson and Durbin, 2012): 7.9kb, SOAPdenovo2 (Luo *et al.*, 2012): 3.8kb. The range of values – 1.2kb to 80kb – is similar

[2] *DISCOVAR de novo* is developed out of *Discovar* the de novo local assembly variant caller (Weisenfeld *et al.*, 2014)

to the range of NG50 values we saw for McCortex and SGA in the simulation on a single 1Mb region of human (Section 4.2.4) and in the whole genome assembly above. The wide range of metrics from individual assemblers indicates that the k parameter can be as important as the choice of assembler. For further discussion on comparing assemblers, including for different genomes, see Section 1.5.4.

5.2 Bacteria: *Klebsiella pneumoniae*

Klebsiella pneumoniae is a gram-negative bacteria that usually lives harmlessly in the mouth and gut of humans. However in the event of a weakened immune system, it can establish pathogenic colonies in the lung leading to inflammation and bleeding. It is also found in some cases of urinary tract infections and antibiotic pan-resistent strains of *K. pneumoniae* have been found in patients (Elemam *et al.*, 2009). This clinical significance makes it an interesting bacterial sample on which to evaluate McCortex.

We used Illumina MiSeq (2x151bp paired-end reads, 93X) and HiSeq (2x301bp paired-end reads, 42X) data from Sheppard *et al.* (2016) and the reference from NCBI (*GCF_000016305.1_ASM1630v1*), which is 5,315,120 bases long. McCortex was assessed in assembly and small and large variant calling. McCortex, Cortex (Iqbal *et al.*, 2012), Platypus (Rimmer *et al.*, 2014) and FreeBayes (Garrison and Marth, 2012) calls were used to compare variant calling. To validate our calls, we used a high quality PacBio (SMRT) assembly (Sheppard *et al.*, 2016) which has been finished using the Illumina reads with Quiver (Chin *et al.*, 2013) to correct small errors.

5.2.1 Building the graph with McCortex

McCortex requires overlapping paired end reads to be merged before running the read threading step (to generate links). As this problem has been solved by several programs, we have not re-implemented this feature into McCortex. Instead a third party program should be run by the user first. This is not currently part of the standard McCortex pipeline as it depends on the sequence preparation and platform. The Illumina data we used has a fragment length distribution that means some read pairs may overlap. Ideally we would merge them for optimal results. Instead we proceeded just using single-ended reads to make graph links.

	Uncleaned	Cleaned	
$k$mers ($k = 31$)	14,887,495	5,532,318	(37%)
$k$mers ($k = 61$)	15,862,168	5,537,956	(35%)
$k$mers with links ($k = 31$)	7,337	1,472	(20%)
number of links ($k = 31$)	235,810	41,050	(17%)
$k$mers with links ($k = 61$)	3,284	625	(19%)
number of links ($k = 61$)	68,293	13,538	(20%)

Table 5.11: *K. pneumoniae* graph and link cleaning. The input data set is Illumina 93X MiSeq 2x151bp reads and 42X HiSeq 2x301bp reads from Sheppard *et al.* (2016). McCortex treated reads as single-ended. The graph was cleaned with a cleaning threshold of < 7 for both $k = 31, 61$.

We built a single graph from the MiSeq and HiSeq data (at $k = 31, 61$). Low quality bases (less than phred score 10) were set to `N`, PCR-duplicate removal was not performed. $k$mers were cleaned with thresholds of < 7 ($k = 31$ and 61). Table 5.11 shows metrics for graph construction. Cleaning removed $\sim$65 % of $k$mers, resulting in graphs of $\sim$5.5 million $k$mers. Cortex chose to clean at < 14. Single-ended read links were generated and cleaned at < 7 ($k = 31$ and 61), which reduced the number of $k$mers with links from 7,337 and 3,284 to 1,472 and 625 ($k = 31$ and $k = 61$ respectively). Cleaning links with a threshold of < 7, for sequencing data with a mean read length of 135 and using Equation 2.27, the longest link we can expect is 188 bases. Therefore we will be unable to resolve any perfect repeats longer than 188 bases.

5.2.2 De novo assembly

Metric	$k = 31$ without links	$k = 31$ with links
Number of contigs	5,021	4,339
Sum of lengths	11,208,857	16,034,059
Min contig length	31	31
Max contig length	104,811	132,595
Median contig length	37	64
Mean contig length	2,232	3,695
N50	23,002	35,086

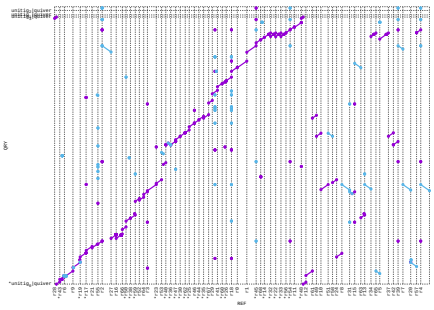
Table 5.12: *K. pneumoniae* McCortex assembly metrics for non-redundant contigs with and without links. The input data set is 151bp 2x93X MiSeq and 2x301bp 42X HiSeq. McCortex treated reads as single-ended and assembled at $k = 31$.

Metric	$k = 31$ without links	$k = 31$ with links
Contigs selected	146	77
Sum of lengths	5,676,068	5,663,695
Min contig length	22,801	48,635
Max contig length	104,811	132,595
Median contig length	33,140	67,956
Mean contig length	38,877	73,554
N50	40,675	72,239

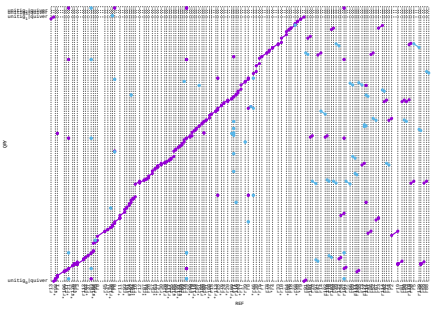
Table 5.13: Filtered *K. pneumoniae* McCortex assembly metrics for contigs assembled with and without links. Contigs from Table 5.12 were filtered by taking the longest contigs that make up $\sim 5.7 Mbp$ of sequence. The input data set is 2x151bp 93X MiSeq and 2x301bp 42X HiSeq. McCortex treated reads as single-ended and assembled at $k = 31$.

We used McCortex to construct contigs at $k = 31$ with and without using links. After contig generation, contigs which are exact substrings of others are removed. Assembly metrics for the non-redundant contigs are shown in Table 5.12. We sorted the contigs by length and selected the N longest contigs such that their lengths summed to approximate the reference length (shown in Table 5.13). In the absence of a reference, an estimate of genome size can be made from the $k$mer coverage distribution. Many of the short contigs that were not selected may be due to uncleaned sequencing error, particularly at sites of PCR-duplicates. Non-clonal bacterial samples or low frequency mutations in the sample will produce variants which will generate additional short contigs.

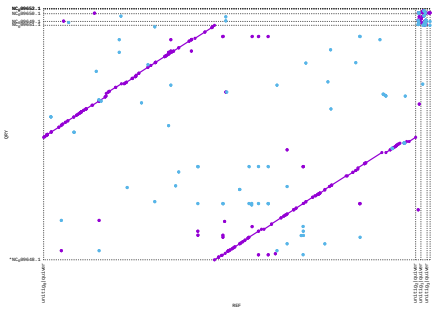
MUMMER (Kurtz *et al.*, 2004) plots of the two sets of contigs and NCBI reference each aligned to the PacBio assembly are shown in Figure 5.3. Overall the contigs generated with links do not appear to generate more mis-assemblies, indicating that contigs that could not be placed on the diagonal are perhaps short duplicate contigs with small mismatches vs. larger contigs in the set.



(a) PacBio assembly vs. contigs (with links).



(b) PacBio assembly vs. contigs (no links).



(c) Reference vs. PacBio assembly

Figure 5.3: MUMMER plots of *K. pneumoniae* assembly with/without links at $k = 31$. Assembled contigs are rearranged by MUMMER to maximise the contigs on the diagonal. Forward strand is shown in purple, reverse in light blue. A perfect assembly would appear as a single line across the diagonal. The input data set is Illumina 93X MiSeq 2x151bp reads and 42X HiSeq 2x301bp reads from Sheppard *et al.* (2016). McCortex treated reads as single-ended and assembled at $k = 31$.

Links have made an obvious improvement in the size of the contigs assembled. The assembly could possibly be improved by removing spurious bubbles in the graph – we did not try running the bubble popping step before generating links and assembling the contigs. If the bubbles are caused by sequencing errors, PCR duplicate removal could help reduce the number of sequencing errors that survive error cleaning.

5.2.3 Calling small events

To assess McCortex on calling small variants (SNPs, indels) in bacterial data, we ran the bubble and breakpoint callers. Results were also generated with Freebayes (Garrison and Marth, 2012), Platypus (Rimmer *et al.*, 2014) and Cortex (Iqbal *et al.*, 2012) to compare[3].

When calling with the McCortex bubble caller both the reference and sample were specified as haploid

[3]Scripts to re-run can be found in the *results/klebsiella/kleb_pneumoniae* of the McCortex repository

samples. Since we are calling between two haploid samples all Bubble and Breakpoint calls should be biallelic, with one allele in the reference and one in the sample. Flanks were required to map to the reference with a MAPQ$\geq$ 30. The breakpoint caller required a run of 5 $k$mers to flank a variant. McCortex was run at $k = 31$ and $k = 61$ and the results were merged. Reads were treated as single-ended.

SNP truth set

To assess the sensitivity of our variant calls, we needed to generate a truth set. MUMMER (Kurtz *et al.*, 2004) creates a global alignment of two whole genomes, and can be used to call SNPs. Since the Illumina reads and PacBio assembly are made from the same strain, variants in the Illumina reads should also be in the PacBio assembly. We ran MUMMER to compare the reference to the PacBio assembly and call SNPs. The Illumina reads were mapped to the reference using BWA MEM (Li, 2013). SNPs called by MUMMER were kept if they had 90% coverage on the non-reference allele in the mapped reads and if there was only one SNP called at that position (i.e. they were biallelic). The numbers of SNPs called by MUMMER are shown in Table 5.14.

Step	Remaining SNPs	
MUMMER SNPs called	35,169	
Removed multiallelics	33,344	(94.8%)
Removed > 10% non-ALT	31,145	(88.6%)

Table 5.14: SNP set generated from Illumina reads, PacBio assembly and the *K. pneumoniae* reference using MUMMER to compare the assembly and reference and BWA to map reads to the reference. Multiallelics and sites with more than 10% read coverage on the reference allele were filtered out.

Validating calls

Variants calls were validated by mapping the non-reference allele, along with 50 bases flanking either side from the reference genome, to the PacBio assembly. If the contig mapped, we checked if the allele matched the reference and if the adjacent bases mapped the correct distance apart (for indels)[4]. Validated calls were permitted to have insertions/deletions/SNPs in their flanks versus the reference. Contigs were considered as unmapped if they had a MAPQ $<$ 20 – supplementary and secondary mappings were ignored. The rate of validation was only reported for variants that mapped back onto the reference. Larger calls are less likely to map back to the reference.

[4]Analysis used *pysam*: `https://github.com/pysam-developers/pysam`

Small events results

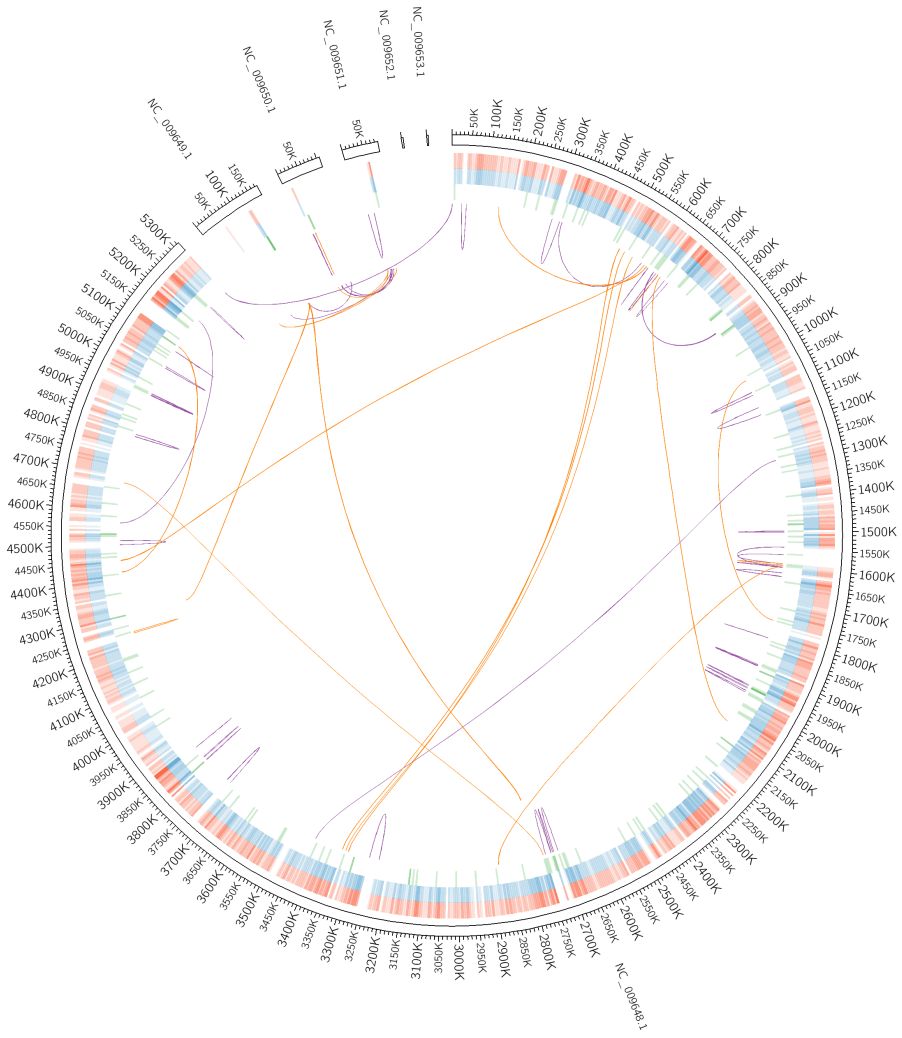


Figure 5.4: McCortex Breakpoint calls made with links at $k = 31$ on Illumina reads from *Klebsiella pneumoniae*, illustrated with Circos plot (Krzywinski *et al.*, 2009) generated by McCortex. Genome coordinates are written on the outside. The first three tracks show events in 5kbp bins. SNPs/MNPs are shown in red. Small insertions are deletions (<200bp) are shown in blue and green respectively. Purple lines indicate breakpoint causing larger indel events. Orange lines indicate breakpoints where the opposite strands join in the sample, which indicates an inverted section was moved or copied. There are 16,705 SNPs/MNPs, 17,217 small insertions, 165 small deletions, 74 breakpoints between the same strand, 39 that jump strands.

