A thesis submitted to

University College London

for the degree

of

Doctor of Philosophy

High-Throughput Linkage Analysis Pipeline

Mehmet Tekman

August 2016

Declaration

I, Mehmet Tekman, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Publications Arising From This Thesis

[1] S. Drury, C. Boustred, **M. Tekman**, H. Stanescu, R. Kleta, N. Lench, L. S. Chitty, and R. H. Scott, "A novel homozygous ercc5 truncating mutation in a family with prenatal arthrogryposis-further evidence of genotype-phenotype correlation," *American Journal of Medical Genetics Part A*, vol. 164, no. 7, pp. 1777–1783, 2014.

[2] N. Mencacci, I. Rubio-Agusti, A. Zdebik, F. Asmus, M. Ludtmann, M. Ryten, V. Plagnol, A.-K. Hauser, S. a. Bandres-Ciga, C. Bettencourt, P. Forabosco, D. Hughes, M. Soutar, K. Peall, H. Morris, D. Trabzuni, **M. Tekman**, H. Stanescu, R. Kleta, M. Carecchio, G. Zorzi, N. Nardocci, B. Garavaglia, E. Lohmann, A. Weissbach, C. Klein, J. Hardy, A. Pittman, T. Foltynie, A. Abramov, T. Gasser, K. Bhatia, and N. Wood, "A Missense Mutation in KCTD17 Causes Autosomal Dominant Myoclonus-Dystonia," *American Journal of Human Genetics*, vol. 96, no. 6, pp. 938–947, Jun. 2015.

[3] P. Le Quesne Stabej, H. J. Williams, C. James, **M. Tekman**, H. C. Stanescu, R. Kleta, L. Ocaka, F. Lescai, H. L. Storr, M. Bitner Glindzicz, C. Bacchelli, G. S. Conway, and GOSgene, "STAG3 truncating variant as the cause of primary ovarian insufficiency," *Eur J Hum Genet*, vol. 24, no. 1, pp. 135–138, Jan. 2016.

[4] D. M. Rowczenio, D. S. Iancu, H. Trojer, J. A. Gilbertson, J. D. Gillmore, A. D. Wechalekar, **M. Tekman**, H. C. Stanescu, R. Kleta, T. Lane, P. N. Hawkins, and H. J. Lachmann, "Autosomal dominant familial mediterranean fever in northern european caucasians associated with deletion of p.m694 residue-a case series and genetic exploration," *Rheumatology*, 2016. [*In Press*]

Abstract

The new paradigm in genetics is more sequence analysis driven, but linkage studies are frequently adopted in parallel to pinpoint loci of interest within the vast torrent of sequence data. Linkage analysis alone can identify a single causative gene under the scope of a rare disease model for the relatively low cost of a genotyping array, and has the added advantage of reconstituting genotypes of individuals absent from the analysis via haplotype reconstruction.

Here we present our comprehensive linkage analysis pipeline consisting of a collection of well-established tools and utilities (GRR, Merlin, Alohomora) to perform analysis under all penetrance models (dominant/recessive, autosomal/X-linked) as well as large complex consanguineous pedigrees. Pre-analysis filtering selects a subset of markers indicative of informative meioses, and qualitative tests are performed to check for gender, relationship, and Mendelian inheritance consistency. A reliable lineup of linkage analysis suites (Allegro, GeneHunter, Simwalk) compute LOD scores to produce fast multi-core genome-wide and chromosome-specific linkage plots complete with sub-banding overlays and peak validation. Limitations in pedigree creation and post-analysis haplotype examination applications (HaploPainter) further prompted the development of a new visualization tool, HaploHTML5, built upon the latest advances in the HTML5 web schema. Pedigrees are drawn and analysed in-browser, and haploblock resolution is performed using the novel approach of a best-first path-finding algorithm (A*) implemented in pure JavaScript.

Small (<19-bit) pedigrees with an informative input set of 40,000 markers were processed in under 15 minutes (single core) and 5 minutes (multi-core). Complex (>19-bit, inbred) pedigrees required extensive code and platform-specific modifications to cater for outdated software (Allegro) which increased the single-core run-time to hours (19 to 23-bit) and days (>23-bit). Correct X-linked haploblock resolution was determined via HaploHTML5, and side-by-side case/control evaluation was facilitated.

Note: This thesis was built using LaTeX 2_ε under the *hyperref* package that converts all references to figures, tables, glossaries, acronyms, and citations into clickable hyperlinks within the PDF. It is therefore better viewed using software, than through a hard copy for ease of use, though page numbers to references are given where applicable.

Throughout the thesis, the first mention of each glossary term and program listing is marked in blue, and links to an index. Glossaries are indexed starting at page 220, and full program listings are given on page 78.

List of Figures

List of Tables

List of Code Listings

Source Code and Hosting

High-Throughput Linkage Analysis Pipeline:

- Source: `https://bitbucket.org/mtekman/linkage_pipeline`

HaploHTML5 Web Application:

- Source: `https://bitbucket.org/mtekman/haplo_html5`
- Hosted: `http://mtekman.bitbucket.org/haplo_html5/`

Contents

Acknowledgements

I dedicate this work to my family.

I would like to express my deepest gratitude to my extremely patient and supportive supervisors:

Dr. Horia Stanescu for his continuously awe-inspiring dedication to the tireless pursuit of knowledge and beauty, and whose mentoring and understanding has meant more to me than I'd willingly admit when sober.

Prof. Robert Kleta for his inexhaustible energy and lazer-like focus that has likely scorched entire forests in order to ensure that I never get lost in the woods for too long.

I would also like to give a great deal thanks to following people:

Dr. Monika Mozere my good colleague and desk-mate for her fearless upbeat confidence, and my much-needed window in the biologist's world.

Prof. Detlef Bockenhauer for his professionalism, politeness and the excellent barbecues.

Dr. Kevin Bryson for accommodating and supporting the research arrangement.

To the guys; Dr. Alan Medlar, Dr. Enriko Klootvijk, Dr. Greg Jacquillet, Dr. Anselm Zdebik, Dr. Stephen Walsh, Daan Viering, Chris Cheshire, and Evans Asowata.

To the girls; Vaksha Patel, Dr. Daniela Iancu, Dr. Naomi Issler, Dr Joanna Marks, Dr. Simona Dumitriu, Anne Kesselheim, Priya Outtandy, Felice Leung, Barbera Jensen, Sanjana Gupta, and Stephanie Dufek.

My final and biggest thanks goes to:

The DeSouza's; whose various ups and downs provided a welcome distraction from my own, and whose unquestioning sociability and hospitality defies all logic.

Aytuğ; for her generous warmth, kindness, and ever-patient support.

1. Background

Here we define some fundamental molecular biology from a genetics perspective, building first from the components of DNA and moving up towards the nature of inheritance from a theoretical and computational standpoint. For those who already have a firm understanding of these concepts, this chapter may be skipped. Otherwise, the aim here is to define the terminology and outline the principles that will be used in the rest of the thesis.

1.1 DNA

Each organism is defined by the data encoded within DNA, a long molecule made of up sub-units of nucleotides that constitutes the genome of the organism. The order of these nucleotides form an ordered sequence specific to that individual, and ultimately encode the genetic information that makes that individual unique.

DNA is packed into chromosomes, two long strands of chained nucleotides bound together by hydrogen bonds between nucleotide pairs on opposing strands. Nucleotides consist of a molecule made up of a phosphate group, 2-deoxyribose, and a nucleobase (or base). There are four possible bases: adenine, cytosine, guanine, or thymine (or A, C, T, G). All adenine bases are paired with thymine, and all cytosine bases are paired with guanine. These form AT and GC pairs across the two strands forming what are known as base pairs (or bp). The sugar and phosphate groups alternate in a chain along the side of DNA, lending a backbone structure on the outer edge of the bases that forms the eponymous helical shape known as the double helix. In humans, there are 3.3 billion base pairs of DNA.

1.1.1 Polarity

Every strand of DNA has a defined polarity, for the sake of a consistent reading orientation. The two strands of the double helix lie in opposite directions of one another, such that the starting end of one strand is the finishing end of the other. The starting end is referred to as 5' (five prime), and the finishing end is known as a 3'. These numbers are named in accordance to the clockwise ordering of the carbon atoms in the deoxyribose molecule.

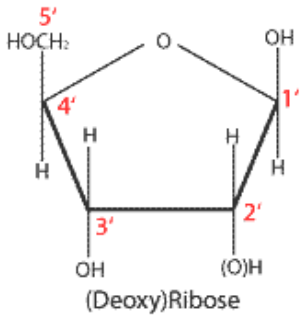

Figure 1.1: The 5' end is terminated with a phosphate, and the 3' end with a hydroxyl sugar group (*Image courtesy of Wikimedia Commons*)

For consistency, biologists follow the convention of defining the 5' to 3' orientation as the forward (or 'sense') strand. The 3' to 5' orientation is the reverse (or 'antisense') strand.

1.1.2 Chromosomes

Chromosomes hold packaged DNA in the form of chromatin, as well as specialised proteins that perform chromosome-specific tasks such as a reading, replicating, and repairing the DNA. There are many chromosomes within a cell; in a human there are 23 distinct chromosomes within a haploid cell such as gametes (sperm and unfertilized eggs), and 23 distinct pairs of chromosomes in a diploid cell which contain an individual's entire genome.

At the end stages of cell division the structure of chromosomes can be clearly seen, with two arms extending out from each strand, with the chromosomes bound together in the middle. The p and the q arm of each chromosome denotes the small and larger arm respectively. These arms are split into bands as defined by the density of the chromatin in the region, and a specific locus can be specified by, e.g. 16p5.2, which denotes that the locus of interest lies on chromosome 16, on the p-arm, in band 5, sub-band 2.

The middle section of the chromosome in known as the centromere and is purely structural; it does not encode any data, but holds pairs of chromosomes together. The ends are capped with telomeres which act as disposable buffers, and are repeatedly truncated during cell division.

Pairs of chromosomes are known as homologs, and are split into two groups: autosomal chromosomes, and allosomal chromosomes (or 'sex-specific' chromosomes).

The autosomes are the most common with 22 pairs of chromosomes within the group. The last remaining pair are therefore of the allosomal variety, and determines the sex of an individual during sexual reproduction and are typically what are known as X and Y chromosomes. Females have a homologous pair of X-X chromosomes, whereas males have an X and Y.

Each pair of chromosomes has a complete set of genes, assuming no duplications or deletions. These two copies of each gene are referred as alleles of the gene, though the term itself can describe any locus between chromosome pairs.

1.1.3 Genes and Function

Genes are the parts of DNA that are mostly responsible for generating a phenotype, the observable physical characteristics of an organism. The sections of DNA that comprise gene data are known as the coding regions or exons, and are delimited by codons which are triplet groups of nucleobases that together form a code to be read by RNA polymerase enzymes, and ribosome proteins that translate it for protein

synthesis in an uninterrupted run of codons known as the open reading frame.

Coding regions are sparsely populated in the human genome, occupying 1-2% of the sequence. The remainder consists of either unknown or regulatory regions responsible for gene expression. Regions between genes are known as intergenic and may play some regulatory roles, but are not typically bound to any particular gene.

Genes are made up of four main components: 5'UTR, exons, introns, splice sites, and 3'UTR.

UTR (or 'untranslated regions') cap the gene extremes with a 5' upstream end containing binding sites to initiate transcription via a start codon, and a 3' downstream end with sites to terminate the process via a stop codon. The transcription process essentially clones a given DNA segment into a precursor RNA molecule called messenger RNA (or mRNA), which is similar to the DNA molecule except that the sugar is a ribose instead of a deoxyribose, and thymine is replaced with a base called uracil (U).

Exons constitute the coding regions of the gene, alternated by introns which are non-coding. Splice sites flank the exon-intron boundaries and are responsible for the splicing process which binds coding (exon) regions together during transcription for the open reading frame into spliced mRNA. The splice mRNA is then translated into a protein by a ribosome, with several different proteins being made from the same DNA template due to splice variations. For reference, there are currently approximately 60,000 genes in the human genome[1] that give rise to 45 million proteins[2].

Proteins are a product of their amino-acid sequence derived from DNA codons, as well as the emergent properties from the contorted three dimensional structure they exhibit due to their folded arrangement. There are 20 different amino-acid bases that are one-to-many mapped from 64 different DNA codons. This redundancy has some interesting caveats, the most notable being that there is only one start codon (ATG) and three stop codons (TAA, TGA, TAG).

[1] As of Human Genome version GRCh38, reference Gene database.
[2] As of UniProtKB database release version June 2016, 75% of which are predicted[1].

1.2 Heredity

During sexual reproduction, chromosomes from both the mother and father are assorted and fused together to create new chromosomes that are passed on into their offspring. This 'reshuffling' of DNA is what creates such variation in organisms, such that the chance of any two individuals of the same organism being clones of each other are extremely slim.

Chromosomes are assorted during the process of meiosis, where during reproduction a diploid cell will split into four distinct haploid cells (gametes). The maternal and paternal gametes then merge to form a diploid zygote which contains DNA from both parents.

1.2.1 Meiosis

Meiosis is split into three phases: *meiosis S*, *meiosis I*, and *meiosis II*.

In the S-phase, homologs from the chromosomes of each parent are replicated. This results in a cell that has twice the number of chromosomes.

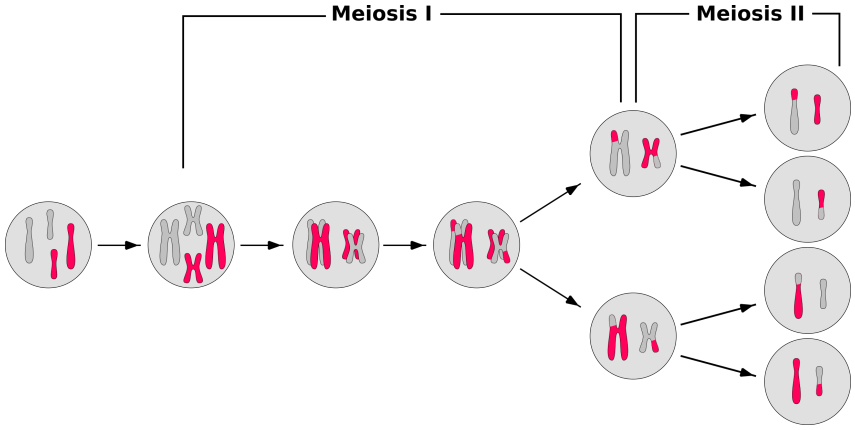

Figure 1.2: fig:back:meiosis

i Meiosis I

The meiosis I phase is by far the longest phase of meiosis, and is split into nine stages.

First, homologs from each parent pair up and exchange the DNA in their sister chromatids (**Prophase I**). Sister chromatids of each chromosome then detach into thin threads (**Leptotene**), and the chromosomes then align into homologous chromosome pairs that snap together outwards in a zipper-like fashion from the centromere, forming a bivalent (**Zygotene**).

The chromatids now exchange homologous regions of DNA to non-sister chromatids, recombining their data in a non-deleterious fashion (**Pachytene**). Homologous chromosomes then separate and uncoil to allow limited transcription, and all bivalents condense such that points of recombination entangle (**Diplotene**). A meiotic spindle becomes pronounced between sister chromatids. Genetic content then migrates to the two poles of the cell (**Diakinesis**) .

The bivalents migrate to a metaphase plate plane and are randomly oriented to one another as a precursor to the independent assortment of the chromosomes (**Metaphase I**). Homologous chromosomes then move to opposite poles, with the sister chromatids remaining intact as the homologs are separated (**Anaphase I**). The meiotic spindle disappears and cell de-condensates and lengthens, completing cell division (**Telophase I**).

The two resultant daughter cells have half the number of original chromosomes, but each chromosome is made up of a pair of chromatids.

ii Meiosis II

Meiosis II can be seen as a simplified version of the cell division in the previous phase, as many of the steps are repeated.

Chromatids once again shorten and thicken (**Prophase II**), and as before the meiotic spindle surfaces dragging material to the poles (**Metaphase II**). The sister

chromatids then segregate towards the poles and group to become sister chromosomes (**Anaphase II**), and finally the meiotic spindle is dissembled and the chromosomes lengthened (**Telophase II**).

This results in four haploid gametes from the two daughter cells in meiosis I.

1.2.2 Recombination

Recombination occurs during meiosis I, when the bivalents are formed from the parental homologs bound by connections known as chiasmata. The process involves splitting and recombining segments of DNA segments across sister chromatids at the chiasmata. This results in exchanged genetic material at a specific point on a pair of chromosomes known as a crossover, with a single crossover event occurring per meioses.

The chance of a crossover event occurring is based on can be determined between two loci. This probability (or recombination frequency) is in the range [0,0.5] increasing with the distance between the loci.

For example, two adjacent loci in close proximity to one another will have a near zero chance of a recombination event occurring. However, if the loci were on different chromosomes, then it's safe to assume that the two loci will segregate independently, with a probability tending towards 0.5.

Any deviation from the expectation inferred from the recombination frequency would assume that there was linkage disequilibrium between the two locations and that the chance of a recombination is biased.

Recombination frequencies are not uniform throughout a chromosome, as the nature and density of the underlying chromatin influences the chance across a given band. The frequencies are also sex-specific, since women are more prone to crossover events than men.

Haldane's model of recombination modelled crossovers as a Poisson[3] distribution,

[3] See Appendix page 244.

defining the Morgan, a unit of genetic distance, that details the expected number of crossovers between two loci [2]. The Morgan is defined such that there is (on average) 1 expected crossover event occurring at a distance 1 Morgan. In practical circumstances, the centiMorgan sub-unit is used to refer to 0.01 expected crossovers occurring at a distance of 1 cM.

$$\theta = \frac{1}{2}(1 - e^{-2d}) \tag{1.1}$$

Note that Haldane's model does not model crossover interference, where the act of a crossover prevents the nearby act of another crossover from happening.

It is clear that recombination events are instrumental to our understanding of how offspring inherit traits from their parents, specifically where the crossovers occur so that we can trace the flow of data throughout the generations.

In order to do so, we must first map chromosomes so that we can identify loci closely related to the inherited traits in question.

1.3 Molecular Maps

Historically, mapping traits to a locus was performed through extensive breeding experiments and then tabulating the number of different traits that appeared. These traits would then act as sign-posts for where the trait or, more typically, the disease phenotype would lie. The resolution of these methods were limited however, with the disease locus being only distinguishable between different chromosomes.

Advances in the field now make use of numerous flags or markers, evenly-spaced across the chromosome such that the phenotype can be located by any two flanking markers that surround the region of interest. The effectiveness of their usage is determined by how well they conform to following principles:

Known locus The trait of interest lies in an unknown location, but it's position can be inferred relative to a marker with a known position.

Polymorphic The marker must denote some point of variability within a population. The human genome consists of many variations between individuals, as many as 150 million[4], but this only comprises 1% of the genome[5]. Each distinct variation in a marker is referred to as an allele, and the marker is said to be biallelic if it has two possible states (and triallelic for three, quadrillelic for four, etc). Bialellic markers are typically referred to as 'binary' markers, with higher-indexed markers being the polymorphic ones.

Co-dominance If a marker is not polymorphic in a unique way, then no information can be inferred between individuals and all modes of inheritance would be equally likely. The principle of co-dominance asserts that all possible states of a marker are distinct from one another.

Hardy-Weinberg Equilibrium The HWE is the model that assumes that allele and genotype frequencies remain constant in the absence of external interference. A biallelic marker with a minor allele frequency of p and a major allele frequency of $q = (1 - p)$ will distribute genotypes with a $p^2 : 2pq : q^2$ ratio.

Low mutation Rate In order to be certain that the marker found in an individual was inherited from a parent, the marker must have a low rate of mutation in the general population.

1.3.1 Markers

It is always beneficial to genotype all individuals of the same family using the same type of marker to ensure consistency by using the same set of genotype loci that are crucial in following the disease locus across generations.

Molecular markers are based upon the type of variation within the DNA they are recording, and variants typically come in two sorts: tandem repeats, and point

[4] As of dbSNP version 146 [3].

[5] As verified by a cursory scan over raw FASTA output files, via: `echo 'grep -c -oP "[ACTGactg]{1}" chr*.fa | awk -F:'{s+=$2} END {print s}'`

mutations.

i Tandem Repeats

These are sections of the genome where one or more nucleotides are repeated in succession an unspecified amount of times[6]. The nature of these repeats is unknown, but they are thought to be caused by repeated strand-slippage in the replication process of early microbes [4]. Nonetheless, they serve as useful markers since they are an inheritable and observable, and thus informative for linkage.

There are several classes of tandem repeats, each class determined by either/both the number of repeats and the length of the repeating sequence. A minisatellite or *variable number tandem repeat* (VNTR) is any repeating sequence approximately 10-60 nucleotides in length, and a microsatellite or *short tandem repeat* (STR) is typically fewer than 10 nucleotides.

ii Point Mutations

A point mutation or *single nucleotide polymorphism* (SNP) denote a variation in the genome of a single base pair. Despite the potential for a SNP to be quadrilellic most are biallelic, lending a lower level of informativeness than repeat markers. Their advantage lies in the sheer density of their numbers, covering the genome at approximately 200bp intervals within coding and non-coding regions alike. With SNPs, a disease locus can be identified at the inter-gene level, and it is common to have two or more SNPs within the same exon of a gene.

The low level of polymorphism does indeed pose a problem when trying to trace which parent a genotype descended from, and is one of main points of contention in this thesis.

[6]Though we should assume a minimum of at least three repeats for a pattern to be detected.

iii dbSNP

Classically the minor allele frequencies of SNPs was said to be no less than 1% to have useable information content, but the most current SNP database (dbSNPv142) also caters for SNPs with much lower frequencies due to typically large sample sizes used in the genotyping process. The general format for SNP IDs is the reference SNP (or 'rs') identifier followed by a unique number that distinguishes it from other SNPs (e.g. rs1234567). Due to the large number of SNPs being registered into dbSNP from several different sources, it is not uncommon to see two different SNPs specifying the same variant, or for a SNP with such a low minor allele frequency ($<0.0001\%$) that it should hardly be called a SNP at all. Indeed, dbSNP rigorously strives to resolve these problematic SNPs with every release, but this leads to inconsistencies between versions and binds the informative of any SNP study to a specific dbSNP release version.

1.3.2 Linkage Maps

Linkage maps hold subsets of markers thought to be informative for the particular study in question. There are two main types of maps - physical and genetic, each with their own uses but for ultimately different purposes.

i Physical Maps

Physical maps ensure that markers represent actual points on the genome, e.g. rs1234567 $\rightarrow$ chr7:97795920. The variant can be found by selecting the correct chromosome and offsetting the number of the base pairs from the start of the chromosome (chromosome 7 in this case).

This is very helpful when wanting the know the actual position of a mutation, but is of no use in linkage studies.

ii Genetic Maps

Genetic maps work on a fundamentally different principle to physical maps. Where physical maps infer actual base-pair distances between adjacent markers, genetic maps use recombination frequencies to predict the occurrence of crossovers. Genetic maps use units of centiMorgan as opposed to base-pairs, and though both scale linearly in accordance with one another, there are notable differences when dealing with genders due to the different recombination frequencies.

The recombination frequencies between markers in genetic maps are utilized effectively by the linkage analysis for determining statistically accurate occurrences of crossover events between from parent to offspring. This information along with a known inheritance model is paramount in precisely determining where points of recombination could occur.

1.4 Modes of Inheritance

Inheritance models dictate the pathways that alleles segregate across generations. An individual with the same allele across both homologous chromosomes is said to be homozygous, whereas an individual with differing alleles at these homologs are said to be heterozygous. To delve into how these types of alleles interact, we must first stop to appreciate the foundation of modern genetics that these modes were built upon: Mendelian inheritance.

1.4.1 Mendel's Laws

Gregor Mendel was a botanist monk who outlined the first basic principles of genetics through his breeding experiments with peas. Though some of his observations have not stood incontestable through the passages of time, they are still of great import and he is still heralded as the "the father of genetics":

i Law of Segregation

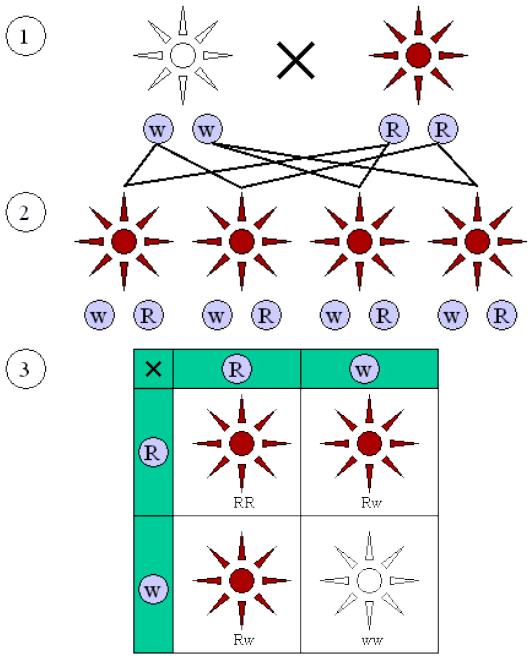

Figure 1.3: Segregation diagram of traits with Punnett square (Image courtesy of Wikimedia Commons)

Every diploid organism has two alleles that segregate during gamete formation to create diploid offspring that bears just one of the alleles from that parent (i.e. each offspring inherits one allele from their parents). As shown in Figure 1.3, this has the interesting effect such that two homozygous parents with genotypes ww and RR respectively will *always* have heterozygous offspring with wR genotypes, yet these heterozygous offspring can further mate to reconstitute their parent's genotypes.

ii Law of Independent Assortment

Separate alleles allude to separate traits, and their transmission from parent to offspring are independent of one another. A zygote can end up with any combination of parental alleles, and may end up with genotypes that are entirely different from

their parents. Offspring that (by chance) have the same genotypes as any of their parents are called recombinant, and offspring with differing genotypes from any of their parents are called non-recombinant.

iii Law of Dominance

If the presence of a single form of an allele is enough to produce the phenotype, then that heterozygous allele is said to be dominant (as is the case with wR, where the R trait dominates the w trait). Conversely, if a single form of an allele is not enough to produce the phenotype, then that heterozygous allele is said to be recessive (as is the case with wR, where the w trait is dominated by the R trait and cannot be expressed).

1.4.2 Mendel's Laws Revised

Though Mendel was ahead of his time in describing genes as allelic traits, he did not take into account that genes from diploid individuals may have more than two alleles (i.e. multi-allelic) due to variations within a population. He also made the erroneous assumption[7] that one allele maps to one trait, where in reality there are numerous polygenic traits produced by the interaction of many alleles.

In relation to his third law, the concept of co-dominance is missed; if the two potential phenotypes within heterozygous alleles are independent of each other and do not conflict, they can both be expressed. Alternately, if the two potential phenotypes within heterozygous alleles are capable of conflicting or interacting with one another, then the phenotype is usually a 'blend' of the two, and said to be of incomplete dominance since neither allele is completely dominant over the other.

[7] One that likely every geneticist makes when first entering the field!

1.4.3 Pedigrees

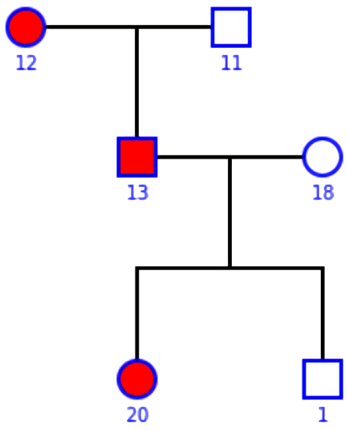

Figure 1.4: Typical dominant pedigree, with females represented as circles and males as squares. Affectation is represented by a red colour fill.

Individuals from the same family share a pedigree, and each individual is grouped into two classes: founders and non-founders. Founders are individuals who contribute unique founder alleles to the pedigree, and the non-founders are the individuals who inherit them. The larger the pedigree the more complex it is to process, with a function that is referred to as the bit size.

$$B = 2n - f - g \tag{1.2}$$

where:

- $\boldsymbol{n}$ number of non-founders
- $\boldsymbol{f}$ number of founders
- $\boldsymbol{g}$ number of ungenotyped founder couples

Inbred pedigrees where related individuals have produced offspring are said to be consanguineous.

1.4.4 Inheritance Models

Inheriting a disease allele is not a straightforward process, since the disease may not be visible in every individual who has the allele. For a given individual it is assumed that the disease status is always known[8]. There is an empirical probability that they will present with the disease phenotype, given a set of disease alleles. This is the derived through what is known as a penetrance function:

$$P(D|A) = [0, 1] \tag{1.3}$$

where:

D Affected or Unaffected

A Genotype belonging to {GG,Gg,gg} where G is the disease allele.

i Autosomal Dominant

If the disease segregates through an autosomal dominant model, then the disease trait is dominant and the penetrance function is thus: $P(D|GG) = P(D|Gg) = 1$, and $P(D|gg) = 0$. Affected individuals are present at all generations, and because the disease trait lies within the autosomes (chromosomes 1 to 22), affectation is non-gender specific and both sexes have equal probability in inheriting the trait.

ii Autosomal Recessive

Autosomal recessive is a more constrained mode, where instead the disease trait is recessive and the penetrance function is limited to: $P(D|GG) = 1$, and $P(D|Gg) = P(D|gg) = 0$

Here, affected individuals appear to skip generations since parents carrying a heterozygous genotype will not present the phenotype, but instead be carriers of the disease trait that can be reconstituted as a homozygous disease allele in their

[8] At least in binary traits which are present at birth.

offspring. In consanguineous pedigrees, the likelihood of this happening is significantly increased due to the multiple pathways the disease allele can take, with each generation having potentially double the chance of inheriting the disease allele after each subsequent consanguineous pairing.

iii X-linked Dominant

The disease trait rests solely on chromosome X. Due to males having only one X-chromosome and one Y-chromosome, this simplifies the inheritance pattern such that males can only inherit the single Y-chromosome from their father, and females can only inherit the single X-chromosome from their father - removing much ambiguity. The male's X-chromosome can still be one of the two maternal X-chromosomes, and likewise the female's maternal X-chromosome can be one of the two as well. X-linked dominant pedigrees can be identified by a distinct lack of father-to-son transmission, since that would imply that the Y-chromosome is the disease allele. Due to dominant nature of the trait, the penetrance function is identical to that of autosomal dominant.

iv X-linked Recessive

In X-linked recessive diseases, both of the inherited X-chromosomes must contain the disease allele (i.e. be homozygous), with the exception of males who need only copy to express the phenotype since the X-chromosome inherently dominates the Y-chromosome. Males cannot be carriers in X-linked pedigrees making them autozygous. Typically the pedigree exhibits more affected males than females, with skipped generations where the mother is only a carrier. The penetrance function is gender-specific, taking on the same rules for females as autosomal recessive, but differing for males:

$$P(D|GG) = 1, P(D|Gg) = P(D|gg) = 0 \qquad \text{(if Female)}$$

$$P(D|GG) = N/A, P(D|Gg) = 1, \text{ and } P(D|gg) = 0 \qquad \text{(if Male)}$$

1.5 Haploblocks

An individual's genotype is usually acquired in an unphased manner, meaning that that genotyping process acquired the alleles present at a locus, but the ordering was unimportant so that the maternal and paternal allele are ambiguous.

A genotype with a known path of inheritance for its alleles is said to be phased, and a phased genotype is also known as a haplotype. A haplotype typically spans a single locus similar to an allele, but a haplotype is encapsulated and a restricted to a larger group of haplotypes known as a haploblock, which represent the intervals between two recombination events[9].

These blocks represent the splitting of founder alleles which occurs at every meiosis, effectively halving the founder data within the allele at each generation.

A founder allele stretching across m markers and undergoing n meiosis, will have an expected length of $\frac{m}{n}$ markers over n generations.

In Figure 1.5 the offspring is recombinant at that particular locus due to a crossover in the maternal alleles. Note that at the level of a single marker it is not clear which way a genotype is phased, e.g. the sixth marker where the maternal alleles (2,2) and the paternal alleles (2,2) leave much ambiguity to which particular chromosome the offspring inherited their alleles (2,2).

[9] Including the recombination that occurs at the start/end of each chromosome.

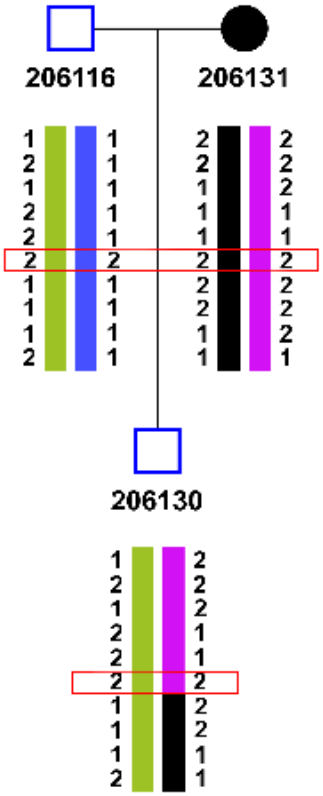

Figure 1.5: Four complete haploblocks from two founders inherited by their non-founder offspring. The sixth marker is highlighted due to ambiguous genotypes.

Resolving ambiguous genotypes to specific chromosomes requires providing the genotype with some neighbouring context such that it knows whether it is in a recombination hotspot, or whether it lies within an existing block.

It is the aim of this thesis to derive methods to explore such problems, but in order to do so we must first understand how haplotypes are generated through linkage analysis.

1.6 Linkage Analysis and Haplotype Generation

In this chapter we look at the underlying principles that power linkage analysis; the general methodology, as well as the algorithms best suited for generating haplotypes from incomplete genotypes.

1.6.1 Theory of Linkage Analysis

In the previous chapter we discussed how genotypes are mapped to specific markers, and that a disease locus is usually only flanked by the nearest adjacent markers which may or may not be within the same haploblock, and thus may or may not be co-segragated by the disease allele during meiosis. If a given marker co-segregates with a disease allele over multiple recombination events, then it is safe to assume that they are in close proximity of one another. Such bound loci are said to be *in linkage* with one another, and when one undergoes a meiosis; so does the other.

In linkage analysis, this translates as pinpointing the multiple subsets of markers with the fewest number of crossovers between themselves, such that the the number of meioses in the cases is minimized and the controls are maximized (i.e. which markers/traits are in linkage with one another in affected individuals, but are not in linkage within unaffected individuals). Classical linkage analysis involves doing this by direct means; counting each crossover event for each marker against every other marker, but the complexity of such a task scales exponentially for large sets and is highly impractical.

Modern linkage analysis determines this using statistical means; through the filter of the correct genetic model to establish a correct penetrance function, the assumption of linkage equilibrium between markers to simplify computation by discounting crossover interference, the exclusion of monozygotic twins[10], and the known recombination frequencies between markers as well as their ordering. These are then used to establish a likelihood score that rates each marker upon the aforementioned principles such that disease loci can be found by scanning for the highest likelihood.

[10] Identical genotypes in twins bias the analysis since meioses may not be independent.

i Likelihood and LODs

This likelihood is known as a logarithm of the odds (LOD) score, and is a ratio that compares the likelihood of any two traits being in linkage compared to them not being in linkage.

$$LOD(\theta) = log_{10}(\frac{L(\hat{\theta})}{L(\theta = 0.5)} \tag{1.4}$$

where:

$\theta = \frac{1}{2}(1 - e^{-2d})$ The Haldane map function that estimates the recombination frequency given the genetic distance in centiMorgan (as defined in equation 1.1 on page 21).

$\hat{\theta}$ The observed recombination frequency.

$\theta = 0.5$ The frequency of two traits segregating independently (are essentially unlinked).

$L(\theta)$ The joint likelihood

The joint likelihood is main core of what the linkage analysis tries to solve, iterating over all individuals and markers, as defined by:

$$L = \sum_{g1} \cdots \sum_{gn} \prod P(Y_i|g_i) \prod_{k,l,m} P(g_m|g_k, g_l) \tag{1.5}$$

where:

g_n genotypes of individual n

Y_n phenotype of individual n

$P(Y_i|g_i)$ penetrance probability applied over all individuals. This is the probability of a phenotype given a genotype.

$P(g_j)$ founder probability applied to founders only. Modelled under Hardy-Weinberg and assumes linkage equilibrium.

$P(g_m|g_k, g_l)$ transmission probability. Probability of non-founder genotypes (m) given parental genotypes (k and l).

There is a narrow window of relevance when dealing with LOD scores, typically any score less than -2 suggests complete lack of linkage and though may seem uninformative, is actually a good indicator of where a disease locus *cannot* be.

A LOD score greater than or equal to 3 is considered to be of great significance for pinpointing a disease locus, but the reasoning for this is mostly historical.

1.6.2 Linkage Algorithms

Though the methodology between algorithms varies somewhat, all work under the same principle of determining a LOD score and generating missing genotypes, though some are more engineered towards resolving phase than others.

i Elston-Stewart

The very first linkage algorithm first computed upon paper, by performing a task known as peeling upon a pedigree.

A connected graph of individuals (as nodes) and relationships (as edges) would be recursively peeled in a depth-first manner, by collapsing LOD score calculations onto members of a pedigree known as *pivots*, who are key to the pedigree such that their removal would create two detached pedigrees [5].

At each stage, the algorithm considers a simple nuclear family (mother, father, offspring(s)) and performs a LOD score computation of the offspring inheriting their founder parents genotypes, then flattening the score up a generation into that founder parent.

This eventually resorts in a single node that contains the total LOD summation of the entire pedigree for a given marker.

Over time, programs such as **Linkage** allowed multiple markers to be considered at the same time and gave rise to multi-point parametric linkage analysis that became the standard for subsequent linkage packages.

The algorithm scales linearly with the number of individuals in a pedigree, but

exponentially with the number of markers for a multi-point analysis; a severe limiting step in an age where the density of markers were on the increase.

Generating fast and reasonable accurate LOD results became the main aim for a lot of programs, each performing a different approximation upon the LOD score calculations to generate a faster analysis. The program **Vitesse** was boasted as being able to analyse 8 markers jointly using "fuzzy inheritance" [6].

Since haplotype reconstruction requires precise determination of a missing genotype, the Elston-Stewart algorithm was lacking in this regard.

ii Lander-Green

The era of haplotypes began with the Lander-Green algorithm, which aims to describe a probability distribution over all possible ways traits may segregate between individuals based on their genotypes for a given marker, whilst being dependent on adjacent markers.

Under Lander-Green, a pedigree is represented using an *inheritance vector*, which shows the transition of founder alleles to non-founders [7].

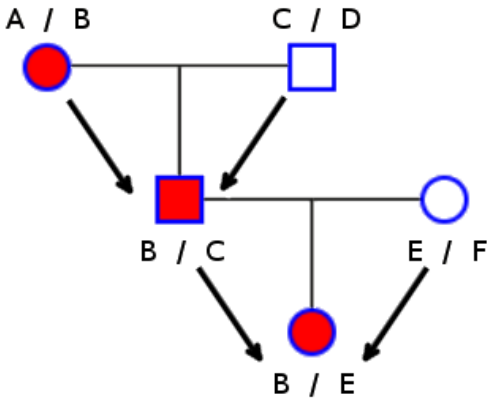

Figure 1.6: A pedigree of phased individuals with inheritance vectors represented as arrows. Here the order of the genotypes are {maternal,paternal} indexed to {0,1}

The compact representation of the inheritance vector in Figure 1.6 would be 1000 since it is only the transition from the first generation mother → child where the second allele (B→1) is transmitted, whereas it is always the first allele(0) in all other cases.

The importance of this for haplotype reconstruction is apparent, for it considers the complete inheritance probability distribution and any bisection of this set would capture a consistent set of genotypes as described by their vectors.

This was the first linkage analysis that made use of a Hidden Markov Model (HMM), which denotes a dependent chain of unobserved hidden states with state probabilities, that transition between each other under transmission probabilities.

Kruglyak readily applied this to a a linkage model where the states mapped the possible inheritance vectors, and the transmission probabilities were the genotype likelihoods [8]. The Elston-Stewart algorithm was then used to sum over the inheritance probability distribution at each locus.

Over time, a few notable speedups were incorporated without the cost of accuracy incurred by approximation:

Inheritance Reduction where potential inheritance vectors with incompatible genotypes between parent and offspring can be pruned from the analysis.

Founder Symmetry where inheritance vectors that differ only in a founder's phase are equal, which lends massive importance in the regard that most pedigrees do not set phase at all (Allegro).

Founder Couple Reduction where ungenotyped founder couples are equivalent to each other and need only be evaluated once (Allegro).

Each of these cut the complexity of a pedigree in half, together producing a significant (6x) reduction in bit-size.

The Lander-Green algorithm scales linearly with the number of markers, a much better time complexity than the Elston-Stewart, however suffers drawbacks in the number of individuals in the pedigree where it scales exponentially with each meiosis.

iii Hidden Markov Models and Haplotype Reconstruction

The use of connected graphs in linkage analysis has the side benefit of being able to reconstruct haplotypes within the same run.

In the HMMs, the linkage program transitions between a subset of inheritance vector states to determine the total likelihood by the time the last state in the chain is reached. The path taken by the linkage program is consistent; such that any paths that do not reach the last inheritance vector terminated early and were thus incompatible, but any paths that did make it through were conformed with the genotypes, and all the (now phased) genotypes that came before.

That is not to say that there is only one true path[11] in the HMM that is compatible with the observed genotypes; several may exist. One way of finding the optimal path is to run an inference algorithm through the HMM chain to find the most likely sequence of states given a likelihood.

iv Viterbi Algorithm

One such algorithm is the forward-backward *Viterbi* algorithm. A forward-backward algorithm works on the principle of performing two passes upon a HMM. Given a sequential set of observations:

Forward Pass Starts at the first state and steps forwards towards the end state, computing forward probabilities that set the probabilities of any given state ending at any other subsequent state. In linkage, this is essentially setting a weighted graph based on compatible inheritance vectors from the observed genotypes.

Backwards Pass After the forward pass, computes a set of backward probabilities that sets the probabilities of any given state observing any applicable previous state. This is essentially the same as the previous step.

[11] Though, at the molecular level there is only one true path, since the crossovers occur at discrete loci.

Once performed, a combined probability distribution is obtained to find the mostly likely state at any step in the chain [9].

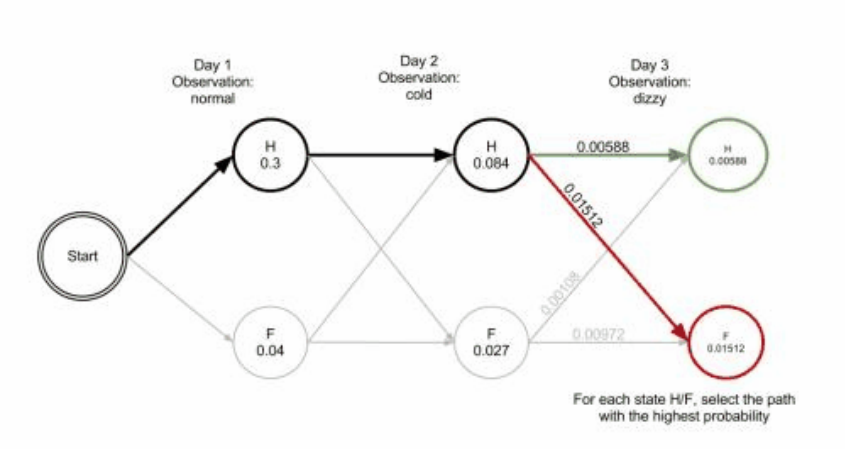

Figure 1.7: Fourth step of Viterbi

The Viterbi algorithm is an extension of this prototype that attempts to maximize the probabilities such that the optimal path can be traced by observing the most likely set of inheritance vectors upon the most likely set of founder alleles.

This is still an approximation upon the data, since the most likely set of inheritance vectors only tells us how the alleles flow from generation to generation, but not necessarily the content of the alleles.

1.7 Summary

We have seen that there are two main limiting steps in linkage analysis algorithms that scale either with the **number of meiosis** or the **number of markers**. The Elston-Stewart algorithm's complexity scaled linearly with the number of meiosis by collapsing computations on pivot individuals, but scaled exponentially with the number of markers.

The Lander-Green-Kruglyak algorithm was the inverse of this; where the number of meiosis caused an exponential increase in complexity due to the possible inheritance vector states doubling at each step. The number of markers scaled linearly however, and this is what made it the most widely-adopted algorithm in linkage analysis programs, since the density of genotyping chips have only increased over the time.

Hidden Markov Models lend their use to inheritance or descent diagrams that allow us to set phase at each genotype by following the most likely set of inheritance vectors [10].

Haplotype reconstruction is by no means deterministic, as the optimal path as derived from the Viterbi algorithm may be just as likely as another path, especially for incomplete data where the potential alleles of untyped genotypes may have equal likelihood.

The strength of the analysis ultimately lies in the observed genotypes, and some are more informative than others. In the next chapter we outline a linkage analysis pipeline that aims to achieve high quality analyses through a combination of elegant filtering protocols, as well as brute-force hardware maximisation.

2. Methods - Pipeline

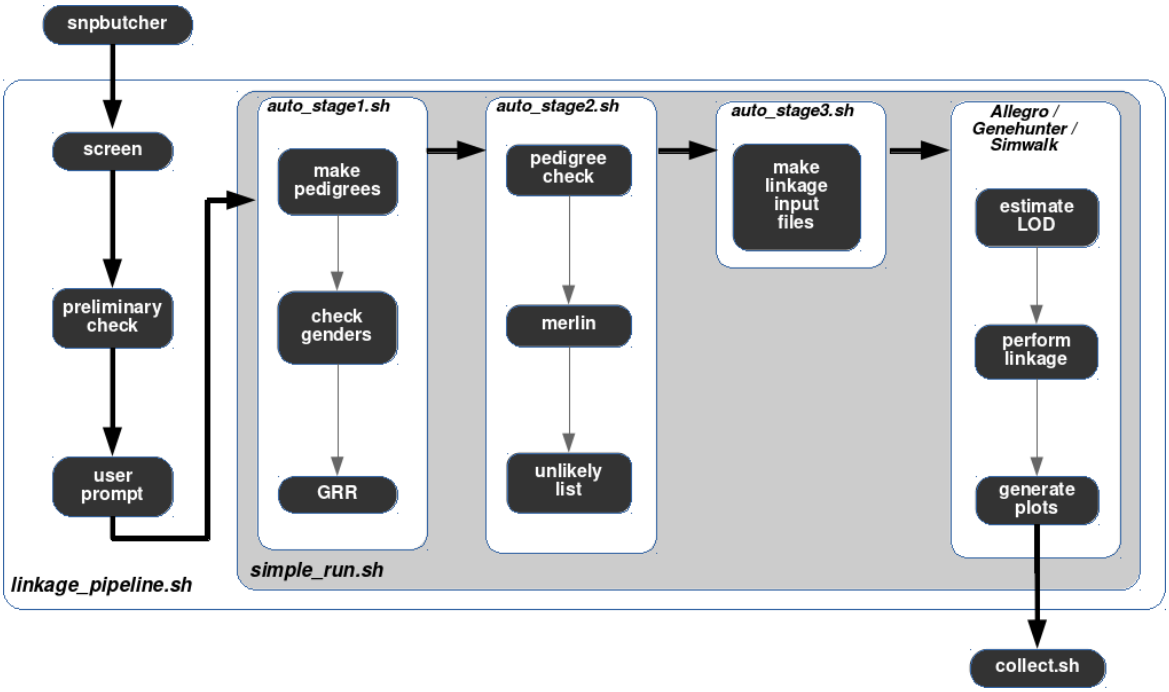

Figure 2.1: Pipeline overview

2.1 Linkage Pipelines and Processing

2.1.1 Input Data

The strength behind any linkage analysis does not lie solely with the linkage program, but with the quality of the input data that the program processes. An input data set consisting of mostly uninformative markers will not yield any meaningful output data, since the linkage program will be processing ambiguous data and subsequent runs may produce wildly varying results. In order to preserve some consistency across subsequent runs, we must be more selective in our choice of input data such that we only include markers that have a high information content in relation to the sample data set.

i Pedigree (pedfile.pro)

This file contains the information upon each family member, with every line denoting a single individual in standard LINKAGE[12] format.