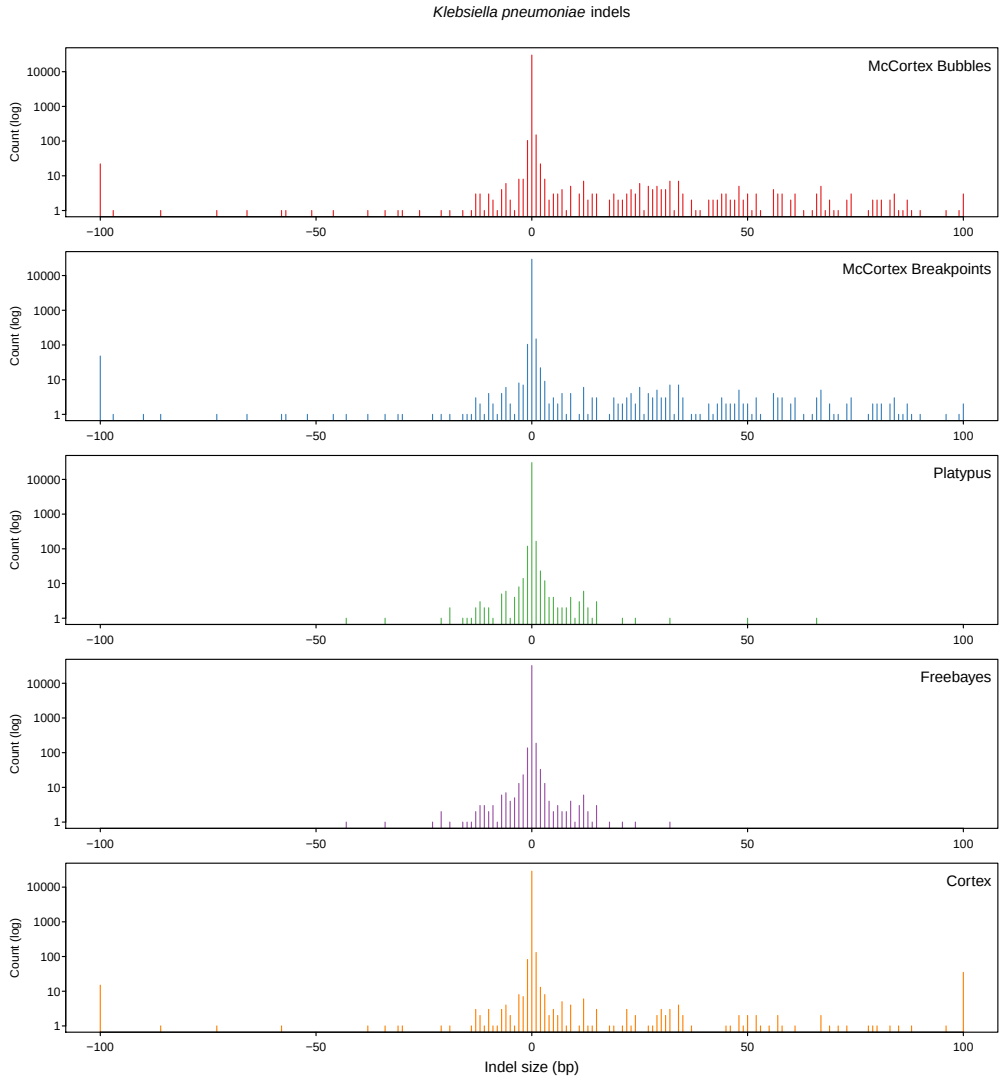


Figure 5.5: Indel calls in *Klebsiella pneumoniae*. 'Indel size' is defined as $length(ALT) - length(REF)$ where REF, ALT are the reference and alternative alleles. McCortex calls were the union of those made at $k = 31, 61$ using links. The input data set is 151bp 93X MiSeq and 301bp 42X HiSeq single-ended reads

.

Table 5.15 shows, for each calling method, the power to recover MUMMER SNPs and the rate of mismatch of the calls that mapped to the PacBio assembly. Calls that mapped and validated against the PacBio assembly were decomposed into SNPs, indels and complex events using `vcfallelicprimitives` from *vcflib*[5] – they are summarised in Table 5.16. The indel size distribution for the validated and decomposed variants is shown in Figure 5.5. A Circos plot (Krzywinski *et al.*, 2009) of McCortex Breakpoint calls is shown in 5.4.

[5] *vcflib*:`https://github.com/ekg/vcflib`

Caller	SNP Power (%)	All mapped calls FPR (%)
BWA + FreeBayes	99.5	0.15
BWA + Platypus	95.8	0.15
Cortex $k = 31, 61$	94.5	1.25
McCortex bubbles $k = 31, 61$	96.1	1.43
McCortex bubbles $k = 31, 61$ + links	96.2	1.43
McCortex breakpoints $k = 31, 61$	95.7	0.88
McCortex breakpoints $k = 31, 61$ + links	95.8	0.90

Table 5.15: SNP calling in *K. pneumoniae* with multiple variant callers. SNP Power is how many of the MUMMER SNP calls each caller made. False positive Rate (FPR) is calculated from the fraction of calls that map and don't match the PacBio assembly. The input data set is Illumina 93X MiSeq 2x151bp reads and 42X HiSeq 2x301bp reads from Sheppard *et al.* (2016). McCortex treated reads as single-ended.

	Validated Calls		
Caller	SNPs	Indels	Affected bases
BWA + Freebayes	*32,325	488	33,764
BWA + Platypus	30,472	414	31,785
Cortex	28,877	414	69,083
McCortex bubbles	30,086	561	62,130
McCortex breakpoints	29,522	*570	*102,259

Table 5.16: Validated calls in *Klebsiella pneumoniae* with multiple variant callers. Once calls were validated by mapping to a PacBio-based assembly, we decomposed variants using `vcfallelicprimitives` from *vcflib*. The highest value for each column is highlighted with *. 'Affected bases' is the sum of the length of longest allele for each variant. The input data set is Illumina 93X MiSeq 2x151bp reads and 42X HiSeq 2x301bp reads from Sheppard *et al.* (2016). McCortex treated reads as single-ended and called at both $k = 31$, $k = 61$, (with links) taking the union of resulting calls.

McCortex makes small improvements on Cortex, with the Breakpoint caller having a better discovery power and false positive rate. As in the human calls, the Bubble Caller finds more SNPs than the Breakpoint Caller. For both Breakpoint and Bubble callers, links appear to only add 0.1% to the sensitivity when calling SNPs.

McCortex Bubble and Breakpoint callers beat other callers for the number of indels called as well as for the total affected bases. The breakpoint caller finds three times as many affected bases as mapping based calling.

The false discovery rate for McCortex calls when calling in haploid samples may be improved by genotyping the reference and samples as diploids and removing variants where either is called as heterozygous. This would help to remove calls resulting from sequencing error (heterozygous in the sample) or paralogous sequence (heterozygous in the sample or reference graph).

5.2.4 Large events

Finally to assess the ability of McCortex to call very large events, we called bubbles using links and a maximum allele length of 200,000 bp. All calls are biallelic since we have specified both the reference and sample colours as haploid. Calls were filtered down to those that had at least 100bp length difference between allele lengths. We made 36,346 bubble calls, 92 had length difference greater than 100bp, which was reduced to 62 when we removed duplicate calls. Duplicate calls result from walking a bubble in opposite directions.

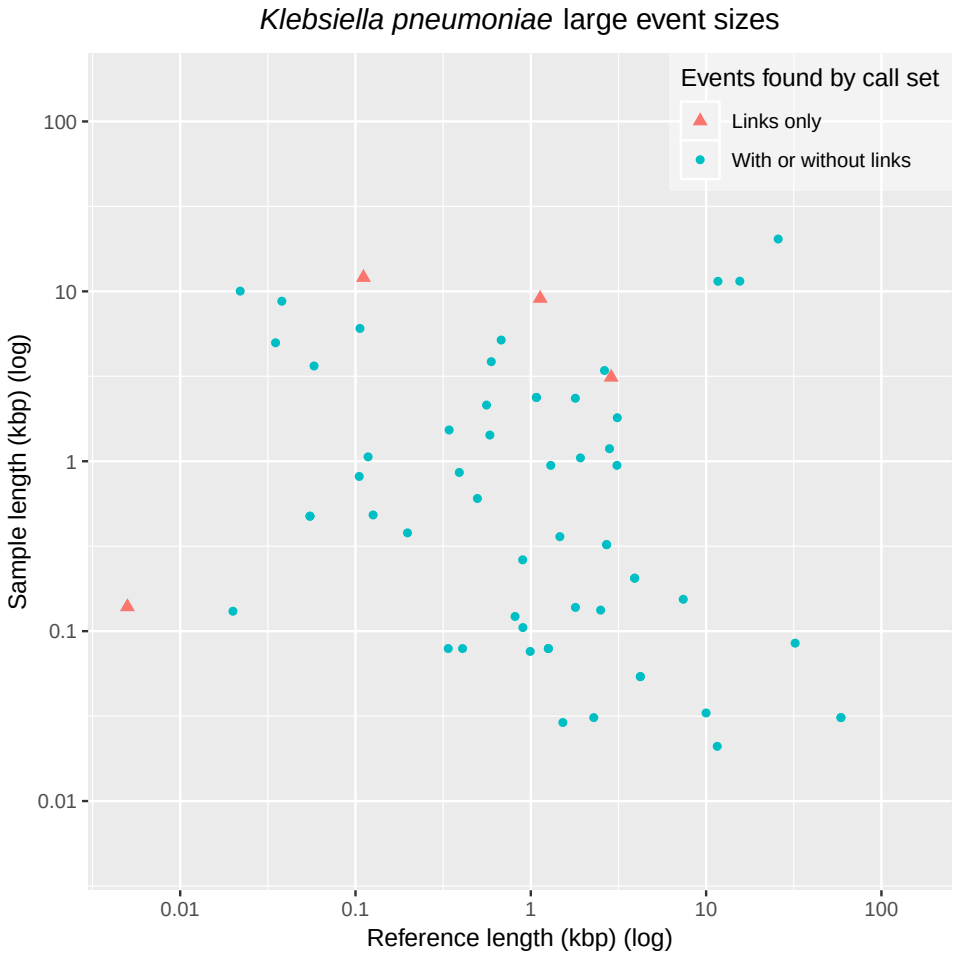


Figure 5.6: *K. pneumoniae*: event sizes of validated bubble calls where alleles have at least 100bp difference in length. Calls were made with McCortex bubble caller ($k = 31$) with and without links. Input data sets are 2x151bp 93X MiSeq and 2x301bp 42X HiSeq. Reads were treated as single-ended. Calls were validated by mapping alleles with flanks to the sample PacBio assembly. Triangles indicate calls found only with links, circles indicate calls found with or without using de Bruijn Graph links.

Reference length (bp)	Sample length (bp)	Affected bases
111	12,057	12,057
1,131	9,083	1,131
2,874	3,120	3,120
5	139	139

Table 5.17: Validated large events found in *K. pneumoniae* with McCortex Bubble caller that could be found only when using a de Bruijn Graph with Links. 'Affected bases' is the sum of the length of longest allele for each variant. The input data set is Illumina 93X MiSeq 2x151bp reads and 42X HiSeq 2x301bp reads from Sheppard *et al.* (2016). McCortex treated reads as single-ended and called at both $k = 31$, $k = 61$, (with links) taking the union of resulting calls.

These large event calls were validated by mapping both alleles (with their flanks) to both the reference used in calling and to the PacBio sample assembly. Of the 62 calls, 2 did not validate as large events but instead as clustered SNPs. Both of these unvalidated calls contained an allele that mapped perfectly to the reference and to the PacBio assembly with single base mismatches, and a second allele that mapped the other way around. All the remaining 60 calls consisted of one allele that mapped perfectly (exact match) to the reference and one allele that mapped perfectly to the PacBio assembly. The allele that did not map perfectly either did not map at all or mapped with insertions, deletions and/or soft clipping. The sizes of the validated calls are shown in Figure 5.6. Calling without links found 56 of the 60 validated calls as well as the 2 unvalidated calls. The calls missed when not using links (see Table 5.17) included the second and sixth longest sample alleles (12kbp and 9kbp respectively).

Using link information clearly improves our ability to call large events. Using single-ended reads of 150 and 300 bases and $k = 31$ we are able to identify and recover 140,054 bases in the sample that are missing from the reference. This is 17% more bases than we can recover without using links (115,655 bases). Additionally, these events are resolved to base pair resolution.

The sum of reference allele bases is 297,312 with links and 293,191 without – they are very similar because we do not use links with the reference sample. Adding links to the reference graph would be a relatively simple improvement. By interrogating the de Bruijn Graph of the reference, we find that 1.8% of 31-mers in the *K. pneumoniae* reference occur more than once (in the human reference *hg19*, 3% of 31-mers occur more than once).

5.3 Discussion

McCortex, like other graph and *k*mer-based variant callers, has limited power in finding small scale variation such as isolated SNPs and small indels. However as shown in *K. pneumoniae*, McCortex decomposed calls can affect three times as many bases as those generated with the latest mapping

based callers. We are able to find insertions into the reference of up to ∼12kbp and deletions of up to ∼60kbp.

Merging overlapping read pairs would allow us to use the read pairing when adding links and should give an improvement in the size and number of events that we can call. The breakpoint caller should give larger calls than those we produced with the bubble caller, since it does not need to assemble the reference allele. Instead it only needs to rejoin the reference as it exploits the fact that the reference has already been assembled.

Applying bubble popping to our data could have improved or worsened our ability to call large events in human and bacteria. Merits and demerits of bubble popping on assembly lengths are discussed in Section 4.2.2.

Multithreading implemented by McCortex has greatly improved the speed of constructing and operating on de Bruijn Graphs when running on real data. Our high memory server has 1TiB of RAM and 64 cores, and is used exclusively for high memory jobs. If jobs on the server utilise the same proportion of the available CPU and memory, a 150GiB RAM job should use ∼10 cores. Doing so ensures that the available CPU is fully utilised. By exploiting multiple CPU cores, jobs finish faster and we can release memory resources sooner. When using single threaded Cortex, CPU resources were seriously under utilised, leaving 90% of processing power unused. The result was memory being tied up for longer and delaying compute jobs.

Chapter 6

Discussion

In chapter 1 we reviewed the literature on genome sequencing, assembly and variant calling, highlighting the limitations of both reference-biased variant calling and de Bruijn Graphs.

In chapter 2 we introduced a new data structure, the Linked de Bruijn Graph, which used an augmented de Bruijn Graph to implement a string graph.

In chapter 3 we showed how the Linked de Bruijn Graph can be implemented and set out the tools we have developed to construct and clean graphs.

In chapter 4 we described various algorithms that can exploit the Linked de Bruijn Graph to assemble genomes, call variants and correct sequencing errors in reads. We showed that they are viable using simulated data.

In chapter 5 we applied our methods to human and *K. pneumoniae* sequencing data, showing that de Bruijn Graph links improve error correction, assembly and indel calling.

In this final chapter we have discussed the strengths and weaknesses of our algorithms, competing algorithms and some alternative ideas.

6.1 Link annotations

Linked de Bruijn Graphs allow annotation of a de Bruijn Graph with sequence data from multiple sources including sequence not used to build the initial graph. Links can be used seamlessly with multiple samples simultaneously, supporting multi-coloured de Bruijn Graphs. Memory and construction time are both linear in complexity and it is possible to use this data structure with mammalian samples using modern

hardware. There is clearly scope for improvement in implementation to reduce both run time and memory usage, and in this section we discuss some of the alternatives.

6.1.1 Links vs. paired $k$mers

Paired $k$mers as implemented in the SPAdes assembler (Medvedev *et al.*, 2011) (discussed in Section 1.5) are tolerant of mistakes between the $k$mers, whereas links are not as tolerant of errors. Sequence must be corrected to be a perfect match to the de Bruijn Graph in order to be converted to links and used in a Linked de Bruijn Graph. Long reads from sequencing platforms with high error rates are likely to contain at least one uncorrectable error, which would interrupt long link generation. This means that using links on such a platform would not exploit all the connectivity information in reads – longer distance connectivity would be lost due to the high error rate. This is similar to the observation that not all paired-end reads are joined by gap filling steps (see Section 5.1.1) and therefore some mate pair information is lost. Paired $k$mers still store connectivity information in the graph even if the path between the $k$mers cannot be resolved.

Paired $k$mers also have the same memory requirements regardless of the gap between the $k$mers. Links on the other hand require more memory for longer connectivity.

Paired $k$mers are not suitable for multi-sample genome comparison as they separate the samples in the graph. Links on the other hand do not stop the de Bruijn Graph collapsing down at areas of sequence identity, but instead aid the resolution of collapsed repeats during graph analysis.

6.1.2 Encoding links as succinct trees

Links attached to a $k$mer form a tree of diverging junction choices. With complete link information, the number of leaves equals the number of occurrences of the $k$mer in the de Bruijn Graph. Our implementation does not store links as trees, instead each link attached to a $k$mer is stored separately as a string of junction choices. The early junction choices that are shared by most links are stored redundantly, clearly missing some opportunities to optimise our memory usage. The simple use-case for a better data structure is when links have been generated and we need only a static data structure without coverage information. This could be used when reading and not altering the graph, for example when assembling or calling variants. For this we could store the link-tree for each $k$mer using a binary encoding, where each vertex is represented by four bits – each indicating whether it is followed by a $\{A, C, G, T\}$ base in the tree (see Figure 6.1). Vertices are visited in pre-order – visit a vertex, then each of its children. Leaf vertices are represented by four zeros. Whether this would be more efficient has yet to be seen using

real data. If we want to store counts with the graph we can do so with an array (of length equal to the number of vertices) that visits the tree vertices in the same order.

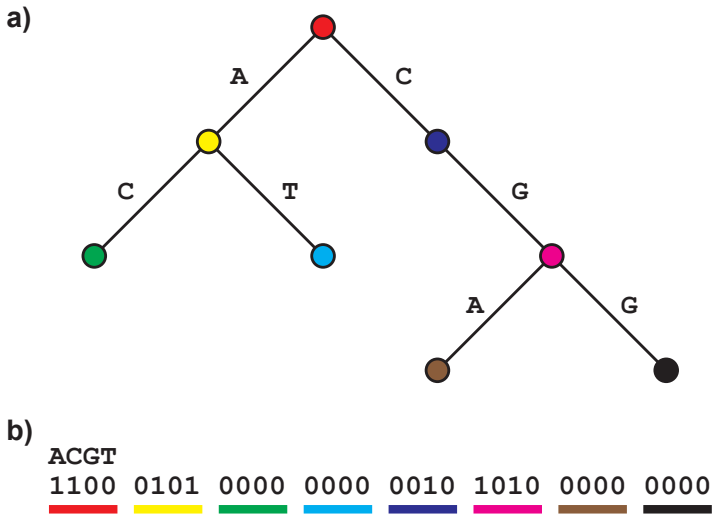


Figure 6.1: Binary encoding of a tree, which could be used to store the tree of links associated with a given $k$mer. a) Link tree where vertices are junctions in the graph and edges represent junction choices. The links represented are: AC,AT,CGA,CGG b) Encoding of tree from (a), each vertex is represented by four bits indicating if they have an $\{A, C, G, T\}$ branch below them. vertices are visited in pre-order – visit a vertex then each of its children. Leaf vertices are encoded as four zeros.

Alternatively links could be stored using generative encodings with bit trees and arithmetic coding as was used by Kingsford and Patro (2015) to compress reads against a de Bruijn Graph. This approach requires all reads to be sorted first and we would still need a hash table to map starting $k$mers to a tree of links starting from that $k$mer.

Our implementation stores links with $k$mers in the de Bruijn Graph. Although dependent on $k$mer size and genome complexity, the number of $k$mers with links attached is much smaller than the total number of $k$mers in the de Bruijn Graph. If we instead used a separate smaller hash table to map $k$mers to links, we could reduce memory usage by approximately 20GiB ($8 \times G$, where G is total $k$mers).

6.1.3 Storing links as ranked paths

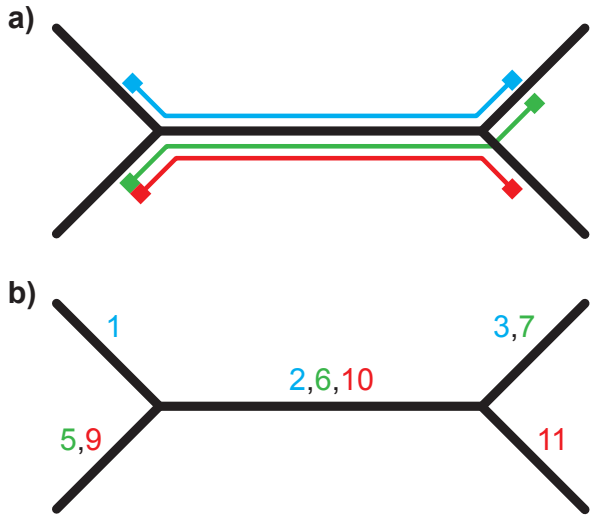


Figure 6.2: An alternative link representation: storing links with ranked paths. The de Bruijn Graph is shown in a simplified form showing the edges but omitting the vertices – black lines illustrate a run of *k*mers (a unitig). (a) Conventional link annotation on a de Bruijn Graph: a bi-directional pair of links are shown by each coloured line. Squares indicate *k*mers storing link information. (b) Alternative link annotation for the same Linked de Bruijn Graph shown in (a), storing links as consecutive numbers inserted into sets associated with unitigs. Links add an incrementing number to each of the unitigs they visit in order.

Links could be encoded by storing, for each unitig, a set of *link path integers*. Links add an incrementing number to each of the unitigs they visit in order (see Figure 6.2). A greedy algorithm could be used to pick the starting number for each link when it is added so as to avoid links interfering with other local links. One advantage of such an annotation is that it allows us to edit the graph after adding links. This would improve certain functions that alter the graph (such as bubble popping, see Section 4.2.3), allowing them to use read information.

The concept of threading reads through a de Bruijn Graph and storing them this way was proposed by Holley *et al.* (2017) who implemented *DARRC* – an alignment-free and reference-free compression program for highly redundant sequence data from related samples and species (pan-genomic sequence data). *DARRC* uses a *guided de Bruijn Graph* – an annotated coloured de Bruijn Graph – to losslessly encode and decode sequences. This graph is constructed by threading reads and storing their paths in the graph. More information is stored than in a Linked de Bruijn Graph, since it includes read starts and ends. *DARRC* uses minimizers to merge reads, and uses de Bruijn Graph colours to partition the graph into separate reads – the coloured de Bruijn Graph is implemented with a *Bloom Filter Trie* and can be updated with new sequences. The succinct data structure used by *DARRC* is very elegant.

6.1.4 Alternative link encodings

De Bruijn graph links describe paths through the graph and can therefore be simply represented as a set of DNA sequences. Therefore we can use existing methods to store them compressed on disk with random access with methods such as DEEZ (Hach *et al.*, 2014). Benoit *et al.* (2015) also used read paths through a probabilistic de Bruijn Graph for reference-free read compression, implemented as *LEON*.

Sirén (2017) have already proposed using an FM-index to store paths through a variant graph. The run-length encoding (RLE) used in many FM-index implementations would reduce the memory required to store many redundant junction choices in links. Using an FM-index to store links would remove the need for a new entry for sub-paths, as they could be found as substrings of the full link path.

Huang and Liao (2016) claimed to combine string graphs with de Bruijn Graphs in the *StriDe* assembler by treating the SGA FM-index as a de Bruijn Graph and generating $k$mer statistics from it. This construct is built for single bacterial genome assembly and they show this works well for that use case, however it is unlikely to scale to a mammalian genome let alone multiple genomes as the Linked de Bruijn Graph is designed for. Interestingly, whereas StriDe brings de Bruijn features to a string graph, the Linked de Bruijn Graph does the opposite.

The LOGAN graph (Bolger *et al.*, 2017) augmented a succinct de Bruijn Graph by storing read 'routing' information on each $k$mer that a read passes through. This stores the same connectivity information as links, along with read starts and ends. Read starts/ends are not usually informative to assembly however they allow LOGAN to losslessly recover all reads used to construct the graph. Storing connectivity information on each vertex does not scale well with long reads – the memory requirement is $O(N * R)$ where N is the number of reads and R is the read length – however the memory impact is reduced by sorting reads locally and using Run Length Encoding (RLE). This means that unlike with links, memory usage scales with coverage, even beyond the point of adding new connectivity information. As LOGAN is not designed to facilitate multi-sample (pan-genome) graphs, it is not clear how this approach would work for such graphs which require not compacting the de Bruijn Graph to facilitate loading individual graphs into a coloured de Bruijn Graph. The authors do not discuss graph traversal or how to use read information in assembly. Although the authors do not go into detail about how they construct graphs, the memory sizes of their graphs are very impressive.

6.2 Variant calling

Although $k$mer-based methods have reduced sensitivity for calling SNPs compared to mapping-based approaches, we have demonstrated that McCortex has better power on calling large events (see Section 5.2.3).

There are some graph sub-structures from which we cannot resolve variants – an example is given in Figure 6.3. Such an example is likely to occur in low complexity repeat regions of the genome, which we know to be prone to mutation. These are the regions where graph-based approaches ought to have an advantage over mapping based approaches.