An individual's line consists of data that numerically maps their sex (male→1, female→2, unknown→0) and their phenotypic status (unaffected→1, affected→2, unknown→0). Depending on the program used, an unknown sex may not be permitted. While it is not uncommon to see an unknown affectation, it is generally of no use to include it within a linkage run as it adds no power to the analysis.

Other fields record the relationship of that individual to other family members (mother, father). This mother-father-offspring grouping is referred to as a trio.

FamilyID	PersonID	FatherID	MotherID	Sex	Affectation
311	101	0	0	1	1
311	102	0	0	2	2
311	201	101	102	1	2
311	202	101	102	2	1

Listing 2.1: An example pedigree file, showing members of family 311, with mother and father of 102 and 101 respectively, and their offspring 201 and 202.

Siblings are not stated within a trio, as this would introduce a great deal of redundancy for individuals that share multiple siblings[13]. Instead, these relationships are inferred under the rubric that trios who share the same set of parents must be siblings. Similarly, (great-)grandparents and (great-)grandchildren are inferred from recursing up through a parent's identifying record, or down through an offspring's.

Individuals without parents are given a null pointer to refer to[14], but they are only of use to an analysis if they have offspring (and are founders).

Extra fields are also permitted, but these are typically used to store genotype data which can be housed in a separate file.

[12] Also referred to as pre-MAKEPED format.
[13] who would then be stated multiple times at later points in the file.
[14] usually a '0', where '0' is a protected identifier for a non-existent individual.

Individual IDs can be alphanumeric, but most linkage programs were written at a time where it was more efficient to use plain integers, and pre-cautious measures have taught us to follow this convention.

Another convention of ours is to follow an individual identifier naming scheme of: <generation><gender (odd/even)>.

Generation begins at 1 for the first generation, and becomes 2 for the second generation and so forth. The gender specifier is the lowest unused integer that is odd for males and even for females.

E.g. 12 the first described female in the first generation, 45 the third described male in the fourth generation (where 41 and 43 have already been used).

It is not a necessary rubric, but it facilitates in the creation of large (and often consanguineous) pedigrees.

ii Genotypes (genotypes.dat)

If the pedigree file does not contain genotypic data, then it is stored separately here. This is typically a truncated readout of the genotyping chip output file.

The full genotype output report is paired with map data that contains marker positions as well as the AB genotypic call for each individual that was genotyped under the same chipset. Calls include, but are not limited to, a combination of A and/or B.

The first type of calls (AA, AB, BB) relate to the actual binary allele that each individual is supposed to exhibit in an present/not-present manner. These are pure genotypes where phase is not present, and thus a call of BA would make no sense in this context.

The second type of calls (NC, - -, ??) are error messages and attribute a misfire (or 'No Call') of the chip such that the allele could not be determined. Often the positional data relating to the markers are split into their own map file, leaving just a simple table of marker names and sample data.

```
         12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28
rs12345  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA  AA
rs12346  AB  AA  AA  AB  BB  BB  NC  BB  AA  BB  AB  NC  NC  AB  BB  AB  BB
rs12456  BB  AB  AA  AB  AA  BB  AA  AA  AA  BB  AB  AB  AA  AA  AA  AB  AB
```

Listing 2.2: Sample genotyping output readout file, with marker identifiers (without positions) as row headers, and sample identifiers as column headers.

In a given sample context, not all markers are as informative as each other. Informative markers are those that satisfy the following criteria:

Varied Set There is enough variation in the genotypes for all samples that the marker covers. This ensures that the linkage algorithm can distinguish the descent of maternal and paternal alleles without too much ambiguity.

High Quality A good proportion (approximately at least 80%) of the calls are not misfires. One or two No Calls in a small set can be resolved by the linkage analysis to reconstitute the missing genotypes from incomplete data, but only within reasonable limits.

Non-Zero Allele Frequencies (optional) Each marker is host to meta-statistics based upon the pool of samples it was used upon, the most relevant statistic being the minor allele frequency, which alludes to the representation of the less-likely allele in the overall population. Markers with 0:1 or 1:0 proportions are effectively homozygous[15] across the entire population and can be said to not contribute a varied enough set for linkage.

Known Inheritance Pattern (optional) The HapMap project was the first international effort to create a haplotype map of the human genome. Markers within this database are extremely useful for linkage since the markers tend to be quite common under the criteria that each allele is present in at least 1% of the population.

[15] Though the genotype may indeed be heterozygous.

iii Minor Allele Frequency (maf.txt)

This file contains a map of markers and their associative minor allele frequencies in context to a super-population (European, African, Asian).

In the earlier days of linkage pipeline filtering, this was typically based upon historical HapMap data, but in more current renditions can be derived from a variety of different sources; the most recent being the calculated allele frequencies from the 1000 Genomes Project (Pilot 1 data) via the **bcftools** commandline utilities.

The markers in this file are used as selection criteria for generating a map, such that any markers with extremely (un)prevalent frequencies can be filtered out due to lack of informativeness.

iv Genotype Map and Linkage Map Creation (map.txt)

The genotype map is the file that comes paired with the genotype call data, such that the markers described in the output point to a physical or genetic map. Where genetic map data is not present, the physical location is used in place under the well-grounded assumption that genetic distance scales with physical distance, and that the marker ordering remains the same.

Chr	Name	0.000000	0	Name	x
01	rs3094315	0.752565	752565	rs3094315	x
01	rs2073813	0.753540	753540	rs2073813	x
01	rs2905040	0.770215	770215	rs2905040	x
01	rs12124819	0.776545	776545	rs12124819	x

Listing 2.3: Map file describing markers in a genotyping chipset. Headers denote: chromosome, marker identifier, genetic map distance (cM), and physical map distance.

For a given genotype chipset there is an associated map file, and as the progression of chip-sequencing technology has rapidly developed, the map files have gotten significantly larger in accordance. As of 2012, Illumina's HumanOmni1S BeadChip kit boasts just over 1 million markers[16].

[16] See: http://support.illumina.com/array/array_kits/humanomni1s-8_beadchip_kit/.

As stated before, Lander-Green based linkage analysis scales linearly with the number of markers, making it computationally impractical to use the entire map set. Instead we use a smaller sub-selection of informative SNPs under the same principles described on page 44.

Genotyping chipsets have fixed buffer sizes, and so if there are not enough members in a pedigree to satisfy the buffer, it is often economical to fill empty slots with individuals from unrelated pedigrees. As a result, this can lead to a very varied set of alleles across a given marker, artificially making it seem more informative (in context of the pedigree in question) than it actually is.

Though there is nothing wrong in including uninformative SNPs in a linkage analysis, it is up to the researcher whether they wish to trim the genotypes file so that it only contains members of the pedigree.

The Python tool **snpbutcher** was developed to facilitate in the creation of these maps under the aforementioned filtering precedents, with the added inclusion of spacing criterion, with all parameters saved to a log file.

Linkage analyses before dense genotyping chipsets allowed for a spacing requirement of 0.2 cM ($\sim$ 10,000 markers) between any two adjacent markers, but subsequent analyses in more modern times have iteratively led to a default spacing requirement of approximately 0.05 cM ($\sim$ 40,000 markers), keeping the marker count low for the linkage algorithm without compromising the power of the analysis.

The "resolution" of a linkage analysis is heavily dependent upon the number of markers used in the analysis, since the spacing between the flanking markers that the disease locus co-segregates with has a centiMorgan precision equal to the minimum spacing parameter used to generate the map.

2.2 Pre-Runtime Configuration

2.2.1 Folder Conventions

All four input files (pedfile.pro, map.txt, maf.txt, genotypes.dat) are placed into their own folder, following a rigid folder naming convention of:

/path/to/working/directory/<projectID>_<projectName>/<runNumber>[_runName]_<date>

E.g. **/scratch/Linkage/124_boneitis/02_correctedpedigree_20120423**, where here **/scratch/Linkage/** is the folder of the working directory relative to the root file system, 124_boneitis refers to a unique project identifier of 124, and the description of the project's phenotype[17].

02_correctedpedigree_20120423 identifies that this is the second linkage run on the same input data set (corrected for minor pedigree changes) on the 23rd April 2012.

This ensures that each linkage project is uniquely identified by a project identifier, such that subsequent analyses can be run in the same master folder to reduce the number of repeating run numbers in the parent folder.

Once an analysis is fully complete (i.e. it is understood that there will be no more subsequent runs on the same input data set for at least a period of a month), then the project should be archived into allocated storage.

2.2.2 User Confirmation and Setup

Once the input files have been set, the **linkage_pipeline.sh** script is called to perform a thorough assessment upon the input files to determine if they are indeed primed correctly for analysis, and then acts as as a pre-analysis screening for the user to set run parameters and perform a double-confirmation that the set parameters are indeed correct.

[17] Boneitis is a dystrophic bone disorder that if left untreated can lead to comically accelerated TV references.

In order, the tasks are performed are:

Encapsulation

The GNU-based **screen** session is initiated (if not already present); a program that allows the user to re-access an ongoing process. Multiple linkage analyses can be run at the same time through different screen sessions without interfering with one another, though this is not encouraged for large projects since they would still be competing for the same system resources.

Display Detection

This portion of the script detects whether there is a user logged on remotely or locally, and forwards graphical applications accordingly so that they appear either on their machine, or locally on the host machine. This has the added benefit of the user being able to inspect the data remotely during the pre-filter stage described later.

Input Parsing

The input files are checked for consistency with one another, most of the work being delegated to the Python script **prelim_check.py**, which asserts the following:

(a) Folder naming conventions are adhered to in the current working directory.

(b) Genotype file has correct headers (i.e. are present and numeric), and that marker identifiers are also present.

(c) Pedigree file specifies the correct project ID, contains all minimally required columns (family, id, father, mother, sex, and affectation), and has no duplicate individuals.

(d) The genotype and pedigree file both contain the same set of IDs. Non-fatal error messages occur when the individuals specified in the pedigree are only a subset of all the individuals genotyped under the same

chipset, likely because the user did not trim the genotypes beforehand. In this case, the user is prompted to visually evaluate the IDs that will not be assessed in the linkage and asked for confirmation, where a termination signal is sent upon refusal.

Run Configuration

The user is prompted for analysis-specific options: dominant or recessive; autosomal[18] or X-linked; single-core or multi-core. Since chromosomes segregate independently, the multi-core option allows for multiple chromosomes to be evaluated simultaneously using the GNU **parallel** framework. Once set, the user is presented with the options they just specified, and if all is correct, a log of what was just entered is stored in a hidden file to be used in readme generation later. At this stage, the appropriate pipeline script is started.

2.3 Runtime Filtering

Filtering is the first and foremost important step in the entire pipeline. The genotypes, pedigrees, and markers are all checked for relational consistency and preliminary tests are run to better gauge the data. The filtering step is split into three main script components: **auto_stage1.sh**, **auto_stage2.sh**, and **auto_stage3.sh**.

2.3.1 auto_stage1.sh

This script is split into four main components: pedigree checking, gender checking, relationship priming, relationship checking.

[18] It should be noted that even if an autosomal analysis is specified, the genotypes must still contain chromosome X markers for accurate gender detection in the next section.

i Pedigree Check

The first stage involves re-checking the pedigree for inconsistencies as before (duplicates, project IDs), but the Perl program **HaploPainter** goes deeper to also follow the relationship in all trios and check that each member is defined. If no errors are discovered, a PDF of the pedigree is generated and the pipeline resumes.

ii Gender Check

Here, the Perl program **Alohomora** reads in the map file and the MAF file, where the minor allele frequencies are mapped to the markers and then compared with the genotypes by counting the number of heterozygous SNPs in chromosome X [11]. Gender detection utilizes **pedcheck** to determine the ratio of the heterozygous alleles of females against the hemizygous alleles of males towards the q-arm of chromosome X, due to a shortened Y chromosome in the males [12]. Errors at this stage are rare, but are often caused by sample mix up, and can be rectified by changing pedigree members to their correct genders in the input pedfile.

The program outputs a relationship file containing a mixture of pedigree data and genotypes which is to be used in the next step.

iii Relationship Priming

This step is a correctional one, only coming into effect if the output relationship file from the previous step is empty. This occurs if the MAF file contains allele frequencies for no more than six markers that are present in the map, showing poor overlap.

The frequencies in the MAF file apply directly to HapMap markers, but the HapMap project has not kept up with the increasingly dense genotyping chipsets, so modern allele frequencies need to be generated from new data (i.e. 1000 Genomes allele frequencies from sub-populations).

This can be achieved by regenerating the MAF file from new data sources, or creating a "dummy" allele frequencies by assuming that all alleles are equally represented across populations. This is a depreciated step that should never be used for a true analysis, but it exists in the case where the user merely wants to gauge the overall linkage before embarking on a more thorough study.

In this case, a new MAF file is generated and the previous step is re-run to generate a valid relationship input file.

iv Relationship Checking

The next step is the visual inspection of the genotypes through the genetics relationship representation program **GRR**. GRR evaluates the genotype distribution of between related individuals and determines how much allele-sharing is occurring compared to how much is expected, classed by their respective levels of relation. Unrelated individuals will share fewer alleles than more related individuals, and this can be plotted in a graph with axes of standard error vs standard deviation [13].

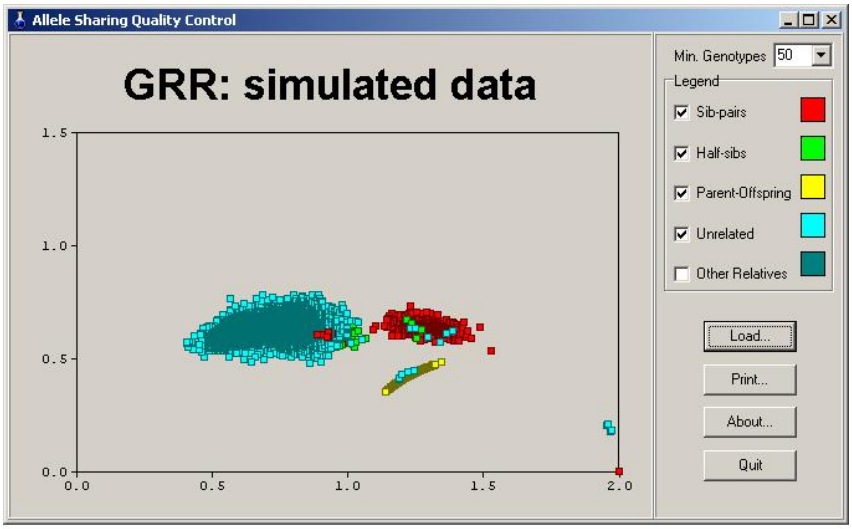

Figure 2.2: A simulated data set showing relationship clusters coloured by type and plotted by mean against standard deviation.

GRR uses *identity-by-state* (IBS) allele sharing in its metrics[19], allowing it to identify monozygotic twins (who share identical alleles) without having trace haplotypes.

Each point is formed by querying the relationships between any two individuals, coloured by the type of relationship specified in the pedigree, and plotting the mean against the standard deviation of the allele's being shared.

This forms distinct clusters, where each cluster is typically a representation of a single relationship type (siblings, half-siblings, parent-offspring, unrelated). This display is crucial, for if there is invalid clustering or mixing, then these are alarm signals for the user to terminate the pipeline.

In Figure 2.2, we can see a single red dot in the bottom-lefthand corner. Red indicates that the pedigree states that they are siblings, but they are somehow skewed far from the main Sib-pair group. GRR allows you to click on the dot and examine the relationship, and in this case it becomes apparent that this sibling relation has identified a monozygotic twin pair who share identical alleles, giving them an extremely high IBS score.

There are also two unrelated relationships with a near-perfect high mean score, indicating once more a duplicated set of alleles, but in this case likely a sample or pedigree mix up; either the same individual was genotyped twice, or the pedigree did not state they were related, or the pedigree specifies different individuals for the same set of genotypes.

The green Half-sibs seem to be split into different groups of Unrelated and Sib-pairs, as well as a small cluster of Sib-pairs within the Unrelated. Both hint that a restructure of the pedigree may be required due to erroneous information specified in the pedigree, likely due to obfuscated parentage.

Should the relationships cluster improperly, the user must terminate the pipeline and re-evaluate their input before continuing.

[19] as opposed to IBD metric which would be computationally expensive.

2.3.2 auto_stage2.sh

This stage performs fast preliminary linkage that ultimately checks for Mendelian inheritance, and gives an approximate overview of what linkage scores we can expect.

i Mendelian Inheritance Check

Mendelian errors are detected very quickly through the examination of trios, where a marker is flagged if a child's genotype contains an allele that could not have possibly been inheriting from their parents. Such errors can be indicative of non-Mendellian inheritance models such as trinucleotide repeat disorders or genomic imprinting [14], but are typically more associated with the low error rate (0.01%) of genotyping chipsets [15].

A list of markers with unlikely genotypes are generated from this checking step, and plotted in a chart of physical position against error frequency. For n families, the maximum error count is n, and markers with an error frequency greater than 1 are very indicative of poor quality.

In a typical linkage analysis of 40,000 markers an average of 5 erroneous markers is expected, and this scales linearly with the number of markers.

Higher numbers of erroneous markers are not uncommon (10 - 30) as long as they are sparsely distributed across the genome. Closely packed clusters of markers numbering greater than 50 usually suggest bad parentage in the pedigree, and several clusters totalling more than 150 are a clear sign that the entire pedigree requires re-verification.

This is a user-evaluated step, with a PDF of the plot being displayed on the screen, so it is up to the user to terminate the pipeline if the situation requires it.

ii Preliminary Linkage Overview

Once the erroneous markers from the list of unlikely genotypes have been removed, the linkage analysis program **Merlin** is started. The error detection suite de-

tects genotyping errors by inferring inconsistent haploblocks in otherwise consistent stretches of alleles between siblings [16]. Such blocks are easily detectable by searching for double recombinations within small marker intervals where the likelihood of such an event occurring is extremely improbable.

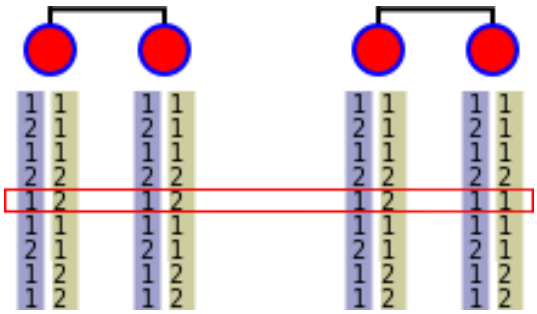

Figure 2.3: Two siblings have identical genotypes over a given allele (left), compared to the same alleles but with one contradicting genotype (right), indicating an unlikely double recombination event or a genotyping error.

Another output list of erroneous markers is generated from this analysis and passed into the next section. Merlin performs a rudimentary pass over a small subset of these markers ($\sim$ 5,000 markers) to give a preliminary linkage plot that works to at least exclude regions of linkage.

2.3.3 auto_stage3.sh

This stage does not perform any filtering, but uses Merlin to generate input files for the main three linkage programs in the pipeline: **Allegro**, **Simwalk**, and **GeneHunter**.

Each program is given its own folder, and pre-existing folders (from a previous run where errors were flagged) are renamed with a timestamp suffix (e.g. allegro $\rightarrow$ allegro-2016-04-02_10-58).

2.4 Runtime Linkage

Only Allegro and GeneHunter are run by default, and they are run sequentially so as not to impede the performances of one another. Though Simwalk input files are generated, they are not used unless further analysis is required.

This is the longest stage of the entire pipeline, with the previous filtering (auto) stages taking no longer than 5 minutes. Here, the rest of the pipeline can take anything from 10 minutes for small pedigrees, and then any interval from 10 hours to over a 100 hours for pedigrees larger than 19 bits.

2.4.1 Preliminary LOD estimation

The first step is to produce a maximum estimated LOD (ELOD) score, using a simulated data upon the pedigree. This estimate is produced by the Python script **clodhopper.py** which generates false linkage files for both Allegro and GeneHunter using the shortest chromosome input files as templates (chromosome 21).

The maximum ELOD score is parsed straight from the output files, storing largest result in a parent folder, and then all related files are deleted in preparation for the real linkage analysis. Scores between Allegro and GeneHunter tend to differ by only $\pm$ 0.2. Any larger discrepancies are suspect, and usually prompt a Simwalk run as a precautionary third control.

2.4.2 Simwalk

Though Simwalk is not run by default it is useful to discuss before delving into Allegro and GeneHunter, since it is the more classically robust linkage program with a longer publication history [17]. It is generally one order slower than more modern linkage programs and is not able to process X-linked inheritance models effectively, which is why it is not included by default. Analysis on moderately sized 19-bit pedigrees can take several days to complete. There are methods we designed to increase its throughput, but these are discussed later (see page 58).

2.4.3 GeneHunter

GeneHunter is one of the core linkage programs, and is used as the main secondary control to Allegro. It is by far the most verbose of the linkage programs, and terminal multiplexers such as screen should take care not to enter a buffer mode (such as copy) should the buffer grow too large and unresponsive from the copious verbose output. It is wise to disconnect from a screen session to let this part of the pipeline run unmonitored, only checking in periodically from time to time if warranted.

2.4.4 Allegro

The fastest, memory compact, and consistently accurate linkage program in the entire runtime linkage suite. Allegro's main strength comes from its ability to perform multi-point parametric linkage and haplotype reconstruction within the same pass. For a pedigree under 19-bits, it performs linkage upon the entire genome with an average time of 10 mins. The real bottleneck occurs for any pedigrees outside of this limit, but this is addressed in the next chapter (see the Large Pedigrees section on page 61).

2.5 Post-Linkage Processing

After each linkage step, clean up operations are performed to reclaim the disk space generated by the temporary files that were created at runtime. This largely involves recursively crawling through the directories within the parent folder, and removing any folders that match numeric-only filenames[20].

Filtering is performed via the script **messner**[21] which parses the output the linkage files in each of the three linkage program folders (Allegro, GeneHunter, Simwalk) and extracts the any linkage regions with a score above the estimated LOD score,

[20] The find command paired with a good regular expression is an excellent one-liner method of discovering these folders: `find ./ -type d -exec [ ls {} | grep \b[0-9]+\b ] && rm {} \;`

[21] See Program Listing section at end of chapter.

as well as the name of the flanking markers for each region. This result is stored in a messner_<program>.txt file in the parent folder.

Conversion operations are also performed at this step, with the **chromosomes.sh** taking the output result files of the linkage program folder it is invoked within (Allegro, GeneHunter, Simwalk), and generating graphical plots with the data by using the graphing tool **GNUplot** which outputs the graph in postscript format, later converted to PDF via the **ps2pdf** system utilities. The **mkhaplotypes.sh** script converts the haplotype output files generated by Allegro (ihaplo.out) and feeds them into **HaploPainter**, which then renders them using the Cairo library, and outputs them in the PDF format.

This happens automatically, with the final message of the pipeline prompting the user to run **readme_gen.py**, which generates the readme file using the saved logs from snpbutcher and pre-linkage user criterion to fill in the core of the data such as: project ID, genotyping chipset, number of markers used, recessive/dominant, and autosomal/X-linked. It is up to the user to fill in other fields such as: original genotypes filename, name of user/technician, and any extra notes specific to the run.

At the end of the pipeline, all files instrumental to the analysis of the data are extracted into a separate directory (outside of the parent folder) through the **collect.sh** script. The files collected are the 4 input files (map, maf, genotypes, pedigree), the readme file, the mendelian errors plot, the GRR relationship plot, the messner file, the linkage plot PDFs, the haplotype PDFs, and the pedigree PDF.

2.6 Linkage Pipeline Upgrades and Revisions

All that was described in the previous chapter was how the linkage program operates under normal input, the main dividing factor between normal and "unnormal" input being whether the pedigree size is above 19-bits, and/or is X-linked, and/or if the run is requested to be multicore.

2.6.1 Parallelization

The need for parallelization in linkage becomes apparent almost immediately after the first run; chromosomes are evaluated independently, and the program uses only a single core to perform the computation using a small percentage of the total RAM, whilst other cores lay idle.

An ideal run would utilize all the cores/threads available on a system at any given time, by processing several chromosomes at the same time. Such a process must first be thread safe, meaning that any two instances of the same program either use independent output resources (temporary files, output files, blocks of memory) or any shared resources are accessed mutually exclusively such that they do not write to the same resource at the same time[22].

Three scripts were developed under the parallel framework to ease this process, by queuing up chromosomes (or 'jobs') in a waiting buffer, and dispatching from it whenever a core (or 'resource') became available. The script **gh_parallel.sh** dispatched jobs from GeneHunter, and the script **allegro_parallel.sh** did the same from Allegro. The script **simwalk_multicore.sh** worked under the same principle, but operated differently because Simwalk splits its input not just into chromosomes, but smaller fixed-size loci. This has the benefit of being able to partially process an entire chromosome, since some of the sub-input files might have been processed and others not. This also allows the script to be *resumeable*, in the manner that the script can be terminated prematurely, and then re-run, but it would not start from scratch upon each chromosome, but would resume from the files it had not processed yet. The parallel description in the Program Listing (page 78) goes into further detail about the nature of these scripts.

[22] See Appendix page 243 for a quick overview on Semaphores

2.6.2 Distribution

The initial pipeline was built for a very specific platform in mind: Ubuntu 10.04. Ubuntu is a Linux operating system (OS) that is built pre-packaged with utilities, drivers, and codecs, that make it work for many general purposes without much prior configuration from the user. It is actively developed with two main major release models: Bleeding Edge (all updates, and support for a year until the next Bleeding Edge release), and Long Term Release (major updates only and supported for 3 years until the next LTR). Its base is derived from a more stable Linux OS (Debian) which exists upstream and is the source of all its major core package and kernel updates.

Linux OS's are structured under a tree schema, with all locations being derived from the root[23] / and most Linux subsystems exist off this root[24], locations of note being: /**usr**/**bin** and /**usr**/**local**/**bin**, each being included in the systems' PATH variable which allows all files within those locations[25] to be seen by the entire system without having to type out the entire filename.

E.g. if the binary **echo** is placed in the location /**usr**/**local**/**bin**/, it now exists as /**usr**/**local**/**bin**/**echo**. It is cumbersome to have to type out /**usr**/**local**/**bin**/**echo** every time we wish to use the command, so if the path /**usr**/**local**/**bin** is included within our PATH variable, we can use the command by simply calling 'echo'.

The pipelines interconnecting scripts, program scripts, and binaries, were solely contained within the /**usr**/**bin**/ and /**usr**/**local**/**bin** locations, meaning that they were mixed amongst the operating systems binaries making it very hard to distinguish between the two. One of the first operations performed in the second iteration

[23] Not to be confused with /**root**/ which is where the root user stores their files, the root user having full access to every part of the system. Other (less-privileged) users keep their personal files in '/home'.

[24] See page 243 in the Appendix for an overview of the Linux filesystem structure.

[25] It should be noted that inclusion is not recursive; any items within sub-folders at a given location are not included.

of the pipeline was extracting the binaries specific to the pipeline from these locations and storing them in separate locations.

The method required involved a brute-force approach of cloning the operating system and programatically deleting any scripts or programs not called by the pipeline. The **ldd** utility was also employed in this regard to discover the locations of libraries, and library dependencies so that these could be extracted too. It was thought at the time that the pipeline was version specific to the utilities it depended upon.

Once extracted, an installer script was developed and through trial-and-error and most of the vast library dependencies were removed as the pipeline became less Ubuntu/Debian based and more Linux-centric. The pipeline was then moved under the **git** school of **version control**, so that further development could be managed more effectively with a private backup on the code hosting site bitbucket.org. Changes to the pipeline could be performed upon separate feature branches without affecting the base code, and if the modifications were stable and successful they could be merged into base (or 'main') branch.

Under this new distribution model, the pipeline has since been ported onto more modern versions of Ubuntu (from version 12.04 to 15.04) as well as other Linux OS's such as Arch and Gentoo.

2.6.3 X-linkage

The first iteration of the pipeline was unable to generate the relevant input files required for X-linked analyses. Setting the Allegro and GeneHunter input flags to run chromosome X (as per their respective manuals) worked to no avail, instead resulting in numerous errors detailing inconsistent markers and penetrances; the main problem stemming from the two separate male and female penetrances required in X-linked inheritance models (see page 30 in the Background for more details).

There were several stages where this problem could originate, the most likely being the Alohomora input file generation stage, but closer inspection revealed that

the program merely re-formats the data and makes no assumptions upon the genetic model. The real source of the error messages was from the separate linkage programs themselves.

Upon thorough cross-checking between the input formats of the Allegro and GeneHunter, separate male and female penetrances could be set [26], and correct run configuration files could be generated for the setup. This process was automated via the script **x_makedat.py** which modifies the chromosome X "datain" input files by setting the "use X" flag[27], and producing an extra line of male-specific penetrances[28] specific to the dominant/recessive models within GeneHunter.

Another script (xallegroinhaploout) generates the correct the Allegro run parameters in an almost identical manner. The same script is also used after a linkage run to convert haplotype output files into a more consistent formatting scheme, where males which only have 1 allele haplotyped in the analysis, now have two (a Y-chromosome consisting of entirely zeroes), aiding in the haplotype visualization process when inspecting the results.

2.6.4 Large Pedigrees

Despite evidence pointing to the contrary [18, 19], neither Allegro nor GeneHunter could handle large bit-size pedigrees very well during our pipeline runs. Even if the number of informative markers were reduced to a 1000 marker range, which was well below the recommended limit for precision linkage, the programs would either terminate early or throw an unhandled exception during runtime.

Such behaviour are typical of out-of-memory / insufficient-resource errors, but most linkage runs use below 500MB of memory, and our hardware was more than capable of handling any resource constraints[29]. The problem was most likely occurring once again in the input files.

[26] As per the GeneHunter Manual, see:
https://www.helmholtz-muenchen.de/fileadmin/GENEPI/downloads/ghm-3.1.pdf.

[27] $0 \rightarrow 1$, top line third column.

[28] 7th line denotes female penetrances (3 values). Male penetrances duplicate a line above this, modifying the second value $0 \rightarrow 1$ if dominant.

[29] See Appendix page 231 on Hardware Requirements for further details

i GeneHunter

GeneHunter was relatively easy to fix , by simply changing its default MAXBITS size setting 19 to 30. This worked well for pedigrees within a range of 19 to 22 bits, but any larger and GeneHunter would split the pedigree or truncate individuals altogether in order to spare the computation. Flags to prevent these processes were alternated in numerous runs, but GeneHunter would either override them if it deemed that it could not or did not have the resources to perform the computation. The flags modified were:

- **PENETRANCE RESTRICTION** Heterozygous penetrances never drop out of a set range.
- **UNTYPED FOUNDERS <boolean>** Use untyped founders in the analysis.
- **SIMULATE UNTYPED <boolean>** Reconstruct missing genotypes, useful for haplotype reconstruction, but can be skipped for linkage.
- **DISCARD <boolean>** Remove less informative individuals.
- **MAXIMISATION (standard|dense|grid|small)** Changes the way the disease model parameters are varied when calculating scores.
- **MAXBITS <integer>** Increasing the default number of bits that the program deems as "large".
- **SKIP LARGE <boolean>** If set to false, it would attempt to process large pedigrees.
- **INCREMENT <integer>** Increase distance at which program bisects markers to compute score. In order to reduce system resources it was thought that fewer markers may ease the complexity of the analysis.

Due to the unresolved splitting problem in large pedigrees for GeneHunter, Allegro is the only linkage program used for large bit-sized pedigrees.

ii Allegro

Allegro required much more time and thought, resulting in a costly solution that involved patching a bug within the program binary itself.

The method in which Allegro spared itself extra computation was by making use of Multi Terminal Binary Decision Diagrams MTBDDs which provided a compact structure to access and store ongoing calculations without duplication or redundancy [20]. The MTBDDs were implemented via the Colarado University Decision Diagrams (CUDD) library [21], which at the time of Allegro's conception was on version 2.4.1 and included within the program source. Careful debugging of the errors thrown by Allegro led to this CUDD source, and it was assumed that the fault lay there.

Updating the library to the (then) latest version of 2.6 yielded no success[30], and the program would not compile due to various inconsistent function calls between versions. Many of these had been depreciated in favour of new ones that no doubt simplified and/or merged several tasks under a single call within the library itself, but broke specific functionality in Allegro.

The only other option left was to disable the CUDD library altogether, by forcing Allegro to skip the pre-existing computation checks, and attempt to keep all calculations within memory, caching to disk where possible.

This worked somewhat, and pedigrees smaller than 19-bits had no significant change in their performance. Pedigrees larger than 21-bits was the limit where the system resources started to become more scarce. The analyses would run, but very slowly and at the cost of quickly diminishing RAM and disk space. Haplotype reconstruction required almost no disk space, but often over 10 TB[31] of memory to function, the most of which was implemented via VRAM by using disk space as swap space. The default Allegro binary could run both multi-point parametric linkage and

[30] As of 2015 there is now a version 3.0 of the CUDD library, but the changelog reveals that only the overall packaging has changed, and the library code has not been modified since 2.6.

[31] Sadly not a typing error. Tera Bytes.

haplotype reconstruction in the same pass, but the newly patched binary could only now do one at a time.

Large pedigrees can now be run via the **allegro21bitstart.sh** script, which calls the newly compiled allegro binary over the input files, and takes an argument that allows it perform either haplotype reconstruction or multi-point parametric linkage (but not both). The user is greeted with a display detailing how much available swap space, memory, and cores/threads are available. If the scratch partition is not detected, the user is prevented from starting any processes.

The script also has the capability of parallelizing the process, but given the large memory demands of a single instance, this functionality goes largely unused unless the analysis is constrained to the higher (smaller) chromosomes.

2.7 Graphical Tools and Post Analysis Software

Once the data has been collected, it is then available for interpretation to determine whether any linkage peaks were actually discovered, and how informative they are in regard to the regions they are discovered in.

2.7.1 Linkage Visualization

Previously, the output results were simply collated by the GNUplot program and plotted as is, without any modification. This may sound agreeable in theory, but there were several issues that were not being addressed; namely that the plots were using genetic distance, and that all adjacent points were joined together.

Genetic distance, though an extremely useful representation of the expected number of crossovers across a region, does not represent the structure of the chromosome very well. Though genetic distance does scale fairly linearly with physical distance, the position of markers varies largely between different versions of the human genome, leading to some confusion in the actual location of a linkage peak.

The spread of markers are also not taken into account, since most SNPs do not fall within telomeric or centromeric regions, meaning that there should be a noticeable gap within the plot towards the head, tail, and off-middle section.

To add some context to the plots, the overarching script **physical_linkage_plot.sh** was developed with the intention of reading in the output files, and remapping the scores back to physical distance for a specified version of the human genome[32]. This is performed through the Python script **read_all_linkage.py**, which performs an additional quality check of splitting data over impossible regions, such as any linkage peaks spanning the centromere (possible due to the last informative marker on the p-arm having the same score as the first-informative marker on the q-arm). The script works under the principle of bisecting the data at fixed intervals, such that plots are not variably-spaced and prone to joining over long distances.

The re-plotting involves reconstituting the data within a GNUplot environment, and this is done through the script **plot_linkage_physical.py** which not only takes in the remapped data, but also a map file of chromosomal regions that dictates the true length of each chromosome (rather than the length inferred from the first and last markers in the analysis), and another map file of chromosomal bands which contains the names and regions of each band and sub-band of every chromosomal band. These bands are then overlayed upon the linkage plot to provide instantly visible context to some of the more questionable peaks in an analysis by allowing the user to see the density of the band the linkage peak falls within.

Genomewide and individual plots are generated under this method, with the individual plots being generated simultaneously using the parallel framework. The overarching script can handle output data from allegro, simwalk, and swiftlink; producing the same type of plots for each (though titled accordingly) so as to provide some consistency when comparing plots from two different program.

Previously, visual comparison of different linkage results from different programs was impeded by the varying size of the axes due to spacing inconsistencies introduced

[32]Currently hg19 is used, but any build within the UCSC Genome Browser can be used.

by different header labels, and that the analysis was constrained to the size of the analysis and not the size of the chromosome.

The consistent sizing provides a huge advantage when programatically comparing the data, since the points in all the remapped linkage output files all correspond to the same points, and comparing sets is a simple matter of addition/subtraction.

2.7.2 Haplotype Inspection and Rendering

Haplotype inspection is problematic. The allegro output haplotype file (ihaplo.out) follows the pre-MAKEPED structure for the first six columns (family ID, person ID, father ID, mother ID, gender, affectation) and then allocates all other columns to represent alleles horizontally. The marker headers are represented as hard-to-parse column headers in a fixed-width text structure that types them out vertically across multiple lines. An example is given below:

```
                    3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
                    2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5
                    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                    s s s s s s s s s s s s s s s s s s s s s s s s s s s s s s s
                    r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r
41 11 0 0 1 2 1 2 1 1 1 1 2 1 1 1 1 1 1 2 2 1 1 1 2 2 1 1 1 2 2 2 1 1 1 2 2
41 11 0 0 1 2 1 1 2 2 1 1 1 2 1 1 2 2 2 2 2 2 1 1 1 2 2 1 1 1 1 2 2 2 1 1
41 12 0 0 2 1 2 2 1 1 2 2 1 1 2 1 1 1 1 2 2 1 1 2 2 1 1 2 2 2 2 1 1 1 1 1 1
41 12 0 0 2 1 1 1 2 2 1 1 1 1 1 2 2 2 1 2 1 2 2 2 1 2 1 2 1 1 1 2 1 2 2 1 1
```

Listing 2.4: A small representation of an ihaplo.out file for two individuals (11 and 12, family 41). The rs marker identifiers are listed vertically as per convention.

Dealing with the actual file involves transposing and trimming the headers, and then inspecting the haplotypes with a text editor that uses window-space buffering such that the entire file is not evaluated upon opening. For example, text editors that do not have horizontal-buffering such as **gedit** and **notepad** experience severe bottlenecks.

Of course, browsing haplotype data alone is not enough for a haplotype analysis, since the real interest in the analysis is the representation of the haplotype blocks which cannot be shown in a plain text format. An application capable of resolving and representing the data as distinct blocks under a genetics model is required for a true analysis.

i HaploPainter

HaploPainter is the one of the most commonly used open source pedigree and haplotype rendering tools, built upon a framework of powerful and well-maintained Perl libraries such as Cairo and Tk, it can produce a wide variety of output formats from the graphics that it renders [22]. It is primarily used by researchers for drawing pedigrees, but is also used in rendering haplotypes.

Researchers can examine an allegro output haplotype file by loading it into the application, which then renders the pedigree and then the complete haplotypes trailing underneath each individual.

Though seemingly sensible in relation to the pedigree structure, this ultimately leads to several problems:

Pan and Zoom Inspection In order to inspect the haplotypes, the user is forced to pan and zoom across the screen in haphazard manner; the sensitivity of movement is set by the program and not appropriately scaled for zoom level, meaning it is a frequent problem for the user to overshoot their region of interest. Distressingly, the arrow keys are bound to other functions, and so the user must navigate sections using the mouse alone.

Miniscule Fonts in Large Sets Each haplotype representation alludes to a specific chromosome, which depending on which chromosome is being analysed can range greatly between 200 (e.g. chr21) to 2000 (e.g. chr1) for a typical 40k marker SNP analysis. Small marker sets can be easily rendered with normal fonts since the pedigree does not need to be shrunk to fit into working space of the application. Larger sets are more problematic, with minuscule fonts being required to fit everything into the working space. These fonts do not scale very well when being exported into other formats such as JPG or even lossless formats such as PNG or PDF.

Out of Range in Large Sets The problem of minuscule fonts leads to another issue that becomes apparent as soon as the user attempts to zoom into a region of interest: the minuscule fonts rendered may be just outside of the reach of the maximum zoom level, making the marker names and genotypes unreadable. Exporting to an image format such as JPG or PNG will only make this problem more apparent, since fonts will be rasterized into the image canvas at a fixed resolution, which even if set high will still not scale the fonts well.

Couple or sibling Analysis Only Since haplotypes trail vertically with the pedigree, a given region of interest is not contained within a single viewport[33], but appears once for each generation. For large complex pedigrees, the viewport quickly becomes localised to each set of offspring. This restricts the region of interest to a side-by-side comparison of either siblings or couples, which is somewhat limiting in dominant pedigrees when case and control individuals can appear in any generation.

Unsupported X-linked Haplotypes At the time of HaploPainter's release, X-linked analyses were not as common as autosomal, mainly due to the distribution of SNPs in earlier genotyping chipsets. As a result, the X chromosome was likely not extensively tested, and bugs were likely to manifest at some point.

ii haplo_region.py

In order to address the minuscule fonts as well as the pan and zoom limitations, the extremely feature-rich script **haploregion.py** was developed to produce a subset haplotypes file that would reduce the number of markers to just the region of interest. The region of interest is usually common regions of heterozygosity/homozygosity across cases which the script also produces.

[33] Cousins in the same generation may be vertically aligned, but likely a great deal of horizontal scrolling back and forth would be required to compare genotypes between them.

The script has many functions:

Producing Haplotype Subsets It does this by taking pairs of markers as argument as well as an optional flanking parameter. The script transposes the headers in the original file, and attempts to match the marker-pairs specified in the arguments to find consecutive non-overlapping regions within the file. The flanking parameter (by default set to 0) specifies the amount at which the region(s) of interest is expanded outwards, such that a flanking parameter of 10 would expand 10 markers left and 10 markers right of the region specified by the marker pair being evaluated. Due to the script usually preceding the modified HaploPainterRFH script, it requires not only the ihaplo.out file but also the map.N file (where N is a chromosome number) to be in the same directory.

Detecting Regions of Homology The script can also parse a ihaplo.out file to look for regions of heterozygosity and/or homozygosity between cases, depending on the inclusion of the respective settings specified in the arguments. If control exclusion is desired, this can also be optionally specified such that only regions of homology that exist in cases and not in controls are produced. A flag to set the minimum size of a stretch of homology can also be set, with the default value set to 2 such that sparse regions of homology spanning a single marker are not included in the final result (though of course this can be overridden). The general output of this is a list of marker pairs, but a debug flag can be specified to show the step-by-step working of the entire process. The debug output consists of a table with marker identifiers as rows and individuals (with identifying case/control tags) as columns, the data depicting the genotype of a specific individual-marker, as well as a boolean true/false score at the end of each row to show evaluation of cases (vs controls (if specified)) and current size of the stretch of homology.

Haplotype Extraction Haplotypes in the ihaplo.out file are depicted horizontally which is not the ideal format when trying to copy/paste haplotypes spanning a given region, since regions will need to be copied line-by-line manually by the user, which could introduce errors. The haplotype extraction mode works under the default assumption that all genotypes from all individuals are to be extracted, and it does so by transposing the headers and the data separately and then printing the data out into a table such that marker identifiers are rows, and individual identifiers are columns. It should be noted that this is the same format as the debug mode used in detecting regions of homology (outlined in the previous bullet point), but this mode does not perform any homology checks, and also takes optional parameters to specify a subset of individuals and/or a subset of markers. The resultant output file is then in a much more useable format, and copy/pasting haplotypes in a given region is facilitated via the conventional method of selecting a region of text that spans multiple lines.

Using the **haploregion.py** script in precession with **HaploPainterRFH** (a modified HaploPainter clone) produces a much more useable representation of the haplotypes that the user does not need to extensively pan/zoom through (see page 185 in the Results section for example images).