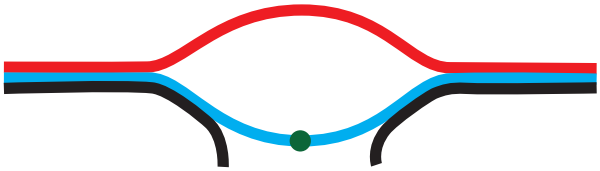


Figure 6.3: Some variants cannot be assembled if we start assembly from the flanking $k$mers. The reference genome is shown in red, a single sample using blue and black (repeats in the genome). If we attempt to assemble the sample allele from either of the flanking $k$mers, we hit forks that cannot be resolved. If instead we seed assembly from the green $k$mer in the middle, we are able to assemble a contig that includes the sample allele and flanking sequence. This contig can then be mapped to a reference to find the reference allele.

We propose a new variant caller[1] (the 'breakout caller') which seeds assembly from 'novel $k$mers' and assembles out from them. As it assembles it marks $k$mers as visited. Novel $k$mers marked as visited are not used to seed assembly. Assembly continues until we meet the reference or another sample for a given number of bases. This idea was briefly discussed in Section 4.4.2.

When calling with a reference 'novel $k$mers' would be those not in the reference and we assemble until we hit the reference at both sides in consistent places. Without a reference, 'novel $k$mers' are those that appear in some but not all samples. In this setting we must assemble contigs in one or more samples and map the reads to an assembled genome in order to place them.

The advantage to this approach is that we give ourselves more possible seeds with which to reconstruct a variant. Both the bubble and breakpoint callers required assembly to start at either side of the variant. This new variant caller would resolve bubbles of the structure illustrated in Figure 6.3 that is currently missed by both our calling algorithms.

[1] Thanks to constructive discussions with Kiran Garimella.

6.2.1 Variant calling development

Low complexity regions house a disproportionate amount of genetic variation (Weisenfeld *et al.*, 2014) and also suffer from poorer variant calling rates (Li, 2014b, Weisenfeld *et al.*, 2014). Therefore we argue that it is important to develop new variant calling approaches for these low complexity regions.

In this thesis, we presented two approaches for variant calling in (Linked) de Bruijn Graphs. Firstly *complex bubble calling* finds, for all $k$mer pairs (K_1, K_2) in the graph, all paths shorter than some threshold that start at K_1 and finish at K_2. This algorithm finds bubbles that themselves contain bubbles – simplistic bubble calling would only find the elementary bubbles in this scenario. Through mapping we place these bubbles onto a reference genome. Secondly, we presented *breakpoint calling* which leverages a reference genome to find samples that diverge from and rejoin the reference.

Once placed on a reference, we convert variant calls to VCF files. To genotype samples using only their $k$mer composition, we developed an algorithm to obtain allele $k$mer counts even when variants were close to each other ($< k$ bases between them) and implemented a diploid genotyping model using these $k$mer counts. This model would need further work to be extended to support multi-allelic sites.

In this work, we have not implemented a variant quality score ('QUAL' column in a VCF file). One could be calculated using a Hardy-Weinberg equilibrium model if multiple samples are available, or else using genotype likelihood, local graph complexity, variant mapping quality, $k$mer coverage and link counts.

Inversion calling could probably be improved for McCortex, as it has not been designed specifically for calling them. *Complex bubble calling* will discover inversions if links can resolve them (i.e. shorter than a read or read-pair) and *breakpoint calling* should discover ones longer than that (assuming a good reference is available). Perfect inversions generate particular graph motifs – bubbles at both breakpoints – which calling algorithms can be designed to find, for example TakeABreak from Lemaitre *et al.* (2014) (see Section 1.7). This algorithm could be re-implemented as part of McCortex and therefore exploit links when traversing the inversion.

An alternative approach to de novo variant assembly is to assemble each sample into a haploid genome and compare those genomes (Lv *et al.*, 2016). This provides the advantages of more rigorous assembly and longer contigs as well as lower memory requirements when comparing samples. However assembly throws away information that is present in the reads e.g. in bubble popping and collapsing down repeats. It is further limited by not sharing information across samples during assembly (such as using low coverage data across multiple individuals) where each sample provides weak evidence of a variant that is significant only when seen in multiple samples. Assembling sequence data into an intermediate graph representation

with coverage or confidence information may provide a solution, though this is very similar to the cleaned de Bruijn Graph with error corrected links proposed in this thesis.

6.3 Error correction

We implemented a simple error correction algorithm to facilitate filling gaps in reads when mapping them to the graph. Although developed for the read threading step, it can be used on its own for read error correction, allowing us to assess its effectiveness. We were impressed by its performance but found it not as good as existing approaches such as *bfc* (Li, 2015a) or as sophisticated as Quake (Kelley *et al.*, 2010). We saw much better performance in assembly when carrying out error correction with *bfc* before running McCortex (see Section 4.2.4).

6.4 Implementation

McCortex is implemented as a set of tools, each of which is implemented with multithreading, an improvement over its single threaded predecessor Cortex. Multithreading greatly improves CPU utilisation, speeding up jobs (measured by wall clock time) and reducing total memory utilisation (wall clock time $\times$ memory required). Additionally when only $k$mer/sample presence/absence information is needed (e.g. most operations traversing the graph), McCortex loads bit-packed $k$mer/colour presence matrices instead of the $k$mer-colour counts that Cortex did – a reduction of 32$\times$.

The data structures used are optimised for speed and simplicity; there is clearly room for improvement in memory use. Existing implementations of low memory de Bruijn Graphs previously discussed, such as BCALM, could be used for unitig construction of a single sample. Loading a unitig graph takes substantially less memory than a $k$mer graph, which would help those operations that do not need to operate on individual $k$mers: traversal and link generation need only unitig level access to the de Bruijn Graph; $k$mer cleaning requires individual $k$mer access and couldn't operate on a unitig graph. Making more use of compacted de Bruijn Graphs / unitig graphs would require mapping reads to these graphs as well as developing methods to merge unitig graphs to create multi-coloured unitig graphs.

Link construction – by far the most memory demanding step – could be disk based, writing links to disk as they are generated from reads. Disk-based sorting could then be used to group links that start from the same $k$mer and create an indexed link file that could be directly queried on disk during graph traversal. Indexed link files could be merged and cleaned on disk as well. During graph traversal, caching

could be used to optimise performance when repeatedly fetching the same links from disk (e.g. when traversing through adjacent copies of a repeat or walking one of thousands of loaded bacteria genomes). Storing links entirely on disk would reduce Linked De Bruijn Graph memory usage down to that of a plain de Bruijn Graph.

Some regions of a genome graph can have very low complexity, resulting in a large number of links that ultimately do not help traversal as there is not enough information to resolve the local area. Storing these links therefore provides no benefit, so developing a method to strip all links on these unitigs could reduce memory meaningfully. This may slow the exponential increase in link memory required when reducing $k$mer size (shown in Section 2.7).

6.5 Assembly

McCortex was designed around comparing genomes, but the underlying Linked de Bruijn Graph could be a useful component of a genome assembly pipeline along with error cleaning and scaffolding with jump libraries.

De Bruijn Graphs discard connectivity information from reads. The strength of the string graph and Linked de Bruijn Graph is that they retain all the information that is in single-ended reads other than their start and end positions. This means that they are both useful data structures for genome assembly. However, the string graph has the advantage of avoiding using a read more than once in an assembly.

6.5.1 Competing assembly approaches: Faucet

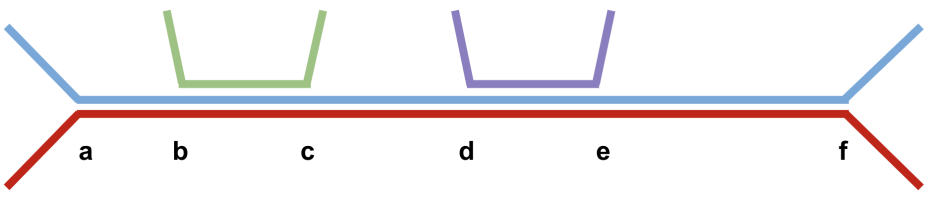


Figure 6.4: A nested tangle in a de Bruijn Graph that can be resolved by a Linked de Bruijn Graph but not by linked $k$mer pairs as in Rozov *et al.* (2018). Each colour is a separate contig used to construct the graph and junctions are labelled $\langle a, ..., f \rangle$. Link-$k$mer pairs would be made between adjacent junctions $\langle a, b, c \rangle$, $\langle b, c, d \rangle$, $\langle c, d, e \rangle$ and $\langle d, e, f \rangle$, which would not resolve the red contig. The linked de Bruijn Graph would reconstruct each of the four input contigs.

The *Faucet* assembler (Rozov *et al.*, 2018) resolves repeats using linked $k$mer pairs generated from adjacent pairs or triplets of junctions on a read. These $k$mer pairs are used to edit the graph and disentangle repeats. By tearing apart tangles in the unitig graph, it is no longer a de Bruijn Graph (two different vertices contain identical $k$mers) – this is the sample assembly idea as in Pevzner and Tang (2001). Linked $k$mer pairs hold strictly less information than links in a Linked de Bruijn Graph, and there are graph motifs that only Linked de Bruijn Graph can disambiguate (see Figure 6.4). *Faucet* is further limited to only resolving tangles with two incoming and two outgoing edges. It is not clear how this approach could be used in a coloured de Bruijn Graph to compare genomes once the de Bruijn Graph property has been broken.

6.6 Conclusion

In this thesis, we have presented a new data structure for de Bruijn Graph assembly – the Linked de Bruijn Graph. We have implemented in McCortex, a modular set of tools for working with Linked de Bruijn Graphs. We showed that links can be used in multicoloured graphs, and that using them reduces the sensitivity to the parameter k and improves assembly. We have presented algorithms that use the Linked de Bruijn Graph for assembly, variant discovery and read error correction. We also presented a new method for variant genotyping using a traditional de Bruijn Graph.

Overall, we found that links on a de Bruijn Graph do not have a meaningful effect on our discovery power for small variants (such as SNPs and short indels). We discovered that coverage, mappability, sequencing error in PCR duplicates and homopolymer runs can have a larger effect on our ability to discover these events than the low-complexity of the genome which links were designed to address. For larger events, such as significant rearrangements and larger indels, links do improve our sensitivity compared to traditional de Bruijn Graphs. McCortex has better sensitivity on large events (when we look at total bases affected) than both existing de novo (plain de Bruijn Graph) and mapping-based variant callers.

Appendices

Appendix A

List of publications

1. *Integrating long-range connectivity information into de Bruijn Graphs*, **I Turner***, K Garimella*, Z Iqbal, G McVean, *Bioinformatics* (2018)

2. *Stable recombination hotspots in birds*, S Singhal, E Leffler, K Sannareddy, **I Turner**, O Venn, D Hooper, A Strand, Q Li, B Raney, C Balakrishnan, S Griffith, G McVean, M Przeworski, *Science* (Nov 2015)

3. *Strong male bias drives germline mutation in chimpanzees*, O Venn, **I Turner**, Iain Mathieson, Natasja de Groot, Ronald Bontrop, G McVean, *Science* (June 2014)

4. *Contributions of intrinsic mutation rate and selfish selection to levels of de nov* HRAS mutations in the paternal germline', E Giannoulatou, G McVean, I B Taylor, S J McGowan, G J Maher, Z Iqbal, S P Pfeifer, **I Turner**, E M M Burkitt Wright, J Shorto, A Itani, K Turner, L Gregory, D Buck, E R Meyts, L H J Looijenga, B Kerr, A O M Wilkie, A Goriely *PNAS* (Dec 2013)

5. *High-throughput microbial population genomics using the Cortex variation assembler*, Z Iqbal, **I Turner**, G McVean *Bioinformatics* (Jan 2013)

6. *De novo assembly and genotyping of variants using colored de Bruijn Graphs*, Z Iqbal, M Caccamo, **I Turner**, P Flicek, G McVean *Nature Genetics* (Jan 2012)

Appendix B

Haploid assembly simulation

Haploid assembly simulation from Section 4.2.4 without using *bfc* error correction.

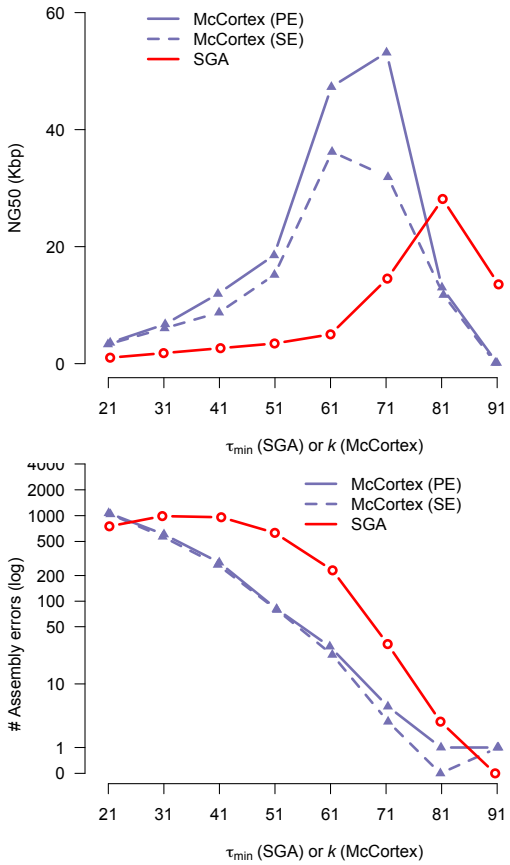


Figure B.1: Assembling simulated reads from a haploid genome with varied *k*mer-size. Source genome is haploid 1Mbp from GRCh37 chr22:17,000,000-17,999,999. Reads are 100bp paired-end with 200bp insert size and 0.5% sequence base error. McCortex is shown in purple – solid line is using paired-end reads (PE), dashed line is using single reads only (SE). SGA results are shown in red.