2.8 Program Listing

Symbols

Allegro Version 2.0f, Linkage program extensively used in the pipeline for all sized pedigrees. Makes use of MTBDD for compactness, and performs haplotype reconstruction in the same run as multi-point parametric linkage in small pedigrees. Larger pedigrees require a modified Allegro version to run where MTBDD is switched off. 54, 125, 149, 193

Alohomora A linkage analysis suite written in Perl, primarily used for its quality control modules such as pedigree checking and Mendelian error checking. It is also an excellent format conversion tool, and is primarily responsible for taking the input files set by the user at the beginning of the linkage analysis and converting into input formats for programs such as Merlin, Allegro, GeneHunter, and Simwalk. 50

CMake A cross-platform tool that assists in the creation of C++ Make files during program compilation. Relatively lightweight compared to qmake. 80

CPAN Utility to download Perl packages from the Comprehensive Perl Archive Network. 197

GNUplot GNU plotting tool that produces postscript files using a scripting language known as Ghostscript. 57

GRR The only windows binary in the entire program listing. The program has no commandline interface and can only be run graphically. In order to load genotypes, we must simulate mouseclicks via Xautomation and regularly poll the display server via Xwininfo to verify that the correct window has appeared in order to resume automation. 51, 146, 192

GeneHunter GeneHunter-ModScore v3, Linkage program used as a control for small pedigrees. Not as fast as Allegro, and has a tendency to split large pedigrees in order compute an analysis. Not used in large pedigrees. 54, 125, 149

HaploPainterRFH A modified clone of HaploPainter, with extra functionality added to be tailored more to the type inspection performed at the RFH. New features include the optional parameters of reading in a messner out-

put file and/or pairs of markers, plotting a transparent box of blue and/or red to highlight multiple regions of interest across multiple generations. A limitation of the script (as with the original HaploPainter) is that it requires both the ihaplo.out data file and the map.N map file (where N is a chromosome number) with a direct 1:1 mapping of each marker in the map file to each marker in the data file, with no tolerance for any overlaps. As a result, any modification of the ihaplo.out file (via haploregion.py) must be met with the same modifications within the map.N file. 70, 85, 184, 195

HaploPainter A Perl script/program that relies on the TkInter framework for drawing and parsing pedigrees with an interactive graphical user interface (GUI). It has a background batch mode that enables it to run without the need to draw anything, and it can also parse and draw haplotypes. Should an invalid pedigree be given to it, it reports to standard error and terminates. 50, 57, 85, 146, 194

Linkage One of the first general-purpose linkage program suites for use by the genetics community. Ground breaking at the time, but could only effectively carry out calculations when a single locus was used [23]. 35

Merlin One of the fastest linkage application suites; capable of parametric and non-parametric analysis, as well as various utilities such as gender checking, pedigree checking, and genotyping error detection. Unable to handle pedigrees over 18 bits without splitting. 53

Simwalk Written in Fortran, it is one of the most trusted comprehensive linkage analysis suites spanning two decades. The current version is 2.91 and it performs parametric linkage, non-parametric linkage, and rudimentary haplotype reconstruction by searching for the optimal chain amongst the various Monte Carlo Markov Chain simulations it performs. On top of the three main input files (pedigree, map, genotypes), Simwalk also requires a penetrance file to complement the map as well as a custom script file to perform user-specified instructions that dictate the type of analysis. Input files are split into chromosomes, and then further split into overlapping marker windows with a 25-marker length. The output files generated come in three main prefix types: SCORE, ERROR, STATS, and VIDEO; each being tailed

by an integer that maps it to a specific chromosome, and another integer to indicate which marker window is being examined. LOD scores are parsed from the SCORE files, and associative statistics with significance scores are detailed in the STATS files. ERROR files is where standard error is duplicated, and VIDEO is where standard output is duplicated. 54, 125, 170, 193, 203

SublimeText Fancy text editor with syntax highlighting, multiple variable renaming, and general project control. 96

Vitesse One of the the first major linkage programs capable of multi-point linkage analysis for reasonably sized pedigrees, using "fuzzy inheritance" to recode genotypes in a stochastic manner to infer transmission probabilities [6]. 36

allegro21bitstart.sh 64

allegro_parallel.sh Script that parallelizes Allegro runs so that several chromosomes run at once. For small pedigrees it is possible to maximize all threads, but larger pedigrees typically only run 2 threads at most due to RAM constraints and disk-write/paging bottlenecks. 58

auto_stage1.sh First filtering script: Generates pedigrees, primes the MAF file, performs gender checking, and plots GRR. 49

auto_stage2.sh Second filtering script: Produces Merlin plot, runs pedigree check to find unlikely genotypes list. 49

auto_stage3.sh Third filtering script: Creates linkage input files based upon the unlikely genotypes file for Allegro, GeneHunter, and Simwalk. 49

chromosomes.sh Generates chromosome-specific linkage plots via GNUplot. 57

clodhopper.py A Python script to estimate the maximum LOD score of an analysis for either Simwalk, GeneHunter, or Allegro. The input files generated are program specific, but all work under the same principle of creating three distinct haplotype blocks: affected individuals (12^g, 11^h, 12^g), unaffected individuals with unaffected parents (12^g, 12^h, 12^g), and unaffected individuals with affected parents (12^g, 22^h, 12^g), where g and h are positive integers and $g \neq h$. The implication being that all affected individuals are subject to the same (in this case, double) recombination event to create a singular well-defined linkage peak spanning h markers. It is then trivial for linkage

programs to evaluate this. 55

collect.sh Collects all relevant linkage data from the working directory to be packaged for review. This involves: GRR charts, Mendelian error plots, Pedigrees, Linkage plots (converted from PS to PDF) and haplotypes. 57, 200

cygwin A Linux environment and shell made accessible from Windows. Programs must still be compiled and built under cygwin to operate under it. 81

echo Prints a line of text. 59

emacs Fancy curses editor with syntax highlighting, autocompletion extentions, custom Macros. Very feature-rich. 96

gedit Simple plain text editor. 66

gh_parallel.sh Clone of the allegro_parallel.sh script rewritten for GeneHunter using the parallel framework. Works well on small pedigrees, but is ineffective for large pedigrees. 58

git A popular type of version control. 60

haploregion.py A Python script to address the large marker sets in haplotype output files from Allegro (ihaplo.out). It can produce haplotype subsets, detect regions of homology, and extract haplotypes. Works in conjunction with HaploPainterRFH for which it provides input files. 68, 70, 183

ldd Shows the shared libraries called by a program by inspecting the programs' dependencies. 60

linkage_pipeline.sh Overarching pipeline script responsible for parsing user inputs and starting a screen session, before calling the appropriate linkage pipeline script. 47

messner Python script to parse the output linkage files of Simwalk/GeneHunter/Allegro and find linkage peaks higher than an amount specified by an argument (determined by clodhopper.py). GeneHunter positions do not match any real marker positions, so the script has to bisect the data in order to closely-approximate scores for GeneHunter. 56, 183

mkhaplotypes.sh Runs HaploPainter upon ihaplo.out Allegro data to be converted into PDF. 57

nano Simple curses plain text editor, some syntax highlighting. 96

notepad Simple plain text editor. 66

parallel The GNU parallel framework is an encapsulating program that takes an existing program and a list of argument parameters, and then dispatches each argument to a separate clone of the program, running them simultaneously. It is implemented via a queue-and-dispatch system, where jobs are added sequentially to a queue buffer, and are dispatched in a first-come-first-serve manner to a given work resource until there are no more resources left. As soon as a job finishes, a resource is freed and another job can be dispatched from the queue. The number of active jobs at any one time can be set manually and can also be dynamically changed during runtime. For efficiency, the number of active jobs should be the number of cores/threads that the CPU can handle, though if the task is memory-intensive then the number of jobs should be limited to the total amount of available RAM divided by the maximum memory usage of that task. Most linkage programs process an entire chromosome at once, and so jobs are queued as chromosomes, with small chromosomes such as chr21 finishing quickly compared to chr1. This is problematic if chromosomes are queued sequentially (either ascending or descending), as the lower-numbered chromosomes can only be effectively processed two at a time whereas the higher-numbered chromosomes can be processed five at a time, requiring a daemon to watch over RAM usage and change the number of active jobs parameter accordingly. This is not always effective, since the daemon must know beforehand how much system resources will be consumed for a given chromosome, and this varies with bit-size and the number of markers (which varies greatly between projects) and so the daemon must make conservative estimates so that the system does not run out of memory and begin killing processes. A better compromise which requires much less oversight is to alternate chromosomes by counting from either end and meeting in the middle (e.g. the queue order would be: 1, 22, 2, 21, 3, 20, 4, 19, 5, 18, 6, 17, 7, 16, 8, 15, 9, 14, 10, 13, 11, 12) with at most 3 jobs running at once. Linkage programs that utilize a sliding-window approach (such as Simwalk) split input data evenly into windows of n markers, regardless of chromosome, and so jobs can be more efficiently queued without any resource monitoring at all. 49, 76

pedcheck Binary that detects Mendelian errors and generates the list of unlikely markers to be excluded from the analysis. Called from Alohomora. 50

physical_linkage_plot.sh Plots Allegro/Simwalk/Swiftlink data using physical map instead of genetic map, moving genetic maps into their own "Alohomora" folder. Calling script for read_all_linkage.py and plot_linkage_physical.py which it uses to produce chromosome-specific and genomewide plots. 65

plot_linkage_physical.py Plots physical linkage map in Mbp for a specific chromosome or genomewide, with parameters to specify chromosomal bands, and a threshold amount to unlink markers spaced apart more than a certain threshold amount (default 1000 Mbp), as well as a minimum LOD specifier. 65

prelim_check.py Compares the individuals found in the genotypes file header against those found in the pedfile name column to see how the sets intersect. If any are missing, a warning is thrown to the user to either terminate the application or ignore. 48

ps2pdf converts postscript files to PDF. 57

qmake A cross-platform tool that assists in the creation of C++ Make files during program compilation. Relies on the Qt framework. 80

read_all_linkage.py Makes a genomewide physical linkage map for UCSC genome graphs with LOD scores for Allegro/Swiftlink/Simwalk/GeneHunter. Must be run in their respective folders, and given the map file as first argument. A minimum LOD parameter can also be specified (default is -3). 65

readme_gen.py Readme generator that automatically detects project name and ID, bit score, date, number of markers, and looks for configuration files to determine the type of filtering performed at the snpbutcher stage as well as the type of pipeline analysis (autosomal/X-linked)(recessive/dominant)(singlecore/multicore) as written by the pipeline log file. It still relies on manual input of the technician's name, the genotype chipset, and the original genotypes file. 57

rsync Better copy utility with speed optimizations and resumeability features that make it ideal for making backups with. Can operate using its own rsync protocol, or otherwise uses ssh to securely transfer files. 201

scp Secure copy utility that operates through ssh, allowing the remote transfer of

files. 201

screen A program that allows for multiple users to access the same terminal session and co-operatively witness and interact with users sharing the same session. It has the added benefit that it detaches the parent process from the application it was spawned from (usually the desktop manager), such that the process is not killed when the application is closed. De-parenting a process is not exclusive to a screen session, but re-accessing the same terminal to resume interacting with an ongoing process is the main benefit of the program (though the tmux program is also capable of this). 48, 201

simwalk_multicore.sh Overarching script that queues parallel jobs to Simwalk. Simwalk uses a sliding window approach, and its input files are split accordingly, meaning that analyses are not locked to a single chromosome but are distributed as smaller pieces that makes them much more even to distribute. The script is fully resumeable by simply checking for the appropriate output file and determining whether it is populated with the correct number of markers before moving on to another job. Very efficiently parallelized. 58

snpbutcher snpbutcher v2.8, a Python script that filters a genotyping map file into a subselection of informative markers based upon MAFs and genotypes. By default it searches for informative SNPs using HapMap project data [24], but if none is found or the –nohapmap flag is given, then this step is skipped. It can be told to space markers by cM distances (default=0.05) or to determine spacing by a maximum number of desired SNPs. Under these options, uninformative SNPs and those with non-existent minor allele frequencies are skipped. 46

ssh Open-source secure shell protocol to remotely log in to a remote machine that runs a SSH daemon and is listening for incoming connections. For security the default listening port of 22 is changed to 6622 to hide it from over-curious ping probes. 201

sudo Utility that allows a user to perform tasks under the security privileges of another user (default:root). 226

version control A means to store incremental changes between two file edits and synchronize across multiple hosts. Edits can be rewound, merged with other

edits, and forked into separate branches of development. Useful for coding. 60

vi Simple curses plain text editor. 96

wine Compatability layer to run Windows programs on Linux. Not an emulator, instead actual DLL libraries are rewritten and compiled under Linux. 200

x_makedat.py This script converts the standard Alohomora output DAT (and Allegro/GeneHunter input) files and resets the headers so that an X-linked analysis is specified. 61

zenity Utility to show and manipulate graphical dialogs and return input back to the command line. Uses GTK framework. 78

3. Methods - HaploHTML5: A Comprehensive Pedigree and Haplotype Analysis Suite

The problems with current haplotype visualization tools highlighted in the previous section prompted the need for a new tool that would meet the shortcomings of its predecessors. Such a tool would have to be developed with a more modern perspective on the needs of both the geneticists and the data they deal with.

3.1 Application Requirements

In order for the application to be of use in a post-analysis haplotypes context, the following specifications must be met:

Haploblock Resolution Haplotypes must be correctly parsed and fielded into distinct allele blocks that describe recombination intervals within founder alleles. This applies to all penetrance models; recessive/dominant, autosomal/X-linked.

Haploblock Rendering The application must be able to correctly render the blocks into coloured rectangles behind the haplotypes, with each coloured rectangle representing a found allele group, and any recombination events being represented as a distinct break between two coloured blocks.

Pedigree Generation In order for the application to be a true replacement for HaploPainter, it should also be possible to draw entirely new pedigrees, and export them to supported formats.

Format Flexible Haplotype files can come in a variety of output formats; Simwalk, GeneHunter, Allegro, and many others. The application should be able to detect and read from these formats, as well as export to them.

User Accessible It should not be hard for the user to obtain, install, and run the application. There should also (ideally) be no licensing barriers preventing the user from using the application in any kind of free context.

Intuitive Analysis Interface The application should enable the user to compare any number of differently affected individuals in a determined region/locus of their own choosing. The region of interest should be simple to specify, as well as being easy and fast to manipulate without hindering the overall analysis process.

(Optional) Analysis Tools It is likely that the user will be inspecting the haplotypes with the intention of discovering regions of homology between cases and controls. It would be beneficial to the user to programatically aid this process by providing a utility of some design.

(Optional) Resumeable The user should be able to resume an ongoing analysis without having to reload the same data twice.

(Optional) Annotation Tools The option to add custom annotations visible either across the entire view, or within a specific region of interest.

(Optional) Shareable The user should be able to share an analysis with another user. Privacy controls would be required, and patient identifiers would have to be hidden to alleviate patient data sensitivities.

Before development can begin to address the requirements set out in this schema, we must first explore the various frameworks that would be used to create such an application.

3.2 Development Frameworks

The first step in any development process is to choose a basis that would allow us to program more efficiently without having to "reinvent the wheel". There are many ways to perform the same task, but some ways are more geared towards certain types of tasks than others, with each method having their own respective advantages and disadvantages.

E.g. a data-mining task that requires some mathematical capabilities as well as the ability to read/write to file. If hardware constraints are not an issue, then a large encompassing framework that provides a vast array of utilities such as file operations, operating system non-specifics, and even its own filtering functions may suit the task well, albeit at the cost of high memory and CPU resources. However if the focus is more upon optimization, then maybe a smaller underlying framework that provides only basic file utilities may be sufficient for the task.

The problem, of course, is the constant balance between defining just the right amount of specificity in the task at hand without compromising the ability to implement further features at a later time. A constant mantra spoken to generations of university undergraduates in computer science: "Never optimize too early".

3.2.1 Programming Frameworks

Programming is a transferable skill; practice in one will aid in the practice of another, and most imperative languages share a great deal of similarity between each other in terms of structure and syntax. Here we will compare a wide range of popular programming languages: Bash, C++, Java, Javascript, Perl, and Python. There are many concepts that a developer must consider when choosing the appropriate language to program in. Different languages offer a different selection of the features and concepts outlined below.

i Licensing and Access

The applications produced by different languages are typically split into two groups; scripts and programs, each with their own forms of accession and distribution.

Scripting languages such as Bash, Javascript, Perl, and Python [25] are high-level runtime dependent languages written in plain-text into files that can be directly executed by their respective interpreters. Scripts are said to be open source, where to share or distribute the script is to openly share the code source with all users.

Programs allude to languages where all the files containing the code are used to generate a binary that contains a reduction of all the files related to the application parsed into machine code; namely C++ and Java. These binaries are distributed to end-users but the users cannot see the code source used to create the binary[34]. It is then up to developer to share their code (making the program open source) so that the user can compile the binary themselves, or the developer can choose not to share their code at all (making the program closed-source) and it is then up to the user to decide whether the binary is trustworthy or not[35].

Open-source applications are also avenues for the misuse of the developers original intention to the distribution and modification of their code[36].

ii Multi-Threading

There are times when an application often needs to execute multiple tasks simultaneously; usually either to speed up the computation of an operation by delegating it over multiple processes, or to provide operability and feedback to the user about an ongoing background task. Some languages implement threading automatically[37], but others may require more work from the developer.

[34] here are however numerous methods to reverse-engineer the code source from a binary.

[35] The problem with closed-source development is the room for abuse in which code can be written with malicious intent without the end-user knowing about it.

[36] For an overview of the various types of licensing in digital media, please see the Licensing section on page 237 in the Appendix.

[37] Even abstracting it from the developer entirely by incorporating it intrinsically into the language. See page 95 on Javascript.

Threading is mainly implemented via three methods:

1. **Run and Wait**

   A thread is bound to a function and executed immediately in parallel to the normal flow of code execution (a method known as forking). The rest of the code execution resumes normally until it reaches a portion in the code where it is specified that it must wait and block all further execution until the thread finishes, where control is then handed back to the main process. Numerous threads can be spawned, and numerous block calls can be made. Bash implements this behaviour by default using the '&' modifier to fork off a process and the 'wait' command to block. C++, Java, and Perl incorporate this basic forking methodology too in their respective standard template libraries.

2. **Queue and Dispatch**

   Threads are not executed straight away but are treated as jobs that are added to a queue. Jobs are then dispatched from this queue on a first-come first-serve basis to any available threads. If there are no available threads, then the jobs wait in the queue until there are. This method is used primarily in batch processing where the job types are similar and the code is parallelizing data would otherwise have been iterated through via a standard for loop. Not many languages incorporate this natively, with Bash relying on the parallel framework[38], and other languages dependent on external implementations.

3. **Asynchronous Callback**

   This structure does not follow the traditional sequential flow of control; rather that there is no "main" flow of control, and all code blocks instead inhabit separate realms which are all assumed to be independent of each other and can be executed concurrently, unless a dependency between a group of blocks is detected in which case blocks within the group are executed sequentially.

[38] See **parallel** specification within the Program Listings (page 78).

E.g. A,B,C,D,E,F are six separate functions, some sharing inter related variables. C is spawned from D, and E is also spawned from D, but variables in E depend upon the outcome of C. F is spawned from B but depends upon the outcome of E. A representation of this scenario can be seen in figure 3.1.

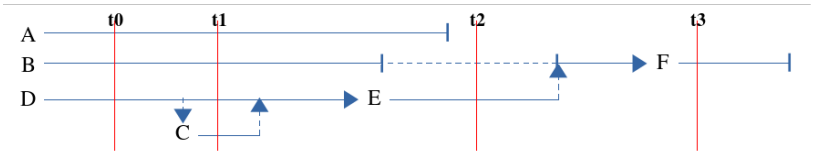

Figure 3.1: Horizontal lines depicting the number of active threads with the respect to a horizontal time axis. Vertical down lines depict spawned processes, and up lines indicate the return of code control to another process. Dashed lines represent a process waiting for another. Arrows highlight callbacks where the functions are executed upon the termination of another. At t0 there are 3 concurrent processes; t1 there are 4; t2 and t3 there is 1.

This asynchronous callback structure is native to Javascript, where there is no guarantee to the order in which two separate lines of code will execute; the interpreter decides what load order is best based upon a dependency model it creates. The Qt framework which has bindings in both C++, Python, and Java, also incorporates this model using a "signals-and-slots" architecture where functions are bound to 'slots' (code blocks) which emit 'signals' that prompt other slots into action, where a termination signal is a holder for a callback function.

Each type of multi-threading process is suited to different tasks, but the asynchronous callback architecture removes a lot of the user-imposed blocking processes and automatically derives the most efficient method to handle concurrent tasks. In order for a developer to effectively wield the other two methods, they must have at least a general understanding of mutual exclusive variables and semaphores[39].

[39] See Semaphores section in Appendix on page 243.

iii Graphical Libraries

Graphical libraries allow a developer to use existing styles and display methods that can aid them in the application development process, without the need to write their own graphical libraries from scratch. A good graphical library has its own set of buttons, windows, progress indicators, input fields, and other form selection items that the user can pick and choose from.

A common application paradigm is the Model-View-Controller (MVC) principle which separates application components into the Model (which holds the data in memory or local storage), the View (which is the graphical front-end and contains placeholders for Model data to be represented within), and the Controller (which interacts between the View and the Model).

Most languages have libraries that incorporate this popular model in some way, the exception being Bash which is not suited for graphical applications but has graphical input helper utilities such as **zenity**, which can be used to sequentially prompt the user for input.

The Qt framework has a graphical interface in which a developer can actually draw the input form by dragging and dropping buttons and other display fields into horizontal/vertical/grid-like layouts to automatically space components[40]. Javascript under a browser context has the entirety of the HTML5 specification to play with; the most relevant item being the canvas specifier, which allows for direct drawing to a webpage without any layout constraints at all [26].

iv Application Programming Interface (API)

The Application Programmers Interface (API) is a full index of how the developer interacts with the language or framework. If the API has a complex function hierarchy or class structure (or lack thereof), it may require a steep learning curve until the developer fully grasps how to wield the various features of the language/framework

[40] Or the developer can ignore tiling altogether and manually specify their own dimensions.

effectively. If the API is more straight-forward, it is quite likely that the developer will not have to waste time on API specifics and can focus on tackling more application-specific tasks.

Good APIs also must be well documented, meaning that the official documentation provides a good explanation of the concepts and functions behind the API, as well as small working examples for those who are already familiar with the concepts and merely want a quick-start. Another feature of good APIs is balancing the fine line between no development and rigorous active development; the former being limiting factor in future improvements, and the latter being a general hindrance in current development if methods are constantly being added or depreciated.

Perl and Bash are not as modern in terms of active development then Python, C++, Java, or Javascript. As a result their documentation is somewhat harder to find, but they have their own active communities with users who can provide excellent feedback and examples. The Qt framework has a clear and extensible API for all supported binding platforms, and is well documented with good examples and code standards. Javascript is somewhat of a scripter's preference and is one of the most rapidly evolving and diversely populated languages out there, essentially meaning that stable API's are hard to come by, and projects need to be version specific to the framework's used.

v Portability and Performance

Some languages can be ported more readily between different systems than others. If the developer already knows what specific system(s) their application will run in, then portability may not be a problem. The cost of portability is the resource cost of the language interpreter (or runtime environment) required to run the application. If the language is compiled, then depending upon whether it has been translated into byte-code (for an optimized interpreter to process), or machine-code (for the CPU to process), then this resource cost is greatly reduced.

Languages that are very feature-rich often experience slowdowns associated with the bloated libraries and background daemons required to run them. This is often the case with high-level languages that provide a level of abstraction from the developer in order to simplify the development process as much as possible. Advances in compiler and toolchain technologies [27] have reduced the performance cost of these high-level languages by providing optimization techniques that can reduce the code into a platform-independent intermediate representation which can be converted directly into machine code. Programming now exists in an age where there is greater emphasis in producing clean and readable code with the intention of modularity and team collaboration, rather than needlessly efficient code from conception; all the optimization is done by the interpreter or compiler[41]. To go further, it is even said that there is no such thing as a "true" low-level language anymore, since even sequential assembly code fed straight to the CPU still undergoes optimization by categorizing related variables into separate dependency trees which are then executed in parallel to speed up computation. The advent of multi-core CPUs has given rise to advanced instruction sets, allowing for data to be channelled into hardware CPU parallelisms such as: Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), and Multiple Instruction Single Data (MISD).

C++ is compiled straight to machine-code but is historically not a very portable language; code written for one platform will likely not translate well to another. Cross-platform frameworks and compilers have been created to address this such as Qt (via **qmake**), and **CMake**, but the process requires much prior configuration and often OS-specific macros need to be included to get an application running.

Python, Java, and Javascript are interpreted languages that require their respective runtime environments to work, but as stated previously massive advancements have been made with compiler targets that provide speedups such as just-in-time (JIT) temporary binaries for portions of code known to otherwise create slowdowns.

[41] See Compilers section in Appendix on page 239.

Bash exists solely on UNIX and UNIX-like systems, and has only been ported to Windows systems via compatibility environments such as **cygwin**.

vi Programming Style

There are several styles or paradigms in programming to aid in the process of either keeping functions within a specific scope, or providing flexible contractions for familiar or overly-used code blocks:

Object-oriented programming with a focus of encapsulating functions into 'classes' that are specific to them, and creating 'instances' of a class, which are able to hide and share methods and variables between other instances of the same class or other classes.

Recursive programming which enables the same function to be called within itself repeatedly until some terminal condition is reached, using minimal amount of code.

Lambda programming a style that borrows heavily from recursive strategies to perform the same task upon an array of items for the purposes of mapping, filtering, or condensing the array.

Pointers the ability for the developer to access specific system resources for the intention of optimization. An example would be accessing a portion of memory populated by another function. Pointers allow for direct modification to a variable by proving a reference to either another process in control of it or the actual address of where that variable lies within memory. This removes the need for the default procedure of modifying a variable wherein the contents of the variable are directly copied into a temporary buffer by the function using it, operated upon, and then copied back (or overwritten) to the original variable. Many high-level languages manage this automatically by providing the developer with a pointer for large variables and accessing functions associated

with the pointer by abstracted means. More low-level core languages leave it up to the developer to create pointers and to delete them after usage.

Garbage collection a background process belonging to the runtime environment that automatically detects when a function or variable falls out of 'scope', i.e. it can no longer be accessed after a certain point, or is not used again in later portions of the code. Items that fall out of scope still occupy portions of the memory and should be removed. The garbage collector performs routine sweeps upon the memory used by the application and seeks and destroys these pointers in order to free up resources. Garbage collection is a costly process that requires pausing the execution of the current application in order to perform its task before resuming it again[42].

Table 3.1 outlines and compares the differences between the various languages being compared with respect to the programming styles described in this section.

Language	*Object-Oriented*	*Recursive -Functions*	*Lambda*	*Pointers*	*Garbage Collection*
Bash	N	Y	N	N	N[1]
C++	Y	Y	N[2]	Y	N[3]
Java	Y	Y	N	N	Y
Javascript	Y	Y	Y	N	Y
Perl	N	Y	N	Y	Y
Python	Y	Y	Y	N	Y

Table 3.1: Comparison of different programming styles. Notes:[1]variables must be unset, [2]C++11 specification does support this but not widely used, [3]however smart pointers perform their own [28]

[42]Indeed, some languages do not implement it at all in the interests of speed and optimization, and leave it up to the developer to clean up their own garbage!

vii Type and String Handling

The aim of a data type is to capture the information within an item of data using the fewest amounts of bits possible. Snippets of text containing numeric data could be more efficiently used if the numeric data was stored in a numeric type. Most programming languages are built upon a foundation of well-established data types called primitives[43], which are known for their efficient handling of information. Primitives allow the developer to store the data in set compact representations of their own volition. However, some languages perform automatic typing such that the type of variable is never explicitly stated but inferred from the context in which it is used. This has the advantage of freeing the developer from hardware constraints and overflow/underflow errors, but may represent large amounts of data inefficiently. Some languages do not even use primitives, but treat all data as an object, a concept borrowed from object-oriented languages that allows a variable to have functions and properties attached to it.

String data is harder to quantify, since a string is essentially just an array of characters, and depending upon what character encoding is used (ASCII, Unicode-16, Unicode-32) string can take up an arbitrary amount of data. Strings[44] are immutable, meaning that they are not editable at runtime and require powerful libraries to workaround this behaviour.

C++ and Java classically follow the primitives directive, but newer versions also allow the inclusion of automatic type specifiers. Python distinguishes between decimals and floating-point numbers, but automatically scales the number of bits required to represent both. Javascript treats all numeric data as floats, but has devices to remove redundancy over number arrays.

C++ has poor native runtime string handling, with modifications on strings requiring unintuitive function calls to C methods. Java, Javascript, Python, and

[43] See Appendix page 233.
[44] See Appendix page 234.

Perl have excellent string handling, with string being a class with basic methods that allow a string to be split, copied, and appended/concatenated.

Bash takes a different approach altogether and treats everything as a string, with numeric operations requiring the string to be fed to an external program for processing (where upon return it is converted back to a string). It is not without consequence that string manipulation in Bash is unparalleled, with native concatenation and many methods to perform variable substitutions as well as in-built regular expression (regex[45]) matching.

3.2.2 Conclusion of Programming Languages

Programming is a transferable skill; knowledge in one will aid in the knowledge of another, and most imperative languages share a great deal of similarity between each other in terms of syntax.

It is clear that C++ will offer greater control and optimization to an application, but will be prone to portability hangups as well as various bugs related to type checking and pointer misuse; issues that could severely hinder the development process. Further, C++ does not have intuitive graphics support and requires extensive frameworks (such as Qt) to perform even basic drawing tasks. The same is true for Java, with the extra caveat that it would require a resource-heavy runtime environment for the application to run as well as the constant threat of Java updates to deter users from using it at all.

Perl (and to some extent Python) are composed of modules that are compiled upon installation and run just as fast as any binary except for the overhead in repeatedly calling it from the runtime environment. Perl and Bash are excellent scripting languages, but can get very lengthy to manage even if the source is split over several files.

In a graphics context, Bash does not have any graphical frameworks associated

[45]See Appendix page 235.

with it since it is primarily used to automate system calls. Perl on the other hand has good bindings with visual libraries such as Tk and Cairo, and it was with this language and these very frameworks that **HaploPainter** was written. However a cursory glance at the source code reveals a very lengthy (albeit well-formatted) list of functions and variables that would require extremely careful deliberation to program with. The **HaploPainterRFH** modification only appended 50 lines of code to the original, but the patching was not a straightforward process; repeated scrolling up and down the code to check if a variable was still within scope.

In terms of the application requirements, the victor seems quite clear; Javascript is a well supported scripting language that caters for a variety of different programming styles, as well as excellent graphics support and an inexhaustible choice of graphics libraries and supporting frameworks to choose from.

Javascript's asynchronous callback structure can take some getting used to, but allows the developer to focus on the task at hand, rather than worrying about specific multi-core optimizations. In-browser development has the added benefit of being able to quickly see the changes made at any stage in the development process due to the quick startup time needed to refresh an application. Many modern browsers even allow for runtime editing such that variables can be modified in real time with changes being displayed immediately; a feature that should not be under sold, for it greatly increases the speed of the development process.

One concept that was not explored in the previous section due to the uncertain nature of it is the "shelf life" of a language. The easier it is for a language to program in, the more likely it will have longer popularity amongst developers as well as better future support since more people would have invested their time in it.

Languages come and go, but Javascript is here to stay with every major browser supporting some version of it; an act that prompted the drive for hardware accelerated graphics via the WebGL and canvas initiative that has only further ensured the future of the language.

3.2.3 A Case For Open Source

Before we go any further, the need for free and open source software (FOSS) within academia should be addressed. The role of scientists and researchers is to push the boundaries of their understanding of a concept in order to make new discoveries that benefits not only them, but their field, and in turn, science in general.

The scientific principle works under the notion of iterating towards a consistent truth model through the use of reproducible peer-reviewed experiments and analyses. In order for this to be the case, other scientists must be able to assess these experiments by accessing the same data sets and the same tools or methods to reproduce the same results. The whole process must be transparent; with no ambiguity at any stage that could cast doubt on the experiment in question.

With open source software, an inquisitive scientist has the freedom to take an existing method or tool and tailor it better to their experiment[46], and a reviewer can then go through the source code and understand the method in which data was analysed.

The transparency of this process becomes somewhat clouded as soon as more black-box methods are employed; software binaries or proprietary owned cloud-computing models.

Binaries can offer more clarity if their source code is available, since the code can be compiled and tested against an input data set to make sure the result between two binaries of the same stated source is actually the case.

Closed source binaries are more untrustworthy. They may employ known methods which can be evidenced by the input set and the processed output set, but there is no actual guarantee that the program performed the method correctly (e.g it may have seen that input set before and produced the output from a stored source, the method employed might be badly implemented and only works in a few use-cases, etc.).

[46] Depending on licensing. See Appendix page 237.

Proprietary cloud-computing solutions are even worse in this regard; essentially forcing the researcher to surrender their data to another location so that they can perform all their tests offsite within the platform itself. The researcher may then be at the mercy of the platform when it comes to releasing their results and showing their methods.

The issue of privacy and ease-of-use is another factor. The ongoing battle to keep free information access as open as possible has been met with many pitfalls; net neutrality, IP blocking, DNS ownership, purposefully weakened encryption, and untrustworthy proprietary security protocols.

However, client security remains a boundary that not many are willing to compromise upon, mostly because of the privacy issues related to client data. As a result only an open source language could be trusted to run on the client machines, and a large-scale effort has been undertaken to ensure that this is the case with Javascript[47].

3.3 HTML5 and Javascript

Before HTML5, media support for web browsers was managed by extensive (insecure) plugins that each browser handled in their own different ways (ActiveX[48], and NPAPI[49]).

Flash used to be the main plugin that covered all the media requirements that browsers would not; video/audio, and graphics. Other plugins also existed (Shockwave, Silverlight, media player extensions) but these suffered from cross-platform incompatibility issues and licensing disagreements.

Flash is scripted in a language called Actionscript which is also ECMAScript based, and developed a very longstanding loyal following with the new generation of

[47] Even corporate institutions such as Google, Apple, and Microsoft have aided and given support to this initative [29].

[48] Traditionally supported by Microsoft, though no longer in their current Edge browser.

[49] Used by Java, Silverlight, Unity, and others. Chromium based browsers no longer support it and use their own PPAPI (Pepper API plugin API) [30].

game developers and web advert designers because of the ease-of-use it provided for dealing with vector graphics.

However the closed-source nature of Flash[50], paired with continuous security risks and update requests, the non-free proprietary nature of its IDE, and its reluctant uptake of hardware acceleration made its approval wane over time. The final nail in coffin was sealed however, when Adobe dropped Flash support for Linux altogether in 2012[51] and effectively managed to distance itself from a large portion of its fanbase.

This created a noticeable gap in trusted content, and the need for a new specification that could provide media support without the need for insecure/proprietary plugins prompted the conception of the HTML5 schema that all modern browsers strive to follow.

Javascript previously was just a language to manage the various background tasks in webpages, but the arrival of the HTML5 schema as well as developments in ECMAScript6 have made the language more graphics-centric within browsers, and multiple visual libraries exist to cater for the various 3D and 2D contexts that the new canvas element can cater for [31].

3.3.1 Javascript Overview

Javascript's open source asynchronous callback structure may seem like a major diversion from standard programming styles and principles, but the (somewhat reluctant) uptake of said principles in other languages only reinforces the notion that Javascript is setting the stage for things to come.

Javascript in the context of general web page management makes use of the Document Object Model (DOM) for manipulating elements of a page. The DOM model is a nested collection of inter-related objects following a parent-child hierarchy such that each child can reference their containing parent, and each parent can iterate through their child elements. Elements are accessed using unique identifying tags, or via multiply-specifiable class names.

[50] Created by Macromedia and later acquired by Adobe.
[51] See: http://www.adobe.com/devnet/flashplatform/whitepapers/roadmap.html

Javascript follows this Object model faithfully by treating all variables as Objects that can be extended by methods. Primitives do exist, but they are interchangeable depending on context. Indeed, a long running joke amongst developers is how the equality/assignment '=' operator takes on multiple meanings depending on context[52].

Numeric data used to be a problem in Javascript prior to ECMAScript version 6, since the numbers would be converted into a floating-point type that would take 32 or 64 bits of memory depending on the platform.

This large overhead does not translate well for small numbers, but makes sense in the automatic pointer control context; since all data are Objects and Objects are passed as references which are simply memory addresses (that are once again either 32-bit or 64-bit depending on the platform). It therefore does not matter significantly if individual numbers are passed this way since they occupy the same space in memory under the Object model.

However, large collections of numbers (or number arrays) quickly begin to eat at memory resources under this model. To get around this, Type Arrays were introduced in ECMAScript6 which bounds numeric data into elements that fit them better; with signed and unsigned arrays with element sizes of 8/16/32/64 bits.

Consider haplotype data which in the case of SNPs represents exactly 3 values: 1 (first allele), 2 (not first allele), and 0 (error). This can be captured very easily by 2-bits ($2^2 = 4$ unique addresses), but under the old array model each haplotype would be contained under 32-bits on a 32-bit platform.

Assuming 1,000 markers for a given chromosome, this translates to 32,000 bits to capture the haplotypes with 30,000 of those bits being padded with nothing but zeros.

52

E.g.1. a = [] ? 1 : 2, is a standard terniary operator which in most languages would ask, "is array true ? If so return value 1, otherwise return value 2".
E.g.2. a = [] == true ? 1: 2, is the same statement this time directly asking the question of whether the array is true, however Javascript is now forced to treat the array as a primitive in order to perform the comparison, but converts the array into a string in order for the statement to make sense, essentially evaluating to ' "" == true ' which is false.

An 8-bit Typed Array saves a lot of this redundancy and uses at most 8,000 bits to represent the data which is a memory saving multiplier of 4 to 8 times depending on platform. 8-bit arrays allow for each element to have (2^8 =) 256 possible values which very easily covers the number of alleles represented by polymorphic markers.

With the ECMAScript6 specification came class and inheritance models, which already existed under the Object model but now provided bindings for the more familiar syntax that object-oriented developers were used to.

3.4 Javascript and Hardware-Accelerated Graphics

The graphical contexts now supplied by the HTML5 schema provided a canvas for 3D or 2D graphics to be drawn into by either software or hardware.

With the need for graphics on the web coming into fruition, a new initiative was developed to provide Javascript bindings to the OpenGL API [32], aptly named WebGL [33], which provides syntactic-sugar to the low-level graphics API specification OpenGL.

All graphics card vendors cater to both OpenGL and DirectX API standards [34], but it is only the former which has open source implementations (MESA) and generally better cross-platform support.

The canvas element not only allows for graphics via the 2D and 3D WebGL API, but also has accelerated 2D graphics of its own. In somewhat of a misnomer, canvas-accelerated graphics are commonly generalised under the WebGL banner, despite using very different APIs to render graphics. However for the purpose of conciseness we will also adopt this convention, and silently note that there are three contexts that the canvas element supports: 3D and 2D (WebGL API, hardware-accelerated), and another 2D (Canvas API, hardware or software accelerated).

3.4.1 General Runtime Performance

Javascript is incorporated differently into the web-engines of different web-browsers, each with comparable levels of performance and ECMAScript compliance. There are two main competing open source web-engines in active development:

Gecko Incorporated by Firefox and Netscape, using the SpiderMonkey Javascript engine.

WebKit Exists within Safari and Chromium derivatives (e.g. Google Chrome), which uses either the JavascriptCore or V8 engine

It is hard to objectively benchmark the ECMCAScript engines in a graphical application by themselves without taking the web-engine into consideration [35]. They all approximately have the same performance (even on mobile)[36], but differences can arise depending on external contributing factors such as OS and browser, as we will see later in the Results (Section 4.3.4).

3.4.2 WebGL Frameworks

Various intermediate Javascript libraries provide layers of abstraction to separate web developers from the convoluted hardware-specific syntax that usually only game programmers for enterprise-level games have to deal with.

There is a multitude of these fast-growing graphical frameworks, each divided into a 3D or 2D context.

3D frameworks are built upon the Three.js library which provides direct C-like almost 1:1 mappings of various OpenGL calls. The library by itself is primarily used by graphics card aficionados with prior experience in the GPU programming, but is of no practical use to developers diving into the field for the first time.

A more commonly used syntactically pleasant framework that is built upon *Three.js* is *Babylon.js*, which provides basic functionality without the complex calls related to exhibiting finer control over a 3D scene.

At the early stages of development for our applications, a 3D framework was considered with the intention of representing haplotypes via a zoomable interface which would focus individuals under inspection into the foreground, and move the remainder into a background using 3D transitions; but this is would require a "2.5D" library at best, since the same functionality could be performed via a simple layering structure. Using a 3D framework would only increase the system requirements for the application without providing that much advantage in usage.

The application was programmed with transitions and other graphical methods in mind (e.g. creating shapes, text, colours), but these would be implemented via an interface library so that the underlying library could be swappable if required[53]. This is useful if the framework loses functionality in future browsers, and another one can be quickly implemented without needing a fine rewrite of the full code, since all changes required would be contained in a single interface.

The remainder of this section will focus on the 2D frameworks considered to aid in this process.

i Canvas (pure)

This is essentially the raw Canvas without supporting frameworks whatsoever.

At code conception, graphics were drawn straight onto the canvas context itself. Drawing functions all follow the same schema of grabbing a context, initialising a 'path' (a group that contains points), and then manipulating that path via primitive Canvas API calls such as 'rect' (rectangle), 'arc' (arcs and circles), and 'lineTo'/'moveTo' for more complex shapes such as diamonds and zigzags.

[53] e.g. To create a red square, the function "drawSquare(10, red)" from my interface library would suffice, without having to worry about the underlying graphics framework used. If direct Canvas was used, the function would bind to "Canvas.drawRect({width:10, height:10, fill: 'red'})". If another framework was used, it may bind to "Other.drawSquare(10,red)".

```
function drawDiamond([c_x,c_y], color, nodeSize)
{
    ctx.beginPath();

    ctx.moveTo(c_x - nodeSize, c_y);
    ctx.lineTo(c_x, c_y - nodeSize);  ctx.lineTo(c_x + nodeSize, c_y);
    ctx.lineTo(c_x, c_y + nodeSize);  ctx.lineTo(c_x - nodeSize, c_y);

    ctx.fillStyle = color;
    ctx.fill(); ctx.stroke();
    ctx.closePath();
}
```

Listing 3.1: Example Canvas API call for drawing a diamond

It should be noted that a shape is not returned after a function call for later manipulation or property editing, but is drawn directly to the screen and would need to be repeat 'clear()' and 'draw()' calls to implement any changes.

This grew cumbersome over time and the need for more useful features such as layering, grouping, shape management, and efficient screen redraws became more apparent.

ii PIXI

PIXI can render graphics via both the Canvas and WebGL(2D) APIs, but it is extremely tailored to delivering fast graphics and is more suitable in a game development environment, likely paired with a physics engine [37].

Under such a model, the physics engine would pre-compute the locations of all the shapes and objects within a simulation, and the PIXI renderer would then draw the final positions to screen.

It does not do any grouping or layering by default, simply renders graphics under a specific 2D context.

It is mentioned here since this is the first framework that was attempted after direct Canvas rendering in the hopes that it would simplify the drawing process at least. This it did, but the duplicity required in shape management became hard to manage and it was quickly abandoned.

iii jCanvas

This library provides layers, animations, and also makes use of Javascript's asynchronous callback structure through the extension of event listeners, where functions are bound to specific events such as mouse clicks or keyboard presses, or even more complex events such as object collision detection [38].

It provides its own complete shape library for drawing nice primitive shapes such as rectangles, circles, lines, and custom shapes.

```
function drawDiamond([c_x,c_y], color, size){
    Canvas.drawRect({
        fillStyle: color,
        x: c_x, y: c_y,
        width: size,
        height: size,
        rotation: 45
    });
}
```

Listing 3.2: jCanvas function to draw a diagonal by simply rotating a square.

This greatly simplifies in the rendering of shapes, but it should once again be noted that the graphics are drawn directly to screen, so the entire scene must be composed first through external shape management before being drawn to screen in order to reduce the number of redraws.

jCanvas does not use WebGL, so it can only draw using the Canvas API and hardware-acceleration depends upon the support of the platform.

Another drawback is the dependency of this framework of jQuery; a large library used for a manner of other web related tasks such as simplifying common tasks such as transitioning DOM elements, table management, and form input.

jQuery has been in wide circulation since 2007 and has been a polarizing source of contention between web developers: for some, jQuery is a do-everything library that greatly simplifies web development and empowers the developer to make extremely rich-looking interactive web pages; for others, it is an extremely bloated resource-heavy library that fundamentally changes the default Javascript syntax and prevents developers from implementing their own methods.

Efficient resource-usage should not be a concern of a developer in a modern age, but jQuery is a rapidly growing library and lot of utilities it sports are of no relevance in a canvas context. This likely suggests that the jCanvas developer(s) preferred the jQuery syntax over pure Javascript and were willing to take on the risk of any potential slowdowns in order achieve it.

iv KineticJS

KineticJS follows the same easy syntax as jCanvas, but with the addition of a core interaction shape model. Shapes can now be created first, modified, added to layers, and assigned other custom properties before being rendered onto the screen. This greatly simplifies the development process, since changes can be made to the active scene and rendered/updated at the developers will.

```
function drawDiamond([c_x,c_y], color, size){
    return new Kinetic.Rect({
        fillStyle: color,
        x: c_x, y: c_y,
        width: size,
        height: size,
        rotation: 45
    });
}
```

Listing 3.3: KineticJS function to draw a diagonal (a rotated square) and returns an object.

KineticJS makes use of layers and groups. Groups are shapes that move together when one of them is transformed in terms of position, rotation, or scale [39].

Layers are interestingly separate canvas elements altogether: that is, elements in two different layers are animated separately onto separate canvases, and then overlayed upon one another to create the illusion of layering. This may seem needlessly contrived, but it enables the developer to manage their elements efficiently by restricting more repeatedly updated shapes into a more 'active' layer, so that a full scene redraw is not being called for the otherwise more static elements which would exist on another layer.

There is a limit to this speedup, and any performance gains begin to falter if too many layers are used, with a good balanced number being two active layers and one static reserve.

The framework comes with its own transition library, custom event listeners and handlers. It does not rely on any other framework, and can render graphics using both canvas and WebGL (2D) contexts.

In terms of optimization, the library has stopped being actively developed and has remained stable since 2014[54], meaning the API will not change during application development. Documentation is clear, and ample examples are provided, as well as a strong community and fanbase to offer insight and advice.

KineticJS seems to provide the relevant WebGL canvas extensions required to create a web application without over breaching the application requirements and retaining an optimized minimality in its features, and it was for this reason why it was used as the main underlying framework in the thesis.

3.5 Application Development

The application was developed at first using small terminal text editors such as **nano** and **vi**, but as the need for multiple buffers to be open at the same time became apparent, **emacs** was adopted. Emacs is a terminal text editor, but has modules that allow for word autocompletion, compilation, command testing, and many more via the (M)EPLA[55] repository [40].

In later stages of development the graphical **SublimeText** editor was used, providing the standard IDE features such as split views, multiple renaming, function usage, and code syntax highlighting (with support for the ECMAScript6 schema).

The application process was split primarily into two main groups of focus: the user interface, and the theoretical backbone. A great deal of work was performed in

[54] See: https://github.com/ericdrowell/KineticJS.

[55] (Milkypostman's) Emacs Lisp Package Archive, a large collection of script extensions for Emacs https://github.com/melpa/melpa.

user interface section for rendering pedigrees, but these will be discussed later. We will first explore the theoretical core of the application that tackles the two main issues of this thesis: Haploblock Resolution, and X-Linked Inheritance.

3.5.1 Haploblock Resolution

Haploblock resolution is the method of phasing a set of genotypes from related individuals to generate haplotypes, which are then used to determine points of recombination between parent-offspring trios under the heuristic of a genetic model and a likelihood over a locus.

Table 3.2 shows the four main trio cases that cater for the combination of different parental alleles that are presented, as well as the possible child alleles from such variations.

Trio Genotype Cases	*Parental Alleles*	*Child Alleles*		*Possible*		*Allele Desc.*	
		Possible	*Illegal*	*R.*	*A.*	*PS*	*Type*
Homozygous	AA | AA	AA	AB or BB	Y	Y	N	O
	BB | BB	BB	AB or AA	N	Y	N	O
	AA | BB	AB	AA or BB	Y	Y	Y	B
Heterozygous	AB | AB	AA	-	Y	N	Y	S
		BB		Y	N	Y	S
		AB		N	Y	N	O
Homozygous and Heterozygous	AA | AB	AA	BB	Y	Y	Y	B
		AB		N	Y	Y	B
	BB | AB	BB	AA	Y	Y	Y	B
		AB		N	Y	Y	B

Table 3.2: Trio Allele Cases showing applicable cases on separate lines, with possible and impossible child alleles, and parental alleles separated by a vertical pipe. A, R, and P denote whether the possible child alleles are Ambiguous, Recombinant, and Parental-Specificity.

For the sake of readability and clarity, we will define three types of alleles: *sealed* (unambiguous), *bound* (ambiguous, parent specific), and *orphaned* (ambiguous, non-parent specific).

The majority of ambiguous alleles stem from the combination of homozygous

and heterozygous parental alleles, however they are all of the bound type since the attributing parent of an allele is always known even if the specific allele is not (see the PS field).

The most ambiguous group by far are the identical homozygous parental alleles, containing only orphaned alleles without any parental-specificity at all.

Though the heterozygous parental alleles and the non-identical homozygous parental alleles seem to share the same degree of ambiguity, with 2 out of 3 cases being of the sealed type, it is the latter group that has full parental-specificity.

It is interesting to note that of the 10 individual cases presented only 2 are unambiguously phased, but that 5 of the remaining 8 ambiguous cases are parent-specific; illustrating that though the task at hand may seem difficult, it is only a third of the cases that we are completely unsure about.

3.5.2 Approaches

The key to resolving haploblocks lies in making use of the sealed trio cases in order to solve for the ambiguous ones.

Here we will discuss the three main implementations attempted within the application under this premise: Neighbouring States, Iterative Inspection, and Path Finding.

3.5.3 Neighbouring States

The neighbouring states approach follows a very straightforward methodology: upon finding an ambiguous trio of genotypes at a given marker locus, find the nearest set of flanking markers that contain unambiguous trios.

This enables the ambiguous trio to 'orient' itself relative to the flanking unambiguous trios. This is a fairly common technique used in many problems of a similar nature, and Illumina actually employ a variant of this technique in their Top-Bottom encoding for representing the direction of ambiguous SNPs [41].

i Theory

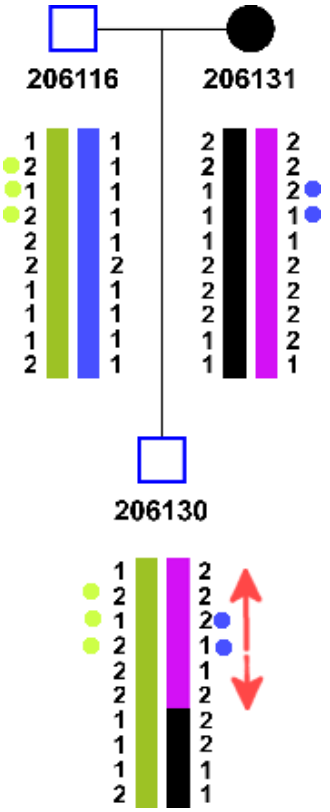

Figure 3.2: A simple parent-offspring trio, with coloured blocks representing founder allele groups.

As highlighted in Figure 3.2, the approach operates by crawling up and down a genotype of interest to find flanking unambiguous neighbouring alleles. If the neighbouring haplogroups are the same, they are absorbed into the group. If they differ, an informed choice on the point of recombination must be made.

Genotypes are reduced into two simple cases; one where flanking sealed genotypes are of the same founder allele group, and one where the flanking alleles differ as shown in Figure 3.3. In the former case, the uncertain genotype is encompassed totally by the group, but the latter case requires more thought.

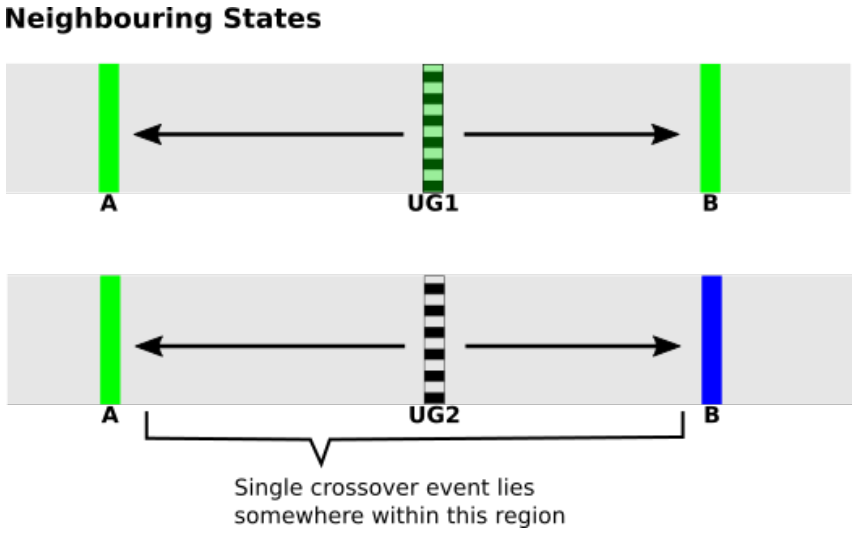

Figure 3.3: Two possible neighbouring state scenarios, where uncertain genotypes (UG1 and UG2) are flanked by sealed genotypes with: (Top) the same founder allele group, (Bottom) different founder allele groups.

When flanking sealed genotypes differ, there are limited resolution strategies that can be taken to investigate where a point of recombination has occurred:

1. **Offspring Validation** Checking within the genotypes of descendants for sealed genotypes at the same locus.
2. **Random Assignment** Picking one of two blocks.
3. **Iterative Block Estimation** Estimating the size of founder haploblocks and determining which encompassing block is more valid.

   Each strategy comes with their own advantages and drawbacks, but these are discussed in more detail in the conclusion.

ii Implementation

The pseudocode code below demonstrates the underlying method used to iterate through a list of markers, extract the genotype trios for a given marker, and then crawl back and forward from that marker location to find the nearest unambiguously phased markers.

```
FOR m IN RANGE [0, max]:

    marker         :=  markerList[m]
    genotypesTrio  :=  getGenotypeTrios(marker)

    IF isAmbiguous( genotypesTrio ):

        UNTIL flankingBackMarker EXISTS:

            FOR b IN RANGE [0, m-1]:
                backmarker         := markerList[b]
                backgenotypesTrio  := getGenotypeTrios(backmarker)

                IF NOT isAmbiguous( backgenotypesTrio ):
                    flankingBackMarker := backmarker
                    BREAK

        UNTIL flankingForwMarker EXISTS:

            FOR b IN RANGE [m+1, max]:
                forwmarker         := markerList[b]
                forwgenotypesTrio  := getGenotypeTrios(forwmarker)

                IF NOT isAmbiguous( forwgenotypesTrio ):
                    flankingForwMarker := forwmarker;
                    BREAK;

        flankingBackGenotypes := getGenotypeTrios( flankingBackMarker )
        flankingForwGenotypes := getGenotypeTrios( flankingForwMarker )

        resolvePhase( genotypesTrio,
            flankingBackGenotypes,
            flankingForwGenotypes )
    ELSE:
        resolvePhase( genotypesTrio )
```

Listing 3.4: Pseudocode for Neighbouring States approach.

There is some redundancy in this, for contiguous ambiguous regions will likely crawl over the same positions multiple times to reach the same unambiguously phased flanking markers, giving an average time complexity of $n * log(n)$.

Actual implementation first mapped out unambiguous markers in an initial pass, and then a second pass bisected the map to find the nearest flanking markers, reducing the time complexity to n.

One more variant of this method that provided 2x speedup in the process was to use the newly resolved genotype trio states within the ongoing analysis. This means that the unambiguous back marker in subsequent iterations will always be the directly adjacent previous marker because it would have been resolved in the previous iteration; essentially meaning that only a forward search is required.

iii Discussion of Approach

The 2x speedup variant method raises the concern that the search may be greedily optimistic; with uncertain genotypes towards the end of analysis being too dependent upon the uncertain genotypes analysed just prior. This could effectively "box" the later uncertain genotype trios into more restricted groups, since the former iterations aggressively remove potential groups from the back-selection; creating the tendency of resisting crossover events in favour of maximising the current haploblock. Though not necessarily wrong[56], it may create a noticeable disparity in the distribution of haploblock sizes in later generations.

The real core of the neighbouring states approach lies within the *resolvePhase()* function referenced (but not defined) in the pseudocode. The reason for this is due to the the numerous ways that the function was implemented, and ultimately abandoned.

The first approach of Offspring Validation seemed to have a sound basis, where uncertain genotype trios that could not be resolved were then prompted to measure the genotype trios of any direct offspring at the same locus. In theory, if one of the offspring trios were (optimistically) sealed, then this was suggestive of a recombination event. In practice however, if the genotype trio in question was uncertain, then the offspring trios would also tend to be uncertain, forcing the method to move onto the next marker locus in hopes of finding a sealed genotype trio. Often entire uncertain contiguous blocks of genotypes were analysed this way without a sealed genotype ever being found.

The second approach of Random Assignment also quickly ran into complications, where two directly adjacent uncertain genotypes within the same A-B locus were (randomly) assigned to different groups creating an $A-AABBABAB-B$ populated region that suggested an unreasonable amount of crossovers have taken place.

[56] A solution will always be reached since a crossover event will be prompted eventually (at maximum) just before the last flanking sealed genotype.

A considered solution to this problem would be to expand the region uncertain genotypes into one general "uncertain block" and then to assign the entire block to either A or B, creating either an $A-AAAAAAAA-B$ or $A-BBBBBBBB-B$ assignment. This is, of course, a terrible idea, since the point of recombination can occur anywhere within the A-B locus and large block assignments will likely create problems with offspring dependent upon the individual under inspection.

The main problem with the neighbouring states approaches evaluated so far are that they are subject to the local maximization[57] pitfalls. In order to find a more global solution, more attention would need to be given to the global size and ranges of the haploblocks being analysed.

The not yet mentioned third strategy (Iterative Block Estimation) works on this principle, but requires a section of its own to be properly assessed.

3.5.4 Iterative Block Estimation

This method is an extension of the Neighbouring States approach, but with more global limitations in place to focus the analysis in a more structured way.

Ambiguous stretches of alleles that fall within sealed loci sharing the same phase are said to exist within the same haploblock, on the condition that the genetic map distance between them is small enough to cast doubt on a crossover event having occurred.

Conversely, ambiguous stretches that fall within sealed loci which have different phases are indicative of at least one recombination having occurred.

In this case, the genetic distance between the sealed loci by itself is of no assistance, since a crossover event could have occurred at any point in prior meioses. However we can estimate a limit based upon the number of meioses expected for a

[57] A common problem in "hill-climbing" algorithms, where the heuristic to find the "maximum peak" of some search involves climbing upwards and determining the discovery of a peak based upon reaching a plateau. Of course, there could many hills which would require many varied uphill walks with different starting positions to actually find the *global* maximum peak, and not just the *local* one.

given founder allele. For example, a founder allele originally spanning 1000 markers, would expect to span 250 markers after two meiosis[58]. By keeping track of the size of a founder haploblock from sealed genotypes, we can make a more informed decision upon whether an uncertain genotype exists within that haploblock or not.

i Theory and Implementation

An orphaned genotype of an unknown founder allele group U is flanked between a sealed genotype of founder allele group A upstream, and another sealed genotype of founder allele group B downstream. Within the region encompassing the A-B locus, it can then be said one of two events occurred:

Event1 Group A underwent a crossover to group B, and U is either of type of A or B.

Event2 Group A underwent a crossover to group X followed by any number of successive crossovers (to group Y, group Z, etc.) before finally crossing over to group B, where U is of any applicable founder allele group type.

In order to determine which of these two events occurred, we first initially map out the positions of all sealed genotypes in order to gauge the approximate known sizes of the available founder haploblocks. These can then be compared to the expected sizes of the founder haploblocks for that individual in the pedigree when evaluating the orphaned genotype in the A-B locus.

[58] Assuming a non-consanguineous pedigree. When inbreeding occurs, there is always the undismissable case that two haploblocks of the same founder group recombining in an offspring from two different paths of descent may be directly adjacent or in overlapping proximity to one another, making the founder block appear longer than it should.

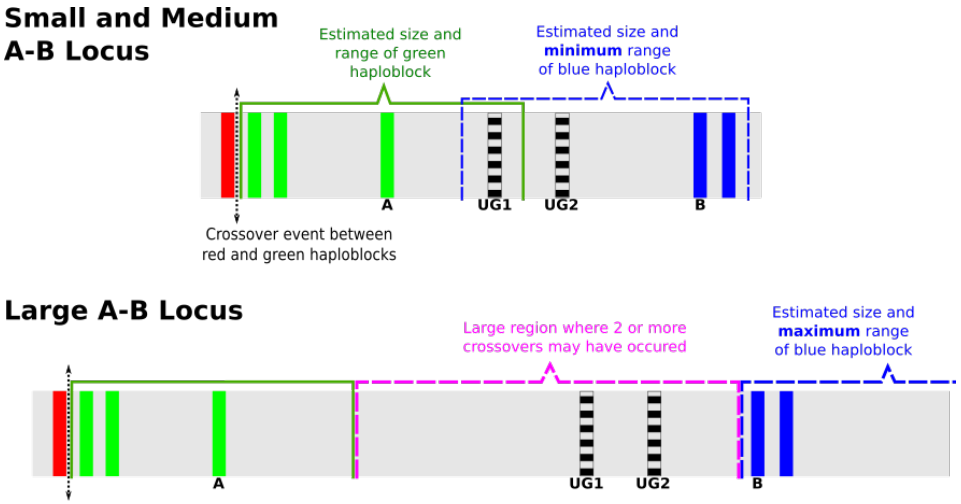

Figure 3.4: Small (top), Medium (top), and Large (bottom) A-B locus showing the types of inspection that can be performed. A and B are sealed genotypes; UG1 and UG2 are uncertain genotypes. Dashed lines indicate regions where the size of a block is known/estimatable but the start position is not. In the small A-B case, UG1 falls within the range of the green haploblock with a known starting position as observed by a prior crossover event. UG1 and UG2 also fall under estimated range of the blue haploblock, though UG1 is less likely because the green haploblock is more definitive.

The rest of the method can be broken down into specific cases of uncertain genotypes within an A-B locus of specific size ranges:

Small If the A-B locus spans a region smaller than the smallest expected founder haploblock, then Event 1 has occurred, and further analysis can be performed to determine whether U is of type A or B.

Medium If the A-B locus is larger than the smallest expected founder haploblock, but smaller than the average expected founder haploblock size; then we must probe the region by alternately assuming U=A and then U=B, and testing if the size of haploblock in question exceeds its expected size, where an Event 1 has occurred upon the success of one of these probes.

Large If the A-B locus is larger than the average expected founder haploblock size, then we must investigate whether two (or more) crossovers have occurred in the A-B locus, and that an entire haploblock (or more) exists within said locus. If so, an Event 2 type has taken place.

The small case is straight-forward to evaluate, since U is encompassed within at most two valid blocks and the resolution is non-problematic. Figure 3.4 provides an example of this (top) in the case of UG1 which falls within the two block ranges, but is better defined in the green block and is more likely to be assigned to it.

The medium case is more involved, where an Event1 is confirmed upon the success of one of the assumptions of U belonging to either A or B. However upon failure of both assumptions, the genotype is flagged for later reattempt after all other uncertain genotypes have been converted into sealed types. A reattempt would ensure a more complete set of sealed genotypes to work with which would have more defined sizes of founder haploblocks, and aid in the resolution process.

The large case is seemingly complex, but large cases (Event 2 types) are only evaluated after all Event1 processing has taken place, allowing the large cases to be evaluated with a more defined model of haploblock ranges and sizes.

Under this more complete model of haploblock sizes, we can then probe U against each of the available founder allele groups and test whether the model still holds. For example, if the founder allele group S is identified in a region much further downstream than U, where the region spans a locus of sealed genotypes that is consistent with its expected size, then it is unlikely that a whole S haploblock would also exist within the A-B locus[59].

In the unlikely event that after probing U with the available founder groups, a clear result still cannot be established, then a compatible haploblock group is picked at random (an operation dubbed "Event 3"), since it is impossible to distinguish between any one group.

[59] See footnote 58.

ii Discussion of Approach

The handling of Event3 types may have serious ramifications in later descendants if the wrong haploblock is picked, wherein a complete recursive re-run of that particular individual (and descendants of) would need to be performed with a different random haploblock being assigned. If incompatible haploblocks are still detected in later descendants, then this process is repeated ad infinitum until there are no more blocks to choose from.

The iterative nature of this process can take quite some time to process, since though the method itself runs in approximately $O(n^2)$ time, it may repeat numerously for every Event3 encountered. Computational slowdowns not withstanding, the model cannot handle consanguineous pedigrees effectively, since much of the resolution involved in the Event2 process depends upon the assumption of non-repeating haploblocks, which may exist if there are multiple paths of descent a haploblock can take to reach an individual.

The coding and maintenance of this method also became extremely lengthy, as separate classes had to be written to handle each event. Solutions to transparent biological processes should be natural and simple, and this method was quickly becoming unmanageable in its attempt to separately cater for all event types. Further, Event2 and Event3 types had a tendency to continuously recurse down through the pedigree often repeating previous analyses to favour the current individual in question. By the time root founder couples were analysed a complete consistent model was found, but run times exceeded practical limits with massive latency (>10s) between reading in the genotypes and presenting them.

Despite the great deal of time and effort that went into the iterative inspection method, it was ultimately abandoned in pursuit of a more elegant approach.

3.5.5 Path Finding

Path finding algorithms are search algorithms that find an optimal (usually shortest) path between two points though a complex of inter-connected nodes in a graph via weighted edges.

A path-finding search is typically conducted by starting a given start node and exploring all adjacent nodes under some graph-traversal heuristic until the end node is reached. Multiple paths may be discovered leading from the start node to the end node, but each path is graded based upon the total weight of the edges that constitute it.

i Graph Traversal

Graph-traversal techniques are split into two types of (usually recursive) methods: *Depth-first*, and *Breadth-first* searches.

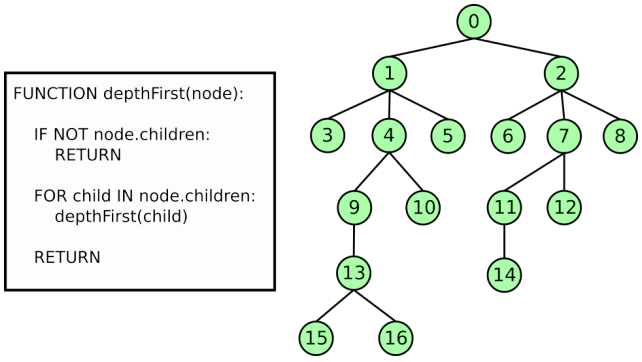

Figure 3.5: Recursive Depth-First function; Expands into each child node until there are no more children to expand into.

Depth-first searches run under the simplistic notion that each node encountered is immediately expanded upon (until there are no more nodes to expand), and only later exploring adjacent nodes at the same point in the path. It is implemented in a trivial recursive fashion outlined in Figure 3.5, where the order of nodes traversed would be: $0, 1, 3, 4, 9, 13, 15, 16, 10, 5, 2, 6, 7, 11, 14, 12, 8$.

Breadth-first searches on the other hand iterate through all nodes available before expanding into another node, where the order of nodes would follow a more standard $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14$.

If the aim of the search is to find a specific node, then the advantages each search strategy can be seen with respect to nodes 8 and 15; where node 15 is found after only six iterations in the depth-first approach (15th in the bread-first search), and node 8 is found after fifteen iterations in the depth-first approach (8th in the bread-first).

Depth-first searches are *optimistic searches* and are prone to being misled; possibly expanding down the wrong path in order to reach a target node that may lie only just adjacent to the root. Breadth-first searches are *conservative searches* and treat all nodes equally such that they assume all paths have an equal chance of leading to the target node.

It is clear that though a breadth-first search has the disadvantage of under-estimating where the target node will, it never over-assumes that the target node is just within reach as in the case with a depth-first search.

Thankfully, there are algorithms that combine the advantages of both methods such that only nodes that are determined to be more likely to yield the target node are expanded upon without the cost of over-estimating, a concept known as an *admissable* heuristic. In order to do this, the edges of the graph need to be *weighted*, meaning that the choice of traversing from one node to another needs to incur a cost in order to actually compare paths.

Such algorithms are known as *Best-First* searches since they expand only the most promising node chosen under a particular heuristic. Though they are seen more as a sub variant of breadth-first searches, they do incorporate depth-first principles.

The application of these algorithms to haploblock resolution may not yet seem apparent because we must first understand the methodology of one particular type of best-first search.

ii Dijkstra's Algorithm

All path-finding algorithms are derived from the Edgar Dijkstra's work, with his landmark paper 1959 paper [42], which outlined a best-first method of finding the shortest path between two points.

Since the nodes in his graph are connected via weighted edges, Dijkstra's algorithm also qualifies as a uniform-cost search algorithm, since it takes into account the (long-term) cost of going down one edge over another[60].

The algorithm works under the heuristic of minimizing the cost of unvisited neighbouring nodes. Its method is outlined as follows:

1. The node now occupied is the current node, n.

2. If n is the target node, terminate the search.

3. Consider all adjacent nodes and the cost of traversing to them. For each adjacent node, α:

   (a) Calculate the cost w, equal to the cost of traversing the weighted edge to it.

   (b) If α already has a cost assigned to it from a previous cost evaluation, and that cost happens to be lower than w, do nothing. Otherwise assign w to α.

4. Remove n from the search and move to the adjacent node with the lowest assigned cost.

5. Go to[61] step 1.

The optimal path is always discovered because much like a breadth-first upon which it is founded, all nodes in turn are considered. For n nodes, the number of unvisited neighbouring nodes considered is $n - 1$, giving it a time complexity of $O(n^2)$.

[60] Though technically the algorithm could operate on any graph with equally weighted edges.

[61] Dijkstra would be spinning in his grave at this careless use of the *GOTO* statement that he so famously campaigned against in favour of the now well precedented *FOR* loop [43].

iii Application to Haploblock Resolution

The search heuristic it employs is of most interest to us, because it works under the principle of trying to minimize a future cost.

If we treat each node as a particular haploblock group and assign zero edge weights between groups of the same type, and non-zero edge weights between groups of different types; we can find an optimal path through our network of nodes that tries to minimize the number of meioses.

This is illustrated in Figure 3.6 as a multi-layered network graph. In layered networks, nodes within the same layer are not connected; only connections between layers are permitted. Each layer represents a single individual-marker locus, and all nodes within a given layer denote the possible founder allele groups for that individual-marker.

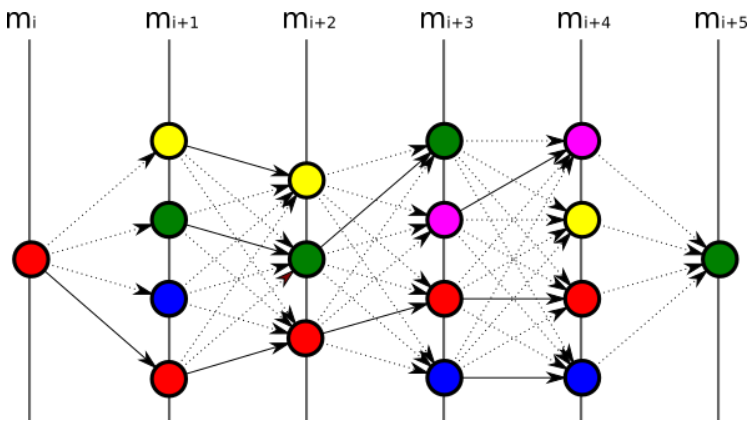

Figure 3.6: Founder allele groups represented by separate colours. Transitions between groups of the same type (bold) incur no edge cost; non-zero weights (dashed) are less favoured.

Nodes that are connected to the nodes of the same group incur no edge cost, whereas nodes that are joined to those of different groups (indicating a meiosis) have non-zero costs associated with them. By traversing through the network under the heuristic of finding the path with the smallest cost, we can minimize the number of meioses and trace an optimal path.

The nature of this task scales with number of markers m, and the number of possible founder allele groups f, effectively placing it on the order polynomial-time, and this can be very resource intensive for practical usage.

The problem with Dijkstra's algorithm is that it explores all routes. This is not necessary, as we will see in the next section.

3.6 A* Best-First

The A* ("A-star") search algorithm is the most widely used path-finding algorithm due to its efficiency and accuracy. Being an extension of Dijkstra's algorithm, it is just as admissible, yet it only considers a small subset of the total nodes in the graph. This is because A* performs optimistic estimates on the cost of an active path without over-breaching the true cost of that path [44].

The algorithm works on the principle of expanding a "frontier" of paths under a simple heuristic of:

$$f(n) = g(n) + h(n) \tag{3.1}$$

where for the last node n on the current path, *g(n)* is the cumulative cost of the path so far, and *h(n)* is the heuristic that estimates the cost of the smallest path to the target node.

All the search algorithms visited so far can be implemented by the A* search algorithm; Dijkstra's being the special case where $h(n) = 0$ (i.e. no estimation is made), as with all Breadth-first searches; Depth-first being $h(n) = X$, where X is a large number greater than the total cost of the search where upon inspection of each node, it is decremented by 1 such that nodes are evaluated in the order they are discovered.

To understand what is meant by the frontier of the search path, consider the state-space grid shown in Figure 3.7. This can be viewed as a network graph where

each square represents a node, and each adjacent square/node is connected to it via an edge with a uniform edge weight cost.

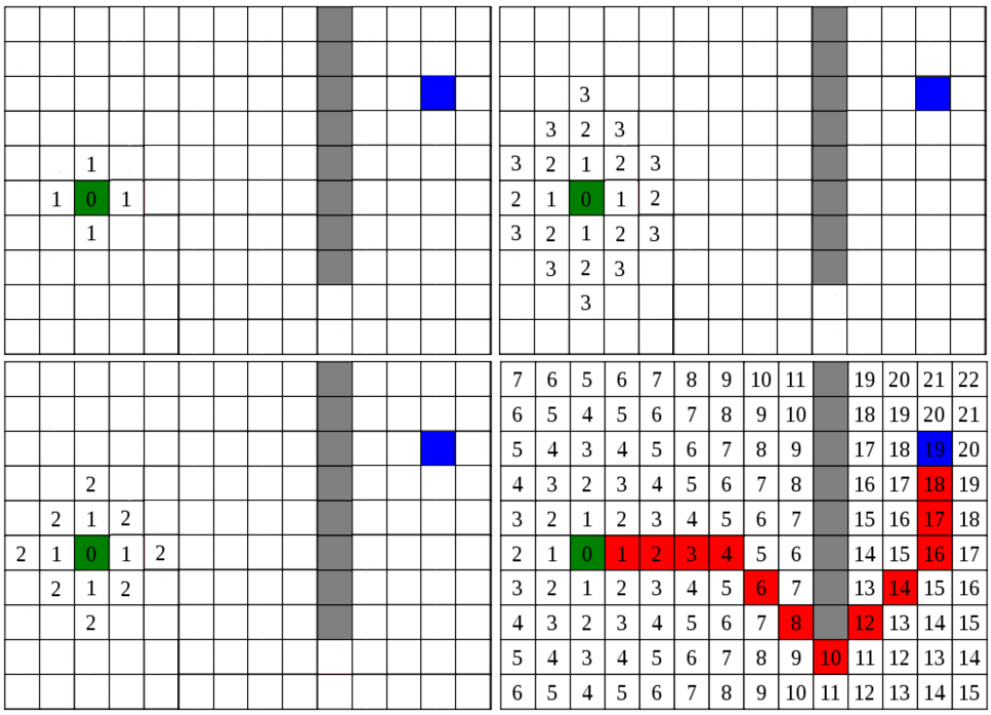

Figure 3.7: State-space diagram showing the optimal path as discovered by an A* best-first search: 1st iteration (Top-Left), 2nd iteration (Bottom-Left), 3rd iteration (Top-Right), 22nd iteration (Bottom-Right).

The green square is the start node, and the blue square is the target end node. All available edges have equal cost, and upon start there four active paths under consideration by the algorithm (up, down, left, right) with a value of 1 . Upon second iteration of the algorithm, the 4 paths can expand, each in 3 more unvisited directions yielding $(4 \times 3 =)$ 12 paths, though four of these actually overlap with same cost; so a total of 8 active paths are under consideration for the next iteration. At the third iteration, we have $(8 \times 3 =)$ 24 paths, though twelve of these overlap with already visited paths and one of them has hit the end of the grid and has nowhere to expand to leaving 11 active paths.

It does not take much imagination to see that the algorithm scales linearly ($4k$, for k iterations), with the overlaps and wall intersections keeping the number of active paths relatively low. For example, by the 19th iteration there should be 76 active paths being considered, but wall constraints have limited this to just 4.

To bring this back to more a genomic context, for f founders in a pedigree, there are a maximum of $2fm$ active paths by the mth marker. For a simple pedigree with 4 founders and 500 markers, a maximum of 4000 active paths would be under consideration.

Of course, haplotype resolution has wall constraints of its own; sealed and bound genotypes collectively reducing the amount of available founder allele groups at specific markers, which work to remove whole path varieties altogether.

Nonetheless, for i individuals in a pedigree with $2i$ chromosomes[62], the delay in processing would be:

$$2i \times p \times d = 2ipd \tag{3.2}$$

where:
p number of active paths
d time takes to process a chromosome

If there are over 20 active paths being explored and each chromosome took 0.05 seconds to precess, then this would result in over 2 seconds of processing per individual, and over 20 seconds of delay for a pedigree with 10 members in it.

Active Queue

Thankfully one of the A* search algorithms greatest strengths is its search heuristic $h(n)$ which only considers a sub-selection of active paths it believes will lead to the target node.

[62]There are of course 23 pairs of chromosomes in an individual, but analyses are chromosome specific so only the current pair is considered.

Active paths are sorted by priority of lowest cumulative score $g(n)$, and the leading P paths are added to a queue of predetermined length P. All paths that are not added into the queue before the next iteration are dropped from the analysis altogether.

This ensures that that there are at maximum P active paths under consideration at any given iteration, and keeps the analysis bounded within reasonable time constraints, for reasonable values of P.

3.6.1 Implementation

Implementing the A* search algorithm within haplotype reconstruction involves two passes of the data; the first pass for populating a chromosome map with possible founder allele groups, and the second pass for performing the search itself.

First Pass: Priming the Data

When pedigree data is first parsed, genotypes for each chromosome are structured into an Allele[63] object which contains four types of data:

data_array A signed 8-bit array (*Int8Array*) to compactly store the genotypes, and is initialised to the actual genotypes described by the input file.

pter_array A temporary standard Array (*Javascript Object*) which carries pointers to founder allele "colour" groups, and is initialised as empty.

unique_groups Another temporary standard Array for storing the unique founder allele groups encountered across the entire chromosome, also initialised as empty.

haplogroup_array A signed 8-bit array (*Int8Array*) to store the resolved founder allele group designations in a compact format, initialised as a null pointer.

The usage of these constructs is carried out in the *haploblock_backend.js* file, and performs the first pass procedure as follows:

[63] See Allele class in pedigree.js for more information.

1. **Founder Initialization** All founder chromosomes are assigned unique colour groups[64] via the function *initFounderAlleles()*. The entire stretch of the pointer array for a founder Allele class is populated with the same colour.

   E.g. pter_array = [[5] , [5], [5], ... , [5]] (for the 5th founder allele being processed.)

2. **Non-Founder to Founder Group Assignment** The function *assignHgroups()* performs a top-down sweep over the pedigree and processes each parent-offspring trio by descending down until there is no more offspring to process. For each trio encountered, a length check is performed on their data to assert that the number of genotypes in the analysis is the same. Then the function *child2parent_link(child, mother, father, family)* performs the brunt of operation, where at each marker locus:

   (a) **Empty Data Assignment** Maternal and paternal genotypes with no data (i.e. have a marker allele index designation of 0) are assigned to a zero colour group (-1).

   (b) **Autosomal Inheritance Modelling** For maternal alleles *m1 m2*, paternal alleles *p1 p2*, and child alleles *c1 c2*; check for genotype equality such that *{c1,c2}* exists within the four valid genotype designations *{ {m1,p1}, {m1,p2}, {m2, p1}, {m2,p2} }*. If a match is found (or multiple) then the parental allele colour groups are pushed onto the child allele colour groups in a mutually-exclusive fashion such that *c1 c2* would only share the same colour groups by chance.

3. **Parental Exclusion** At this stage some clean up is required to deal with the overly ambiguous regions such that the next generation deals with a clean inheritance set of colour groups. The problem is that each child allele could inherit from one of the four parental chromosomes, but needs to do so in an exclusive fashion such that a maternal haploblock does not also appear in the

[64] Unique colours that will be used to represent these groups are also assigned, evenly distributed across the colour spectrum as designated under a HSV colour space (see Colour Space section in Appendix on page 231 for more information).

childs paternal haploblock (i.e. no two blocks can have the same colour across sister alleles).

To counter this, we populate exclusion groups where all the unique colour groups from both maternal alleles across the entire chromosome comprise a maternal exclusion array, and likewise for paternal alleles into a paternal exclusion array. This arrays assist in the A* search later on.

4. **Consanguineous Check** If the maternal/paternal relationship has been labelled consanguineous, then we must slacken the parental exclusion somewhat since having duplicate haploblock groups in both child alleles is now not impossible. To do so, we first find the the intersection of the maternal and paternal unique colour groups, and then remove the colors identified from both groups. At this stage, the pter_array is now fully primed for the A* search algorithm to process.

Second Pass: Path-finding Algorithm

The algorithm is contained within the function *a_star_bestfirst()* which takes a pointer array as argument, as well as an (optional) exclusion list.

First, a function to define whether an item exists within the exclusion list is determined based upon whether the list itself is:

(a) **Undefined / Non-existent** where upon the function simply returns false.

(b) **A single index/number** where a simple equality check is performed.

(c) **A list** where the exclusion list is mapped and items are checked against the map.

The global parameter *MAX_ROUTES* defines the number of active routes at any point within the iteration, and optional global parameters such as allowing single-marker stretches.

With these parameters set, the actual algorithm runs under the pseudo-code representation in Listing 3.5.

```
WHILE algorithm_running AND routes_to_explore:

    // Sort routes in descending order of significance,
    // extract max number of routes
    routes_to_explore = sortAndGetTopRoutes( routes_to_explore , MAX_ROUTES)

    current_route = routes_to_explore[0];        // Examine first route
    route_length = getLength( current_route )
    route_colors = getColors( current_route )

    // Map to store lengths of paths traced by each color
    ordered_routes = {}

    FOR current_color IN route_colors:

        IF inExcludeList( current_color ):
            SKIP

         // Perform lookahead for current group
        stretch_iterator = route_length + 1

        WHILE stretch IS LESS THAN END:

            color_groups_at_marker = getColorGroupsAt( stretch_iterator  )

            // stop lookahead if new color encountered
            // that is not a zero group
            IF NOT current_color EXISTS IN color_groups_at_marker:
                IF NOT color_groups_at_marker EQUAL zero_color_group:
                    BREAK

            stretch_iterator  ++

        // After lookahead for current route and current color is finished,
        //  store color with key as the length of current lookahead stretch
        lookahead_stretch = stretch_iterator  - route_length
        ordered_routes[ lookahead_stretch ] = current_color

    // Examine color keys in descending stretch order
    FOR color_stretch IN descendingKeyOrder( ordered_routes ):

        IF color_stretch LESS THAN OR EQUAL TO 1:
            //Not significant change, dead end route.
            SKIP

        color_for_stretch = ordered_routes[color_stretch];

        // Create a new route, based on the current route with
        // the lookahead color added the same amount of times
        // as the color stretched.
        new_route = clone( current_route )
                + addcolor( color_for_stretch, color_stretch)

        IF length(new_route) EQUALS END          // Found a complete route
            complete_routes.add( new_route )

        ELSE:    // Otherwise, add new route to active search
            routes_to_explore.add( new_route )

    // Remove current route, now expanded by new routes
    routes_to_explore.remove( current_route )
```

Listing 3.5: A* Best-First Haplotype Pseudocode Representation

Essentially the number of active paths expands with the number of available colour groups at the frontier of each path, with every path split into several sub-paths each designated an available colour group. Each of these sub-paths attempts to maximize their colour group as much as possible to create new paths that will be used in the next iteration.

To maximise a colour group, a lookahead is performed that steps forward from the current path's frontier and asserts the colour group exists at each step, whereupon failing the length of the lookahead (the "stretch") is stored to be used as a metric to grade the colour group that performed the stretch over the other colour groups for the same path.

New paths are then formed from the current path, each padded with separate stretches of the colour group that performed the lookahead. These new paths are added to the active search (assuming they satisfy a minimum stretch limitation), and the current path is removed because it is no longer the frontier of the search.

The exclusion list is used during the lookahead process, where any colour groups encountered from the frontier of the search that appear in the exclusion list are not expanded upon. This serves the purpose of separating maternal and paternal blocks from each other in order to prevent illegally inherited blocks from propagating down the pedigree.

The A* best-first search is run under two mutually-exclusive assumptions:

1. Either the chromosome is **maternally-inherited**, therefore the paternal exclusion list should be used. The sister chromosome should perform the opposite (paternally-inherited, maternal exclusion).

2. Or the chromosome is **paternally-inherited**, therefore the maternal exclusion list should be used. The sister chromosome should perform the opposite (maternally-inherited, paternal exclusion).

It may seem that two whole searches are being performed, but it should be noted that a search on a chromosome performed under the wrong assumption will terminate very early due to the colour group inconsistencies being encountered early on.

Only one route can be chosen as the best from the complete routes, and this is determined by sorting the complete routes by ascending number of sets and picking the first route, i.e. the route that has the fewest unique colour groups and number of meioses.

3.6.2 Optimization

Several optimizations are undertaken to reduce the overall complexity and excess system resource usage of the algorithm; and these can be broken down into modes attributed to stages of processing that occur prior to, during, and after the search algorithm has run its course.

i Prior Processing: Step Contraction

The first speedup is introduced during the first pass of procedure, where the overall number of nodes that the search algorithm has to evaluate during the lookahead section is reduced.

Previously, in order for the algorithm to perform a lookahead it had to step through each marker in order to determine whether the colour group under consideration existed at that locus. By adapting the *child2parent_link* function to store a sliding window of the colour groups it processes, a very rudimentary lookahead can be performed within the same pass used to prime the colour groups.

This effectively means that prior information upon the colour groups can be gained and utilized by the A* search, such that nodes are tailored so that the algorithm can pass over known stretches without having to crawl through them first.

Figure 3.8 illustrates this quite effectively; where the information contained within the set has not been transformed under the Lookahead arrangement, only that the more optimal paths are better defined. For the other less optimal paths, nothing has been changed: R-Y-G-B-R-G remains the same under both representations.

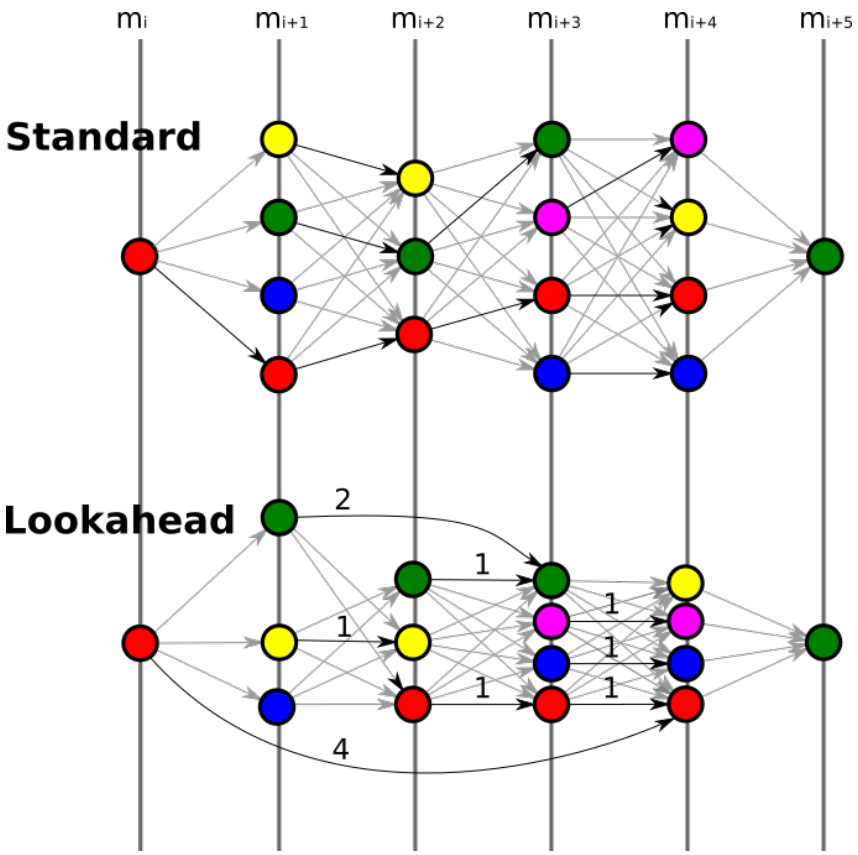

Figure 3.8: Step Contraction, assisting in the A* search algorithm by reducing the number of steps needed to be evaluated, e.g. $R - R - R - R - R - G$ under the Standard priming technique (Top) can be reduced to $R - R - G$ under the Lookahead priming technique.

The gains in this technique may seem small, and indeed the total number of nodes and edges between the Standard and Lookahead representations in Figure 3.8 differs by only 1. However, A* is a best-first algorithm, meaning that only the leading paths in each expanding iteration of the frontier are considered. It is these leading paths that benefit the most from this step contraction, since they are the ones ultimately will have to step through the most.

For a sliding-window of size s stepping through a haploblock of estimated size $\frac{h}{m}$ for m meoises and h initial founder length, this produces a linear reduction on the order of $\frac{h}{sm}$.

ii Runtime Processing

Some back-processing occurs during actual runtime with the intention of improving upon the compactness of the data.

Debug and Release

The A* algorithm is actually first defined in the function *a_star_bestfirst__DEBUG*, which logs to console every step of the analysis for later inspection. This is useful for debugging purposes, but is impractical during standard processing because the JavaScript runtime environment must queue these output messages to be printed in a timely manner, expending system resources.

The flexibility in the JavaScript Object model means that attributes can be bound to a given object during runtime and do not need to be explicitly defined before[65]. Since functions and objects in JavaScript are implemented the same way, this means that functions can be defined at runtime too.

A new optimized "release" function *a_star_bestfirst* is defined that takes the content of the debug function and truncates all logging references in order to boost performance. This also means that two varieties of the function only exist when required (i.e. during processing), and that the optimized function does not need to be manually re-written every time the original function undergoes a revision.

Homology Path Bunching

This extra feature is less a performance optimization and more an analysis optimization. As an analysis progresses through a chromosome, the top N paths begin to "bunch" together in such a way that the MAX_ROUTES limit could compromise the admissibility of the search, if the similarity between active paths becomes too homologous as $N \rightarrow MAX\_ROUTES$.

[65] Several other languages incorporate similar functionality (C++ via virtual functions) but with more rigid parameters.

To prevent this bunching behaviour and encourage a more diverse selection of paths, the number of active paths N only contribute towards the maximum route limit if they do not share the same length score at given iteration. This essentially changes *MAX_ROUTES* to *MAX_TIERED_ROUTES* such that unique sets of active routes are considered.

For example, a *MAX_ROUTES* limit of 4 would be able to give active route lengths of {32, 21, 21, 21, 15 ,12,12} instead of {32, 21, 21, 21}. Paths that have identical scores are usually indicative of a long stretch where (in the case of 21, three) colour groups maintain their colours through an ambiguous region that caters for them all.

By allowing a greater variety of path sizes, earlier branching points in the analysis are not pruned off too eagerly and provide a more diverse set overall.

iii Post Processing: Pointer Cleanup

After the processing is complete and the result of the best-first search is returned unambiguously for both parental alleles, a cleanup operation is performed upon the Allele class containing the data. The pointer array contained within the class was a non-compact representation of all the possible colour groups that could propagate to a given individual through accepted models of inheritance, and was operated upon via the A* search.

Once the search is fully complete for all members of the pedigree, the pointer arrays have served their purpose and are no longer required. The function *removePointers* in the file *pointer_cleanup.js* sweeps down through pedigree and initialises the *haplogroup_array* with a compact signed Int8Array that stores the best path for each individual-chromosome. The pointer array is then requested for a delete by the JavaScript garbage collector.

The memory saved from this cleanup should not be understated: For a typical 1000 marker chromosome stretch with a maximum of 8 possible founder allele groups

(from 4 founders) where each allele group is a (64-bit) reference to a group, this produces a total of ($1000 \times 8 \times 64 = 512000$ bits =) 64 kilobytes per chromosome. By compacting the colour group data into an 8-bit array, the total is shrunk down to ($8 \times 8 \times 1000 = 64000$bits =) 8 kilobytes per chromosome.

It is arguable that the only savings being made are at the kilobyte level and even the most modern platforms by default sport at least 1GB of memory, however the data is now structured in contiguous parts of memory instead of dotted around via pointer arrays and the JavaScript runtime environment can now free up memory and run under a smaller more optimized region of the RAM.

3.7 X-linked Inheritance

As shown in the Autosomal Inheritance implementation on page 123, parental alleles are represented as *{p1 p2},{m1 m2}* pointers which each represent a possible array or founder allele "colour" groups. The valid child allele *{c1 c2}* configurations are shown in Table 3.3 below.

Model	*Parental Alleles*		*Allowed Child Combinations*	*Total*
	Maternal {p1 p2}	*Paternal {m1 m2}*	*{c1 c2}*	
Autosomal	{a1 a2}	{a3 a4}	[a1 a3],[a1 a4],[a2 a3],[a2 a4]	8
X-linked (Female)	{x1 x2}	{x3 y1}	[x1 x3],[x2 x3]	4
X-linked (Male)	{x1 x2}	{x3 y1}	[x1 y1],[x2 y1]	4

Table 3.3: Three separate methods of inheritance. Parental alleles given in curly brackets denote ordered alleles. Child alleles are mutually exclusive of different parental types, and are given in square brackets to denote possible combinations (e.g. [a b] = {a b},{b a}).

It can be seen that the pedigree being dominant/recessive has no effect in the way that the alleles are inherited, the only deciding factor being X-linkage paired with gender-specific cases. Even if the pedigree is consanguineous, the model still holds because any founder blocks that find their way into both alleles of an individual, did

so because their colour groups were contained in the separate maternal and paternal alleles through different paths of descent.

Under the pointer model, X-linked inheritance is not a very involved process since it acts merely as a more constrained version of the autosomal model under different gender designations. The only unknown exists in the X-linked male model, where knowledge of which allele indicates the Y chromosome is required prior to evaluation.

Initially it was a hard-coded requirement in the input file that the second male allele was the Y-chromosome, but this restraint was later removed and resolved in the software via the function *detectYalleles* in *filehandler.js*.

At this current time of writing, Y-chromosome capable genotyping exists but is not effectively utilized in most linkage software, with **Allegro**, **GeneHunter**, and **Simwalk** merely defaulting it to an entire chromosome array of zeroes. The function simply detects these zero alleles in the males when parsing the pedigree, and stores the Y-chromosome as the second allele in the Pedigree class.

In the event that Y-chromosome capable genotyping becomes more widely used, the *detectYalleles* function will need to perform further probing. This will not preclude the rest of the analysis, since the Y-allele will still be assigned to the second allele in males.

3.7.1 Comparison with HaploPainter

One of the core reasons for HaploHTML5's conception was the lack of X-linked inheritance within the more commonly used tool HaploPainter, where haploblocks would be rendered improperly. This was at first thought to be due to the linkage software that performed the haplotype reconstruction assigning the wrong allele indexes to untyped individuals who were reconstituted in the program, but closer inspection revealed that allele indexes were in fact correctly assumed and that the problem lay in the haploblock resolution software.

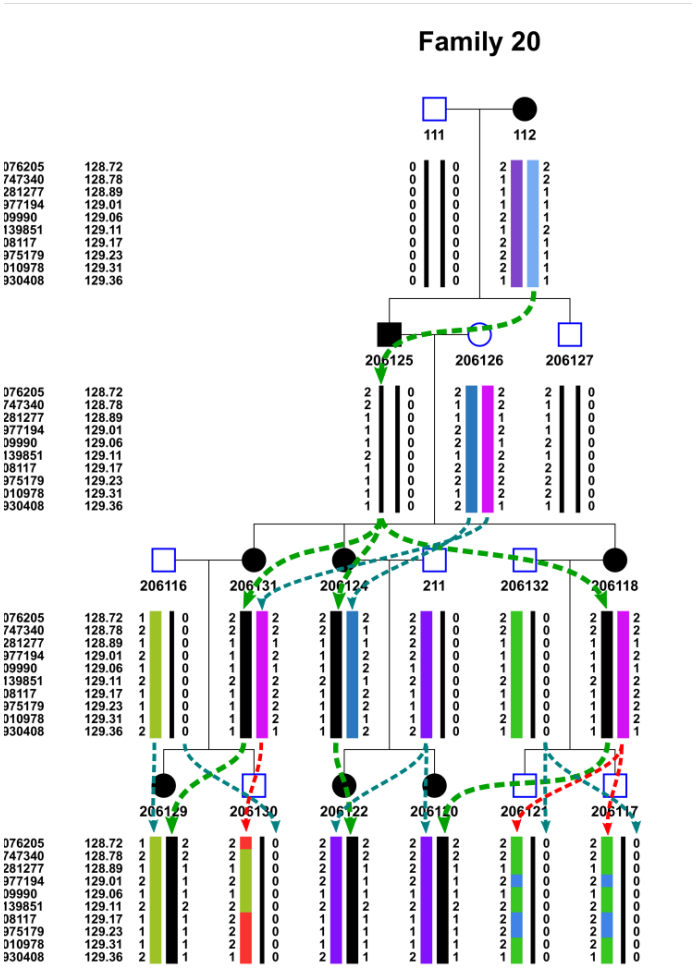

Figure 3.9: Minimum working example of an X-linked Dominant pedigree. Inconsistent colouring within alleles depict HaploPainter's haploblock resolution failing in the case of X-linked pedigrees. Overlayed arrows depict true inheritance paths; black disease allele (green-dashed), correctly inherited alleles (blue-dashed), incorrectly inherited alleles (red-dashed).