Appendix C

Coverage derivation

Given a distribution of coverage *covg* (*k*mer counts or link counts) and a user specified false negative rate (FNR) (default is $FNR = \frac{1}{1000}$), we fit the following model to pick a threshold T, such that fewer than FNR of elements with coverage T are due to error. Elements with coverage less than T are assumed to be due to error.

We assume *k*mers that occur three or fewer times are due to error and fit a Gamma-Poisson mixture distribution to the erroneous coverage (*errcovg*). We then find the lowest level of coverage T such that $errcovg(T)/covg(T) < FNR$. Specifically, we model the probability of seeing an erroneous k-mer with coverage x, $p(x)$, as a Poisson distribution with mean drawn from a gamma distribution:

$$p(x|\alpha,\beta) = \int \frac{\beta^\alpha}{\Gamma\alpha}\mu^{\alpha-1}e^{-\beta\mu}.\frac{e^{-\mu}\mu^x}{x!}d\mu \tag{C.1}$$

$$= \frac{\beta^\alpha}{\Gamma(\alpha)}\frac{1}{x!}\int \mu^{\alpha+x-1}e^{-\mu(1+\beta)}d\mu \tag{C.2}$$

$$= \frac{\beta^\alpha}{\Gamma(\alpha)}\frac{1}{x!}\frac{\Gamma(\alpha+x)}{(1+\beta)^{(\alpha+x)}} \tag{C.3}$$

(C.3) derived from (C.1) using the identity for the Gamma Function $\Gamma(t) = \int x^{t-1}e^{-x}dx$ and setting $x = \mu(1+\beta)$, $t = \alpha + x$, which gives:

$$\Gamma(\alpha+x) = \int (\mu(1+\beta))^{\alpha+x-1} e^{-\mu(1+\beta)} d(\mu(1+\beta)) \tag{C.4}$$

$$\Gamma(\alpha+x) = \int \mu^{\alpha+x-1}(1+\beta)^{\alpha+x-1} e^{-\mu(1+\beta)} d(\mu(1+\beta)) \tag{C.5}$$

$$\frac{\Gamma(\alpha+x)}{(1+\beta)^{\alpha+x}} = \int \mu^{\alpha+x-1} e^{-\mu(1+\beta)} d\mu \tag{C.6}$$

Since observed coverage of errors is conditional on having seen a $k$mer we assume $p(k|k>0,\mu) \sim p(k-1|k>0, \frac{\mu}{e^{-\mu}-1}-1)$. Assume $k$mers with coverage ≤ 3 are due to error. Use this to estimate $\hat{\alpha}$ using (C.3):

$$p(1)/p(0) = \frac{\left(\frac{\beta^\alpha}{\Gamma(\alpha)} \frac{1}{1!} \frac{\Gamma(\alpha+1)}{(1+\beta)^{(\alpha+1)}}\right)}{\left(\frac{\beta^\alpha}{\Gamma(\alpha)} \frac{1}{0!} \frac{\Gamma(\alpha)}{(1+\beta)^\alpha}\right)} \tag{C.7}$$

$$= \frac{\Gamma(\alpha+1)(1+\beta)}{\Gamma(\alpha)} \tag{C.8}$$

$$p(2)/p(1) = \frac{\Gamma(\alpha+2)(1+\beta)}{2\Gamma(\alpha+1)} \tag{C.9}$$

$$\frac{p(2)/p(1)}{p(1)/p(0)} = \frac{\left(\frac{\Gamma(\alpha+2)(1+\beta)}{2\Gamma(\alpha+1)}\right)}{\left(\frac{\Gamma(\alpha+1)(1+\beta)}{\Gamma(\alpha)}\right)} \tag{C.10}$$

$$= \frac{\Gamma(\alpha+2)\Gamma(\alpha)}{2\Gamma(\alpha+1)^2} \tag{C.11}$$

Using $covg(x)$ = Number of $k$mers with coverage x, estimate $\hat{\alpha}$ by finding value that minimises the absolute difference between distribution and coverage data.

$$\hat{\alpha} = \min_\alpha \left| \frac{p(2)/p(1)}{p(1)/p(0)} - \frac{covg(3)/covg(2)}{covg(2)/covg(1)} \right| \tag{C.12}$$

$$\hat{\alpha} = \min_\alpha \left| \frac{\Gamma(\alpha+2)\Gamma(\alpha)}{2\Gamma(\alpha+1)^2} - \frac{covg(3) \times covg(1)}{covg(2)^2} \right| \tag{C.13}$$

Now find $\hat{\beta}$ using $\hat{\alpha}$:

$$
\begin{aligned}
p(1)/p(0) &= covg(2)/covg(1) \\
\frac{\Gamma(\hat{\alpha}+1)(1+\hat{\beta})}{\Gamma(\hat{\alpha})} &= covg(2)/covg(1) \\
\hat{\beta} &= \frac{covg(2)\Gamma(\hat{\alpha})}{covg(1)\Gamma(\hat{\alpha}+1)} - 1
\end{aligned}
$$

Finally:

$$
\begin{aligned}
p(0|\hat{\alpha},\hat{\beta}).c_0 &= covg(1) \\
c_0 &= \frac{covg(1)}{(\hat{\beta}/(1+\hat{\beta}))^{\hat{\alpha}}} \\
&= covg(1).(\hat{\beta}/(1+\hat{\beta}))^{-\hat{\alpha}}
\end{aligned}
$$

Expected number of erroneous $k$mers with coverage x is given by:

$$
errcovg(x) = c_0 \times p(x-1|\hat{\alpha},\hat{\beta}) \tag{C.14}
$$

Appendix D

Greedy maximal substring algorithm

A greedy algorithm with time complexity $O(N^3)$ for finding the set of maximal substrings for any string.

Algorithm 5 $O(N^3)$ algorithm to find all maximal substrings

```
1:  procedure LENGTH(str)                          ▷ Returns the length of string str
2:  procedure SUBSTR(str,a,b)          ▷ Returns substring of str from a...b (inclusive, 0-based)
3:      return str[a ... b]
4:  procedure NOCCUR(str,s)          ▷ Returns no. of (possibly overlapping) occurrences of s in str

5:  procedure MaximalSubstring(str,i,j)
6:      n ← NOCCUR(str, SUBSTR(str, i, j))
7:      if n > 1 then
8:          while i > 0 && NOCCUR(str, SUBSTR(str, i − 1, j)) == n do
9:              i ← i − 1
10:         while j + 1 < LENGTH(str) && NOCCUR(str, SUBSTR(str, i, j + 1)) == n do
11:             j ← j + 1
12:         emit ⟨substr(str, i, j); n⟩

13: procedure FindMaximalSubstrings      ▷ Emits tuples of ⟨substring; count⟩ with duplicates
14:     for i in 0 ... (LENGTH(str) − 1) do
15:         for j in i ... (LENGTH(str) − 1) do
16:             MaximalSubstring(str,i,j)
```

Bibliography

Aardenne-Ehrenfest, V. T. and de Bruijn, N. G. (1951). Circuits and trees in oriented linear graphs. *Simon Stevin : Wis- en Natuurkundig Tijdschrift*, **28**, 203–217.

Aissani, B. (2014). Confounding by linkage disequilibrium. *Journal of human genetics*, **59**(2), 110–115.

Albers, C. A., Lunter, G., MacArthur, D. G., et al. (2011). Dindel: accurate indel calls from short-read data. *Genome research*, **21**(6), 961–973.

Alekseyev, M. and Pevzner, P. (2007). Colored de bruijn graphs and the genome halving problem. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **4**(1), 98–107.

Alipanahi, B., Muggli, M. D., Jundi, M., Noyes, N., and Boucher, C. (2017). Resistome SNP calling via read colored de bruijn graphs. *bioRxiv*, pages 156174+.

Almodaresi, F., Pandey, P., and Patro, R. (2017). Rainbowfish: A succinct colored de bruijn graph representation. *bioRxiv*, pages 138016+.

Bailey, J. A., Yavor, A. M., Massa, H. F., Trask, B. J., and Eichler, E. E. (2001). Segmental duplications: organization and impact within the current human genome project assembly. *Genome research*, **11**(6), 1005–1017.

Bankevich, A., Nurk, S., Antipov, D., et al. (2012). SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology : a journal of computational molecular cell biology*, **19**(5), 455–477.

Batzoglou, S. (2000). *Computational genomics: Mapping, comparison, and annotation of genomes.* Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

Batzoglou, S., Jaffe, D. B., Stanley, K., et al. (2002). ARACHNE: a whole-genome shotgun assembler. *Genome research*, **12**(1), 177–189.

Belton, J.-M., McCord, R. P., Gibcus, J. H., et al. (2012). Hi–C: A comprehensive technique to capture the conformation of genomes. *Methods*, **58**(3), 268–276.

Benoit, G., Lemaitre, C., Lavenier, D., et al. (2015). Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph. *BMC bioinformatics*, **16**.

Bentley, D. R., Balasubramanian, S., Swerdlow, H. P., et al. (2008). Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, **456**(7218), 53–59.

Bentley, J. L. and Sedgewick, R. (1997). Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 360–369, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Birmelé, E., Crescenzi, P., Ferreira, R., et al. (2012). Efficient bubble enumeration in directed graphs. In *SPIRE*, pages 118–129, Cartagena, Colombie.

Birol, I., Jackman, S. D., Nielsen, C. B., et al. (2009). De novo transcriptome assembly with ABySS. *Bioinformatics*, **25**(21), 2872–2877.

Bloom, B. H. (1970). Space/Time trade-offs in hash coding with allowable errors. *Commun. ACM*, **13**(7), 422–426.

Boetzer, M. and Pirovano, W. (2014). SSPACE-LongRead: scaffolding bacterial draft genomes using long read sequence information. *BMC Bioinformatics*, **15**(1), 211+.

Boisvert, S., Laviolette, F., and Corbeil, J. (2010). Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of computational biology : a journal of computational molecular cell biology*, **17**(11), 1519–1533.

Bolger, A. M., Denton, A. K., Bolger, M. E., and Usadel, B. (2017). LOGAN: A framework for LOssless graph-based ANalysis of high throughput sequence data. *bioRxiv*.

Bonfield, J. K. (2014). The scramble conversion tool. *Bioinformatics*, **30**(19), 2818–2819.

Bosi, E., Donati, B., Galardini, M., et al. (2015). MeDuSa: a multi-draft based scaffolder. *Bioinformatics (Oxford, England)*, **31**(15), 2443–2451.

Boucher, C., Bowe, A., Gagie, T., Puglisi, S. J., and Sadakane, K. (2014). Variable-Order de bruijn graphs.

Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de bruijn graphs. In B. Raphael and J. Tang, editors, *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bradnam, K. R., Fass, J. N., Alexandrov, A., et al. (2013). Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, **2**(1), 10+.

Brandt, D. Y., Aguiar, V. R., Bitarello, B. D., et al. (2015). Mapping bias overestimates reference allele frequencies at the HLA genes in the 1000 genomes project phase i data. *G3 (Bethesda, Md.)*, **5**(5), 931–941.

Brankovic, L., Iliopoulos, C. S., Kundu, R., et al. (2016). Linear-time superbubble identification algorithm for genome assembly. *Theoretical Computer Science*, **609**, 374–383.

Bresler, G., Bresler, M., and Tse, D. (2013). Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics*, **14**(5).

Brown, C. T., Howe, A., Zhang, Q., Pyrkosz, A. B., and Brom, T. H. (2012). A Reference-Free algorithm for computational normalization of shotgun sequencing data.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation.*

Burton, J. N., Adey, A., Patwardhan, R. P., et al. (2013). Chromosome-scale scaffolding of de novo genome assemblies based on chromatin interactions. *Nature biotechnology*, **31**(12), 1119–1125.

Cao, H., Wu, H., Luo, R., et al. (2015). De novo assembly of a haplotype-resolved human genome. *Nature biotechnology*, **33**(6), 617–622.

Cazaux, B., Lecroq, T., and Rivals, E. (2014a). From indexing data structures to de bruijn graphs. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, *Combinatorial Pattern Matching*, pages 89–99, Cham. Springer International Publishing.

Cazaux, B., Lecroq, T., and Rivals, E. (2014b). From indexing data structures to de bruijn graphs. In A. Kulikov, S. Kuznetsov, and P. Pevzner, editors, *Combinatorial Pattern Matching*, volume 8486 of *Lecture Notes in Computer Science*, pages 89–99. Springer International Publishing.

Cazaux, B., Lecroq, T., and Rivals, E. (2016). Linking indexing data structures to de bruijn graphs: Construction and update. *Journal of Computer and System Sciences*.

Chaisson, M. J., Brinza, D., and Pevzner, P. A. (2009). De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome research*, **19**(2), 336–346.

Chen, J., Rozowsky, J., Galeev, T. R., et al. (2016). A uniform survey of allele-specific binding and expression over 1000-Genomes-project individuals. *Nature communications*, **7**.

Chen, K., Wallis, J. W., McLellan, M. D., et al. (2009). BreakDancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods*, **6**(9), 677–681.

Chen, K., Chen, L., Fan, X., et al. (2014). TIGRA: A targeted iterative graph routing assembler for breakpoint assembly. *Genome Research*, **24**(2), 310–317.

Chen, L., Liu, P., Evans, T. C., and Ettwiller, L. M. (2017). DNA damage is a pervasive cause of sequencing errors, directly confounding variant identification. *Science*, **355**(6326), 752–756.

Chikhi, R. and Lavenier, D. (2009). Paired-end read length lower bounds for genome re-sequencing. *BMC Bioinformatics*, **10**(Suppl 13), O2+.