Figure 3.9 shows a typical X-linked dominant pedigree as rendered by HaploPainter. There are several issues highlighted in this example:

1. **Non-zero Alleles Assigned Zero-Allele Blocks** This is shown in the left allele of individuals 206125 and 206127, which are represented as narrow blocks normally reserved for the zero-alleles.

2. **New Haploblock Group Assigned to Non-Founder** 206130 shows the appearance of a red block that is neither inherited from parents, grand-parents, or great-grandparents at the current locus. This is a misalignment because an inspection of the allele numbering shows that the block should be inherited from the maternal pink allele (as shown by the red-dashed arrow).

3. **Impossible Recombinations** 206121 and 206117 exhibit multiple recombinations over extremely small genetic distances. For reference, a distance of 1 cM would expect 0.01 recombinations. The genetic distances here are on the order of 0.01 cM, which makes multiple recombinations very unlikely. The red-dashed arrows once again show the true path of inheritance, where both individuals inherit their mother's pink allele as confirmed by the allele numbering.

4. **Non Parental-Offspring Inheritance** The appearance of the blue haploblock in 206121 and 206117 is troubling, since only the mother (206118) could inherit the allele from the founder (206126) and yet doesn't. Even more concerning is the apparent mixing of non-sister alleles in the offspring. Recombinations take place within the parent first before being transmitted, which creates clear maternal and paternal alleles in the offspring. 206121 and 206117 appear to show the green (paternal) allele recombining with the blue (maternally inherited) allele which should be impossible under any inheritance model.

It is likely that the method that HaploPainter uses to assign allele groups does not distinguish between bound and uncertain genotypes, very liberally pushing all possible founder allele groups onto both child alleles without any mutual-exclusivity, effectively resulting in a "mixed bag" of possible colours. Though this gives the algorithm more leeway to find any solution instead of terminating with errors, it does (in this instance) produce incorrect results.

3.8 Pedigree Rendering

A pedigree is populated by reading in a file in the standard MAKEPED linkage format and storing the data in a containing structure (Person class in this case). However this specifies information about each individual in the pedigree and not their relationships to one another.

The function *connectAllIndividuals* (*pedigree.js*) iterates over all stored members and creates connections between each: parent is added to child, and child to parent, with parents being added as mates to each other. Each family is stored in its own separate map so that different families are segregated into their own pedigrees.

The process of actually drawing the pedigree is a complex process because as mentioned on page 115 traversing through a graph involves expanding nodes and all nodes that are connected to them.

Consider now a graph where there is no longer a single root node, but instead several, as in the case of a pedigree with many founders. If node expansion was performed upon each of these individuals, we could descend through the graph by perform a breadth-first search upon all offspring.

A problem will quickly arise however where individuals who exist in the descent path of multiple founders will be evaluated more than once and lead to duplicity in the data if not carefully handled.

The peeling method that Elston and Stewart devised (as mentioned in Background, page 35) was considered as one solution; determining pivot individuals and treating them as special edge cases to be separately handled when drawing the pedigree. However, previous experiences in handling edge-cases (see page 114) had not been straightforward and so a more general solution was derived instead.

The resulting function *populateGrids_and_UniqueObjs* (*init_graph.js*) required iterating through all individuals and populating an array that stored each generation, as well as populating a map which stored unique nodes and vertices associated with pedigree members and their relationships as discussed below.

3.8.1 Populating and Structuring the Pedigree

The grid generation is contained within the class *addFamMap* (*init_ graph.js*), which asserts that each pedigree has a single randomly chosen individual to act as the root node. The sub-function *addNodeArray(person, level)* descends through all relationships associated with that individual (mates, children, mother, father) recursively until there are no more relationships to expand upon.

i Unique Graphics Map

Upon each recursion, a shape representing that individual's gender and affectation is inserted into the graphics map with the individual's identifier being the accessing key. If trio relationships are also detected, then the trio is added as the combination of two unique lines:

Mateline A straight line connecting two individuals who share offspring. Their unique key is paternal ID and the maternal ID in that order.

e.g. 11 (father) and 12 (mother) → "m:11-12"

Childline A right-angled line that connects an offspring to the centre of the Mateline of that offspring's parents. Siblings share the same centre (or 'anchor') but different childlines. Their unique key is the Mateline ID and the offspring's ID in that order.

e.g 14 (offspring of 11 and 12) → "c:m:11-12-14"

Under this format, even if individuals are evaluated twice the map will not contain duplicate data because the order of parental gender and offspring is preserved so that the same relationship is only recorded once.

ii Generational Grid Array

Within the same recursion step that the graphics map is populated, the level parameter records the ascension/descent of the relationship being expanded. With the parameter initially set to 0 for the root individual, parental relationships are expanded with *level - 1* parameter, and offspring with a *level + 1* parameter. This ensures is that not only are all siblings at the same "level", but first-cousins too.

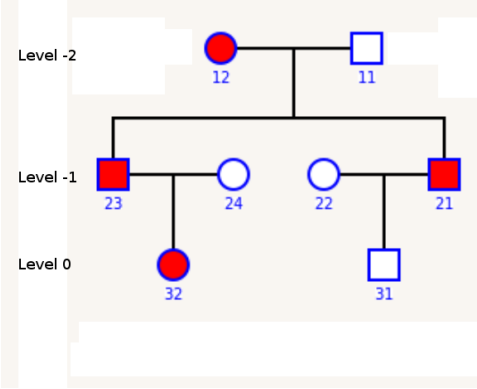

Figure 3.10: Example of pedigree levels and recursion

Consider Figure 3.10 which shows three levels for a group of individuals as determined by recursive function. If we pick 32 as the root individual with a level of 0, then we can trace a route first-cousin 31, via $32 \rightarrow 23 \rightarrow 11 \rightarrow 21 \rightarrow 31$:

1. **Start at Root: 32,** $level = 0$
2. **Expand to Parent** $(level - 1) : 23, level = -1$
3. **Expand to Parent** $(level - 1) : 11, level = -2$
4. **Expand to Offspring** $(level + 1) : 21, level = -1$
5. **Expand to Offspring** $(level + 1) : 31, level = 0$

Thus we have determined that 32 and 31 share the same level and can be drawn at the Y-position in the pedigree as shown in the figure. By expanding upon all individuals, we can very easily deduce the individuals who should constitute a given generation and represent our pedigree accordingly.

iii The Separation of Graphics and Data

The information generated and stored at this stage follows a strict Model-View structure, with the graphical data to be drawn on screen being held entirely within the global graphics map *unique_graph_objs* which is accessed solely through the static library class *uniqueGraphOps* via getters and setters for adding/removing nodes and edges.

This is divorced completely from the the pedigree data that pertains to individuals' information storage (name, gender, parents, offspring, mates) which is held entirely within the global data mappings *family_map* which is likewise accessed solely through its own static library class *familyMapOps* via its own getters and setters for adding/removing individuals.

It cannot be stressed enough that the Model (familyMapOps) and the View (uniqueGraphOps) do not overlap, i.e. one does not have pointers that reference entries in the other or vice versa.

This ensures clean code management and simplifies tasks drawing tasks since the graphics should ultimately be a disposable representation of the pedigree data. The deletion of an individual's graphical node should not also prompt the deletion of that individual's pedigree data.

3.8.2 Determining Pedigree and Penetrance Model

The right penetrance model is key to running the correct analysis upon the pedigree, and the function *determinePedigreeType* (*pedigree.js*) was written to determine whether the pedigree was X-linked/Autosomal, and Dominant/Recessive.

Though Table 3 showed that the analysis does not depend on dominance/recessiveness, it is still useful in the context that it can be used for other programs that may require it without having to prompt the user.

The function iterates through every member of the pedigree and operates as follows:

1. **Detecting X-linked**

   (a) Calls upon the *detectYalleles* function repeatedly to count the number of males with either a single allele, or with an allele consisting entirely of zeroes. This lends some flexibility in the input format, where a single allele need only be specified for a male in a pedigree and the program will automatically detect that the pedigree is X-linked, and will generate a second Y-allele for that male (consisting of zeroes).

   (b) The function asserts that the pedigree is X-linked if the all males in the pedigree are equal to either: the number of males with a single allele, or the number of males with an all zero allele, or any combination of the two.

2. **Detecting Dominant**

   Operates on a much simpler notion of descending through each generation and counting the number of affecteds. If the number of affecteds in each generation is non-zero, and the number of affected generations is the same as the number of generations; then the pedigree is dominant.

Note: Autosomal and Recessiveness are determined by inverting the results of these two methods.

i Detecting Consanguinity

Couples who share either parents, grand-parents, or other immediate ancestors are said to be consanguineous and their mateline indicates this via a double line. Detection of this is actually also implemented by a (much smaller) A* best-first search to find the highest common founder between two mates.

The function *hasConsanguinuity(person1, person2)* (*init_graph.js*) is outlined concisely in the pseudo-code given in Listing 3.6.

Essentially the search begins with the two individuals *p1* and *p2* who are mates. The search runs under the assumption that their ancestors are not related, and so recursively expands up their parents and their parents-parents, etc. until there are no more parents to evaluate.

This populates a global list of founders as derived from separate "routes" starting at *p1* and *p2*. If *p1* and *p2* do not share any of the same founders then they are unrelated. However if they do share founders then there will be duplicates in the global list of founders, because shared founders will be discovered more than once from the two separate starting points.

```
FUNCTION hasConsanguinuity( p1, p2 ):

    complete_routes = []
    routes_to_check = [ p1, p2]

    WHILE routes_to_check NOT EMPTY:
        // get first person from active routes array and remove them
        person = removeFirst( routes_to_check )

        // No parents, terminate active route
        IF NOT ( hasMother( person ) AND hasFather( person ) ):
            complete_routes.add( person )
            SKIP

        // Otherwise add parents to search
        routes_to_check.add( getMother( person ) )
        routes_to_check.add( getFather( person ) )

    // Check for duplicates in complete routes
    RETURN duplicatesExist( complete_routes )
```

Listing 3.6: Consanguinity test via path-finding approach.

3.8.3 Application Methods

The following methods are more focused upon the general appearance and graphics handling of the application.

i Line Management and Node Position Handling

The graphics map contains pointers to all the shapes and lines to be drawn on screen. As the user manipulates these node's positions the relationship lines associated with them must also have their positions updated.

As previously stated, Matelines are bound to the nodes of two individuals whose relationship they represent, and Childlines are bound to the anchor points of Matelines and the node of the depicted offspring. KineticJS creates shape nodes easily enough (squares for males, circles for females, and diamonds for unknowns), and shapes are coloured to represent affection (red for affected, white for unaffected, and grey for unknown).

KineticJS also has a Line class that can represent a straight line between two points, or a path through multiple points under a variety of different styles. However, it does not have a Connector class that can automatically update a line connected between two objects, such that the movement of one object automatically updates the position of the line.

As a result, line management must be performed manually via graphics map. Essentially upon the detection of a single node being moved[66]:

1. All matelines associated with that node must have their positions updated.
2. All childlines associated with each mateline moved in the previous step must also have their positions updated.
3. If the individual that the node represents is not a founder, then their own childline must be updated.
4. Draw updated positions onto screen.

Given that these steps are performed repeatedly in real time until the node is no longer being moved, it may be seen as a costly process. However, it is not recursive (i.e. the positions of grand-parents and grand-children do not need to be taken into consideration), so at most only two generations of childlines are being updated.

[66] Thankfully, nodes can only be moved one at a time thanks to there being only a single mouse pointer. Multi-touch mobile devices may present problems in the future however.

ii Grid Snapping and Family Grouping

In order to reduce some of the draw operations when moving nodes in real-time, a snapping grid was implemented so that nodes cannot be placed just anywhere but must fall in line with some hidden horizontal and vertical guideline intersections; essentially "a grid".

Dragging a node between the points in this grid will not prompt a redraw, where redraws will only occur when the node is locked into one of the points on the grid.

The grid itself is not global to the screen, but is family-specific, where the graphics of each pedigree is contained within their own separate family-group. These groups (and their containing nodes and edges) can be moved freely without any snapping taking place. It is only the movement of nodes within these groups that are subject to grid snapping, and the grid is relative to placement of overall the family-group.

iii Sibling Level Balancing and Mate Swapping

Another advantage of the grid snapping feature is that it facilitates the user placement of first-cousins and siblings at the same level as shown in Figure 3.10 (page 137). To extend this functionality, sibling level balancing was introduced to overcome the tedious task of having to individually move all siblings to the same level.

Siblings positioned at different levels does not makes sense in a generational context and looks unsightly, and it became clear that it would be beneficial to move all siblings automatically. The dragging of a single node prompts the vertical movement of all nodes sharing the same mateline anchors.

In a similar fashion, it also looks unnatural to have a mother and father at two different levels and so parents are also locked vertically such that the movement of one also updates the other. Parents also have the added feature of being dragged as a single unit if they are brought together. This may seem as if it would prevent mates from moving past each other in a certain direction, but automatic mate-swapping occurs to prevent this, allowing the free ordering.

iv Pedigree Creation (PedCreate View)

Pedigrees can be created from scratch without having to load a pedigree file. When creation mode is enabled individuals can be added to families, joined together as mates (with available choices highlighted in red circles), and joined as offspring (with available anchor points appearing within matelines) as depicted in the figure below.

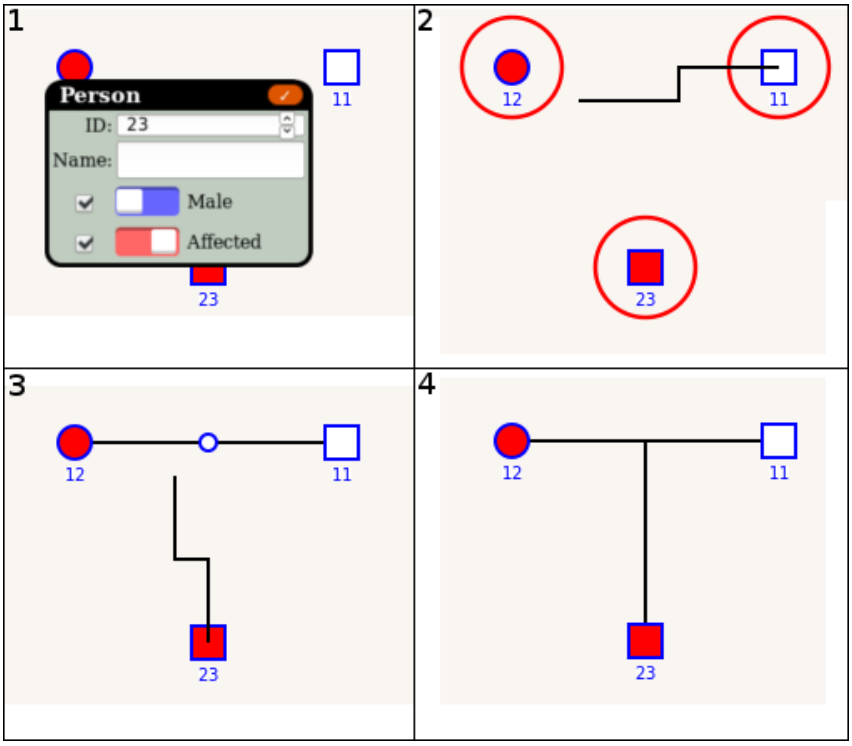

Figure 3.11: The four stages of creating a pedigree: (1) Adding individuals, (2) Joining mates, (3) Joining Offspring, (4) Completing a trio.

Once complete, the pedigree can exported as standard MAKEPED file, as well as optional non-standard tags which contain metadata such as individual names (i.e. not just IDs), and position upon rendering which are interpreted by the application.

v Selection Mode (Selection View)

Prior to HaploHTML5, analysing the haplotypes of individuals required repeatedly scrolling up and down the screen because individuals at different generations would be represented vertically as shown in Figure 3.12.

This is not a problem for small loci, and is even beneficial when trying to see the path of descent. However, examining larger loci becomes difficult when trying to compare allele indexes because the user needs to mentally recount the position of a given marker at each generation.

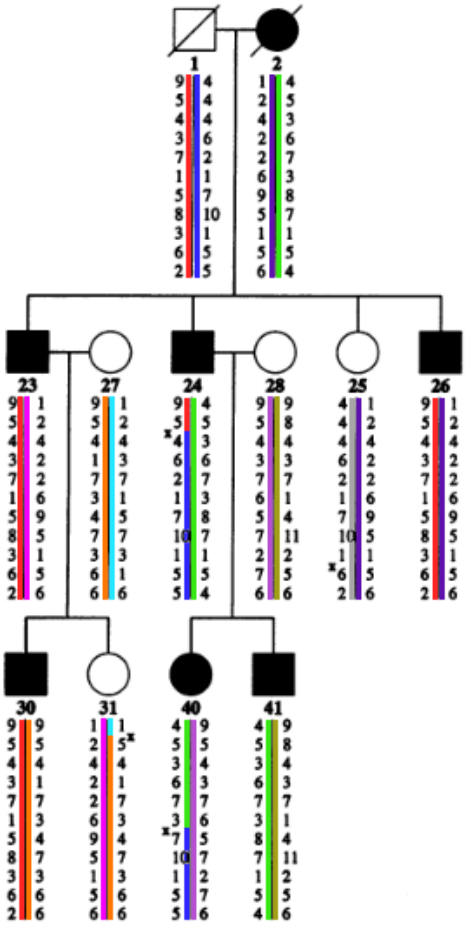

Figure 3.12: Example haplotype analysis spanning three generations and 11 markers.

HaploHTML5 does away with this notion by allowing the user to freely select whichever individuals they want to analyse, whether from the same family or not, and then transitions the individuals onto a side-by-side level view so that haplotypes at a given locus can be be compared across all generations simultaneously as shown in Figure 3.13.

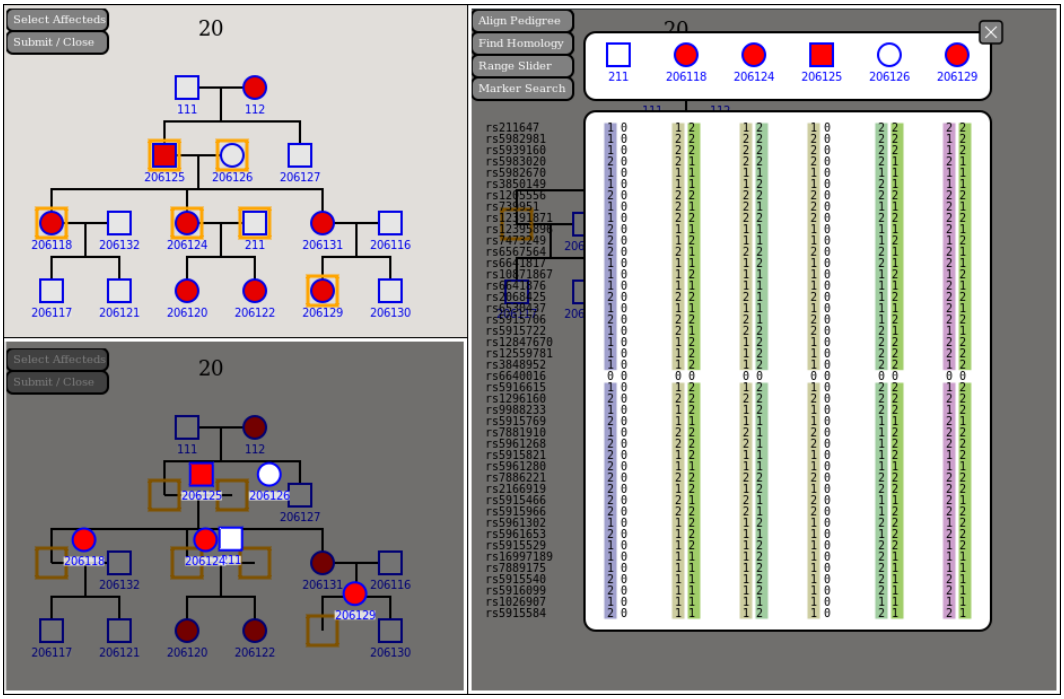

Figure 3.13: Selection mode in three stages; (Top Left) User selection, (Bottom Left) Transitioning to side-by-side view, (Right) Comparison mode

vi Degrees of Separation (Selection View)

When individuals are selected within the Selection View and transferred into the Haploblock View, generational information is generally hidden. In order to give some context to individuals selected in relation to one another, a Degree-Of-Separation (DOS) function was implemented to show by how many generations one related individual was to another.

The function *findDOSinSelection* in file *dos.js* works by grouping siblings under the same node, where only non-founders can be within the same group in order to allow founders to extend their own DOS to other non-founders. Founders are their own sib groups, and for each sibgroup defined, relations are recursed upwards through mates and parents in order to find selected individuals at grouped by level.

The method works well for simple relations, but grows computationally complex for more involved selections which can lead to bad line rendering. Figure 3.14 represents this below, with a clear DOS for the four selected individuals in the left pedigree selection, and inconsistent or missing line renders in the right pedigree selection.

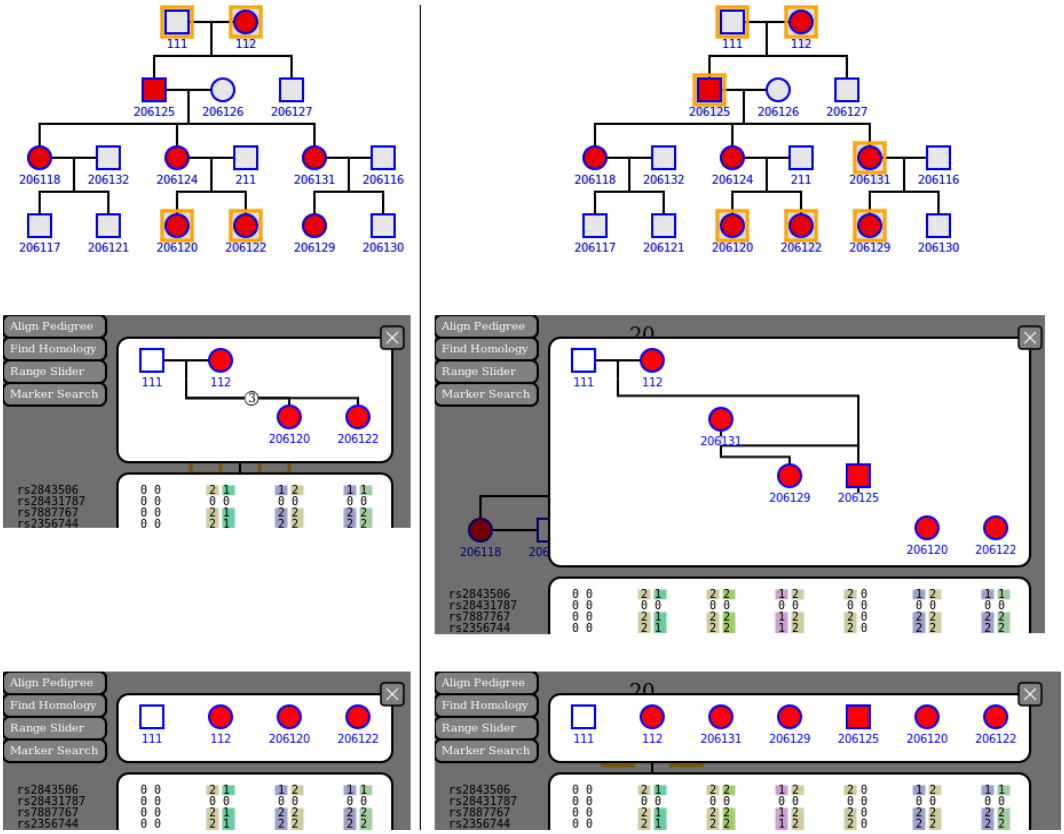

Figure 3.14: Degrees of Seperation (DOS) for two subselections of the same family. (Left) Simple and correct representation of three degrees of seperation between founders and great grandchilden. (Right) Complex and erroneous representation of larger selection spanning more generations. The top image of each column shows the subselection of the pedigree, the middle image shows the DOS view, and the bottom shows the aligned view.

Due to the incompleteness of the method, it was dropped from the release version of the application, and the Haploblock View aligns the pedigrees by default.

vii Haplotype Navigation (Haploblock View)

Haplotypes are initially displayed in a viewport restricted to a particular locus of interest for a default stretch of 30 markers. The user can then move through haplotypes using this viewport by one of:

1. **Scrolling the Mousewheel** Haplotypes are moved up/down by a single marker for each mouse wheel increment.

2. **Dragging and Dropping** Haplotypes can be grabbed directly by the mouse and the view can be shifted up/down by multiple marker stretches.

3. **Range Slider** A slider can also be used to quickly navigate up and down the chromosome. This is by far the quickest method, and has the added bonus that the viewport size can be adjusted by changing the handles of the slider, as well as a quick overview for the scale of the chromosome being represented by the sliders boundaries (see Figure 3.15). Points of recombination are best viewed by this method since quick scrolling makes coloured block changes immediate.

4. **Marker Search** This makes use of HTML5's *datalist* element which populates a list of available markers and assists the user by allowing them to either choose from the entire list of markers via a dropdown menu, or an autocompleted subselection as they begin to type out a marker identifier. This method is the most effective when the locus of region is precisely known (e.g. due to linkage) and the user wants to quickly navigate to a region of interest without having to scroll for it.

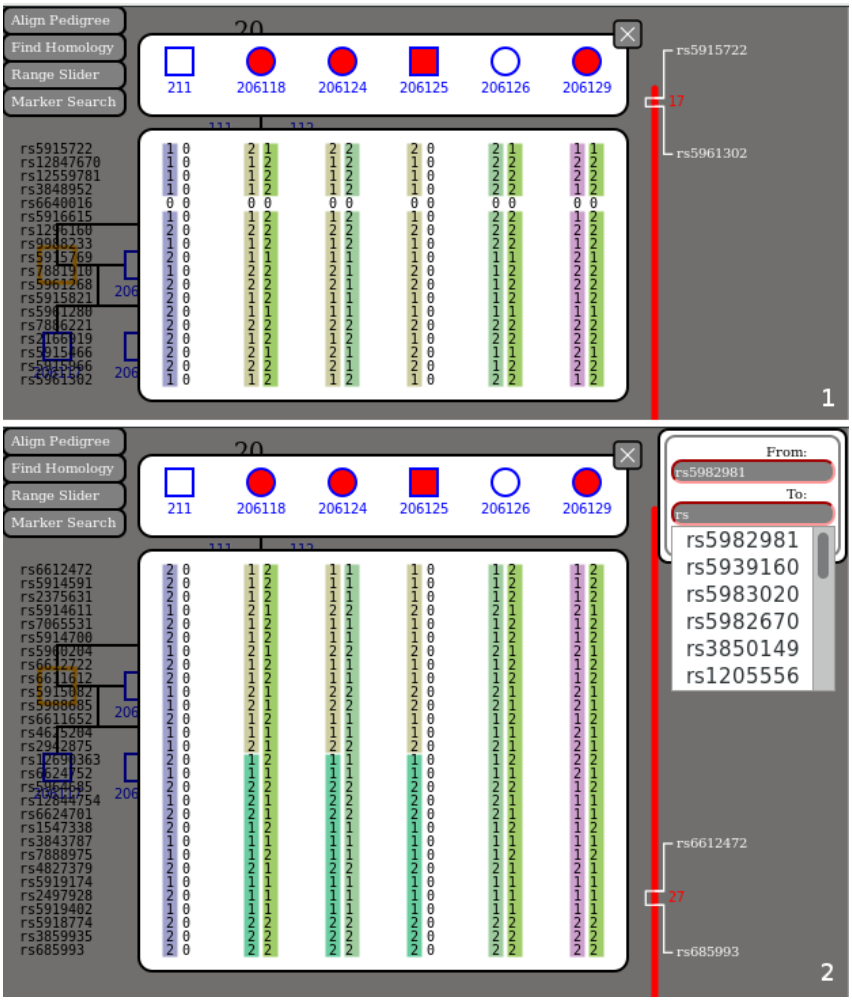

Figure 3.15: (1) Range slider and (2) Marker search with region expansion via slider handles.

viii Analysis Resumeability

The application makes use of the *localStorage* JavaScript implementation to cache the last used pedigree or haplotypes file, so that the user does not need to manually reselect it every time they want to rerun an analysis. The browser simply pulls the file from local storage, and meta flags in the file return to whichever view that the user was last on (PedCreate, Selection, or Haploblock).

3.8.4 Homology Tools

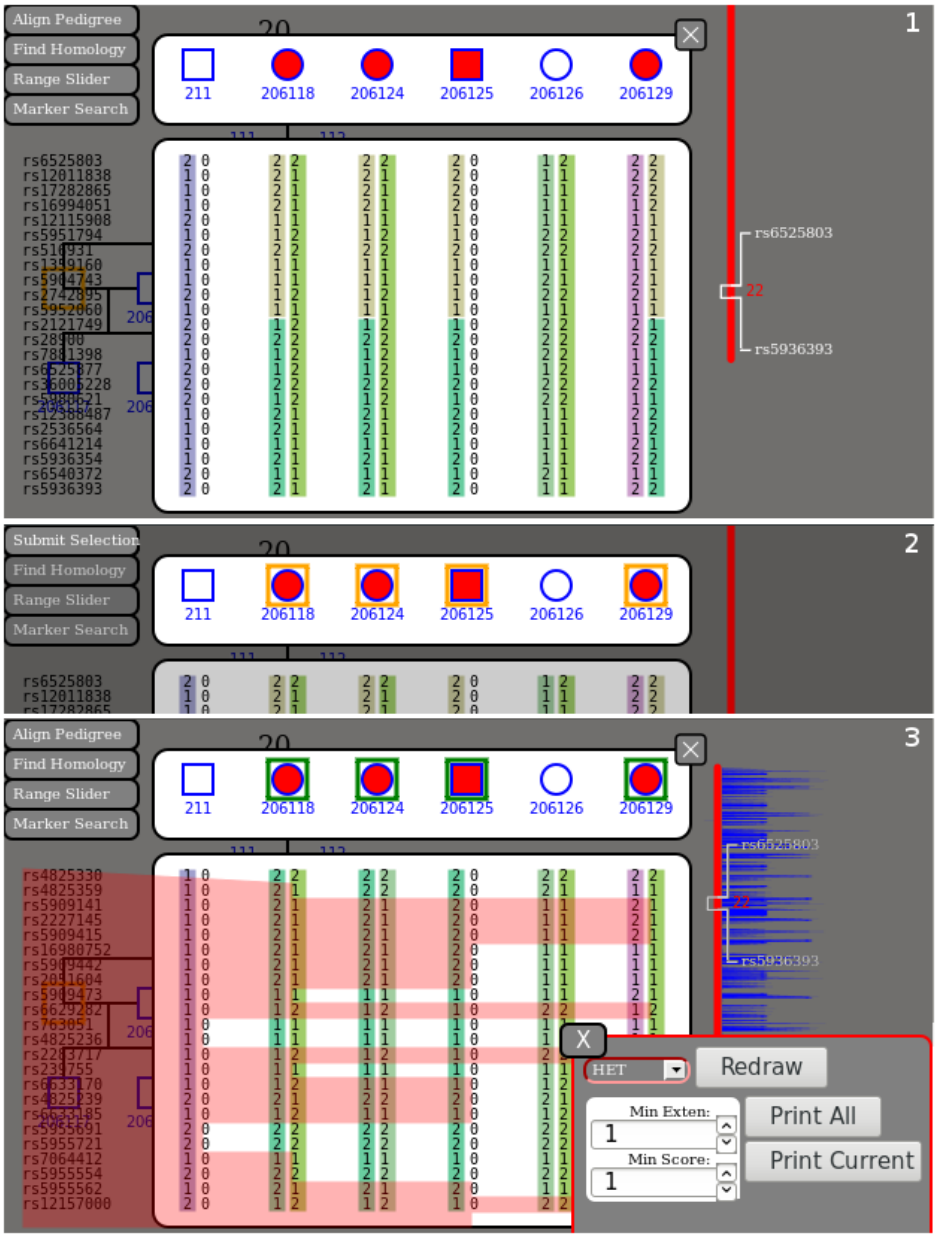

Figure 3.16: Homology Mode activation; (1) Haploview mode, (2) Find homology button pressed where user can select which individuals should be compared for homology (in this case the affecteds), (3) Homology plotted across marker range (blue) and overlayed over haplotypes (red).

One of the main reasons that haploblocks are analysed post-linkage is to find regions of homology pertaining to the linkage region(s). Where the Haploblock View allows the user to make side-by-side comparisons of cases and controls over multiple families, the Homology View further enables a sub-selection of those individuals to be checked for homology, as shown in Figure 3.16 above.

Upon sub-selection submission homology, individual alleles are sorted into groups of affecteds and unaffecteds; the idea being that affected alleles that match to some model contribute to a score and unaffected alleles that match to the same model subtract from the score. Thus for a scenario where all alleles (affecteds and unaffecteds) have the same homology, a total score of zero will be produced.

Scores are calculated for three scenarios:

1. **Homozygous** Both alleles must be identical and must match all other affected alleles to contribute to the score. If only 1 allele of an individual matches, the score will not increase.

2. **Heterozygous** One allele must match at least one allele in all other affected individual alleles to contribute to the score.

3. **Compound Heterozygous** Both alleles must differ and must match all other affected alleles to contribute to the score.

All three scenarios are determined in the same pass, but cannot be represented simultaneously as an overlay, so it is up to the user to select which mode they wish to view, and a redraw is required upon mode change. The user can also filter the data so that scores below a minimum score threshold are not shown, as well as a minimum extension limit so that only regions of homology spanning at minimum a certain amount are shown. This is especially useful when trying to constrain a wide linkage peak to a more narrow region of interest. Recommended threshold values are Min Extension = 2, and Min Score > 3.

3.9 Code Minification and Obfuscation

Before the licensing of the application had been decided, attempts were made to scramble the source code such that it could be incorporated into more restricted models without privacy being a concern.

Browsers do not run closed-source binaries, and deploying a JavaScript application as such through a toolchain that would compile it into a binary would not make sense over the internet, nor would it run in-browser.

The only other option was to somehow encrypt the code itself in a manner that it could decrypted at runtime and run as a normal web application. This process was broken down into six stages:

1. **Minify the code** Code minification is the act of taking a piece of code such as "var myVariable = 123; var myOther = myVariable" and condensing it into a smaller equally valid statement such as "var a=123, b=a";. This was performed using Google's Closure compiler using the *ADVANCED_OPTIMIZATIONS* parameter which not only condensed the code by renaming symbols, but also performing compiler optimizations[67] to reduce the code structure into a more CPU friendly format. All minified code is then consolidated into a single file for easier handling.

2. **Picture Encryption** The Python script *to_image.py* takes in a string of text and converts it into a square image where image transformations are performed upon it. This is what the single consolidated minified code file is fed into. Two key steps occur:

   (a) The code is stored as a long contiguous character array, which is then broken into 32-bit elements, each of which are then broken into four 8-bit

[67] see Appendix page 239

elements that represent the four (8-bit) Red/Green/Blue/Alpha[68] channels of an image (that supports transparency).

(b) The image is encoded with a fairly simple cypher:

i. The diagonal pixels undergo a XOR[69] operation with a value of 247, and are then shifted diagonally 22 places.

ii. All other pixels undergo another XOR operation with a value of 93.

iii. Rows and columns are shuffled according to a shift map.

(c) The image is then saved as a lossless PNG file in order retain the pixel values determined in the previous steps.

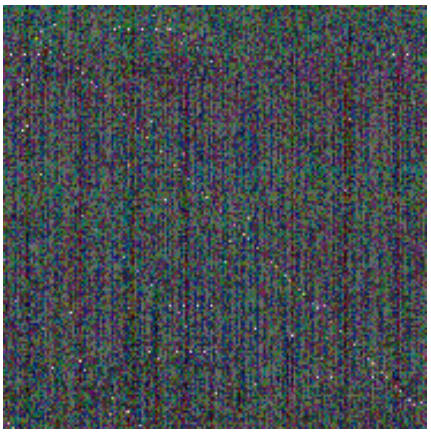

Figure 3.17: Example resultant image from code minification, pixelisation, and encryption.

[68] The alpha channel is a requirement to prevent WebKit powered browsers (Chromium, Safari) from changing RGB values whenever the alpha is unset.

[69] A XOR operation is a reversible bitwise function that essentially transforms $0 \rightarrow 1$, and $1 \rightarrow 0$.

4. Results

In this section we examine the output of seven linkage projects; five small straightforward of which three are non-problematic, and two with larger more complex pedigrees. Due to the focus of the nephrology research group at UCL, all linkage projects follow rare disease models, with a primary aim in determining a single disease locus indicative of a causative variant.

Bit-size	No. of Genotyped Individuals	No. of Markers	Model
3	5	35,155	Autosomal Recessive
5	6	41,479	Autosomal Recessive
7	9	43,421	Autosomal Dominant
9	8	43,421	Autosomal Dominant
15	11	41,480	X-Linked Dominant
18	7	50,110	Autosomal Recessive
21	10	15,914	Autosomal Dominant
23	11	15,914	Autosomal Dominant
29	14	47,595	Autosomal Recessive

Table 4.1: Summary table of the pedigrees evaluated in ascending order of bit-size.

Timings of each pipeline run are considered later in the section, and haplotypes are compared using **HaploPainter** and HaploHTML5.

4.1 Linkage Projects

The small pedigrees underwent two types of runs: single-core and multi-core. Due to the speed at which both types operate on small pedigrees, each type performed 10 trials each. The larger pedigrees were for the most part single-core due to resource limitations, but parallelizations were undertaken where possible.

We will examine the pipeline by looking at the four main visual components of each analysis: the pedigree, the **GRR** relationship charts, the Mendelian error plots,

and finally the linkage plots. Run times will be displayed at the end of the section.

4.1.1 Small Pedigrees

Here we look at a variety of pedigrees, each with an increasingly larger bit-size within the 19-bit limit that defines the threshold for large "big data" pedigrees. Small pedigrees are run through the pipeline without any runtime modifications required, resulting in a complete set of plots from all linkage programs.

3-bit Autosomal Recessive

Figure 4.1 shows a very simple pedigree; depicting one affected male (2053) amongst unaffected male (2052) and female (2054) siblings from two parents (2056 and 2055). All individuals are genotyped; the parents and the affected offspring being the main targets of informative meiosis, and the siblings acting as controls.

The GRR relationship chart shows normal bunching of sib-pairs (red), two close clusters of parent-offspring relations (yellow), and one single unrelated connection (cyan) between the two parents. For small non-consanguineous pedigrees, parents typically exhibit separate clusters of relation to their offspring since allele inheritance is never perfectly even and some offspring will share more alleles with one parent than another.

In larger (usually inbred) pedigrees, these two parental clusters either bunch together due to the parents being more related, or the gap between two outlying parent clusters are filled with other more-related parental clusters to form one large group that is still distinct from other sib-pair and half-sib groups.

The number of markers used for this analysis was 35,155 of which only 9 exhibited Mendelian errors and were filtered out from subsequent analysis. This constitutes 0.025% of the number of markers and is of no cause for concern, allowing the rest of the analysis to progress.

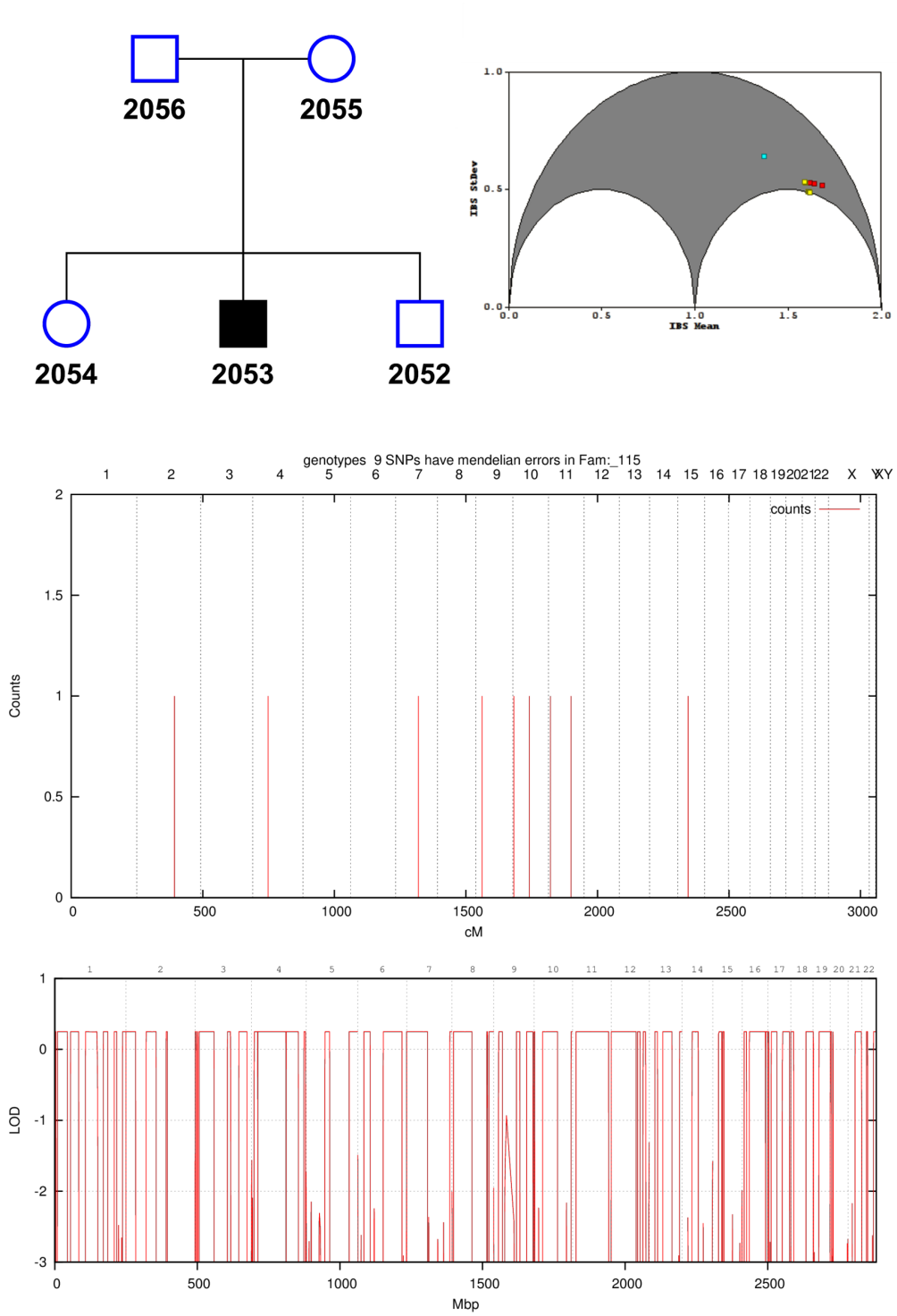

Figure 4.1: 3-bit Autosomal Recessive Pedigree. (Top) Family Tree and GRR, (Middle) Mendelian errors, (Bottom) Genomewide linkage plot.

Due to the low bit-size, the estimated lod score for the pedigree gave a maximum score of 0.2499 (**Allegro** and **GeneHunter**), which is reflected in the genome-wide plot. Unfortunately most regions of the genome reached this score ($\sim>$ 80%) likely due to there not being enough lack of homology between the affected and unaffected siblings. The score is also far below the minimum LOD score for significant linkage (LOD $>$ 3.3 [45]) meaning that the analysis was likely not very useful in identifying or excluding regions of interest for a causative disease locus.

5-bit Autosomal Recessive

This pedigree is very similar to the 3-bit pedigree above, but the bit-size increases by 2-bits for each non-founder added, and here we have one extra affected offspring. The GRR relationship diagram in Figure 4.2 also shows a similar pattern of groups as before, but this time there are more parental clusters which exhibit the faint beginnings of a larger parental cluster.

55 SNPs (of 41,479 markers) were identified having Mendelian errors, with a cluster of 6 towards the telomeric p-arm of chromosome 8. Removing these from the analysis also had negligible impact (0.133%).

The estimated LOD score was 0.8519 (Allegro and GeneHunter) which was once again reached by the linkage with a smaller number of peaks to indicate locus of interest. Though a single causative peak could not be determined, wide regions of exclusion defined chromosomes 2, 3, 6, and 7, as well much narrower peaks genome-wide.

We can also see a few distinct types of peaks: the nicely squared "flat" plateau peaks, the "rounded" bullet-like peaks, and the sharp "shard" peaks. Flat peaks tend to be of more worth to the analysis because they indicate that there are at minimum three markers exhibiting the same LOD score to constitute a left-wall, a right-wall and a point in-between.

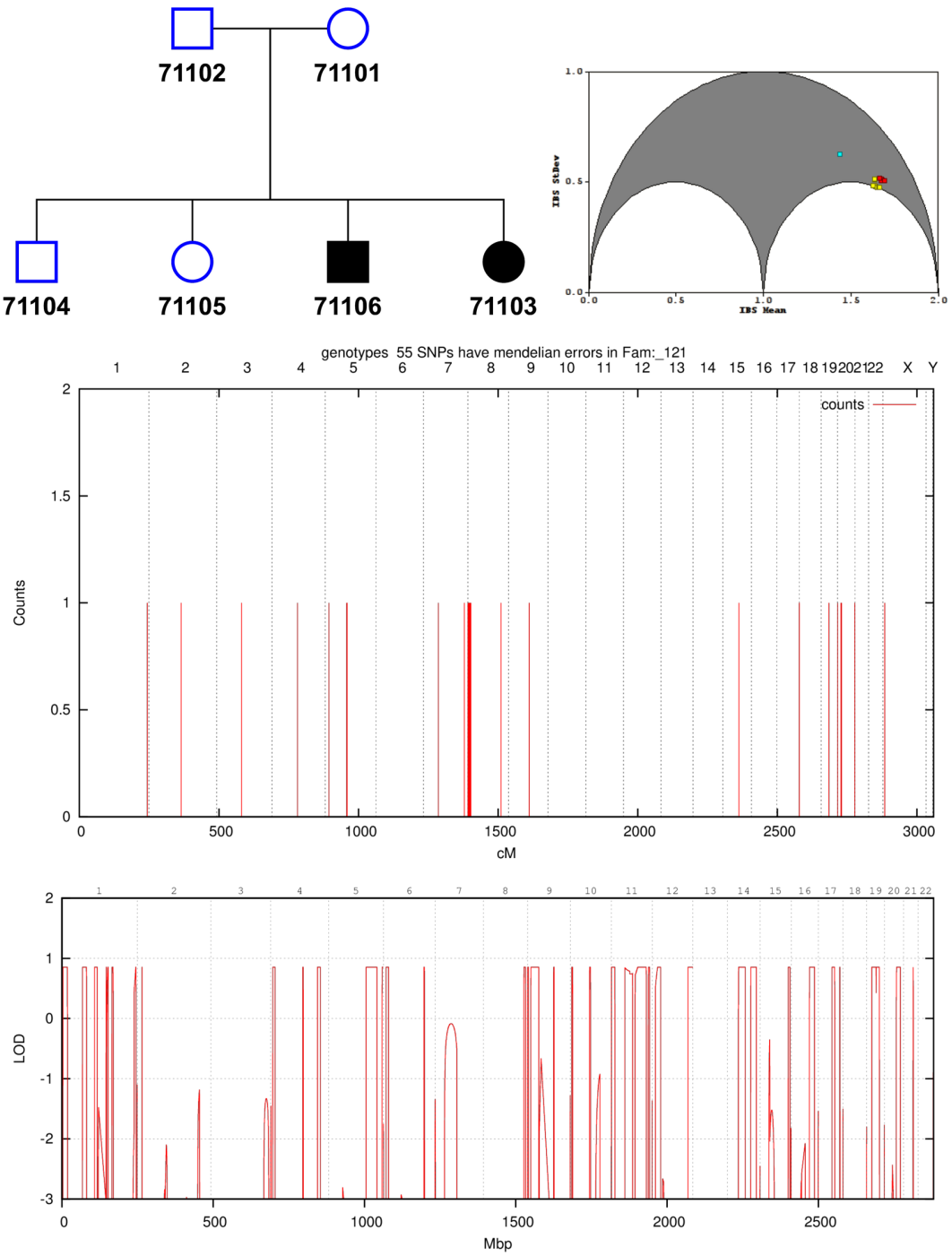

Figure 4.2: 5-bit Autosomal Recessive Pedigree.

Rounded peaks show discord or a lack of resolution within the linkage region. The small peak at chromosome 7 (LOD=-0.1) shows that at maximum there there are three markers constituting the peak with the left and right walls in rough agreement LOD-score wise, but that a single marker in-between them is significantly higher, forcing the plotting program (GNUplot) to smooth the peak into a rounded shape. These types of peaks are extremely common in low-marker analyses ($<$ 5000 SNPs) and are rare in high-marker analyses ($>$ 50,000 SNPs). Shards convey the same level of information, with the further disadvantage of only being constituted by two markers, both with unequal LOD scores.

7-bit and 9-bit Autosomal Dominant

In this analysis we see examples of excessive Mendelian errors, as well as the trial-and-error process of trying to determine the "true" model of a given data set through successive run iterations.

Figure 4.3 shows ten pedigree variants from the same initial base, each constructed to pinpoint the source of the high Mendelian errors shown in Figure 4.4 for the base scenario; 8000 out of 43,421 SNPs (18.4%) . The GRR plot for the run (Figure 4.5, 01) shows inconsistent bunching between sib-pair and parent-offspring groups with some mixing, clearly showing that something was indeed wrong with the pedigree.

The second scenario (run 02) assumed that the majority of errors stemmed from the father (2017) since genetic testing often reveals bad paternity approximately 15% of the time [46]. The father was truncated from the analysis to be replaced by an ungenotyped placeholder parent, which halved the number of errors to 3433 SNPs but also proved that the father was not the main source of the errors, hinting that one of the offspring was responsible. The GRR plot for the scenario (02) reflects this, for there is no longer any mixing between sib-pair and parent-offspring clusters, though the two extremely disparate groups for each relation still remains.

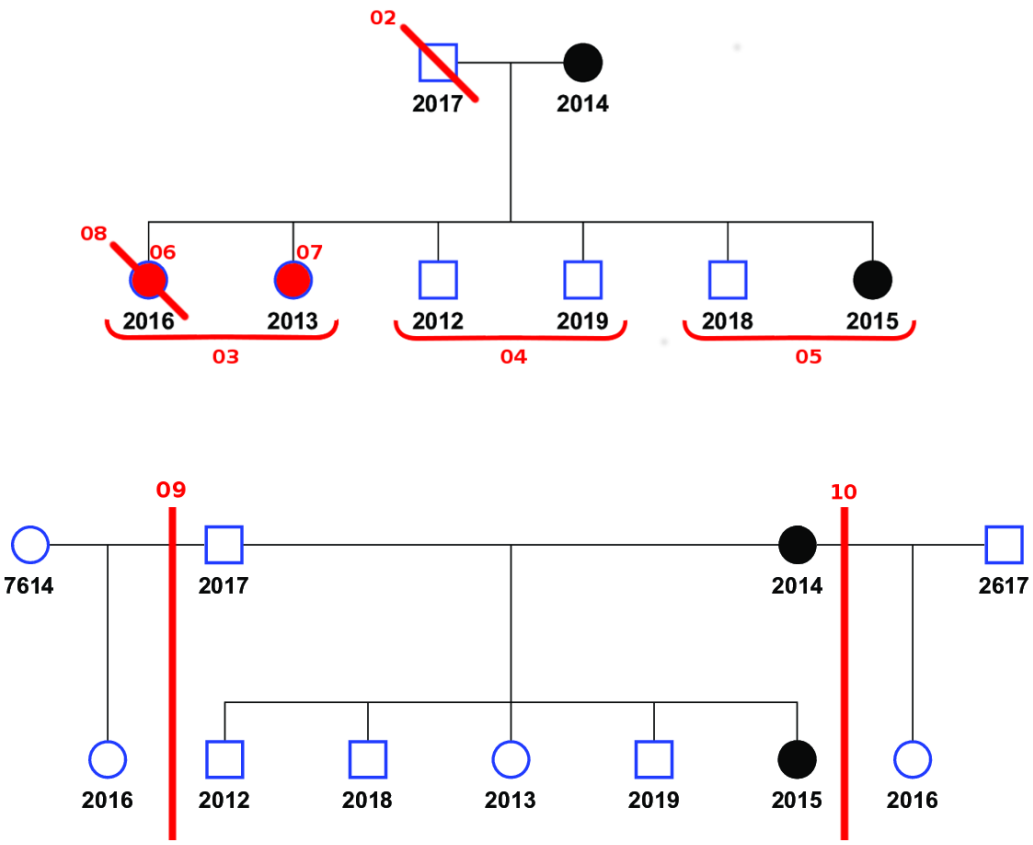

Figure 4.3: Autosomal Dominant Pedigree split into 10 scenarios. The base scenario (Top) exists without any red-line modifications. Individuals with a diagonal red line crossing through them denote a scenario (numbered in red) where that individual was removed from the pedigree. Individuals filled in red denote analyses where only that individual was considered in relation to their parents with all other offspring omitted. Red underlined groups denote parent-sibling analyses where siblings are grouped and evaluated separately. The bottom pedigree is the same as the top base scenario, but with alternate mother and father for individual 2016 making it half-sib with other offspring.

To pinpoint the offspring individual at fault with a reasonably small number of re-runs, offspring were split into pairs and run with their original parents in independent analyses: (run 03) 16 and 13, (run 04) 12 and 19, (run 05) 18 and 15. The analyses were terminated prematurely before the linkage stage in order to compare their relative errors without bias.

Run 03 reproduced over 99% of the original base errors with 7991 SNPs with Mendelian errors, with run 04 and run 05 producing 55 erroneous SNPs between them. The GRR plots also represented this grouping, with the runs 04 and runs 05 showing the classic close groupings of parent-offspring relations orbiting around the main sib-pair cluster, compared to run 03 which spread the parent-offspring relations in a more diverse fashion[70].

This clearly implicated 16 and 13 as the source of the errors, and the next two runs attempted to resolve this by splitting the pair and examining them separately: (run 06) 16 only, and (run 07) 13 only. Figure 4.4 shows that the errors between runs 06 and 07 clearly implicate individual 16 as the source of the errors, since the Mendelian errors once again rises to a high number of 7980 SNPs. GRR confirmed this by producing a cleaner parent-offspring grouping cluster for individual 13.

The next analysis (run 08) omitted 16 from the base analysis, preserving the original parents and offspring. This resulted in only 37 SNPs (0.0852%) with Mendelian errors across the entire analysis. GRR also showed better group clustering for sib-pair and parent-offspring relations, with much less disparity within each group, and no mixing.

Due to the more complete nature of this run, a genomewide linkage plot was also produced (Figure 6 (top)) showing distinct peaks in chromosomes 4, 7, 8, 9, 12, 13, and 17, 20, and 22 each with a LOD score of 1.25. The estimated LOD gave an expected maximum of 1.52 (Allegro and GeneHunter) which was not reached.

[70]Indeed, the relation between mother (14) and child (16) was scored to be the same as the unrelated relationship between the two parents (14 and 17).

Reconfirmation upon the sample data prompted a two more genotype sets to be introduced for a mother and a father of individual 16. Runs 09 and 10 denote alternate testing of each new parent in conjunction with an existing parent in order keep a half-sib relation between 16 and the other offspring. This created some disarray in the GRR plots for run 09 and 10, and even some relation mixing for run 10. The number of Mendelian errors also rose to significant levels (1849 and 2978 SNPs respectively) though the linkage peaks remained relatively unchanged.

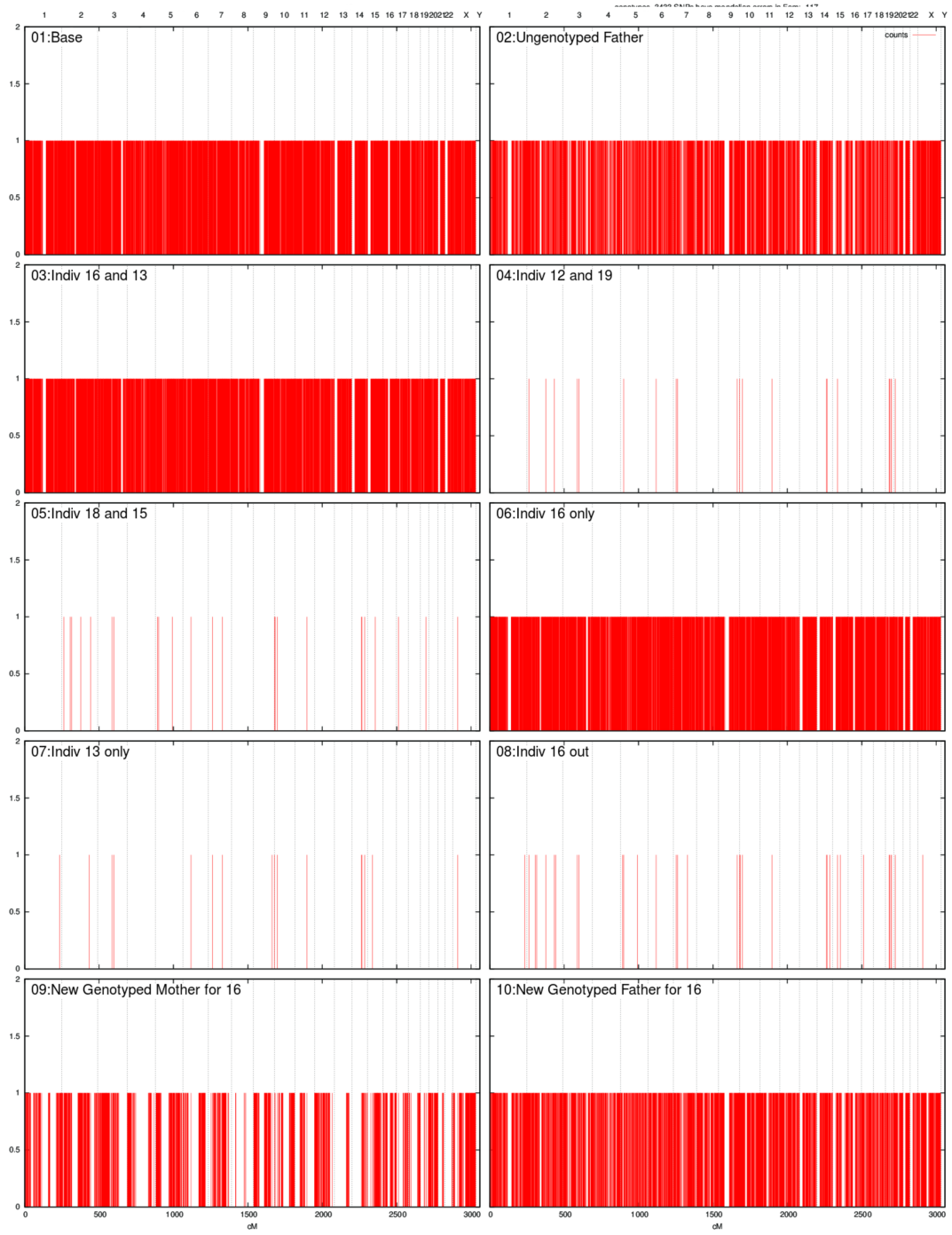

Figure 4.4: Mendelian errors for the ten scenarios depicted in Figure 4.3. Each of the scenarios represent the following number of errors: 01 (all base) 8000 SNPs, 02 (father excluded) 3433 SNPs, 03 (16+13 only) 7991 SNPs, 04 (12+19 only) 24 SNPs, 05 (18+15 only) 26 SNPs, 06 (16 only) 7980 SNPs, 07 (13 only) 18 SNPs, 08 (16 out) 37 SNPs, 09 (new mother for 16) 1849 SNPs, 10 (new father for 16) 2978 SNPs.

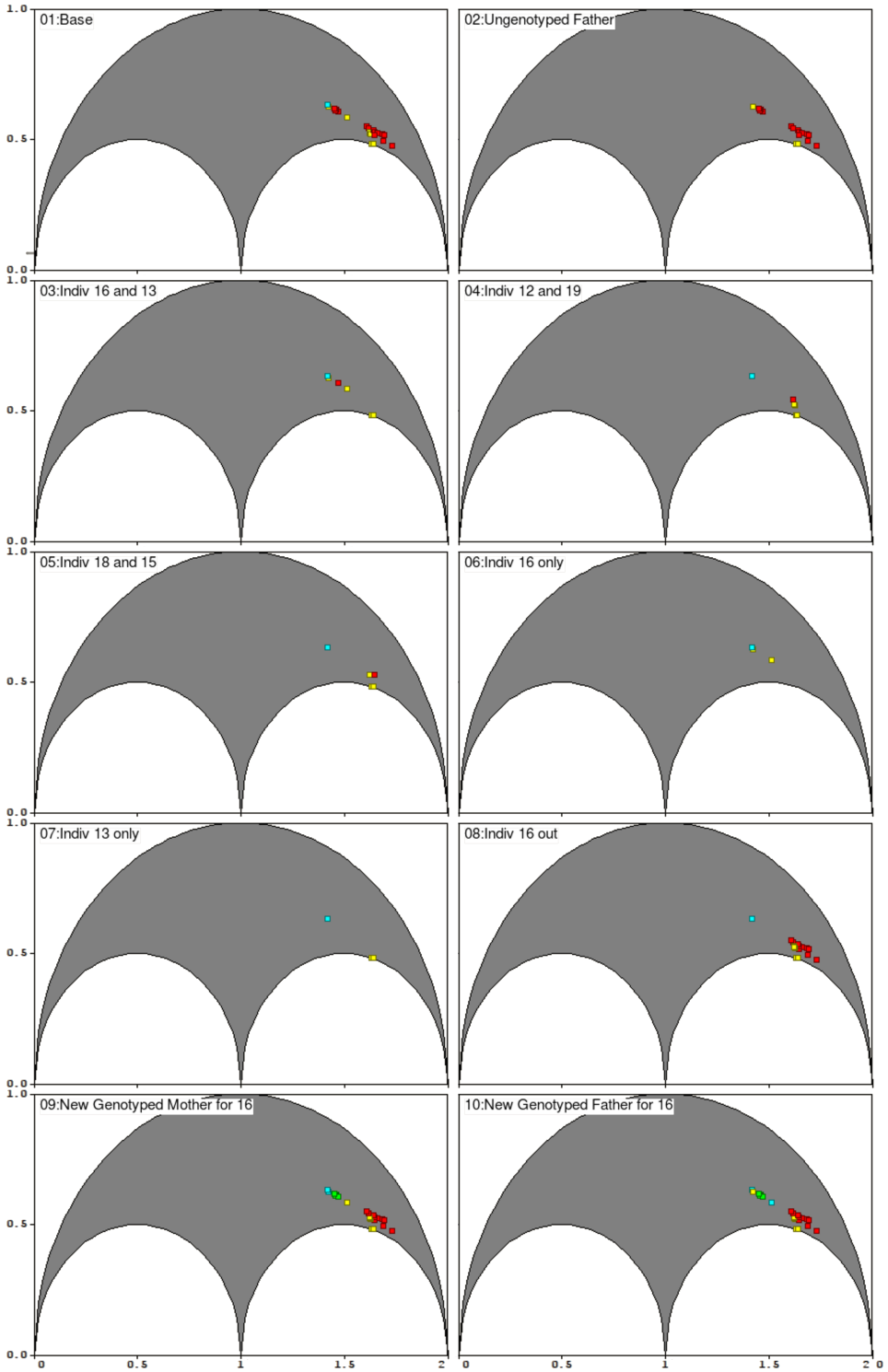

Figure 4.5: GRR plots for the ten scenarios. Identity-by-State (IBS) is plotted as a function of Mean against Standard Deviation and bound within an arced range of possible values.

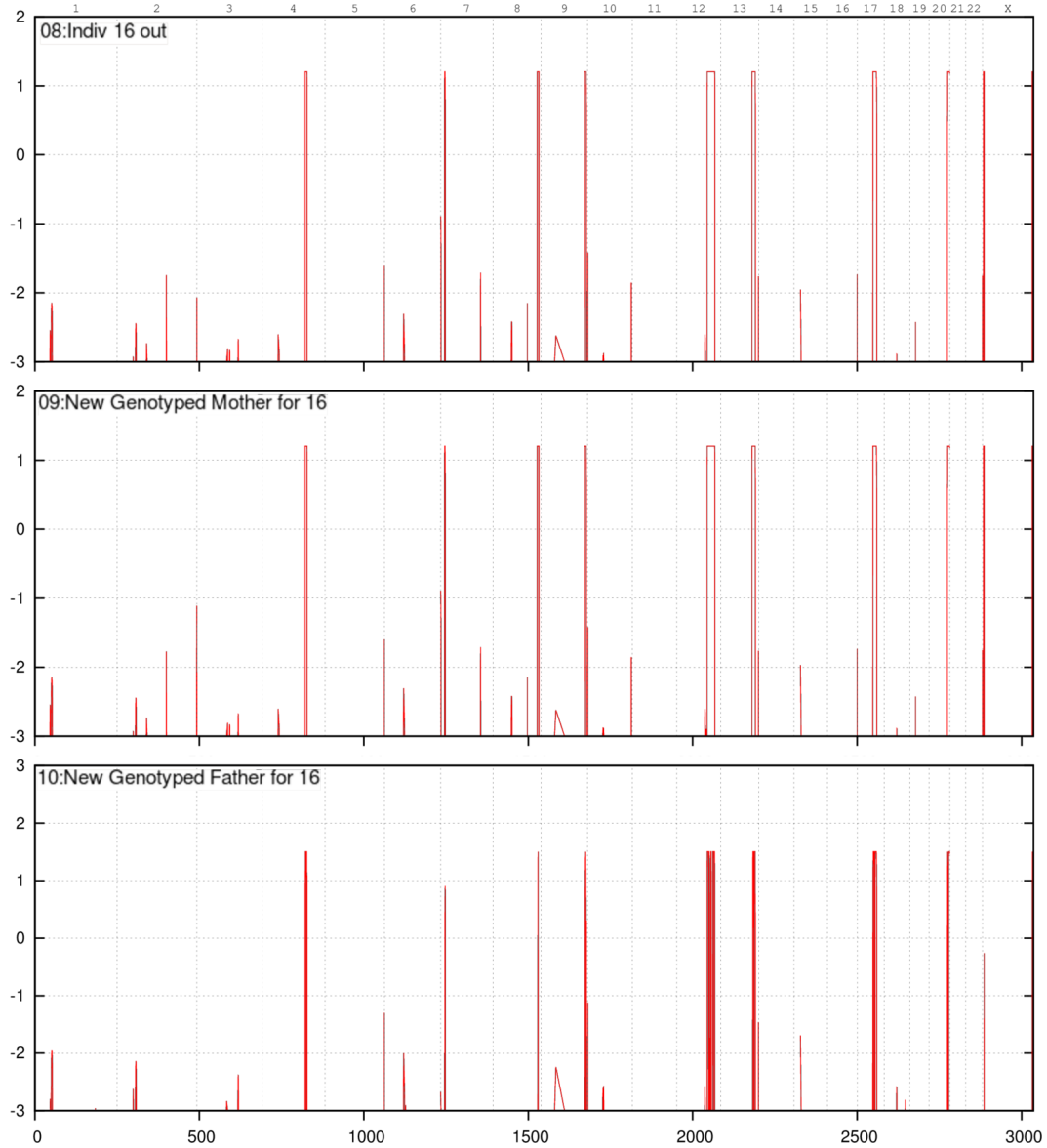

Figure 4.6: Genomewide linkage plots for three of the ten scenarios that did not produce Mendelian errors: (08) with individual 16 omitted, (09) with a new genotyped mother for individual 16, and (10) with a new genotyped father for individual 16.

15-bit X-Linked Dominant

Figure 4.7 below shows a larger pedigree, with three individuals of unknown affectation status. The penetrance model is dominant because an affected individual resides at each generation, and it is suspected to be X-linked because of the absence of male-male transmission though that is not to say that it is not autosomal.

Normally when a penetrance model is fully described in a pedigree, then individuals of unknown affectation are pre-emptively set to either affected or unaffected in order to better accommodate the model. Here the phenotype is not fully penetrant, so ambiguity was purposefully left for the linkage pipeline to interpret.

Despite autosomal dominant and X-linked dominant not being biologically compatible models, computationally they are processed in the same run because each chromosome is treated as fully independent of each other. The linkage pipeline treats X-linked models as "special cases" in conjunction with the standard dominant and recessive runs. It is for this reason that the X chromosome is included with autosomal chromosomes in the genome-wide plots, though the historical threshold for significant linkage is lowered to 2 [47].

The low 162 Mendelian errors are usually negligible for marker set of 41,480 SNPs (0.391%), but the close bunching of 52 SNPs in the q-arm of chromosome X raises some concern on possible genotyping errors. This is later reflected in the genome-wide linkage plot which consists of extremely "noisy" peaks: sharp and intermittent, consisting of no more than two markers before the signal is lost and drops below significance, only to resurrect again within close proximity of the previous signal.

The size and frequency of the peaks indicate at least one of four issues:

1. Recombinations exist abundantly within the three generations of the pedigree.
2. The family was badly genotyped with errors stemming from the genotyping process.
3. The pedigree structure is not correct and the family model is more complex.
4. The penetrance model is incorrect.

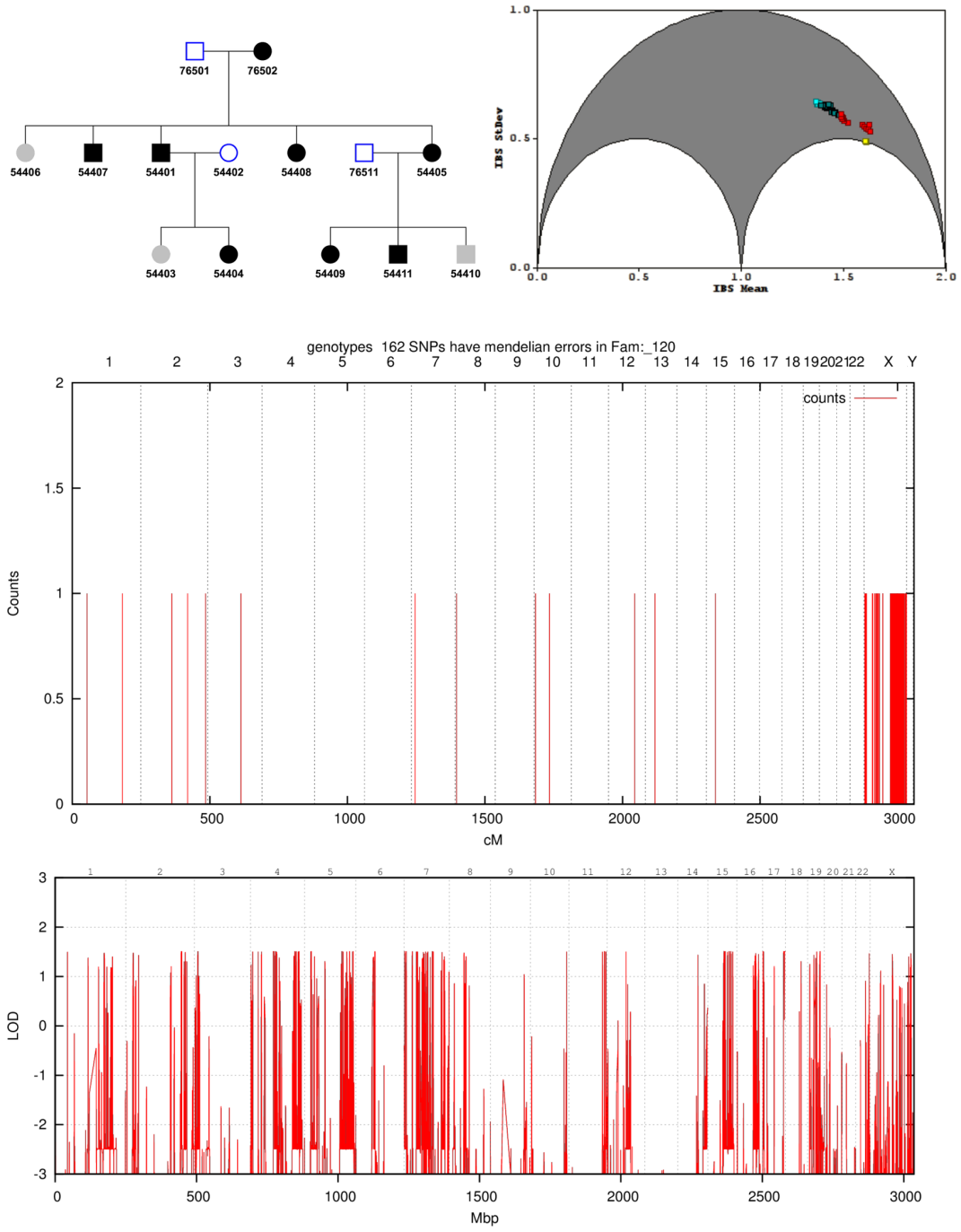

Figure 4.7: X-linked Dominant pedigree, with three individuals of unknown affectation marked in grey (54406, 54403, 54410).

Many of the recombinations occur within extremely small proximity of one another, and since physical distance scales linearly with centiMorgans, a large number of meioses in those regions are extremely unlikely. Similarly, the family could not have been badly genotyped nor could the pedigree structure be incorrect, since the number of Mendelian errors are relatively low.

This defers us to the conclusion that the penetrance model is incorrect, with the affectation of certain individuals (specifically those not already set to 'unknown') not being correctly determined by the physician, possibly due to a late-onset phenotype.

Further probing of the pedigree under different models (autosomal/X-linked recessive) would be required under various combinations of individual affectation in order to stabilize the linkage peaks and determine the true model behind the data.

18-bit Autosomal Recessive

Figure 4.8 shows a family with an inbreeding loop stemming primarily from the founders on the right (101 and 102) that results in a second-cousin consanguineous pairing (401 and 402).

Occasionally members of consanguineous families have a tendency of obscuring such relations, but the members of this family are fully described with their GRR relationship plot clustering in a good distinct groupings. The lack of extra relations in the plot is due to the only 7 of the individuals being genotyped (401, 402, 404, 403, 501, 502, 503).

Of the 50,110 SNPs in the input marker set, 22 negligible Mendelian errors were reported (0.439%). The preliminary LOD-score estimates provided a score of 2.9, which was met accordingly in the actual linkage output with peaks at chromosome 2, 5, and 7. Figure 4.9 shows the zoomed in plot of chromosome 7, with chromosomal bands (and sub-bands) overlayed with a centromere.

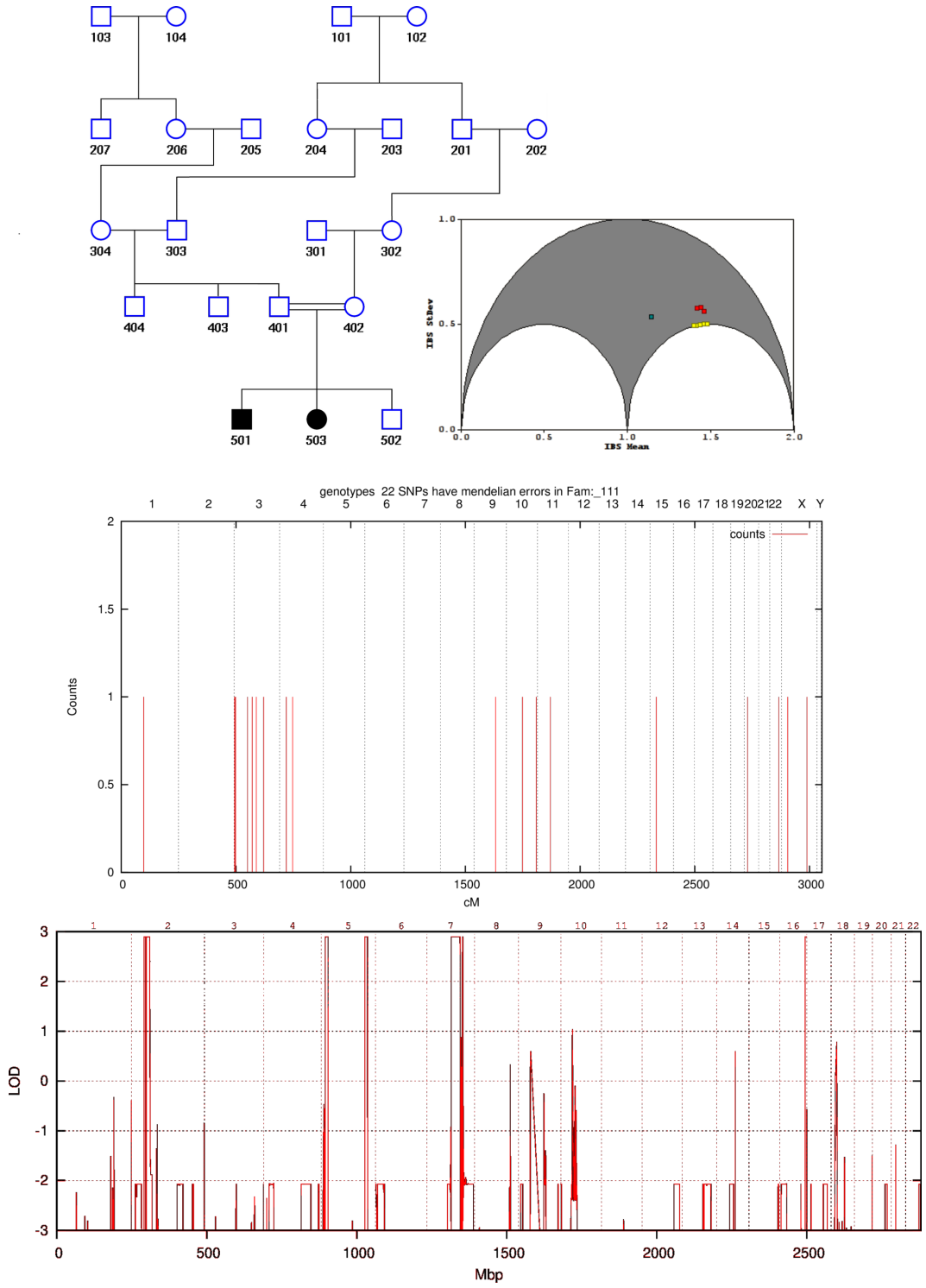

Figure 4.8: 18-bit Autosomal Recessive Pedigree.

The darkness of the band is indicative of the density of the chromatin, hinting at regions of heterochromatin (dark) and euchromatin (light) which provide cues for gene expression. A single characteristic flat peak spans a region of 30 Mbp along the q-arm, followed by sharp recombination peak artefacts.

The peaks at the other chromosomes also display valid linkage, and it is up to the researcher to examine all valid regions through further experimental sequencing analysis in order to truly pinpoint the causative gene/mutation.

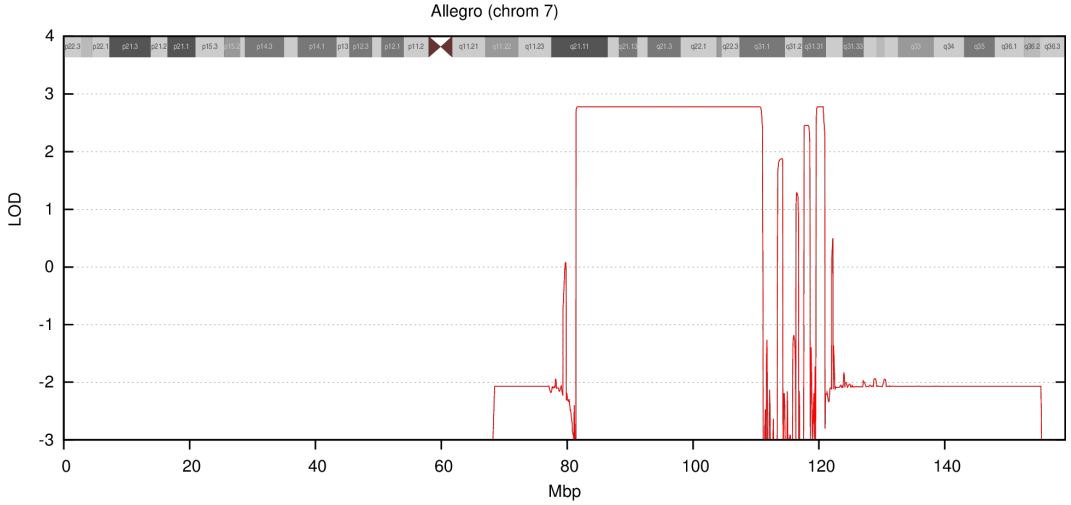

Figure 4.9: 18-bit Autosomal Recessive Pedigree, chromosome 7 (Allegro).

4.1.2 Large Pedigrees

The pedigrees evaluated here required the big data patching scripts mentioned previously in the Methods to operate; the default Allegro and GeneHunter binaries failed due to the combined size of the marker set and the pedigree complexity, requiring the modified Allegro binary to operate as well as some trials with Simwalk.

21-bit and 23-bit Autosomal Recessive Pedigree

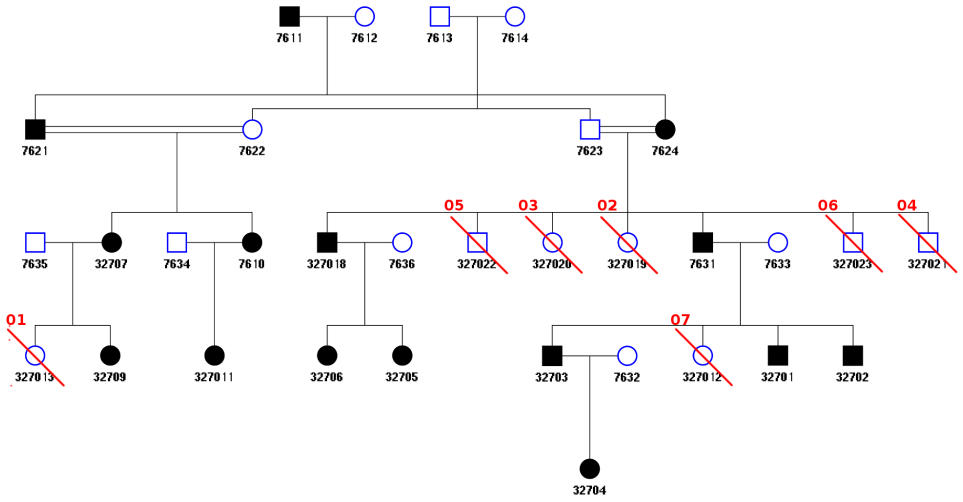

Figure 4.10: 23 bit pedigree with red-lines indicating the individuals omitted from a given analysis.

Here we examine another large consanguineous pedigree with two inbreeding loops occurring both within the second generation (7621 and 7622, 7623 and 7624) from the two founder couples (7611 and 7612, 7613 and 7614) with seven variations upon the inclusion of unaffected individuals spawned from a base analysis where only affected individuals were included. The base analysis has a 21-bit pedigree complexity, and as per equation 1.2 on page 28, each non-founder contributes 2-bits which places all subsequent analyses at 23-bits.

Each of the 8 total scenarios contributed negligible Mendelian errors (in total less than 10 SNPs of the 15,914 in the marker set), with an example chart shown in Figure 4.11 for completion[71]. A total of 18 genotyped individuals contributed to the pedigree (25 individuals) providing a sizeable cluster of 'other' relationships in the GRR plot (Figure 12) for scenario 02 (individual 19[72] omitted). No significant changes in the GRR plots for the other scenarios were observed.

[71] Full LOD scores for all 8 scenarios are provided in the Appendix (page 245).
[72] Individual 19 = 327019, but we have removed the '3270' prefix for readability.

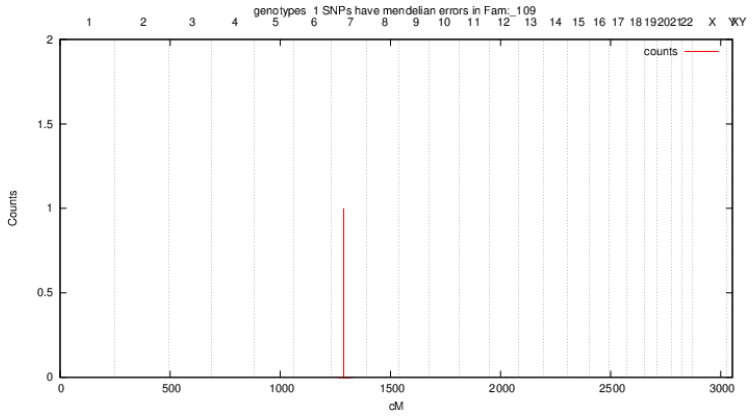

Figure 4.11: Mendelian error for the base scenario (21-bit). All other analyses gave either 0 or at maximum 3 erroneous SNPs.

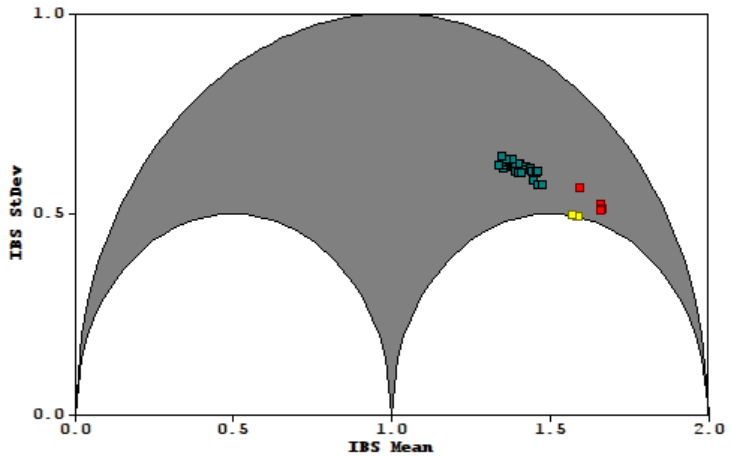

Figure 4.12: GRR relationship chart of a 23-bit pedigree for scenario 02 (individual 327019 omitted).

The maximum LOD score for the base scenario (21-bit) was estimated to 3.31, and the estimate LODs for the other (23-bit) analyses gave an average of 3.62[73].

The independent addition of each of the unaffected individuals for the seven non-base scenarios (individuals: 13, 19, 20, 21, 22, 23, and 12) created small but noticeable differences in linkage plots (Figure 4.13).

[73] see footnote 71.

The base analysis (scenario 0) shows peaks at chromosomes 3, 4 and 11 all reaching the maximum estimated LOD score (3.35) with the peak at chromosome 4 being the broadest. A zoomed in plot of chromosome 4 for that analysis reveals a slight drop in the peak at 45 Mbp (Figure 4.14, top-left). The peaks at chromosomes 3 and 11 are equally viable linkage peaks at this point[74].

By looking at each linkage result evaluated, we can groups the plots into three result types:

1. With the independent additions of individuals 13 (scenario 1), 22 (scenario 5), and 12 (scenario 7), the peak at chromosome 4 increases to the new maximum (3.62) whilst the slight drop disappears, and the peaks at chromosomes 3 and 11 decrease.

2. The independent addition of individuals 19 (scenario 2), 20 (scenario 3) cause the chromosome 4 peak to drop as well as the chromosome 11 peak, leaving a fractured peak at chromosome 3 that raises doubt on the informativeness of the peak.

3. Individuals 21 (scenario 4) and 23 (scenario 6) preserve the peaks at chromosomes 3 and 4, but greatly fractures them both too, hinting at an incompatibility.

The width of the chromosome 4 peak (26.8 Mbp) does not narrow with the addition of individuals 13, 22, and 22 as in the first result type, suggesting that the disease locus is within a reasonably conserved locus across the pedigree. A higher resolution trial would likely be required to narrow the region.

[74] Plots for all chromosomes can be found in the Appendix (page 246).

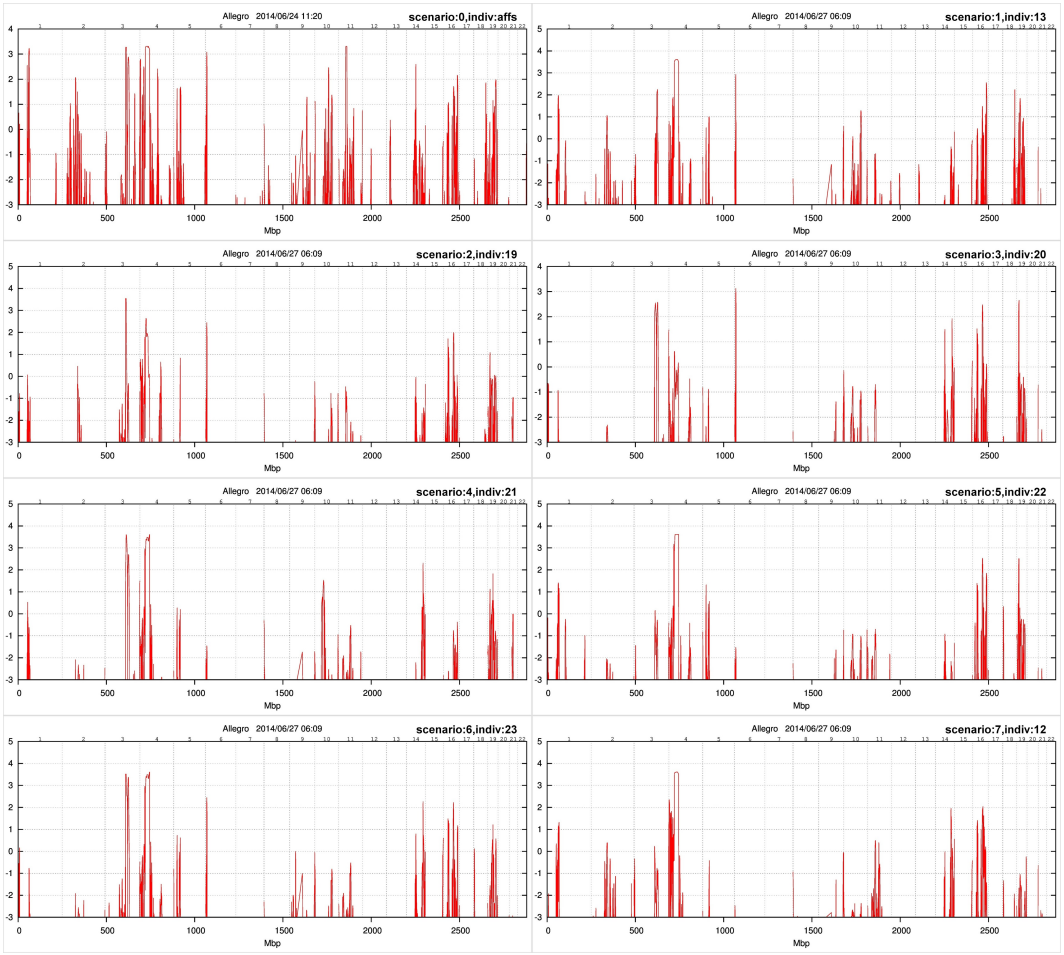

Figure 4.13: Genomewide linkage plots for each of the eight scenarios for the 21 to 23-bit autosomal dominant pedigree.

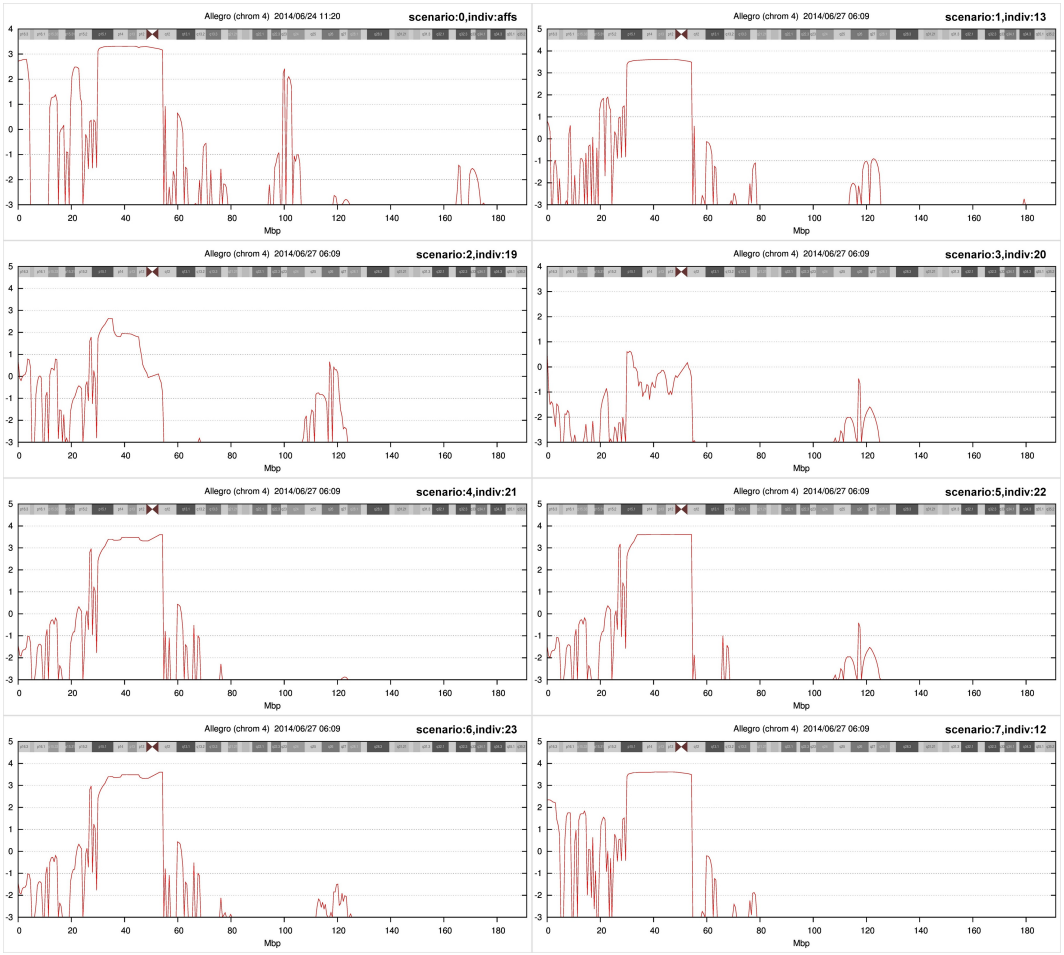

Figure 4.14: Linkage plots of chromosome 4 for each of the eight analyses

i 29-bit Autosomal Recessive

This is the largest family ever considered by the pipeline. Figure 4.15 consists of three inbreeding loops that span four generations stemming from the same founder couple, where all individuals in the last generation were genotyped along with their parents. Parent-offspring genotyping is always more favoured than grandparent-offspring genotyping because it leaves little ambiguity in tracing the path of inheritance of the disease locus, and makes detecting Mendelian errors much more effective.