Chikhi, R. and Lavenier, D. (2011). Localized genome assembly from reads to scaffolds: Practical traversal of the paired string graph. In T. Przytycka and M.-F. Sagot, editors, *Algorithms in Bioinformatics*, volume 6833 of *Lecture Notes in Computer Science*, chapter 4, pages 39–48. Springer Berlin Heidelberg, Berlin, Heidelberg.

Chikhi, R. and Medvedev, P. (2014). Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, **30**(1), 31–37.

Chikhi, R. and Rizk, G. (2013). Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for molecular biology : AMB*, **8**(1), 22+.

Chikhi, R., Limasset, A., Jackman, S., Simpson, J., and Medvedev, P. (2014a). On the representation of de bruijn graphs. In R. Sharan, editor, *Research in Computational Molecular Biology*, volume 8394 of *Lecture Notes in Computer Science*, pages 35–55. Springer International Publishing.

Chikhi, R., Limasset, A., Jackman, S., Simpson, J., and Medvedev, P. (2014b). On the representation of de bruijn graphs.

Chikhi, R., Limasset, A., and Medvedev, P. (2016). Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics (Oxford, England)*, **32**(12), i201–i208.

Chin, C.-S., Alexander, D. H., Marks, P., et al. (2013). Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*, **10**(6), 563–569.

Chin, C.-S., Peluso, P., Sedlazeck, F. J., et al. (2016). Phased diploid genome assembly with single molecule Real-Time sequencing. *bioRxiv*, pages 056887+.

Chinwalla, A. T., Cook, L. L., Delehaunty, K. D., et al. (2002). Initial sequencing and comparative analysis of the mouse genome. *Nature*, **420**(6915), 520–562.

Church, D. M., Schneider, V. A., Graves, T., et al. (2011). Modernizing reference genome assemblies. *PLoS biology*, **9**(7), e1001091+.

Church, D. M., Schneider, V. A., Steinberg, K. M., et al. (2015). Extending reference assembly models. *Genome Biology*, **16**(1), 13+.

Clarke, J., Wu, H.-C. C., Jayasinghe, L., et al. (2009). Continuous base identification for single-molecule nanopore DNA sequencing. *Nature nanotechnology*, **4**(4), 265–270.

Clerry, J. G. (1984). Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, **C-33**(9), 828–834.

Cochrane, G., Cook, C. E., and Birney, E. (2012). The future of DNA sequence archiving. *GigaScience*, **1**(1), 2+.

Conway, T. C. and Bromage, A. J. (2011). Succinct data structures for assembling large genomes. *Bioinformatics*, **27**(4), 479–486.

Crusoe, M. R., Alameldin, H. F., Awad, S., et al. (2015). The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Research*, **4**.

Danecek, P., Auton, A., Abecasis, G., et al. (2011). The variant call format and VCFtools. *Bioinformatics*, **27**(15), 2156–2158.

de Bruijn, N. G. (1946). A combinatorial problem. *Koninklijke Nederlandsche Akademie Van Wetenschappen*, **49**(6), 758–764.

de Bruijn, N. G. (1975). Acknowledgement of priority to c. flye Sainte-Marie on the counting of circular arrangements of 2n zeros and ones that show each n-letter word exactly once. Technical report, Technological University Eindhoven.

Degner, J. F., Marioni, J. C., Pai, A. A., et al. (2009). Effect of read-mapping biases on detecting allele-specific expression from RNA-sequencing data. *Bioinformatics (Oxford, England)*, **25**(24), 3207–3212.

Deorowicz, S., Debudaj-Grabysz, A., and Grabowski, S. (2013). Disk-based k-mer counting on a PC. *BMC Bioinformatics*, **14**(1), 160+.

Deorowicz, S., Kokot, M., Grabowski, S., and Debudaj-Grabysz, A. (2015). KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, **31**(10), 1569–1576.

DePristo, M. A., Banks, E., Poplin, R., et al. (2011). A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature genetics*, **43**(5), 491–498.

Dilthey, A., Cox, C., Iqbal, Z., Nelson, M. R., and McVean, G. (2015). Improved genome inference in the MHC using a population reference graph. *Nat Genet*, **47**(6), 682–688.

Dobin, A., Davis, C. A., Schlesinger, F., et al. (2013). STAR: ultrafast universal RNA-seq aligner. *Bioinformatics (Oxford, England)*, **29**(1), 15–21.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn an efficient, generic c++ library for sequence analysis. *BMC bioinformatics*, **9**(1), 11+.

Drezen, E., Rizk, G., Chikhi, R., et al. (2014). GATB: Genome assembly & analysis tool box. *Bioinformatics*, **30**(20), 2959–2961.

Dutilh, B. E., Schmieder, R., Nulton, J., et al. (2012). Reference-independent comparative metagenomics using cross-assembly: crAss. *Bioinformatics (Oxford, England)*, **28**(24), 3225–3231.

Earl, D., Bradnam, K., St John, J., et al. (2011). Assemblathon 1: a competitive assessment of de novo short read assembly methods. *Genome research*, **21**(12), 2224–2241.

Eberle, M. A., Fritzilas, E., Krusche, P., et al. (2017). A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. *Genome research*, **27**(1), 157–164.

Edwards, A. (1990). Automated DNA sequencing of the human HPRT locus*1. *Genomics*, **6**(4), 593–608.

Eggertsson, H. P., Jonsson, H., Kristmundsdottir, S., et al. (2017). Graphtyper enables population-scale genotyping using pangenome graphs. *Nature genetics*, **49**(11), 1654–1660.

Eid, J., Fehr, A., Gray, J., et al. (2009). Real-time DNA sequencing from single polymerase molecules. *Science (New York, N.Y.)*, **323**(5910), 133–138.

El-Metwally, S., Zakaria, M., and Hamza, T. (2016). LightAssembler: fast and memory-efficient assembly algorithm for high-throughput sequencing reads. *Bioinformatics (Oxford, England)*, **32**(21), 3215–3223.

Elemam, A., Rahimian, J., and Mandell, W. (2009). Infection with panresistant klebsiella pneumoniae: A report of 2 cases and a brief review of the literature. *Clinical Infectious Diseases*, **49**(2), 271–274.

Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., and Woodhull, G. (2003). Graphviz and dynagraph - static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using phred. II. error probabilities. *Genome research*, **8**(3), 186–194.

Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using phred. i. accuracy assessment. *Genome research*, **8**(3), 175–185.

Fang, H., Wu, Y., Narzisi, G., et al. (2014). Reducing INDEL calling errors in whole genome and exome sequencing data. *Genome medicine*, **6**(10), 89+.

Farrant, G. K., Hoebeke, M., Partensky, F., et al. (2015). WiseScaffolder: an algorithm for the semi-automatic scaffolding of next generation sequencing data. *BMC bioinformatics*, **16**(1), 281+.

Farrer, R. A., Kemen, E., Jones, J. D. G., and Studholme, D. J. (2009). De novo assembly of the pseudomonas syringae pv. syringae b728a genome using Illumina/Solexa short sequence reads. *FEMS Microbiology Letters*, **291**(1), 103–111.

Fasulo, D., Halpern, A., Dew, I., and Mobarry, C. (2002). Efficiently detecting polymorphisms during the fragment assembly process. *Bioinformatics (Oxford, England)*, **18 Suppl 1**(suppl 1), S294–S302.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, volume 0, pages 390–398, Los Alamitos, CA, USA. IEEE.

Feuk, L., Carson, A. R., and Scherer, S. W. (2006). Structural variation in the human genome. *Nature reviews. Genetics*, **7**(2), 85–97.

Gagie, T., Navarro, G., Puglisi, S. J., and Sirén, J. (2015). Relative compressed suffix trees.

Gallant, J. (1980). On finding minimal length superstrings. *Journal of Computer and System Sciences*, **20**(1), 50–58.

Garrison, E. and Marth, G. (2012). Haplotype-based variant detection from short-read sequencing.

Garrison, E., Sirén, J., Novak, A. M., et al. (2017). Sequence variation aware genome references and read mapping with the variation graph toolkit. *bioRxiv*.

Garrison, E., Sirén, J., Novak, A. M., et al. (2018). Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*.

Genovese, G., Handsaker, R. E., Li, H., et al. (2013). Using population admixture to help complete maps of the human genome. *Nature genetics*, **45**(4).

Gurevich, A., Saveliev, V., Vyahhi, N., and Tesler, G. (2013). QUAST: quality assessment tool for genome assemblies. *Bioinformatics (Oxford, England)*, **29**(8), 1072–1075.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1 edition.

Hach, F., Numanagic, I., and Sahinalp, S. C. (2014). DeeZ: reference-based compression by local assembly. *Nat Meth*, **11**(11), 1082–1084.

Hajirasouliha, I., Hormozdiari, F., Alkan, C., et al. (2010). Detection and characterization of novel sequence insertions using paired-end next-generation sequencing. *Bioinformatics (Oxford, England)*, **26**(10), 1277–1283.

Hernandez, D., François, P., Farinelli, L., Østerås, M., and Schrenzel, J. (2008). De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, **18**(5), 802–809.

Holley, G., Wittler, R., and Stoye, J. (2016). Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for molecular biology : AMB*, **11**.

Holley, G., Wittler, R., Stoye, J., and Hach, F. (2017). Dynamic Alignment-Free and Reference-Free read compression. In S. C. Sahinalp, editor, *Research in Computational Molecular Biology*, pages 50–65, Cham. Springer International Publishing.

Homer, N., Merriman, B., and Nelson, S. F. (2009). BFAST: an alignment tool for large scale genome resequencing. *PloS one*, **4**(11), e7767+.

Hsi-Yang Fritz, M., Leinonen, R., Cochrane, G., and Birney, E. (2011). Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome research*, **21**(5), 734–740.

Huang, L., Popic, V., and Batzoglou, S. (2013). Short read alignment with populations of genomes. *Bioinformatics*, **29**(13), i361–i370.

Huang, Y.-T. T. and Liao, C.-F. F. (2016). Integration of string and de bruijn graphs for genome assembly. *Bioinformatics (Oxford, England)*, **32**(9), 1301–1307.

Hubbard, T., Barker, D., Birney, E., et al. (2002). The ensembl genome database project. *Nucleic acids research*, **30**(1), 38–41.

Hunt, M., Kikuchi, T., Sanders, M., et al. (2013). REAPR: a universal tool for genome assembly evaluation. *Genome biology*, **14**(5), R47+.

Idury, R. M. and Waterman, M. S. (1995). A new algorithm for DNA sequence assembly. *Journal of computational biology : a journal of computational molecular cell biology*, **2**(2), 291–306.

Ip, C. L. C., Loose, M., Tyson, J. R., et al. (2015). MinION analysis and reference consortium: Phase 1 data release and analysis [version 1; referees: 2 approved]. *F1000Research*, **4**(1075).

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. (2012). De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature Genetics*, **44**(2), 226–232.

Iqbal, Z., Turner, I., and McVean, G. (2013). High-throughput microbial population genomics using the cortex variation assembler. *Bioinformatics*, **29**(2), 275–276.

Iu, Florent'ev, V. L., Khorlin, A. A., Khrapko, K. R., and Shik, V. V. (1988). Determination of the nucleotide sequence of DNA using hybridization with oligonucleotides. a new method. *Doklady Akademii nauk SSSR*, **303**(6), 1508–1511.

Jackman, S. D., Vandervalk, B. P., Mohamadi, H., et al. (2017). ABySS 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome research*, **27**(5), 768–777.

Jacobs, P. A., Wilson, C. M., Sprenkle, J. A., Rosenshein, N. B., and Migeon, B. R. (1980). Mechanism of origin of complete hydatidiform moles. *Nature*, **286**(5774), 714–716.

Jaffe, D. B., Butler, J., Gnerre, S., et al. (2003). Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome research*, **13**(1), 91–96.

Jain, M., Koren, S., Miga, K. H., et al. (2018). Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature biotechnology*, **36**(4), 338–345.

Jones, N. C., Zhi, D., and Raphael, B. J. (2006). AliWABA: alignment on the web through an A-Bruijn approach. *Nucleic acids research*, **34**(Web Server issue).

Kahn, A. B. (1962). Topological sorting of large networks. *Commun. ACM*, **5**(11), 558–562.

Kelley, D. R., Schatz, M. C., and Salzberg, S. L. (2010). Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, **11**(11), R116+.

Kent, W. J., Sugnet, C. W., Furey, T. S., et al. (2002). The human genome browser at UCSC. *Genome Research*, **12**(6), 996–1006.

Kim, D., Pertea, G., Trapnell, C., et al. (2013). TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, **14**(4), R36+.

Kim, D., Langmead, B., and Salzberg, S. L. (2015). HISAT: a fast spliced aligner with low memory requirements. *Nature Methods*, **12**(4), 357–360.

Kingsford, C. and Patro, R. (2015). Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, **31**(12), 1920–1928.

Knuth, D. E. (1973). *The art of computer programming*. Addison-Wesley series in computer science and information processing. Addison-Wesley.

Korbel, J., Abyzov, A., Mu, X., et al. (2009). PEMer: a computational framework with simulation-based error models for inferring genomic structural variants from massive paired-end sequencing data. *Genome biology*, **10**(2), R23+.

Koren, S., Schatz, M. C., Walenz, B. P., et al. (2012). Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature Biotechnology*, **30**(7), 693–700.

Krzywinski, M. I., Schein, J. E., Birol, I., et al. (2009). Circos: An information aesthetic for comparative genomics. *Genome Research*, **19**(9), 1639–1645.

Kurtz, S., Phillippy, A., Delcher, A. L., et al. (2004). Versatile and open software for comparing large genomes. *Genome biology*, **5**(2), R12+.

Kurtz, S., Narechania, A., Stein, J. C., and Ware, D. (2008). A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC genomics*, **9**(1), 517+.

Lam, T. W., Li, R., Tam, A., et al. (2009). High throughput short read alignment via bi-directional BWT. In *Bioinformatics and Biomedicine, 2009. BIBM '09. IEEE International Conference on*, volume 0, pages 31–36, Los Alamitos, CA, USA. IEEE.

Lander, E. S., Linton, L. M., Birren, B., et al. (2001). Initial sequencing and analysis of the human genome. *Nature*, **409**(6822), 860–921.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with bowtie 2. *Nature Methods*, **9**(4), 357–359.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, **10**(3), R25–10.

Layer, R. M., Chiang, C., Quinlan, A. R., and Hall, I. M. (2014). LUMPY: a probabilistic framework for structural variant discovery. *Genome Biology*, **15**(6), R84+.

Lee, H., Gurtowski, J., Yoo, S., et al. (2014). Error correction and assembly complexity of single molecule sequencing reads. *bioRxiv*, pages 006395+.

Lee, S., Hormozdiari, F., Alkan, C., and Brudno, M. (2009). MoDIL: detecting small indels from clone-end sequencing with mixtures of distributions. *Nature methods*, **6**(7), 473–474.

Leggett, R. M. and MacLean, D. (2014). Reference-free SNP detection: dealing with the data deluge. *BMC Genomics*, **15**(Suppl 4), S10+.

Leggett, R. M., Ramirez-Gonzalez, R. H., Verweij, W., et al. (2013). Identifying and classifying trait linked polymorphisms in Non-Reference species by walking coloured de bruijn graphs. *PLoS ONE*, **8**(3), e60058+.

Leitch, I. J. (2007). Genome sizes through the ages. *Heredity*, **99**(2), 121–122.

Lemaitre, C., Ciortuz, L., and Peterlongo, P. (2014). Mapping-Free and Assembly-Free discovery of inversion breakpoints from raw NGS reads. In A.-H. Dediu, C. Martín-Vide, and B. Truthe, editors, *Algorithms for Computational Biology*, volume 8542 of *Lecture Notes in Computer Science*, pages 119–130. Springer International Publishing.

Li, D., Liu, C.-M., Luo, R., Sadakane, K., and Lam, T.-W. (2014). MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph.

Li, D., Liu, C.-M. M., Luo, R., Sadakane, K., and Lam, T.-W. W. (2015). MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics (Oxford, England)*, **31**(10), 1674–1676.

Li, H. (2012). Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, **28**(14), 1838–1844.

Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM.

Li, H. (2014a). Fast construction of FM-index for long sequence reads. *Bioinformatics (Oxford, England)*, **30**(22), 3274–3275.

Li, H. (2014b). Toward better understanding of artifacts in variant calling from high-coverage samples. *Bioinformatics (Oxford, England)*, **30**(20), 2843–2851.

Li, H. (2015a). BFC: correcting illumina sequencing errors. *Bioinformatics*, **31**(17), 2885–2887.

Li, H. (2015b). FermiKit: assembly-based variant calling for illumina resequencing data. *Bioinformatics*, **31**(22), 3694–3696.

Li, H. (2016). Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, **32**(14), 2103–2110.

Li, H. (2018). Minimap2: versatile pairwise alignment for nucleotide sequences.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, **25**(14), 1754–1760.

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

Li, H. and Homer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, **11**(5), 473–483.

Li, H., Ruan, J., and Durbin, R. (2008). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, **18**(11), 1851–1858.

Li, H., Handsaker, B., Wysoker, A., et al. (2009). The sequence Alignment/Map format and SAMtools. *Bioinformatics (Oxford, England)*, **25**(16), 2078–2079.

Li, R., Li, Y., Zheng, H., et al. (2010a). Building the sequence map of the human pan-genome. *Nat Biotech*, **28**(1), 57–63.

Li, R., Zhu, H., Ruan, J., et al. (2010b). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, **20**(2), 265–272.

Li, S., Li, R., Li, H., et al. (2013). SOAPindel: efficient identification of indels from short paired reads. *Genome research*, **23**(1), 195–200.

Lieberman-Aiden, E., van Berkum, N. L., Williams, L., et al. (2009). Comprehensive mapping of Long-Range interactions reveals folding principles of the human genome. *Science*, **326**(5950), 289–293.

Limasset, A., Cazaux, B., Rivals, E., and Peterlongo, P. (2016). Read mapping on de bruijn graphs. *BMC Bioinformatics*, **17**(1).

Liu, B., Yuan, J., Yiu, S.-M. M., et al. (2012). COPE: an accurate k-mer-based pair-end reads connection tool to facilitate genome assembly. *Bioinformatics (Oxford, England)*, **28**(22), 2870–2874.

Liu, B., Guan, D., Teng, M., and Wang, Y. (2015). rHAT: fast alignment of noisy long reads with regional hashing. *Bioinformatics*, pages btv662+.

Liu, B., Guo, H., Brudno, M., and Wang, Y. (2016). deBGA: read alignment with de bruijn graph-based seed and extension. *Bioinformatics (Oxford, England)*, **32**(21), 3224–3232.

Liu, Y., Schröder, J., and Schmidt, B. (2013). Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics (Oxford, England)*, **29**(3), 308–315.

Loman, N. J., Quick, J., and Simpson, J. T. (2015). A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature methods*, **12**(8), 733–735.

Lunter, G. and Goodson, M. (2011). Stampy: A statistical algorithm for sensitive and fast mapping of illumina sequence reads. *Genome Research*, **20**(6), 936–939.

Luo, R., Liu, B., Xie, Y., et al. (2012). SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, **1**(1), 18+.

Luo, R., Wong, T., Zhu, J., et al. (2013). SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner. *PLoS ONE*, **8**(5), e65632+.

Lv, Y., Liu, Y., and Zhao, H. (2016). mInDel: a high-throughput and efficient pipeline for genome-wide InDel marker development. *BMC genomics*, **17**(1).

Maciuca, S., del Ojo Elias, C., McVean, G., and Iqbal, Z. (2016). A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In M. Frith and C. N. Storm Pedersen, editors, *Algorithms in Bioinformatics*, volume 9838 of *Lecture Notes in Computer Science*, pages 222–233. Springer International Publishing.

Magoc, T., Pabinger, S., Canzar, S., et al. (2013). GAGE-b: an evaluation of genome assemblers for bacterial organisms. *Bioinformatics*, **29**(14), 1718–1725.

Magoč, T. and Salzberg, S. L. (2011). FLASH: fast length adjustment of short reads to improve genome assemblies. *Bioinformatics*, **27**(21), 2957–2963.

Maillet, N., Lemaitre, C., Chikhi, R., Lavenier, D., and Peterlongo, P. (2012). Compareads: comparing huge metagenomic experiments. *BMC bioinformatics*, **13 Suppl 19**(Suppl 19), S10+.

Maillet, N., Collet, G., Vannier, T., Lavenier, D., and Peterlongo, P. (2014). Commet: Comparing and combining multiple metagenomic datasets. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 94–98.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**(6), 764–770.

Marcus, S., Lee, H., and Schatz, M. C. (2014). SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics (Oxford, England)*, **30**(24), 3476–3483.

Maretty, L., Jensen, J. M. M., Petersen, B., et al. (2017). Sequencing and de novo assembly of 150 genomes from denmark as a population reference. *Nature*, **548**(7665), 87–91.

Margulies, M., Egholm, M., Altman, W. E., et al. (2005). Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, **437**(7057), 376–380.

Masada, T., Takasu, A., Shibata, Y., and Oguri, K. (2012). Clustering documents with maximal substrings. In R. Zhang, J. Zhang, Z. Zhang, J. Filipe, and J. Cordeiro, editors, *Enterprise Information Systems*, volume 102 of *Lecture Notes in Business Information Processing*, pages 19–34. Springer Berlin Heidelberg.

Mascher, M., Muehlbauer, G. J., Rokhsar, D. S., et al. (2013). Anchoring and ordering NGS contig assemblies by population sequencing (POPSEQ). *The Plant journal : for cell and molecular biology*, **76**(4), 718–727.

Masella, A. P., Bartram, A. K., Truszkowski, J. M., Brown, D. G., and Neufeld, J. D. (2012). PANDAseq: paired-end assembler for illumina sequences. *BMC Bioinformatics*, **13**(1), 31+.

Mccreight, E. M. (1976). A Space-Economical suffix tree construction algorithm. *J. ACM*, **23**(2), 262–272.

McVean, G. A., Altshuler Co-Chair, D. M., Durbin Co-Chair, R. M., et al. (2012). An integrated map of genetic variation from 1,092 human genomes. *Nature*, **491**(7422), 56–65.

Medvedev, P., Pham, S., Chaisson, M., Tesler, G., and Pevzner, P. (2011). Paired de bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. *Journal of computational biology : a journal of computational molecular cell biology*, **18**(11), 1625–1634.

Melsted, P. and Pritchard, J. K. (2011). Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, **12**(1), 333+.

Mikheyev, A. S. and Tin, M. M. Y. (2014). A first look at the oxford nanopore MinION sequencer. *Mol Ecol Resour*, **14**(6), 1097–1102.

Minkin, I., Pham, S., and Medvedev, P. (2017). TwoPaCo: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics (Oxford, England)*, **33**(24), 4024–4032.

Mitzenmacher, M. and Upfal, E. (2017). *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, New York, NY, USA, second edition.

Mohiyuddin, M., Mu, J. C., Li, J., et al. (2015). MetaSV: an accurate and integrative structural-variant caller for next generation sequencing. *Bioinformatics (Oxford, England)*, **31**(16), 2741–2744.

Muggli, M. D., Bowe, A., Noyes, N. R., et al. (2017). Succinct colored de bruijn graphs. *Bioinformatics (Oxford, England)*, **33**(20), 3181–3187.

Mulyukov, Z. and Pevzner, P. A. (2002). EULER-PCR: finishing experiments for repeat resolution. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, **1**, 199–210.

Myers, E. W. (1995). Toward simplifying and accurately formulating fragment assembly. *Journal of computational biology : a journal of computational molecular cell biology*, **2**(2), 275–290.

Myers, E. W. (2005). The fragment assembly string graph. *Bioinformatics*, **21**(suppl 2), ii79–ii85.

Myers, E. W., Sutton, G. G., Delcher, A. L., et al. (2000). A Whole-Genome assembly of drosophila. *Science*, **287**(5461), 2196–2204.

Nakamura, K., Oshima, T., Morimoto, T., et al. (2011). Sequence-specific error profile of illumina sequencers. *Nucleic Acids Research*, **39**(13), e90.

Narzisi, G., O'Rawe, J. A., Iossifov, I., et al. (2014). Accurate de novo and transmitted indel detection in exome-capture data using microassembly. *Nat Meth*, **11**(10), 1033–1036.

Nattestad, M. and Schatz, M. C. (2016). Assemblytics: a web analytics tool for the detection of variants from an assembly. *Bioinformatics*, **32**(19), 3021–3023.

Neale, D. B., Wegrzyn, J. L., Stevens, K. A., et al. (2014). Decoding the massive genome of loblolly pine using haploid DNA and novel assembly strategies. *Genome biology*, **15**(3), R59+.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, **48**(3), 443–453.

Nguyen, N., Hickey, G., Zerbino, D. R., et al. (2015). Building a pan-genome reference for a population. *Journal of computational biology : a journal of computational molecular cell biology*, **22**(5), 387–401.

Nielsen, C. B., Jackman, S. D., Birol, I., and Jones, S. J. (2009). ABySS-explorer: visualizing genome sequence assemblies. *IEEE transactions on visualization and computer graphics*, **15**(6), 881–888.

Nijkamp, J. F., Pop, M., Reinders, M. J. T., and de Ridder, D. (2013). Exploring variation-aware contig graphs for (comparative) metagenomics using MaryGold. *Bioinformatics*, **29**(22), 2826–2834.

Nikolenko, S. I., Korobeynikov, A. I., and Alekseyev, M. A. (2013). BayesHammer: Bayesian clustering for error correction in single-cell sequencing. *BMC genomics*, **14 Suppl 1**(Suppl 1), S7+.

Okanohara, D. and Sadakane, K. (2007). Practical entropy-compressed Rank/Select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 60–70, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Onodera, T., Sadakane, K., and Shibuya, T. (2013). Detecting superbubbles in assembly graphs. In A. Darling and J. Stoye, editors, *Algorithms in Bioinformatics*, volume 8126 of *Lecture Notes in Computer Science*, pages 338–348. Springer Berlin Heidelberg.

O'Rawe, J., Jiang, T., Sun, G., et al. (2013). Low concordance of multiple variant-calling pipelines: practical implications for exome and genome sequencing. *Genome medicine*, **5**(3), 28+.

Pandey, P., Bender, M. A., Johnson, R., and Patro, R. (2017a). deBGR: an efficient and near-exact representation of the weighted de bruijn graph. *Bioinformatics (Oxford, England)*, **33**(14).

Pandey, P., Bender, M. A., Johnson, R., and Patro, R. (2017b). A General-Purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 775–787, New York, NY, USA. ACM.

Parks, D. H., Imelfort, M., Skennerton, C. T., Hugenholtz, P., and Tyson, G. W. (2015). CheckM: assessing the quality of microbial genomes recovered from isolates, single cells, and metagenomes. *Genome research*, **25**(7), 1043–1055.

Paten, B., Novak, A., and Haussler, D. (2014). Mapping to a reference genome structure.

Paten, B., Eizenga, J. M., Rosen, Y. M., et al. (2018). Superbubbles, ultrabubbles, and cacti. *Journal of Computational Biology*, **25**(7), 649–663. PMID: 29461862.