Three pedigrees were used in this analysis, the first being the 29-bit pedigree used to lend power to the linkage study and the latter two (Figure 4.16) not being informative for linkage at all but aided in the haplotype reconstruction process to better understand the disease locus via sideways comparison.

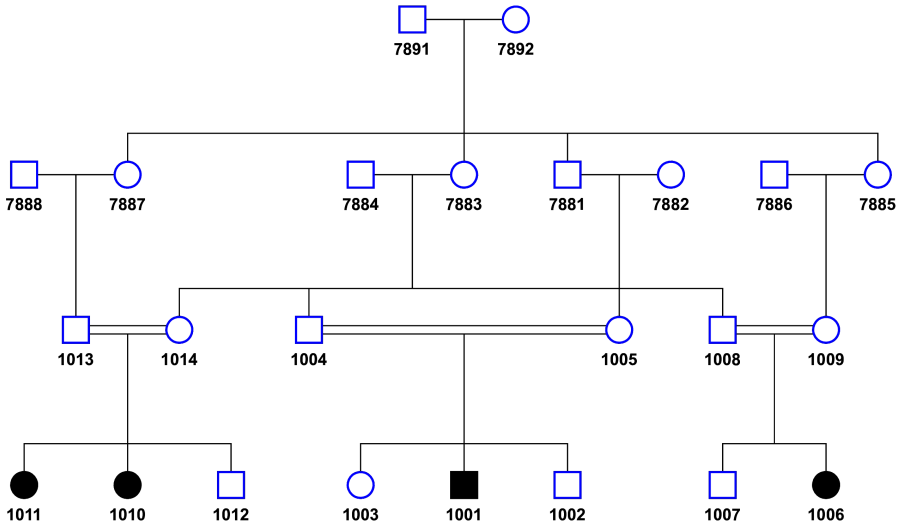

Figure 4.15: 29-bit Autosomal Recessive Pedigree, 4 affecteds in the last generation.

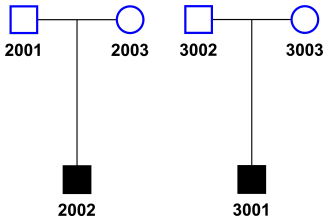

Figure 4.16: Two completely uninformative pedigrees for linkage (bit-size of -1).

The combined Mendelian errors across the three pedigrees amounted to no more than 175 SNPs out of the total 47,595 selected for linkage (0.368%) , with 3 SNPs encountered more than once as shown in Figure 4.17.

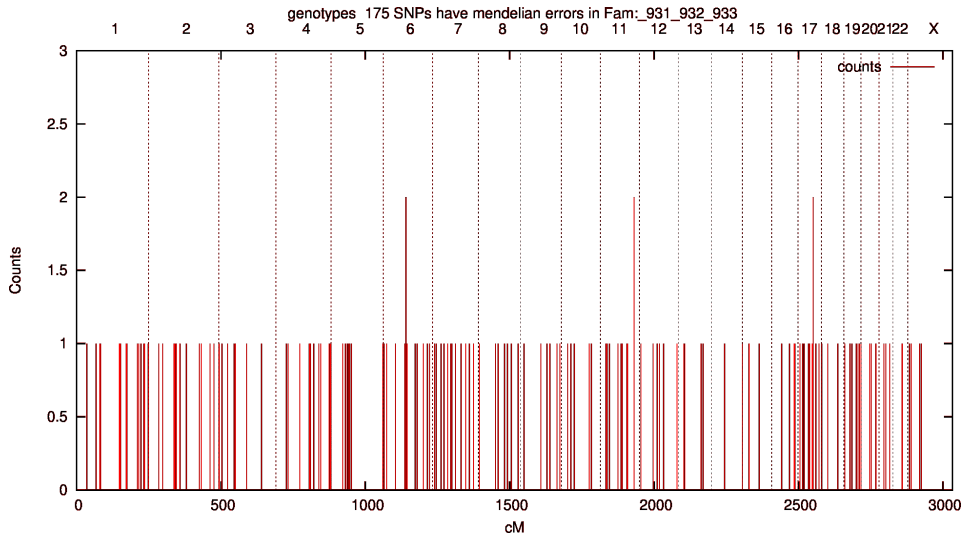

Figure 4.17: Mendelian errors for the three families considered under the 21-bit Autosomal Recessive analysis. The three peaks relate to an overlap of an erroneous marker in two of the three families.

The GRR plot of the first family (Figure 18) showed good distinct relational grouping with some potential border overlap between the sib-pair and parent-offspring relations, but no actual outliers in either group. The GRR plots of the other two families were normal (Figure 19).

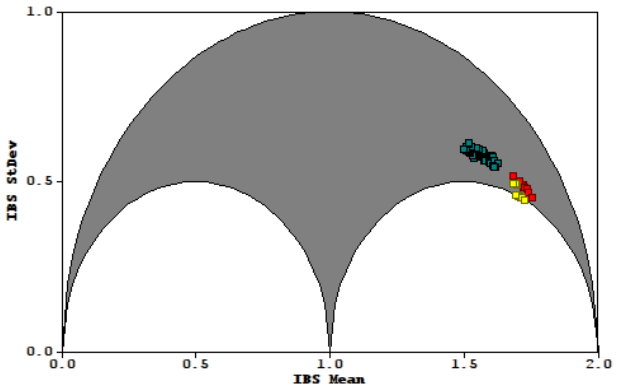

Figure 4.18: GRR plot of 29-bit family.

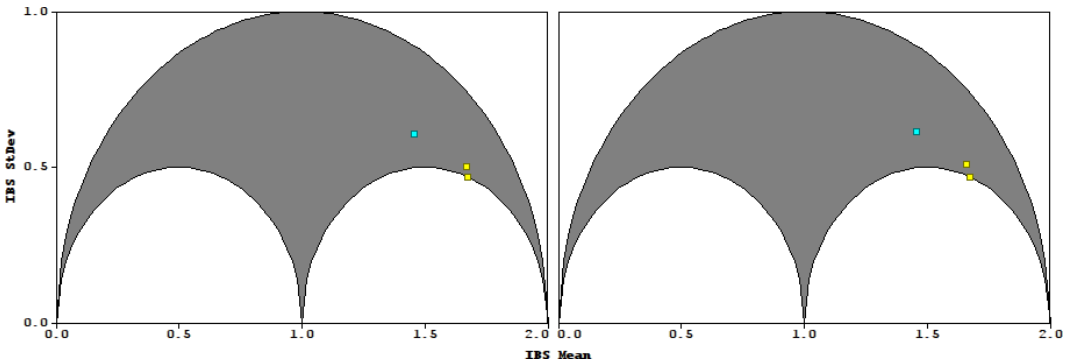

Figure 4.19: GRR plots of the smaller two families that contributed to the 29-bit analysis.

Due to the complexity of the analysis, genome-wide analysis had to be performed via **Simwalk** due to the big data errors mentioned previously (see page 63 of Methods). An estimated maximum LOD score could also not be computed because of this reason since the calculation relied upon GeneHunter and Allegro, both of which were not able to process the analysis in any reasonable timeframe.

The sliding-window approach of Simwalk allowed the parallelization techniques employed by the *simwalk_ multicore.sh* script to fully utilize all threads available on platform without throttling the RAM. Timings of these runs are discussed later in the next section.

The Simwalk run produced a single peak on the p-arm (p13.3 to p13.2) of chromosome 16 with a LOD score of 5.21 (Figures 4.20 and 4.21) . In order to confirm the Simwalk result and to gain haplotype reconstruction, Allegro was required. A single chromosomal analysis was performed on chromosome 16, reproducing the peak at the same locus with a LOD score of 5.23 (Figure 4.22).

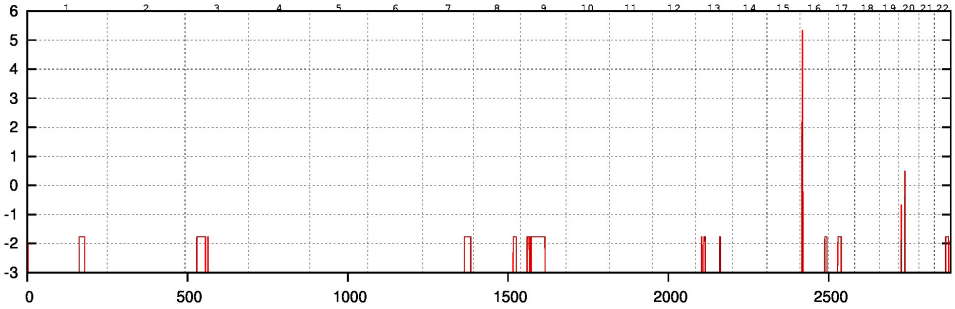

Figure 4.20: Simwalk plot of 29-bit family (Genomewide).

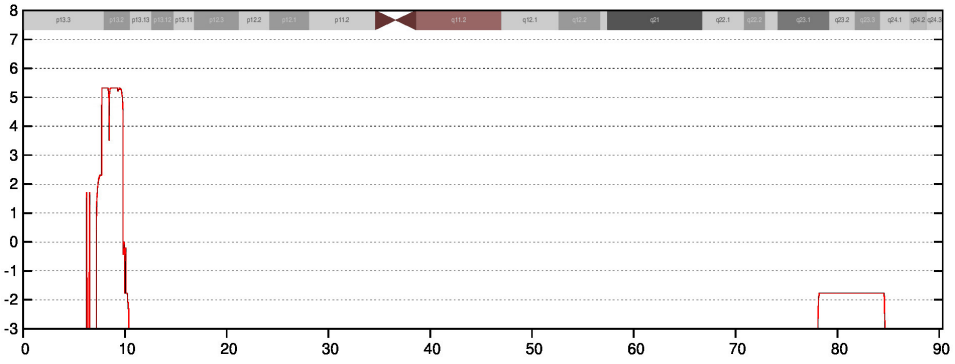

Figure 4.21: Simwalk plot of 29-bit family (chr16).

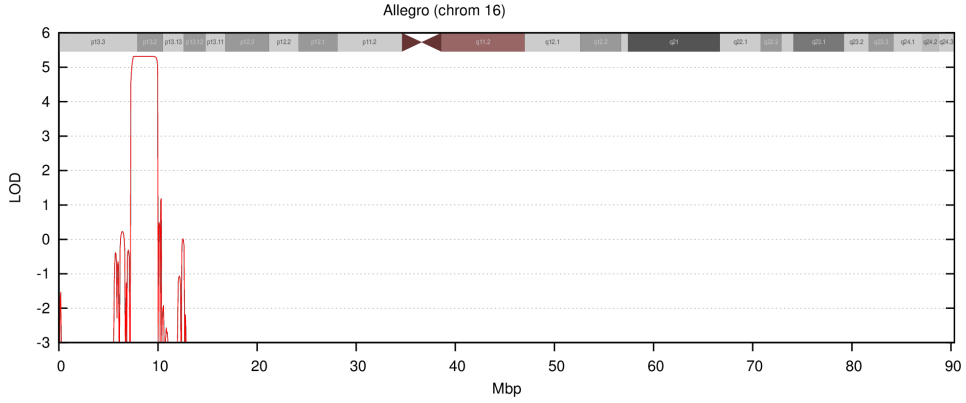

Figure 4.22: Allegro plot of 29-bit families (chr16), combined analysis but only first family contributes to the peak score.

4.2 Linkage Run Times

All analyses were timed using the GNU time utility in order to get the "wall clock" runtimes from the calling (bash) script.

4.2.1 GRR

For all projects with pedigree complexities less than 19-bits, the main limiting factor in their runtimes was the graphical GRR Xautomation script. Table 4.2 below shows how the mean runtime remains at a reasonably constant 122 seconds up to the 19-bit limit, where GRR window polling then begins to take precedence and longer wait times are required for the genotypes to fully load.

Bit-size	No. of Genotyped Individuals	No. of Markers	Total Genotypes	Trials (seconds) 1	2	3	4	Mean
3	5	35,155	75,775	121.91	122.08	121.87	121.62	121.87
5	6	41,479	248,874	121.85	121.52	121.62	121.81	121.70
7	9	43,421	390,789	121.96	121.50	122.29	122.34	122.02
9	8	43,421	347,368	122.18	121.98	122.28	122.33	122.19
15	11	41,480	456,280	122.14	121.51	122.31	122.03	121.99
18	7	50,110	350,770	123.21	122.75	122.59	122.23	122.69
21	10	15,914	159,140	129.82	130.11	130.20	129.15	129.82
23	11	15,914	175,054	152.81	151.11	151.64	152.37	151.99
29	14	47,595	666,330	178.66	180.42	179.31	183.75	180.54

Table 4.2: GRR run times over 4 trials, ordered by ascending bit-size. The number of genotyped individuals and the number of genotypes per individuals are also listed.

A plot of Table 4.2 shows GRR mean run times against normalized bit-size, number of genotyped individuals, number of markers and total genotypes. The total genotypes, bit-size, and number of genotypes appear to follow an upwards sloping power trend, whereas the run time appears to be more independent of the number of markers.

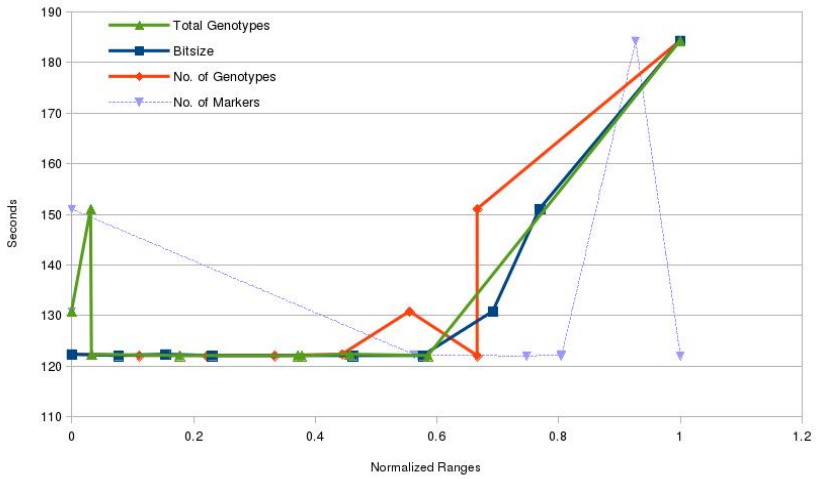

Figure 4.23: GRR run times. (Normalized) Genotypes, Markers, Total Genotypes, and Bitsize against Run Time.

The GRR runtimes appear to be constant at approximately 122 seconds within the established 19 bit limit with a small increase to 123 seconds at 18-bits. Larger pedigrees begin to scale more linearly, rising to 180 by the 29-bit stage, at which stage the contribution of the GRR window polling script runtime becomes negligible in contrast to the Allegro runtimes outlined in Table 4.3 below.

Bit	**SingleCore Runtimes (seconds)**				
Size	**1**	**2**	**3**	**4**	**Mean**
3	81.63	80	83	84.1	82.13
5	86.84	84	84	87.07	85.26
7	146.72	148	148	150.77	148.29
9	378.72	380	379	377.17	378.63
15	775.54	735	742	768.11	755.11
18	2779.57	2,810	2,795	2758.11	2785.59
21	943891.22	940,985	948,115	946515.6	944876.54
23	6531019.41	6780937.4			6655978.41

Table 4.3: Total Genomewide Allegro Single-Core Run Times for MPT and Haplotypes. Single-Core timings for 23-bit pedigree underwent two trials only.

4.2.2 Chromosome-Specific

The following table contains the average chromosome-specific[75] runtimes that comprise the single-core components of Table 4.3.

	Bit-size								
Chrom	**3**	**5**	**7**	**9**	**15**	**18**	**21**	**23***	**29***
1	3.97	4.02	12.88	50.67	138.89	595.14	87849.27	730875.58	-
2	4.51	4.02	10.27	40.16	99.07	433.59	83949.84	635076.98	-
3	4.14	3.63	8.84	29.94	68.51	322.90	76173.28	665699.06	-
4	3.69	3.78	7.93	24.66	48.83	232.71	70044.33	600316.88	-
5	3.39	3.75	7.31	19.69	35.43	177.24	64033.08	493473.21	-
6	3.94	4.72	6.63	16.78	28.98	128.61	59448.95	461203.36	-
7	4.01	3.91	6.19	14.36	23.52	99.59	53240.12	409079.04	-
8	3.68	3.62	5.83	12.47	21.52	74.21	48531.97	378953.42	-
9	2.86	3.39	5.64	11.52	18.19	59.12	44332.31	280859.56	-
10	3.91	3.45	5.51	10.68	16.42	47.62	39859.11	275189.27	-
11	3.63	4.02	5.36	10.35	14.96	38.11	35716.49	259160.94	-
12	3.60	4.56	5.29	9.82	15.22	30.88	32045.78	217073.37	-
13	3.49	3.37	5.17	9.62	15.14	26.25	28290.74	167115.23	-
14	3.09	3.21	5.16	9.57	14.12	23.46	24899.32	148455.32	-
15	2.38	3.09	5.13	9.43	15.16	19.08	21915.39	110163.35	-
16	3.19	3.42	5.06	9.17	13.22	17.93	19353.51	76464.53	315615.23
17	3.77	4.32	5.09	9.07	14.46	17.95	16933.29	81678.43	-
18	3.73	3.69	5.02	9.04	14.43	15.91	14746.18	66359.72	-
19	3.07	3.44	5.08	9.26	13.77	16.14	12955.06	33812.14	-
20	3.31	3.23	5.00	9.01	14.27	15.33	11266.49	19439.38	-
21	3.71	2.91	5.06	9.11	14.18	16.13	10049.32	15547.82	-
22	3.54	3.55	4.99	9.02	13.71	14.22	9065.98	35267.98	-
X	3.52	4.28	9.87	35.33	83.09	363.44	80176.73	494713.86	-
Total	82.13	85.38	148.31	378.73	755.09	2785.56	944876.54	6655978.43	

Table 4.4: Average Single-Core Allegro Run Times (seconds) for Multi-point Parametric Linkage and Haplotype Reconstruction; Bit-Size (columns) against Chromosome (rows). All analyses are the result of 4 trials, except the 23-bit analysis which was only able to run for 2 trials and the 29-bit analysis which only run on chromosome 16, both analyses without haplotypes.

[75] For a full disclosure of all trials run, please see Appendix page 257.

4.2.3 Total Genomewide Singlecore vs Multicore

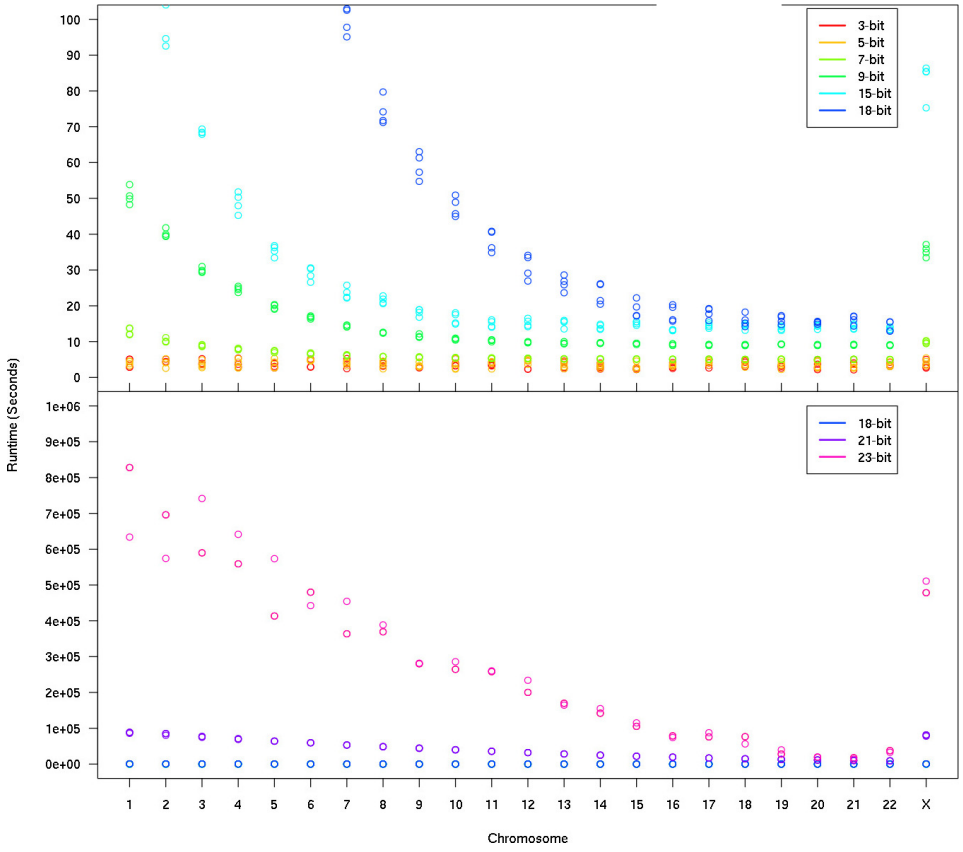

Figure 4.24: Single-core Allegro run times for each chromosome. Small pedigrees from 3-bit to 18-bit with runtimes within [0,400] range (Top), larger pedigrees from 18-bit to 23-bit shown with runtimes as powers of 10 (Bottom).

Figure 4.24 shows the plot of Table 4.4 at different scales.

The top image showing the general logarithmic decrease in runtime with increasing chromosome number (1 to 22). The sharp rise at the end is due to the inclusion of chromosome X which in terms of size is more similar to chromosomes 1 and 2 than chromosome 22. For 3-bit, 5-bit, and 7-bit pedigrees, the time required to process each chromosome is almost constant; the main contributing factor to the timing being the overhead in the system calls related to the loading and unloading

the Allegro binary. Should the Allegro binary remain in memory throughout the duration of a genomewide run, the plots may reflect the size of the chromosomes they process more accurately. 9-bit, 15-bit, and 18-bit pedigrees begin to take on the logarithmic shape that better exemplifies the size of the chromosome, where the overhead in system calls becomes more negligible.

The bottom image represents a zoomed out Y-axis scale of the same data by 5 orders of magnitude. The smaller ({3,5,7,15,18}-bit) pedigrees appear grouped as a flat plot towards the bottom, whilst the 21-bit and 23-bit rise up above it. The difference in runtimes between 21-bit and 23-bit greatly deviates for the lower-number chromosomes, with chromosome 1 taking approximately 8 times longer to process for the latter analysis.

Bit	**Multicore Runtimes (seconds)**				
Size	**#1**	**#2**	**#3**	**#4**	**Mean**
3	55.79	55.88	54	50.37	54.01
5	57.53	60.07	57.24	60.24	58.77
7	62.13	56.32	58.51	60.09	59.26
9	81.7	80.74	86.99	81.97	82.85
15	116.62	108.27	103.63	112.82	110.34
18	389.55	415.02	400.36	404.79	402.43

Table 4.5: Total Genome-wide Allegro Multi-core Run Times for MPT and Haplotypes. Multicore timings for 21-bit and 23-bit analyses were not processed.

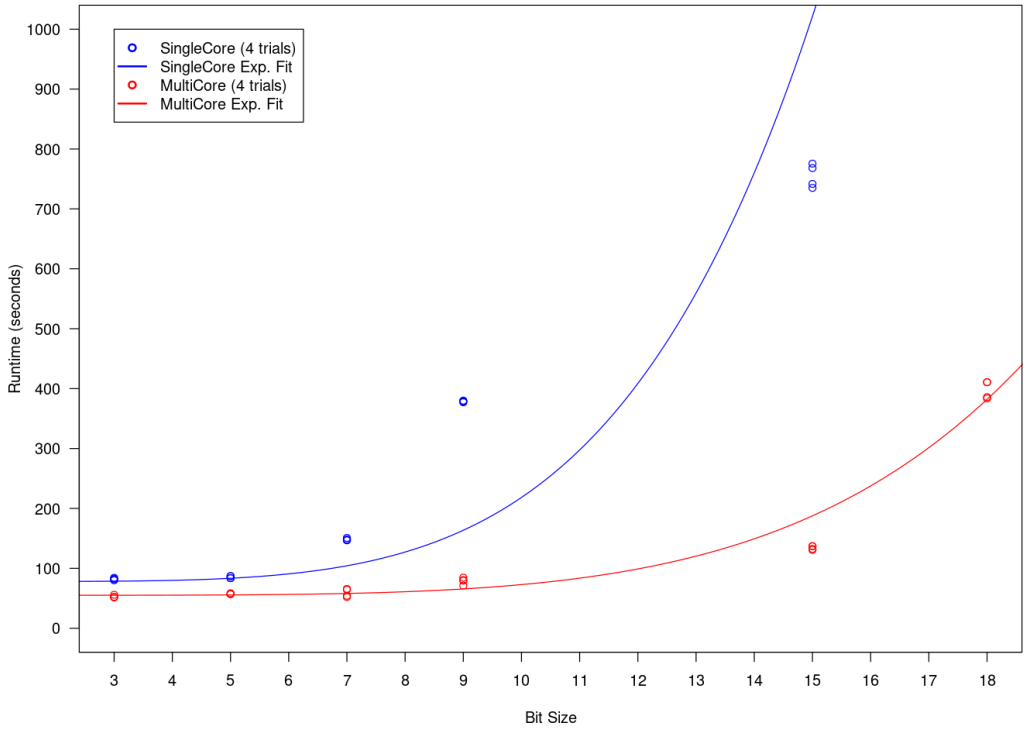

Figure 4.25: Single-Core and Multi-Core Allegro runtimes displayed as a function of bit-size against run time fitted with an exponential fit curve.

The multi-core runtimes shown in Table 4.5 are plotted above in Figure 4.25 alongside the single-core runtimes summarized in Table 4.4 and tabulated verbosely in the Appendix[76] with exponential fit curves[77]. Individual chromosome runtimes for the multi-core analyses were not recorded due to the scope of the parallel evaluation. In the case of multi-core, runtimes remain reasonably linear from 3-bit to 15-bits where a sharp rise in runtime occurs 18-bits due to system resources requested by the active Allegro instances having to compete for memory. This lends a sharper exponential rise in higher-bit multi-core runs compared to single-core runs, despite a lower and flatter outset.

[76] See footnote 75.

[77] Curves were derived by fitting $log(runtime)$ and then iteratively correcting the coefficients with a nonlinear least squares model: $y = 78 + 0.0028x^{4.7}$ (single), $y = 55 + 0.00021x^{4.95}$ (multi).

Due to RAM constraints, the pedigrees larger than 18-bits were not able utilize parallelization effectively, only being able to process a single chromosome at a time (i.e. single-core) and so they were removed from the analysis.

For the 29-bit pedigree, Simwalk performed the genomewide analysis and then Allegro reconfirmed the chromosome 16 peak. Allegro took 315615.20 seconds (3.7 days) to process chromosome 16 alone. A large portion of the processing at that point was memory paging in order to preserve all the LOD calculations being made. Simwalk writes straight to disk and has an extremely small RAM usage globally, so despite being the traditionally slower program (due to the sliding-window approach creating numerous overlaps), it did not suffer from paging bottlenecks. Due to the Simwalk parallelization script not being chromosome-specific (i.e. it executes any window-processing job it is dispatched), only the genome-wide runtime was recorded at 682117.03 seconds ($\sim$ 7.9 days).

4.3 Haplotype Resolution Comparisons

We will look at the haplotypes of three large pedigrees with the following penetrance models: Autosomal Dominant, Autosomal Recessive, X-Dominant.

All the pedigrees viewed in the Results so far were produced by HaploPainter, and here we will examine some of the same pedigrees now rendered by HaploHTML5. Haplotype rendering and resolution between the two programs will also be compared.

4.3.1 23-bit Autosomal Dominant

Family 109 in the figure below consists of many members and so it was prudent to compare only the affecteds at a given locus in order to prioritize the haplotypes on screen. The colouring of the haploblocks is always semi-random so it is the position and lengths of the blocks themselves that we are most concerned with.

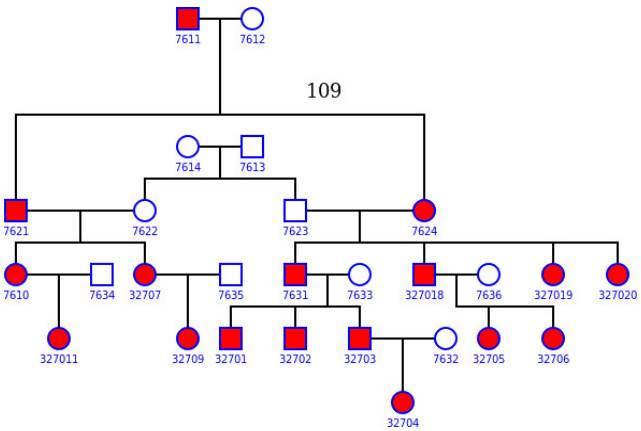

Figure 4.26: HaploHTML5 default rendering of pedigree 109 from Allegro haplotype input file. This is the 23-bit pedigree shown earlier in Figure 4.10 or page 170. Since both founder couples (7611 and 7612, 7613 and 7614) were at the same grid 'level', a line overlap was inevitable and so 7611 and 7612 were raised to better distinguish childlines.

Figure 4.27 shows a side-by-side comparison of the pedigree given in Figure 4.26. The top image shows the HaploPainter rendition[78], and the bottom the HaploHTML5 render.

The first red arrow marks a point between rs7693827 and rs11735648 which refers to a recombination in both haploblock renditions where the founder allele switches to a red block HaploPainter which is equivalent to the green block in HaploHTML5. Both blocks persist fully in the right allele of 32707, 32709, 7610, and 327011. The red arrow at point 3 shows agreability too between the applications, though HaploHTML5 appears to have selected two very similar looking colors for the pink block just above point 3, which looks identical to the right allele of 32704 but has a different set of genotypes. Debugging the allele shows that HaploHTML5 does assign them different color groups, but the groups themselves have high colour homology due to an over abundance of founder alleles (9 founders, 18 alleles) and a limited selection of colors.

[78] manually modified to be horizontally aligned since it normally only produces generation-based vertical plots.

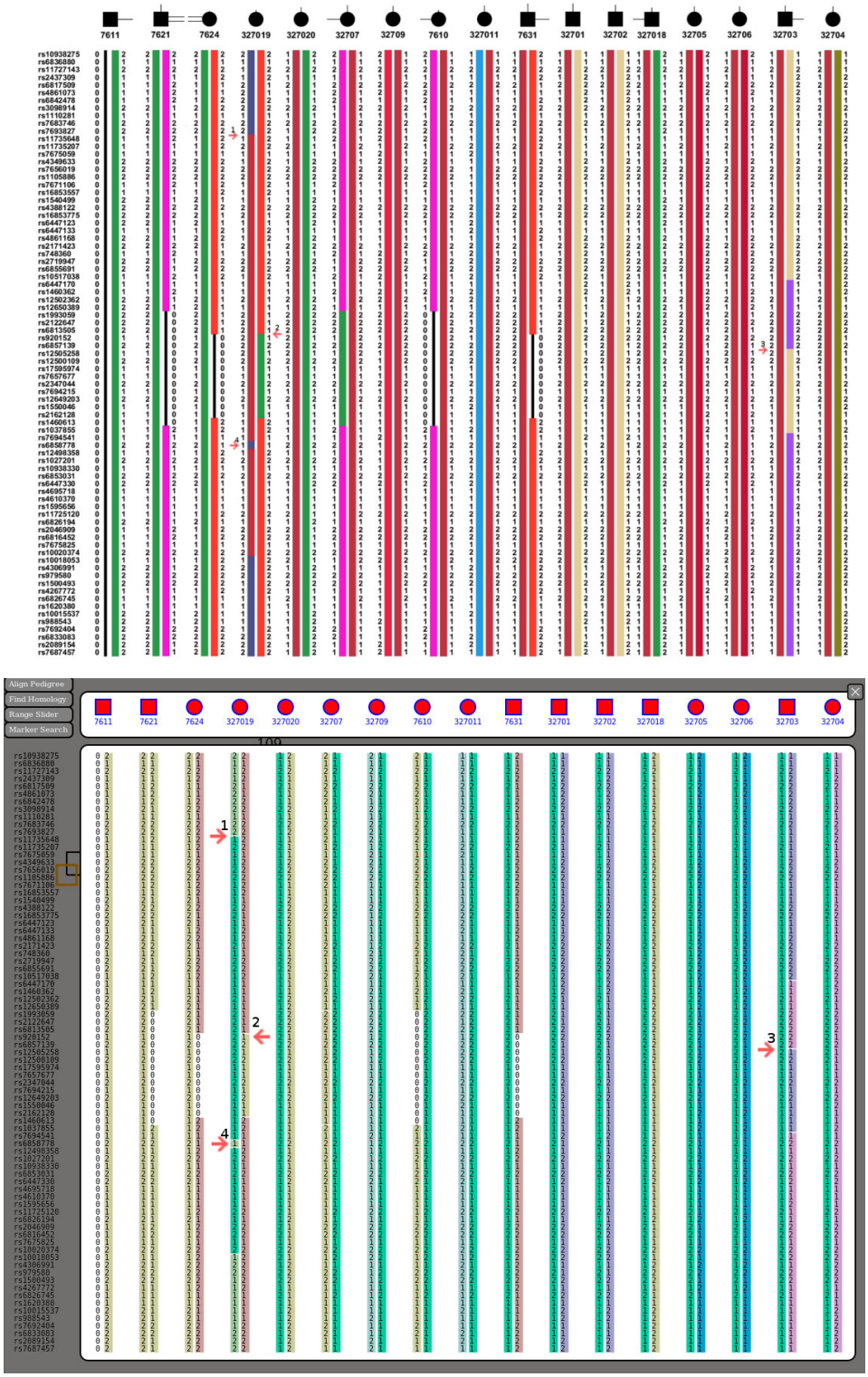

Figure 4.27: Haplotypes of family 109. HaploPainter modified to display affecteds inline (Top). HaploHTML5 default comparison view showing affecteds only (Bottom). Four noteable points of comparison points are highlighted with red arrows.

The red arrow at point 4 shows a genotyping error that is correctly picked up upon by both programs and assigned the first available group outside of the block it resides in.

In terms of design and presentation, HaploPainter prefers to present its haplotypes in a family-tree structure resorting in more manual methods of aligning the individuals of the pedigree for haplotype inspection. HaploPainter also places the text of the haplotype outside of the coloured block. HaploHTML5 presents the text within the block itself in order to preserve horizontal space, and as a result must use lighter shades of colours to improve the readability of the black text.

4.3.2 29-bit Autosomal Recessive

Here we have the three inbreeding loops that we encountered previously in the section, with consanguinity correctly detected for all the couples in the second to last generation. There is only one founder couple, but there is inter-generational marrying that makes it impossible for lines to not overlap (7612 and 7611 for example).

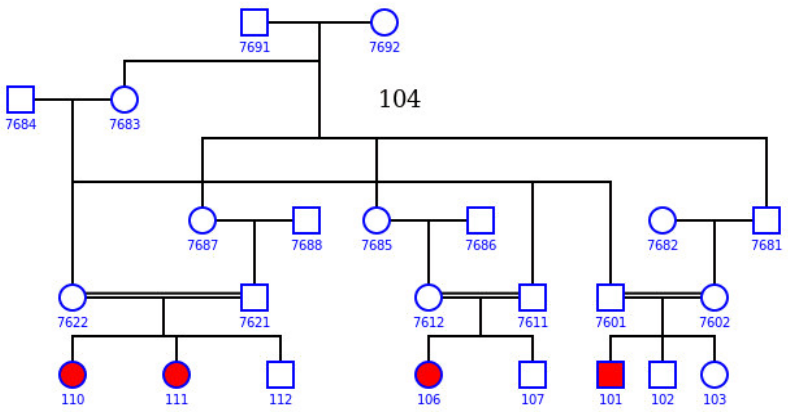

Figure 4.28: HaploHTML5 rendering of family 104, of the 29-bit pedigree originally shown in Figure 4.15 on page 175.

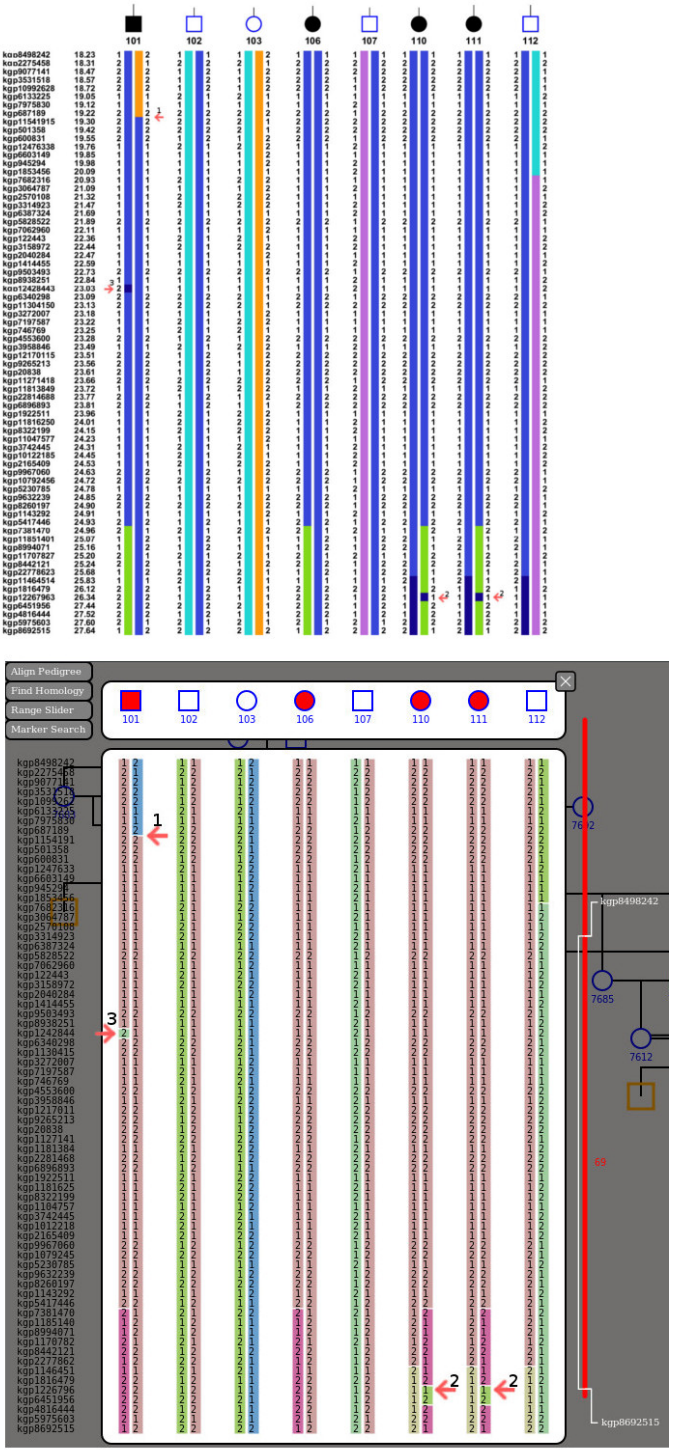

Figure 4.29: Family 104 rendered in HaploPainter and then modified for horizontal alignment (Top) and HaploHTML5 (Bottom).

There are only 4 affected individuals in this pedigree, so we can now include some unaffecteds too (the last generation in this case). Here we once again see good accordance between HaploPainter and HaploHTML5 for the genotyping error at point 3, and the recombination at point 1.

However there is a slight difference at the point 2 location(s) where HaploHTML5 extends the small block at kgp12267963 to two marker loci and not the single locus given by HaploPainter. Both configurations are valid because the purple and green block (in the case of HaploHTML5) both share a '2' allele index at that locus. However, HaploPainter does not maximise the block and makes it appear more as a genotyping error or artefact instead of the small haploblock as HaploHTML5 does. This can be easily rectified by adjusting the *minimum stretch* parameter for the A* path finding algorithm such that it does not consider blocks less spanning less than or equal to two markers to be actual blocks (and thus does not try to maximise it), but this places an unfair constraint on the data because single-locus haploblocks may be valid for many-generational pedigrees with sparse markers.

A better compromise would be to allow the inclusion of marker map data (such as is read in by HaploPainter) so that the algorithm can determine whether the inter-marker distances surrounding a marker is great enough to warrant a haploblock on its own.

Figure 4.30 below compares the homology tools used by the two applications. HaploPainter was adapated to take in **messner** files and marker pairs, where regions of homology could be detected using our **haploregion.py** script. Messner regions are boxed in a blue regions, and regions of homology are surrounded by red boxes where in this case the affecteds are all homozygous for the regions shown, with the extra condition that the unaffecteds are not also similarly homozygous. HaploHTML5 performs all homology detection within the browser, but uses a different approach where it assumes that homology detection is not so binary and assigns a "homology score" at each locus which is overlayed against the haplotypes and marker slider. HaploPainter is much clearer in showing homology in this instance, but HaploHTML5 allows forr some leeway for erroneous genotypes that might otherwise obstruct a binary homology detection.

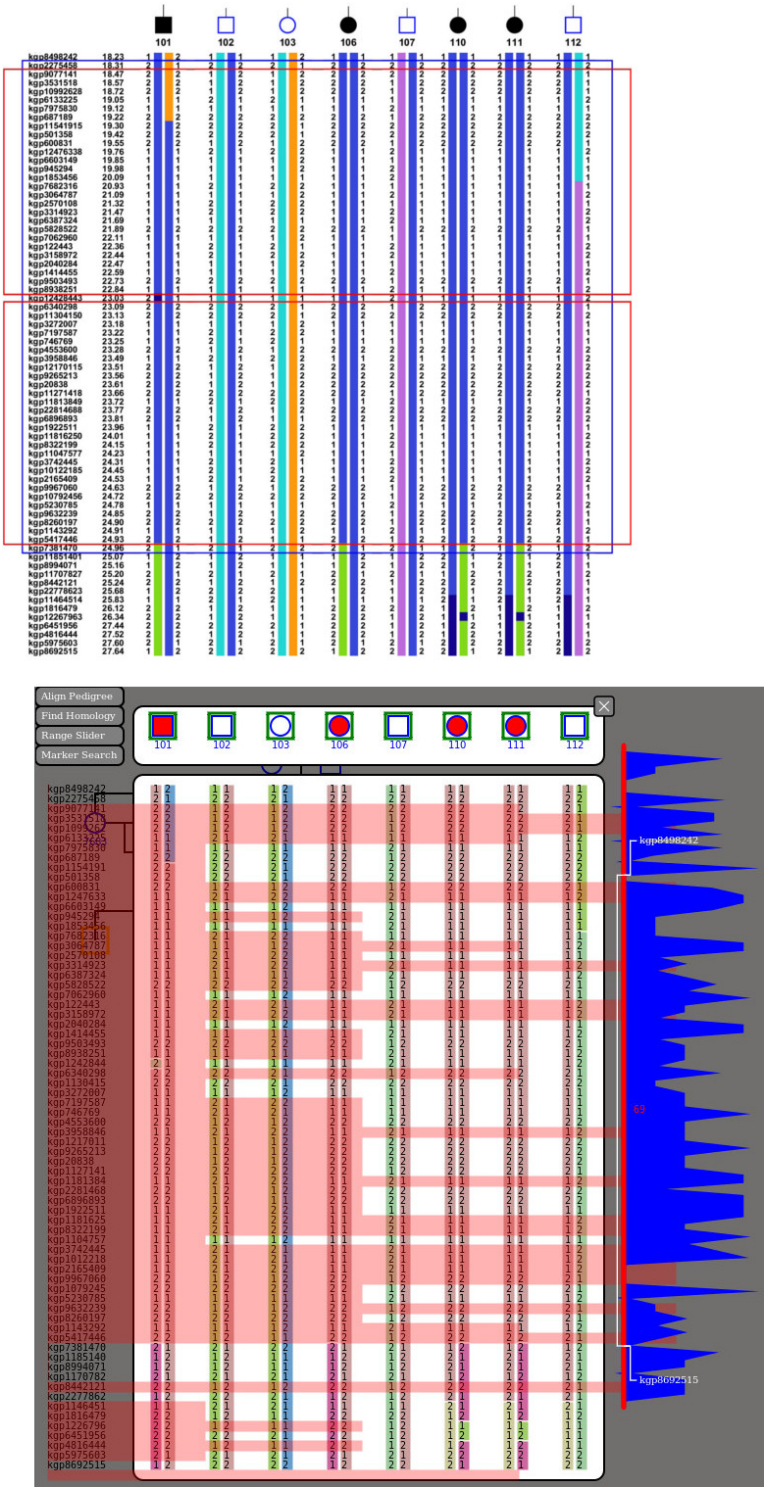

Figure 4.30: Homology Tool comparison of HaploPainter via **HaploPainterRFH** modification script (Top), and HaploHTML5 Homology Mode (Bottom).

4.3.3 15-bit X-Linked Dominant

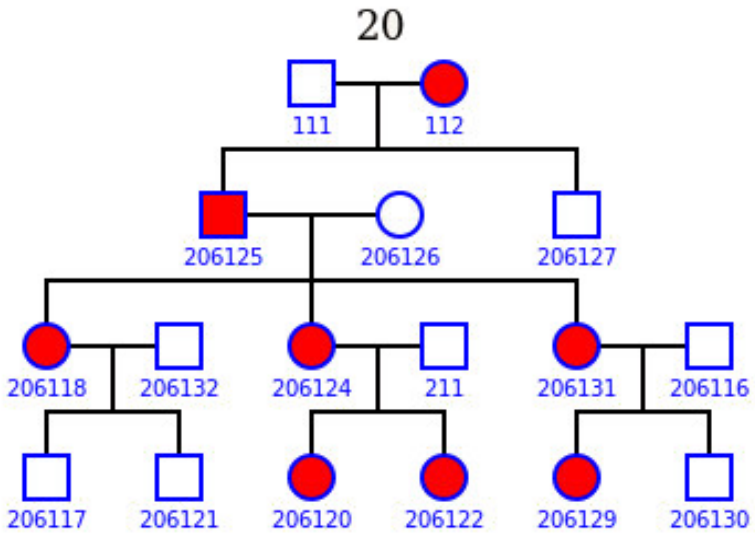

Figure 4.31: X-Linked Pedigree, 17 members of which 8 are affected individuals.

We once again take a look at our X-linked Dominant pedigree mentioned previously on page 166, where we can examine the haplotypes of all individuals in the pedigree.

The figures below provide three ranges of which to examine the X chromosome where recombination errors were observed within HaploPainter. HaploHTML5 sets the correct penetrance model and provides no such recombinations, often providing complete non-recombinants for the loci considered.