Pellicer, J., Fay, M. F., and Leitch, I. J. (2010). The largest eukaryotic genome of them all? *Botanical Journal of the Linnean Society*, **164**(1), 10–15.

Pellow, D., Filippova, D., and Kingsford, C. (2016). Improving bloom filter performance on sequence data using k-mer bloom filters. In M. Singh, editor, *Research in Computational Molecular Biology*, volume 9649 of *Lecture Notes in Computer Science*, pages 137–151. Springer International Publishing.

Peng, Y., Leung, H., Yiu, S. M., and Chin, F. (2010). IDBA – a practical iterative de bruijn graph de novo assembler. In B. Berger, editor, *Research in Computational Molecular Biology*, volume 6044 of *Lecture Notes in Computer Science*, chapter 28, pages 426–440. Springer Berlin Heidelberg, Berlin, Heidelberg.

Pérez, N., Gutierrez, M., and Vera, N. (2016). Computational performance assessment of k-mer counting algorithms. *Journal of computational biology : a journal of computational molecular cell biology*, **23**(4), 248–255.

Pevzner, P. A. (1989). l-Tuple DNA sequencing: computer analysis. *Journal of biomolecular structure & dynamics*, **7**(1), 63–73.

Pevzner, P. A. (1995). DNA physical mapping and alternating eulerian cycles in colored graphs. *Algorithmica*, **13**(1), 77–105.

Pevzner, P. A. and Tang, H. (2001). Fragment assembly with double-barreled data. *Bioinformatics*, **17**(suppl 1), S225–S233.

Pevzner, P. A., Tang, H., and Waterman, M. S. (2001a). An eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, **98**(17), 9748–9753.

Pevzner, P. A., Tang, H., and Waterman, M. S. (2001b). A new approach to fragment assembly in DNA sequencing. In *RECOMB '01: Proceedings of the fifth annual international conference on Computational biology*, pages 256–267, New York, NY, USA. ACM.

Pevzner, P. A., Tang, H., and Tesler, G. (2004). De novo repeat classification and fragment assembly. *Genome Research*, **14**(9), 1786–1796.

Pop, M., Kosack, D. S., and Salzberg, S. L. (2004). Hierarchical scaffolding with bambus. *Genome research*, **14**(1), 149–159.

Pritt, J. and Langmead, B. (2018). FORGe: prioritizing variants for graph genomes. *bioRxiv*.

Prjibelski, A. D., Vasilinetc, I., Bankevich, A., et al. (2014). ExSPAnder: a universal repeat resolver for DNA fragment assembly. *Bioinformatics (Oxford, England)*, **30**(12), i293–i301.

Putnam, N. H., O'Connell, B. L., Stites, J. C., et al. (2016). Chromosome-scale shotgun assembly using an in vitro method for long-range linkage. *Genome Research*, **26**(3), 342–350.

Qin, J., Li, R., Raes, J., et al. (2010). A human gut microbial gene catalogue established by metagenomic sequencing. *Nature*, **464**(7285), 59–65.

Rana, S. B., Zadlock, F. J., Zhang, Z., Murphy, W. R., and Bentivegna, C. S. (2016). Comparison of de novo transcriptome assemblers and k-mer strategies using the killifish, fundulus heteroclitus. *PLoS ONE*, **11**(4), e0153104+.

Rausch, T., Emde, A.-K., Weese, D., et al. (2008). Segment-based multiple sequence alignment. *Bioinformatics*, **24**(16), i187–i192.

Rimmer, A., Phan, H., Mathieson, I., et al. (2014). Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nat Genet*, **46**(8), 912–918.

Rizk, G., Lavenier, D., and Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics (Oxford, England)*, **29**(5), 652–653.

Rizk, G., Gouin, A., Chikhi, R., and Lemaitre, C. (2014). MindTheGap: integrated detection and assembly of short and long insertions. *Bioinformatics*, **30**(24), 3451–3457.

Roberts, L. (1988). Carving up the human genome. *Science (New York, N.Y.)*, **242**(4883), 1244–1246.

Robertson, G., Schein, J., Chiu, R., et al. (2010). De novo assembly and analysis of RNA-seq data. *Nature Methods*, **7**(11), 909–912.

Robinson, J. T., Thorvaldsdóttir, H., Winckler, W., et al. (2011). Integrative genomics viewer. *Nature biotechnology*, **29**(1), 24–26.

Rozov, R., Goldshlager, G., Halperin, E., and Shamir, R. (2018). Faucet: streaming de novo assembly graph construction. *Bioinformatics (Oxford, England)*, **34**(1), 147–154.

Safonova, Y., Bankevich, A., and Pevzner, P. (2014). dipSPAdes: Assembler for highly polymorphic diploid genomes. In R. Sharan, editor, *Research in Computational Molecular Biology*, volume 8394 of *Lecture Notes in Computer Science*, pages 265–279. Springer International Publishing.

Sahlin, K., Vezzi, F., Nystedt, B., Lundeberg, J., and Arvestad, L. (2014). BESST–efficient scaffolding of large fragmented assemblies. *BMC bioinformatics*, **15**(1), 281+.

Salikhov, K., Sacomoto, G., and Kucherov, G. (2013). Using cascading bloom filters to improve the memory usage for de brujin graphs. In A. Darling and J. Stoye, editors, *Algorithms in Bioinformatics*, volume 8126 of *Lecture Notes in Computer Science*, pages 364–376. Springer Berlin Heidelberg.

Salmela, L. and Rivals, E. (2014). LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, **30**(24), 3506–3514.

Satya, R. V. V., Zavaljevski, N., and Reifman, J. (2012). A new strategy to reduce allelic bias in RNA-seq readmapping. *Nucleic acids research*, **40**(16), e127.

Schirmer, M., Ijaz, U. Z., D'Amore, R., et al. (2015). Insight into biases and sequencing errors for amplicon sequencing with the illumina MiSeq platform. *Nucleic acids research*, **43**(6), e37.

Schirmer, M., D'Amore, R., Ijaz, U. Z., Hall, N., and Quince, C. (2016). Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data. *BMC bioinformatics*, **17**.

Schneeberger, K., Hagmann, J., Ossowski, S., et al. (2009). Simultaneous alignment of short reads against multiple genomes. *Genome biology*, **10**(9), R98+.

Schöniger, M. and Waterman, M. S. (1992). A local algorithm for DNA sequence alignment with inversions. *Bulletin of mathematical biology*, **54**(4), 521–536.

Seemann, T. (2013). Ten recommendations for creating usable bioinformatics command line software. *GigaScience*, **2**(1).

Sheppard, A. E., Stoesser, N., Wilson, D. J., et al. (2016). Nested russian Doll-Like genetic mobility drives rapid dissemination of the carbapenem resistance gene blaKPC. *Antimicrobial Agents and Chemotherapy*, **60**(6), 3767–3778.

Silva, G. G. G., Cuevas, D. A., Dutilh, B. E., and Edwards, R. A. (2014). FOCUS: an alignment-free model to identify organisms in metagenomes using non-negative least squares. *PeerJ*, **2**.

Simpson, J. T. and Durbin, R. (2010). Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, **26**(12), i367–i373.

Simpson, J. T. and Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, **22**(3), 549–556.

Simpson, J. T., Wong, K., Jackman, S. D., et al. (2009). ABySS: a parallel assembler for short read sequence data. *Genome research*, **19**(6), 1117–1123.

Sirén, J. (2017). Indexing variation graphs. pages 13–27.

Sirén, J., Välimäki, N., and Mäkinen, V. (2014). Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, **11**(2), 375–388.

Solomon, B. and Kingsford, C. (2016). Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, **34**(3), 300–302.

Solomon, B. and Kingsford, C. (2018). Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *Journal of computational biology : a journal of computational molecular cell biology*.

Song, L., Florea, L., and Langmead, B. (2014). Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*, **15**(11), 509+.

Sović, I., Šikić, M., Wilm, A., et al. (2016). Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nature Communications*, **7**, 11307+.

Steinberg, K. M., Schneider, V. A., Graves-Lindsay, T. A., et al. (2014). Single haplotype assembly of the human genome from a hydatidiform mole. *Genome Research*, **24**(12), gr.180893.114–2076.

Stevenson, K. R., Coolon, J. D., and Wittkopp, P. J. (2013). Sources of bias in measures of allele-specific expression derived from RNA-sequence data aligned to a single reference genome. *BMC genomics*, **14**(1), 536+.

Sung, W.-K. K., Sadakane, K., Shibuya, T., Belorkar, A., and Pyrogova, I. (2015). An o(m log m)-Time algorithm for detecting superbubbles. *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM*, **12**(4), 770–777.

Szmulewicz, M. N., Novick, G. E., and Herrera, R. J. (1998). Effects of alu insertions on gene function. *Electrophoresis*, **19**(8-9), 1260–1264.

Talwalkar, A., Liptrap, J., Newcomb, J., et al. (2014). SMaSH: a benchmarking toolkit for human genome variant calling. *Bioinformatics (Oxford, England)*, **30**(19), 2787–2795.

Tettelin, H., Saunders, N. J., Heidelberg, J., et al. (2000). Complete genome sequence of neisseria meningitidis serogroup b strain MC58. *Science (New York, N.Y.)*, **287**(5459), 1809–1815.

The 1000 Genomes Project Consortium (2010). A map of human genome variation from population-scale sequencing. *Nature*, **467**(7319), 1061–1073.

Trapnell, C., Pachter, L., and Salzberg, S. L. (2009). TopHat: discovering splice junctions with RNA-seq. *Bioinformatics (Oxford, England)*, **25**(9), 1105–1111.

Turner, I., Garimella, K. V., Iqbal, Z., and McVean, G. (2018). Integrating long-range connectivity information into de bruijn graphs. *Bioinformatics*.

Tutte, W. T. and Smith, C. A. B. (1941). On unicursal paths in a network of degree 4. *The American Mathematical Monthly*, **48**(4), 233–237.

Ukkonen, E. (1992). Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, **92**(1), 191–211.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, **14**(3), 249–260.

Ulyantsev, V. I., Kazakov, S. V., Dubinkina, V. B., Tyakht, A. V., and Alexeev, D. G. (2016). MetaFast: fast reference-free graph-based comparison of shotgun metagenomic data. *Bioinformatics (Oxford, England)*, **32**(18), 2760–2767.

Uricaru, R., Rizk, G., Lacroix, V., et al. (2015). Reference-free detection of isolated SNPs. *Nucleic acids research*, **43**(2), e11.

Venter, J. C., Adams, M. D., Myers, E. W., et al. (2001). The sequence of the human genome. *Science*, **291**(5507), 1304–1351.

Vyahhi, N., Pyshkin, A., Pham, S., and Pevzner, P. (2012). From de bruijn graphs to rectangle graphs for genome assembly. In B. Raphael and J. Tang, editors, *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, chapter 20, pages 249–261. Springer Berlin Heidelberg, Berlin, Heidelberg.

Walker, B. J., Abeel, T., Shea, T., et al. (2014). Pilon: An integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PLOS ONE*, **9**(11), e112963+.

Weiner, P. (1973). Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA. IEEE Computer Society.

Weisenfeld, N. I., Yin, S., Sharpe, T., et al. (2014). Comprehensive variation discovery in single human genomes. *Nature Genetics*, **46**(12), 1350–1355.

Wetterstrand, K. A. (2016). DNA sequencing costs: Data from the NHGRI genome sequencing program (GSP).

Wick, R. R., Schultz, M. B., Zobel, J., and Holt, K. E. (2015). Bandage: interactive visualization of de novo genome assemblies. *Bioinformatics*, **31**(20), 3350–3352.

Ye, K., Schulz, M. H., Long, Q., Apweiler, R., and Ning, Z. (2009). Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics (Oxford, England)*, **25**(21), 2865–2871.

Younsi, R. and MacLean, D. (2015). Using 2k + 2 bubble searches to find single nucleotide polymorphisms in k-mer graphs. *Bioinformatics*, **31**(5), 642–646.

Zare, F., Dow, M., Monteleone, N., Hosny, A., and Nabavi, S. (2017). An evaluation of copy number variation detection tools for cancer using whole exome sequencing data. *BMC Bioinformatics*, **18**(1).

Zerbino, D. R. (2009). *Genome assembly and comparison using de Bruijn graphs*. Ph.D. thesis, Darwin College, University of Cambridge, European Molecular Biology Laboratory, Hinxton, Cambridge, CB10 1SD, UK.

Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, **18**(5), 821–829.

Zerbino, D. R., McEwen, G. K., Margulies, E. H., and Birney, E. (2009). Pebble and rock band: Heuristic resolution of repeats and scaffolding in the velvet Short-Read de novo assembler. *PLoS ONE*, **4**(12), e8407+.

Zhang, J., Kobert, K., Flouri, T., and Stamatakis, A. (2014a). PEAR: a fast and accurate illumina Paired-End reAd mergeR. *Bioinformatics (Oxford, England)*, **30**(5), 614–620.

Zhang, Q., Pell, J., Canino-Koning, R., Howe, A. C., and Brown, C. T. (2014b). These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure. *PLOS ONE*, **9**(7), e101271+.

Zhang, Z. D., Du, J., Lam, H., et al. (2011). Identification of genomic indels and structural variations using split reads. *BMC genomics*, **12**(1), 375+.

Zhi, D., Raphael, B. J., Price, A. L., Tang, H., and Pevzner, P. A. (2006). Identifying repeat domains in large genomes. *Genome biology*, **7**(1), R7+.

Zook, J. M., Chapman, B., Wang, J., et al. (2014). Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls. *Nature Biotechnology*, **32**(3), 246–251.