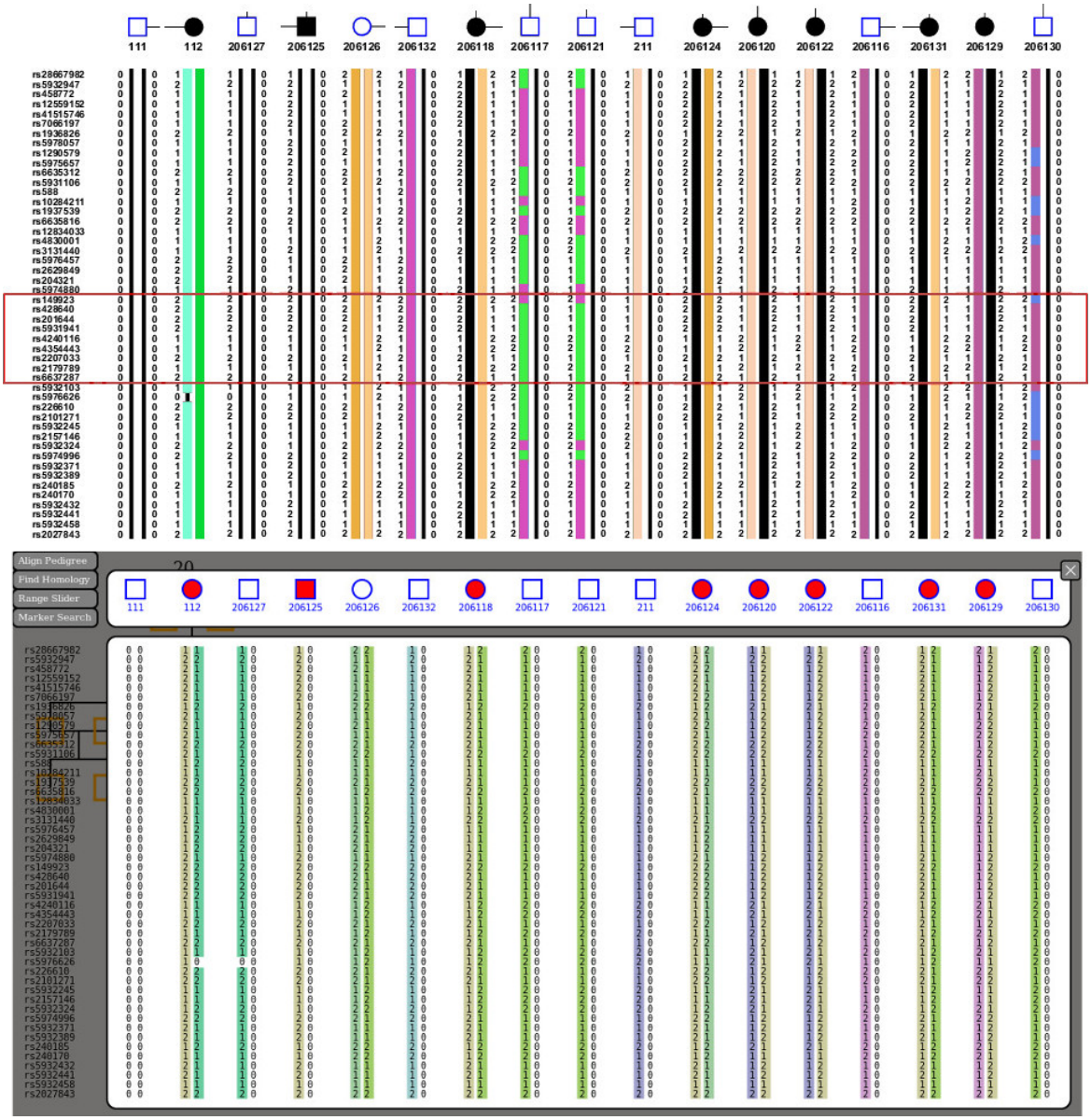

Figure 4.32: Starting telomeric P-arm region of chrX spanning 60 markers. (Top) HaploPainter recombination errors caused by bad X-inheritance. (Bottom) HaploHTML5 complete haploblocks due to correctly parsing the genotypes.

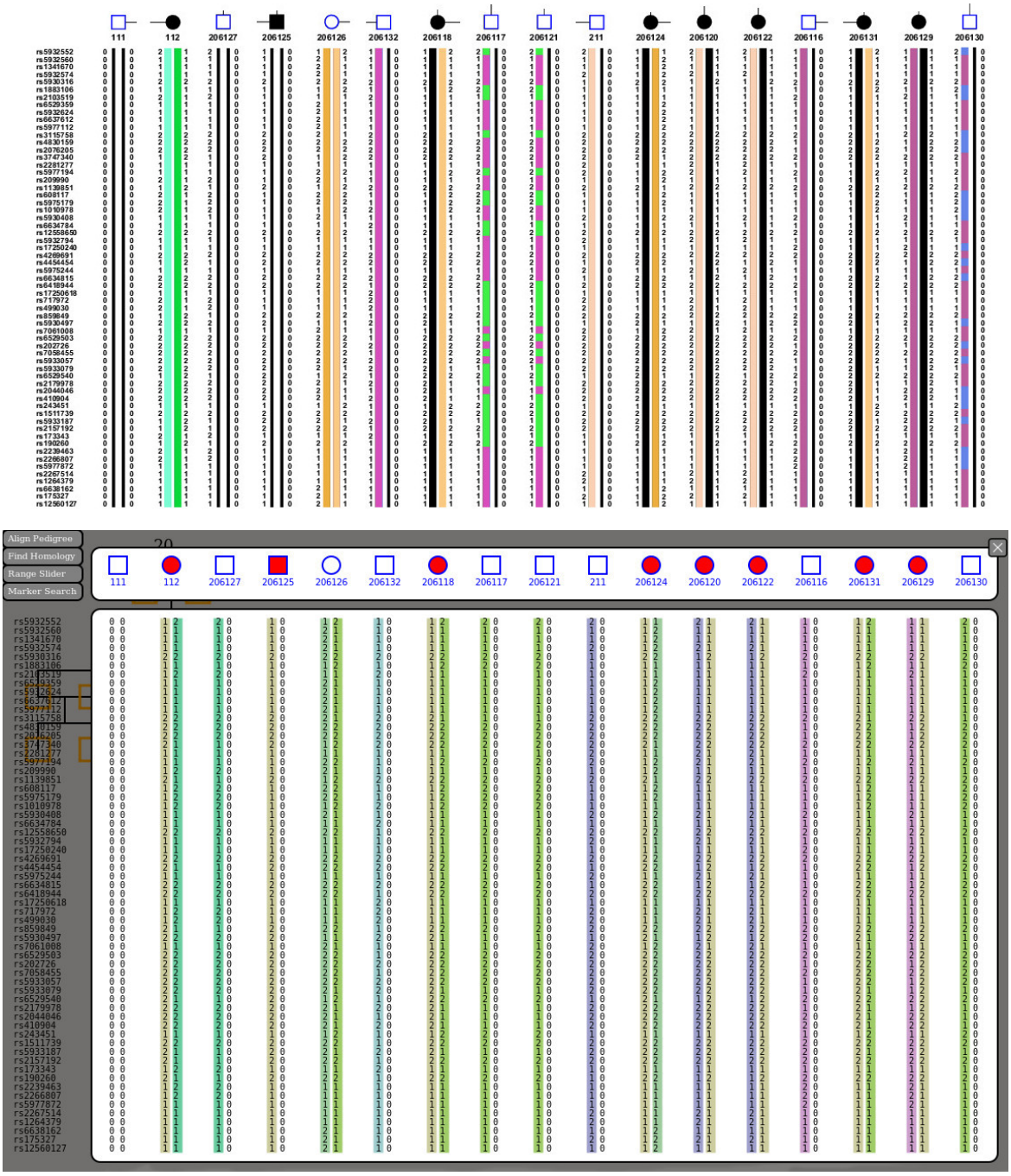

Figure 4.33: Middle region of chrX spanning 72 markers. HaploPainter (Top) and HaploHTML5 (Bottom).

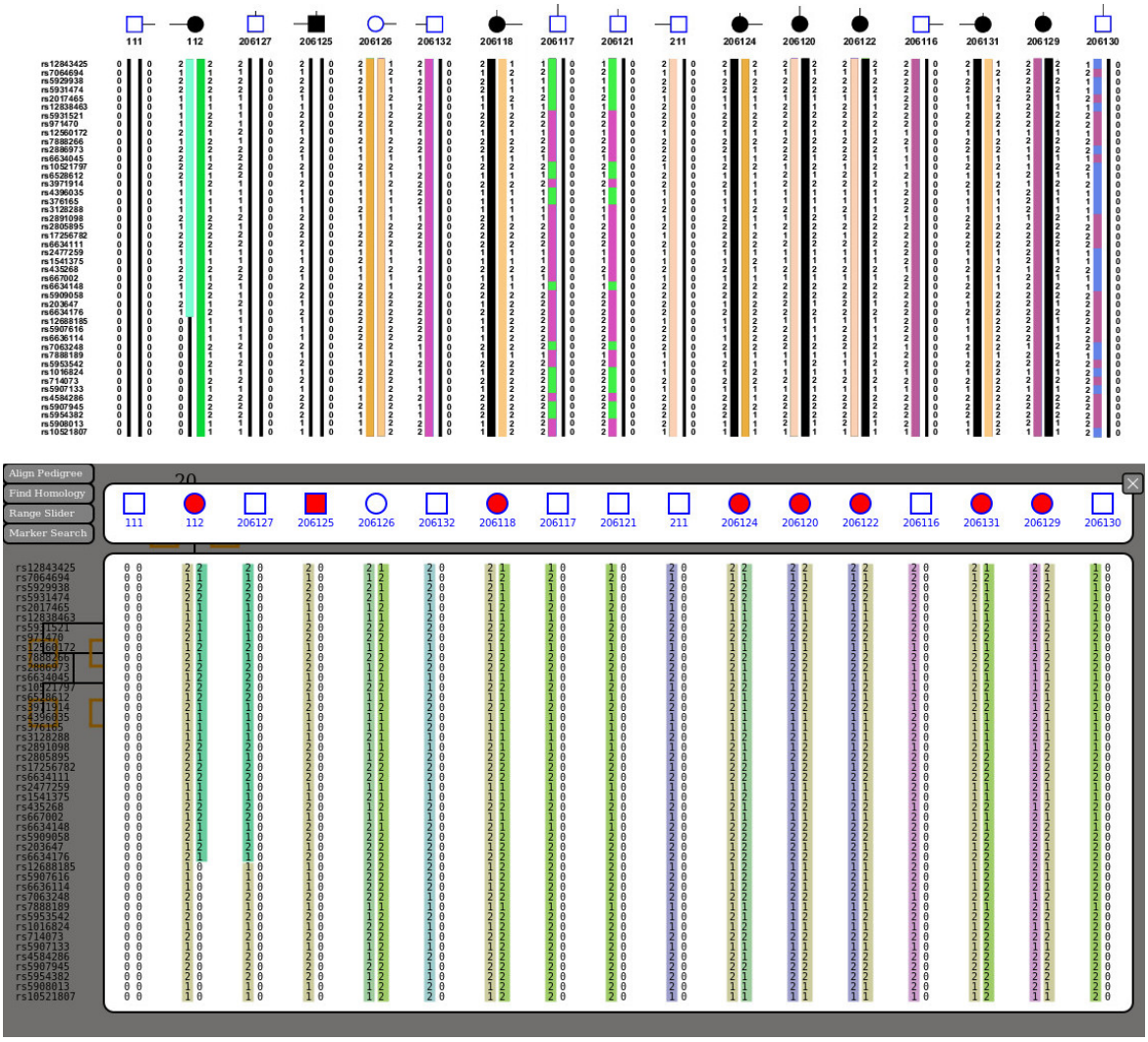

Figure 4.34: Finishing telomeric Q-arm region of chrX spanning 56 markers. HaploPainter (Top) and HaploHTML5 (Bottom).

4.3.4 Performance

In order to gain some understanding in how practical the A* best-first algorithm was on common hardware, we utilized Equation 3.2 on page 121 to determine the actual time it took to process a single chromosome upon different platforms.

The test itself involved the following initialization script:

```
function loadPedFromStorage(){
    document.getElementById("buttons").style.display = 'none'

    var ped_data = localStorage.getItem( localStor.data_save ),
        ped_type = localStorage.getItem( localStor.data_type );

    var pi = new ProcessInput(ped_data, ped_type);
    init();
    delete pi;
}

function init(){
    connectAllIndividuals();
    populateGrids_and_UniqueObjs();

    determinePedigreeType();
    graphInitPos(nodeSize + 10, grid_rezY);

    assignHGroups();
    washMarkerMap();

    populateIndexDataList();

    startSelectionMode();
    for (var key in selection_items){
        var fid_id = key.split('_'),
            fid = fid_id[0],
            id = fid_id[1];

        var item = selection_items[key];

        if (id >= 6 && id <=10){
            item.box.fire('click')
        }
    }
    launchHaplomode();
}

var start_time = performance.now()
loadPedFromStorage();
var end_time = performance.now()

alert("Time :", end_time - start_time);
```

Listing 4.1: Javascript initialization to automatically load pedigree from storage, select all members, and view their haplotypes.

This performs the standard HaploHTML5 functions in the order given below:

1. *loadPedFromStorage* loads a single 9-bit pedigree of 9 members[79] for a 3,561 marker analysis (chromosome 3).
2. *connectAllIndividuals* connects the pedigree.
3. *populateGrids_and_UniqueObjs* populates all drawable objects.

[79] Scenario 10 of the family shown in Figure 4.3 on page 159

4. *determinePedigreeType* determines penetrance and X-linkage.
5. *assignHGroups* creates haplogroups.
6. *washMarkerMap* performs pointer cleanup.
7. *populateIndexDataList* generates a marker list for easy scrolling.
8. *startSelectionMode* initializes selection mode.
9. *item.box.fire('click')* simulates the user clicking on individuals.
10. *launchHaplomode* opens the HaploView for haplotype analysis.

The *alert* function was used instead of the standard *console.log* logging tool because many mobile browsers do not show logging output.

Table 4.6 below shows the runtimes of the above script for each of the two main ECMAScript engines (JavascriptCore, SpiderMonkey) on three different hardware platforms: Desktop (x64, 4GB dedicated graphics, 3.2GHz x 2 CPU, 8GB RAM), Laptop (x86, embedded graphics, 1.67GHz x 2, 8GB RAM), and Mobile (armv7, embedded graphics, 1.16GHz x 2, 1GB RAM).

Hardware and Browser(OS, Engine)		**1**	**2**	**3**	**4**	**Mean**
Desktop	Chromium	0.051	0.042	0.014	0.047	0.038
Laptop	(Windows 8.1, WebKit, V8)	0.185	0.252	0.214	0.210	0.215
Desktop	Firefox	0.023	0.019	0.080	0.063	0.046
Laptop	(Windows 8.1, Gecko, SpiderMonkey)	0.371	0.482	0.472	0.268	0.398
Desktop	Chromium	0.055	0.013	0.025	0.032	0.031
Laptop	(Arch Linux, WebKit, V8)	0.157	0.224	0.229	0.243	0.213
Desktop	Firefox	0.060	0.039	0.059	0.042	0.050
Laptop	(Arch Linux, Gecko, SpiderMonkey)	0.333	0.456	0.508	0.231	0.382
Mobile	WebPirate (Jolla, WebKit, V8)	0.816	1.233	1.095	1.136	1.070
Mobile	Sailfish (Jolla, Gecko, EmbedLite)	1.941	2.014	1.724	2.277	1.989

Table 4.6: Four time trials (seconds) upon Desktop, Laptop, and Mobile hardware, with different ECMAScript engines (V8, SpiderMonkey, EmbedLite).

Due to the various different hardware components each specialized to different tasks, it makes no sense to chart the data in Table 4.6 since the difference between each platform is not measureably distinct enough to form a Y-axis. We can however summarize each hardware device in terms of runtime: Desktop (0.041 s), Laptop (0.302 s), and Mobile (1.523 s).

5. Discussion

5.1 Pipeline Evaluation

Small analyses such as the 3-bit and 5-bit pedigrees examined in the previous chapter are generally not very informative for linkage by themselves. For the 3-bit pedigree we observed multiple low LOD score flat peaks that did not exclude many regions, and for the 5-bit pedigree we saw fewer flat peaks with slightly higher LOD scores and still not that many regions of exclusion.

That is not to say small pedigrees are unusable; weak studies can lend strength to larger analyses where the linkage region is one of a multitude of peaks which can be narrowed down by the extra regions of exclusion given by a smaller analysis. Similarly, large analyses are of great assistance to smaller ones, where we witnessed a 29-bit pedigree lend power to the haplotype analysis of two 3-bit pedigrees in Figure 4.16 on page 175.

Pedigree errors such as those indicated by GRR, gender, or Mendelian errors are typically not hindered by the size of analysis[80] and problems are detected at the trio level. Often, small pedigrees are later expanded into larger ones by genotyping the rest of the family in order to improve the LOD score. By catching these small trio errors early on, a researcher can make a more informed decision on whether to pursue the family for further linkage. As we saw in the pedigree in Figure 4.4 on page 162, parents sometimes have to falsify their children's genealogy for personal and legal reasons, and this can be cause for many a headache for the researcher who has to reconcile the true pedigree.

Medium size analyses (7-bit to 18-bit) are usually in the vicinity where such scenarios occur because the linkage peaks are reasonably high (> 1) and few in number (< 6), meaning that there is more room for a researcher to "play" with

[80] except for **GRR** runtimes, where window polling increases with the number of total genotypes.

analysis by running more variations upon the input sets to maximize the analysis. Again, it should be stated that maximizing an analysis by small modifications to the pedigree and penetrance model are not necessarily a true representation of the phenotype, but only what is expected or allowed according to the classical ideals of genetics (e.g. parent-offspring genotype variance and Mendelian inheritance).

For large analyses ($\geq$ 19-bit) fewer analyses are possible due to time constraints, where Table 4.4 on page 181 shows that the timeframe shifts from a under an hour ($\sim$ 2800 seconds $\sim$ 47 minutes) to a few days ($\sim$ 945000 seconds $\sim$ 11 days).

Hardware limitations (namely RAM) also make **Allegro** largely unuseable on a genome-wide scale because the over 19-bit modification switches off the CUDD capability which would otherwise minimize the number of repeated calculations employed in the linkage. It is better suited to analysing individual chromosomes, preferably higher numbered chromosomes since the lower chromosomes begin to scale exponentially with runtime as shown in Figure 4.24 on page 182. **Simwalk** is of more use on the genome-wide level, but risks not being able to accurately determine X-linked analyses.

5.1.1 Pipeline Summary

The filtering portion of the pipeline seems to work extremely well for all types of pedigrees and penetrances, and is the main strength of the pipeline. The combination of Mendelian errors, GRR relations, gender validation, and unlikely genotypes form easily understandable plots of the quality of the data. Errors are pinpointed to specific families and specific individual(-trios), and the quick runtime of the filtering core ($<$ 5 minutes) means that changes to the pedigree can be tested and evaluated very easily.

In relation to the rest of the pipeline, analyses are typically brute-force driven in order to maximize the LOD score as permitted by the reasonable runtimes for all pedigrees under the 19-bit limit. Over the limit requires more thought before an

analysis is run, often requiring a pedigree to be split into smaller families[81] to form a smaller-bit combined analysis that may provide more visual cues on which specific chromosomes to explore when running the larger pedigree, rather than risk the time cost of running a full genome analysis via Simwalk[82].

5.2 HaploHTML5 Evaluation

5.2.1 HaploBlock Resolution

As shown in Figure 4.29 and Figure 4.27 in Section 4.3 on page 185 HaploHTML5 resolves autosomal haplotypes just as well as HaploPainter, with the added benefit of comparing haplotypes of any selected individuals in a side-by-side view.

The 15-bit X-linked pedigree shown in Figures {4.32, 4.33, 4.34} on pages 193 to 195 shows how the chromosome X limitation inherent in **HaploPainter** is overcome in HaploHTML5, where the highly improbably blocks shown in HaploPainter are resolved correctly using the appropriate penetrance model.

5.2.2 UI Operability

The PedCreate view is intuitive and simple, with two types of relations of possible (mate-mate, and parent-offspring) with valid connections being highlighted to the user via red or white anchor points[83]. Individuals are trivially added, modified, deleted, and can be moved around with their various relationship lines updating their positions (and their siblings and/or mates lines) accordingly. An invidiual's name can also be specified, but it is not a requirement. The resultant pedigree can be exported to pedfile, with optional positional meta tags which can be reread back into the application either by manually selecting the exported file, or by saving the pedigree into local storage and reloading it upon startup.

[81] especially for consanguinous pedigrees.
[82] Or Allegro, if the pedigree is X-linked and time is not a crucial issue.
[83] See Figure 3.11 on page 143 as an example.

The SelectionView mode as shown in Figure 3.13 on page 145 allows for multi-family selection which simplifies the comparison process, especially when comparing the disease locus (found via linkage) within combined analyses. The DOS view outlined in page 145 is in intriguing concept that aims to simplify generational representation upon the selected indivduals under analysis, but ultimately complicates it. It may be better to simply represent the pedigrees of all selected individuals via a small mipmap thumbnail in the corner of the screen.

The Homology tools mode is a useful tool for scoring regions of sequence homology under heterozygous, homozygous, and compound heterozygous models. These scores are plotted vertically against the marker scale (see Figure 3.16 on page 149) for quick scrolling, and can also be output to a text file for later inspection. However, the mode can be confusing to interpret due to the non-binary nature in which it scores homology (see page 150) and the modified **HaploPainterRFH** script is better in highlighting regions of homology as shown in Figure 4.30 on page 191. It would be more beneficial for the end-user if the homology tools merely outlined the regions that met that maximum homology score for the scenario (as in the case of HaploPainterRFH) instead of trying to overlay all the data at once.

5.2.3 Performance

Performance testing of the A* best-first algorithm upon a single set of test data gave negligible runtimes on Desktop (0.041 s) and Laptop (0.302 s). Mobile (1.523 s) produced a minor noticeable delay, but still produced the correct haplotypes, which was unexpected given the lightweight nature of its Javascript engine (EmbedLite) not necessarily incorporating all the features of ECMCAScript6 specification. The overall differences seem to be a factor of 10 from Mobile $\rightarrow$ Laptop $\rightarrow$ Desktop, but the differences between OS and browser upon the same hardware appears to be function specific as expected [48], with minor speed increases in Chromium over Firefox.

HaploPainter runs on Perl and relies upon Tk, Cairo, Sort Naturally, and DBI Perl dependencies to be present in the system. The more dependencies a program has, the less likely it is to be ported to other platforms successfully since it's dependencies must be ported first. Perl itself also is not very common on ARM devices[84]. However, the only dependency that HaploHTML5 requires is a web-browser with Javascript capability, immediately making it much more portable than HaploPainter.

5.2.4 Distribution

The scripting nature of Javascript virtually enforces an open source distribution model upon all web applications created under it. However, language notwithstanding, HaploHTML5 was always going to be released as FOSS, and it was only the enforcement of this principle that prompted the exploration of the obfuscation methods outlined on page 151.

Though the static source files were indeed concealed from the user through said methods, runtime debugging would still be able to recapitulate the body of most functions, albeit the discovery of *all* functions would involve careful (recursive) manual inspection.

The obfuscation methods were therefore ultimately removed from the release version of the application, and it was licensed under GPLv3[85].

5.3 Future Work

The pipeline and application are in a useable state, and both will achieve the tasks they are designed for: performing quality linkage on large data, and rendering haplotypes in-browser. However, there are a few improvements that could be incorporated to make them more intuitive to use at the end-user level, and also adjustments that could be made at the core level to increase performance.

[84]The Jolla mobile device used in the Results chapter runs an OS aimed at Linux and shell enthusiasts; Perl did exist upon it, however none of the Perl dependencies had been ported to it and so HaploPainter could not run on it.

[85]See Licensing section in Appendix on page 237.

5.3.1 Pipeline improvements

i Fixing Installer Issues

The pipeline comes packaged with an installer that is designed to work on all Unix and Linux platforms; and so far it has been successfully upon Ubuntu, Debian, Arch, and Gentoo. The installation is not straightforward however, since every OS does not always use the same package manager as another. Ubuntu and Debian use Aptitude (or simply *apt*), Arch uses *pacman*, and Gentoo uses *portage*.

Their install syntax is very similar, but they are all separate package installers and must be treated as such. To ease this issue, all software packages are kept in a separate file which merely lists them so that they can be fed into each OS's respective package manager. Unfortunately, though there is generally good accordance of package names between different Linux OS's, there can be some minor discrepancies in their naming that can cause issues[86].

Another issue is installing the Perl dependencies (for HaploPainter). If HaploPainter will be depreciated in the future with the uptake of HaploHTML5, then this issue can be easily resolved by simply removing the Perl dependencies from the installer altogether. Otherwise, the problem arises when deciding which means of software distribution to install the Perl dependencies from: the OS package manager, or via **CPAN** [49] which installs Perl modules to user or root directories separate from the system.

The latter is the more compatible option, since CPAN exists on every system that supports Perl, but the packages from CPAN tend to be newer than those distributed by the system repositories and this may cause clashes.

[86] The switchover from Python2 to Python3 for example creates some very ambiguous instances where a package named *python-gobl* is not inherently clear on which version of Python the gobl library is for.

ii Custom Modular Component Ordering

The pipeline as outlined in Figure 2.1 on page 41 exists as two main modules: Filtering (comprising of three sub-modules), and Linkage, with pre-linkage and post-linkage operations.

Quick retesting usually involves modifying the main pipeline script to some degree in order to prematurely terminate the analysis after the Filtering stage. This is trivial for any technician familiar with Bash or Unix who can comment out the relevant portions, but may prove more difficult for those used to graphical interfaces.

A better method would be use a configuration text file that outlines what modules and submodules the pipeline should run before terminating. The default configuration would run all modules in their specific orders, but a custom user configuration could change the order of the modules and better tailor the analysis to their needs (e.g. maximising a pedigree requires only the Filtering module).

iii Automate Large Pedigree Handling

Pedigrees beyond the 19-bit limit are not handled by the default pipeline. Only the pre-linkage and Filtering components are handled and then the pipeline has to terminate in order for the custom scripts that can handle the large pedigrees can be called manually by the user.

This can be easily automated, since the *bit_score.py* script automatically determines the bit-size of any pedigree, and can act as a screening stage for whether the pipeline should run the vanilla Allegro binary, or the custom modified (over 19-bit) Allegro binary. A potential pitfall would be having to declare beforehand whether the analysis should be haplotypes or multi-point parametric driven, since the modified binary cannot do both, but this can be specified in a configuration text file mentioned in the previous subsection.

iv Fixing Allegro

The modified Allegro binary works by deactivating the CUDD library which minimizes the number of unique computations made in the LOD score calculations and vastly improves upon memory and disk resources. Allegro uses version 2.4.1 of the CUDD library, but a later version exists that may fix the problem.

Attempts to incorporate this library into Allegro, but certain function calls that Allegro depended upon were depreciated between versions and remapping them to newer ones proved problematic.

One other solution to this problem would be to rewrite Allegro from scratch. Allegro is written in C/C++ and is well commented in the source code. A more up-to-date rendition of the same program, using some of the newer features of the C++11 specification [50] may greatly benefit the program and even boost performance under the previous C++98 specification under which it was written and compiled.

v GRR Backend

The most problematic component of the pipeline is the graphical GRR interface, which is a Windows binary that does not take any commandline arguments[87]. As a result data cannot be simply fed into the application from the commandline where it performs all computations in the background, instead the application requires a display server to host its graphical interface where user-clicks are then simulated by the system in order to load in the data.

The requirements of GRR are immense; entire display libraries must be installed such as a X11 or Wayland display server, a desktop environment, display fonts, sounds, and all manner of other multimedia dependencies that come packaged with

[87] Despite a pleading email to the developers who promised to add such functionality in the future, only to then respond with silence.

operating systems that require desktop sessions to run. The application also requires **wine** to run because it was originally written for Windows.

A replacement Linux-based binary would be ideal at this stage, and prompting our own rewrite of GRR may be required.

vi Web UI

Each run comes equipped with an automatic readme generator so that each run has a log of what analysis transpired, from what input sources, at what date, with how many markers, and under what penetrance. The penetrance and number of markers are automatically detected from the surrounding input files, but everything else must be typed by the user.

A web front-end would facilitate in this greatly, where the user would simply upload their files to the webserver which would place them in a correctly named folder (see page 47) as specified by a project ID input field within the form, filter for informative markers as set by other options in the form, and would print the details into the readme file without the user having to manually specify anything.

The web interface could also give options for customizing the order of pipeline modules and write an appropriate configuration script.

This could be implemented in pure PHP [51] which can talk to the system and is capable of spawning off bash commands and reporting on their feedback in realtime. The images generated during the Filtering stage (pedigrees, Mendelian errors, GRR) could be displayed within the interface as and when they appear.

The linkage portion of the pipeline would have to be reported to the user via commandline output which would be fed into a textarea in the interface, but once complete the linkage plots could be displayed as tiled images or opened up as separate PDFs in a new tab.

The user would then be able to download their data as a compressed ZIP file as packaged using the normal **collect.sh** script from a button or link that would only become visible once the analysis is complete.

One issue that has been known to arise when implementing similar interfaces in other pipelines[88] is that the pipeline is spawned by the browser which makes the browser the parent process. Once the browser is closed (or a SIGHUP signal is detected) the system will attempt to terminate all child processes too which may kill an ongoing process.

To get around this, the **screen** utility is used to detach the child process from the parent but still maintain control later on. However getting realtime output out of a screen process is not straightforward, and requires taking repeated "snapshots" of the session window and reading the contents.

vii Deployment

There are three ways that the pipeline can be deployed:

1. **Install** - The pipeline is installed onto a Unix or Linux system via the installer script to work alongside the other files and programs on the OS.

2. **Live Environment** - the entire pipeline is placed into a bootable medium (DVD or USB) and all analyses are run in-house off the medium. Live environments are restricted in the notion that the operating system is essentially read-only, which has the benefit of greatly reducing any paging overhead from the OS itself, but means that all created files must be kept in memory. For small analyses this is not a problem, but larger pedigrees will prove a problem.

3. **Web hosting** - the pipeline is hosted from a dedicated server with all optimizations catered towards the specific hardware it runs upon. Users then interface with the pipeline remotely either through the **ssh** utility via the secure SSL protocol, or through a HTTP server to a web interface. Complete analyses are transferred through **scp**, **rsync**, or downloaded from the web interface via HTTP.

[88] Significant work has also been performed upon a High-Throughput *Sequence* Analysis Pipeline.

The first and third option are the most viable, albeit with the install stage requiring the user to take care of their own hardware which may be a large limiting factor in the overall uptake of pipeline usage due to heavy hardware requirements. The web hosting option is more approachable, but requires limiting the number of concurrent users accessing the pipeline at any time since it can only handle one large pedigree at a time, though numerous small pedigrees can be handled at any given time.

viii Effective Filesystem Management

The working directory should be seen as a temporary file system at best, since many linkage programs generate a substantial amount of temporary files during processing. Though modern file systems are excellent at keeping track of and recovering files (a task known as journaling), it is still good practice to "mount a scratch monkey" such that temporary working files are isolated away from archived ones in distinctly separate partitions.

It is preferable to isolate the two file systems on different disks altogether, such that a disk-wide read error on a given hard disk won't affect the operation of another. Most modern file systems have journaling enabled by default, which writes extra data to disk so that a recovery is possible in the event of a crash, but this becomes more problematic with Solid State Drives (SSDs) which have a limited number of read/write operations, and are prone to seizing should an operating system over-breach those limits.

This is not an improbable occurrence; operating systems routinely write temporary files to disk in order to manage RAM constraints in a process known as paging, where not immediately required portions of system memory can be sequestered into slower disk storage to free up space in the RAM for more higher priority tasks.

When a system begins to run low on RAM (as in the frequent case of extensive linkage analysis), a great deal of paging is performed to keep the linkage program in

memory, leading to a substantial amount of disk writes on a journaled file-system. Mechanical (spin) drives can handle these read/write requests robustly, but they are an order of magnitude slower than SSDs and significant bottlenecks can occur where the system has to wait for the disk to be ready in order to perform a block[89] read/write to it.

viii.1 Why Upgrading Disk Technology May Not Necessarily Lend a Performance Increase It is clear that SSDs offer no advantage at all in the use-case of frequent large temporary file activity, since though the operating system will rarely have to wait in order to perform a read/write, the limitations on the number of these operations shortens their lifetime considerably.

The best long-term compromise is to use mechanical disks with a filesystem without journaling; either an older filesystem where journaling was never implemented (though file size constraints may be enacted), or a newer filesystem where journaling is disabled upon initialization. This reduces the total number of read/write operations whilst still ensuring disk longevity.

Recoverability may be jeopardized, but the temporary nature of the files within the system imply that a re-run of the same analysis would not be resumable in any case; for it is the operating system that dispatches the temporary files to the linkage programs, and not the linkage program itself (with the notable exception of **Simwalk**).

RAID Configuration One extra caveat to reduce the number of operations upon a single disk *and* keep some level of redundancy is to distribute the load across several disk drives in a standard redundant array of independent disks (RAID); specifically a RAID-5 level setup. RAID relies on the method of storing contiguous data across several mediums (a concept known as striping), as well as ensuring the

[89] If multiple sequential operations are requested on same contiguous portion of a disk, then the operating system groups the individual requests under a large contiguous 'block' request, that performs one long read/write operation instead of several short ones.

data is correct and error-checkable without having to scan the entire block of data (a concept known as parity). The failure of one disk drive would not be enough to impede the setup and an ongoing analysis could be reconstituted from the other remaining disks, though a minimum of three identically sized disks is required.

These features lend well to a large temporary filesystem, since the storage capacities of multiple disks can now be combined to store a great amount of data, as well as the number of operations on any one disk being reduced by a factor of n (for n disks in the setup). The setup will still have the same read/write speeds as an ordinary disk, but there will not be any added complexity since the operating system will detect it as a single volume.

5.3.2 HaploHTML5 Improvements

Other than the general bugfixes, as well as rewriting the DOS method mentioned previously (page 202), there are numerous improvements that could be made to the application in order to boost its popularity amongst geneticists.

i Founder Colour Group Representation

One current problem with the application is the limited colour space for the founder allele groups to be allocated to. Under the HSV model with a hue scored from 0% to 100% differences in colour become harder to distinguish below the 15% range as shown in Figure 5.1.

For f founders, there are $2f$ unique colours that must be assigned to their respective alleles, meaning that as soon as the pedigree begins to have more than 3 founders[90], the colour groups become more ambiguous.

One solution would be to change the the Value component of the HSV model to take on darker and/or lighter components, which would double and/or triple the number of available colours, but would require changing the colour of the genotypes text which reside on top of the coloured block.

[90] $\frac{100}{15} \sim 7$ unique colour groups for all founder alleles ~ 3 founders.

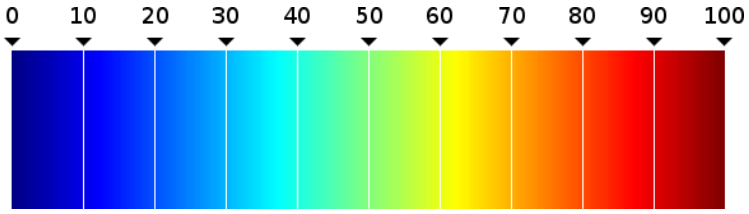

Figure 5.1: Hue Scale (%). Distinct colours are more visible at 15% increments.

It is possible to move the genotype text out next to the block as represented in HaploPainter, but this would reduce the amount of available horizontal space since more width would have to allocated per individual.

Another solution would be to regenerate the founder allele colour groups to only those currently under selection, but introduces the issue of inconsistent colouring across multiple runs on the same data.

ii Analysis Sharing and Privacy

The application currently operates in-browser via either local or web deployment, and analyses are restricted to a single user. In the interests of scientific collaboration, it is likely that the end-user would want to share their analysis with another who may be working on the same project.

In order to achieve this, there are two main avenues of development to explore:

1. Restrict application deployment to cloud-based services, where analyses are stored in a mySQL[91] database and are issued private login keys to restrict access to the data to only permitted users.

   Access can then be metered using established social network authentication services such as Google, Facebook, Microsoft, Twitter, and OpenID via the OAuth2 [53] protocol.

   The issue of patient confidentiality would still remain an issue however, since shared analyses would store patient data remotely on the site host, and could

[91] The mariaDB open source fork to be specific, since mySQL was acquired by Oracle [52].

be subject to vulnerabilities if the host is compromised. To complicate matters, the PedCreate View allows for the option of applying actual names to individual IDs, which is not an issue if the application uses locally-stored data, but is a major concern if such names are stored in a remote server.

A solution to this would be to store two copies of the pedigree: one unmodified within local storage, and the other in remote storage with patient names stripped from the pedigree. The user could then share their analysis with another (with names omitted) over the internet, and still preserve names when working locally on their own machine. However, problems will quickly arise if the other user wishes to modify the pedigree, in which case the locally-stored names become moot, where it may then be beneficial to restrict modification altogether and only allow read-only sharing to solve this problem.

Another concern is that stripping names from the pedigrees may not be enough to preserve patient confidentiality, since the pedigree structure itself may be enough to identify a family, especially if the phenotype is rare.

2. Analyses are encoded within the URL, where nothing is actually stored in the remote database. The pedigree is compressed via LZMA [54] in order to keep the number of characters within the recommended 2000 character limit for URLs [55] and encoded under base64 where the string is then affixed to a GET specifier the URL to be shared.

   Once a recipient clicks on the URL, the GET variable is decoded and uncompressed, where it is then loaded as a regular pedigree file on the recipient's browser.

   Base64 encoding is supported by the HTML5 specification [56], and the lz-string library exists [57] to perform LZMA compression and decompression within Javascript.

   Though data is never stored remotely, this approach does run the risk of anyone with the link being able to view the data.

A combination of both approaches would likely need to be implemented to achieve both privacy and security.

iii Framework Upgrade

Eric Dowell is the author of KineticJS but has since moved onto another project after he completed KineticJS. KineticJS was deemed "complete" in 2014 [58] and has not been worked upon since. This was good news for developers who needed a stable platform to work with, but came with the unspoken agreement that applications developed under it would not be future maintained.

Dowell has since moved onto ConcreteJS [59] which is a more general-purpose 2D library from manipulating shapes and is vastly optimized such that the entire codebase (including documentation) consists of no more than 600 lines. The project is still in the early Alpha phase (v0.1.0), but looks very promising. It shares almost the same syntax as KineticJS and will be non-problematic to implement.

Bibliography

[1] U. Consortium *et al.*, "The universal protein resource (uniprot)," *Nucleic acids research*, vol. 36, no. suppl 1, pp. D190–D195, 2008.

[2] J. Haldane, "The combination of linkage values and the calculation of distances between the loci of linked factors," *J Genet*, vol. 8, no. 29, pp. 299–309, 1919.

[3] S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin, "dbsnp: the ncbi database of genetic variation," *Nucleic acids research*, vol. 29, no. 1, pp. 308–311, 2001.

[4] A. F. Saeed, R. Wang, and S. Wang, "Microsatellites in Pursuit of Microbial Genome Evolution," *Frontiers in Microbiology*, vol. 6, p. 1462, 2015. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4700210/

[5] R. C. Elston, *Elston-Stewart Algorithm*. John Wiley & Sons, Ltd, 2005. [Online]. Available: http://dx.doi.org/10.1002/0470011815.b2a05018

[6] J. R. O'Connell and D. E. Weeks, "The vitesse algorithm for rapid exact multilocus linkage analysis via genotype set–recoding and fuzzy inheritance," *Nature genetics*, vol. 11, no. 4, pp. 402–408, 1995.

[7] E. S. Lander and P. Green, "Construction of multilocus genetic linkage maps in humans," *Proceedings of the National Academy of Sciences*, vol. 84, no. 8, pp. 2363–2367, 1987.

[8] L. Kruglyak, M. J. Daly, M. P. Reeve-Daly, and E. S. Lander, "Parametric and nonparametric linkage analysis: a unified multipoint approach." *American Journal of Human Genetics*, vol. 58, no. 6, pp. 1347–1363, Jun. 1996. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1915045/

[9] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[10] E. Sobel and K. Lange, "Descent graphs in pedigree analysis: applications to haplotyping, location scores, and marker-sharing statistics." *American journal of human genetics*, vol. 58, no. 6, p. 1323, 1996.

[11] F. Rüschendorf and P. Nürnberg, "Alohomora: a tool for linkage analysis using 10k snp array data," *Bioinformatics*, vol. 21, no. 9, pp. 2123–2125, 2005.

[12] J. R. O'Connell and D. E. Weeks, "Pedcheck: a program for identification of genotype incompatibilities in linkage analysis," *The American Journal of Human Genetics*, vol. 63, no. 1, pp. 259–266, 1998.

[13] G. R. Abecasis, S. S. Cherny, W. O. C. Cookson, and L. R. Cardon, "GRR: graphical representation of relationship errors," *Bioinformatics*, vol. 17, no. 8, pp. 742–743, 2001. [Online]. Available: https://www.researchgate.net/profile/Stacey_Cherny/publication/11822219_GRR_Graphical_Representation_of_Relationship_Errors/links/02e7e51dac6047f791000000.pdf

[14] F. O. Walker, "Huntington's disease," *The Lancet*, vol. 369, no. 9557, pp. 218 – 228, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140673607601111

[15] I. W. Saunders, J. Brohede, and G. N. Hannan, "Estimating genotyping error rates from Mendelian errors in SNP array genotypes and their impact on inference," *Genomics*, vol. 90, no. 3, pp. 291–296, Sep. 2007. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S088875430700136X

[16] G. R. Abecasis, S. S. Cherny, W. O. Cookson, and L. R. Cardon, "Merlin-rapid analysis of dense genetic maps using sparse gene flow trees," *Nature Genetics*, vol. 30, no. 1, pp. 97–101, Jan. 2002. [Online]. Available: http://www.nature.com/doifinder/10.1038/ng786

[17] E. Sobel, J. C. Papp, and K. Lange, "Detection and Integration of Genotyping Errors in Statistical Genetics," *American Journal of Human Genetics*, vol. 70, no. 2, pp. 496–508, Feb. 2002. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC384922/

[18] D. F. Gudbjartsson, T. Thorvaldsson, A. Kong, G. Gunnarsson, and A. Ingolfsdottir, "Allegro version 2," *Nat Genet*, vol. 37, no. 10, pp. 1015–1016, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1038/ng1005-1015

[19] M. Brugger and K. Strauch, "Fast Linkage Analysis with MOD Scores Using Algebraic Calculation," *Human Heredity*, vol. 78, no. 3-4, pp. 179–194, 2014. [Online]. Available: http://www.karger.com?doi=10.1159/000369065

[20] E. M. Clarke, M. Fujita, and X. Zhao, "Multi-terminal binary decision diagrams and hybrid decision diagrams," in *Representations of discrete functions*. Springer, 1996, pp. 93–108. [Online]. Available: http://link.springer.com/chapter/10.1007/978-1-4613-1385-4_4

[21] F. Somenzi, "CUDD: CU decision diagram package release 2.3. 0," *University of Colorado at Boulder*, 1998. [Online]. Available: http://www.async.ece.utah.edu/~myers/nobackup/ee5740_98/cudd/cudd.ps

[22] H. Thiele and P. Nürnberg, "Haplopainter: a tool for drawing pedigrees with complex haplotypes," *Bioinformatics*, vol. 21, no. 8, pp. 1730–1732, 2005. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/21/8/1730.abstract

[23] G. Lathrop, J. Lalouel, C. Julier, and J. Ott, "Strategies for multilocus linkage analysis in humans," *Proceedings of the National Academy of Sciences*, vol. 81, no. 11, pp. 3443–3446, 1984.

[24] R. A. Gibbs, J. W. Belmont, P. Hardenbol, T. D. Willis, F. Yu, H. Yang, L.-Y. Ch'ang, W. Huang, B. Liu, Y. Shen *et al.*, "The international hapmap project," *Nature*, vol. 426, no. 6968, pp. 789–796, 2003.

[25] G. VanRossum and F. L. Drake, *The Python Language Reference*. Python software foundation Amsterdam, Netherlands, 2010.

[26] R. Berjon, S. Faulkner, T. Leithead, S. Pfeiffer, E. O'Connor, and E. D. Navara, "HTML5," W3C, Candidate Recommendation, Jul. 2014, http://www.w3.org/TR/2014/CR-html5-20140731/.

[27] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on.* IEEE, 2004, pp. 75–86.

[28] E. C. T. Committee *et al.*, "The embedded c++ specification," 1999.

[29] T. O'Reilly, "Open source paradigm shift," *DiBona, C., Stone, M., Cooper, D.: Open Sources*, vol. 2, pp. 253–272, 2004.

[30] B. Eriksson, "Migrating real-time applications to the web browser," 2012.

[31] E. ECMAScript, E. C. M. Association *et al.*, "Ecmascript language specification," 2011.

[32] OpenGL Website. [Online]. Available: https://www.opengl.org/

[33] WebGL Specification. [Online]. Available: https://www.khronos.org/registry/webgl/specs/latest/2.0/

[34] Microsoft DirectX. Microsoft. [Online]. Available: https://support.microsoft.com/en-gb/kb/179113

[35] J. Nielson, C. Williamson, and M. Arlitt, "Benchmarking modern web browsers," in *2nd IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2008.

[36] A. Charland and B. Leroux, "Mobile application development: web vs. native," *Communications of the ACM*, vol. 54, no. 5, pp. 49–53, 2011.

[37] M. Groves and C. Engler. Pixi. [Online]. Available: https://github.com/pixijs/pixi.js/

[38] C. Evans. jcanvas. [Online]. Available: http://projects.calebevans.me/jcanvas

[39] E. Drowell. (2014) Kineticjs. [Online]. Available: https://github.com/ericdrowell/KineticJS/

[40] R. M. Stallman, *EMACS the extensible, customizable self-documenting display editor*. ACM, 1981, vol. 16, no. 6.

[41] "top/bot" strand and "a/b" allele. [Online]. Available: http://www.illumina.com/documents/products/technotes/technote_topbot.pdf

[42] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[43] ——, "A case against the GOTO statement," *Comm. ACM 11*, March 1968.

[44] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, July 1968.

[45] C. Center, "Genetic dissection of complex traits: guidelines for interpreting and reporting linkage results," *Nat genet*, vol. 11, pp. 241–247, 1995.

[46] J. N. Fenner, "Cross-cultural estimation of the human generation interval for use in genetics-based population divergence studies," *American journal of physical anthropology*, vol. 128, no. 2, pp. 415–423, 2005.

[47] N. E. Morton, "Sequential tests for the detection of linkage," *American journal of human genetics*, vol. 7, no. 3, p. 277, 1955.

[48] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications." *WebApps*, vol. 10, pp. 3–3, 2010.

[49] Comprehensive Perl Archive Network. [Online]. Available: http://www.cpan.org/

[50] C. S. Committee *et al.*, "iso/iec 14882: 2011, standard for programming language c++," Technical report, 2011. http://www. open-std. org/jtc1/sc22/wg21, Tech. Rep., 2011.

[51] S. S. Bakken, Z. Suraski, and E. Schmid, *PHP Manual: Volume 2*. iUniverse, Incorporated, 2000.

[52] (2008, February) Sun microsystems announces completion of mysql acquisition; paves way for secure, open source platform to power the network economy (press release).

[53] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[54] N. Merhav, M. Gutman, and J. Ziv, "On the estimation of the order of a markov chain and universal data compression," *IEEE Transactions on Information Theory*, vol. 35, no. 5, pp. 1014–1019, 1989.

[55] (2006) WWW FAQs: What is the maximum length of a URL? [Online]. Available: https://boutell.com/newfaq/misc/urllength.html

[56] L. Masinter, "The "Data" URL Scheme," 1998.

[57] pieroxy. (2015) LZ-based compression algorithm for JavaScript. [Online]. Available: https://github.com/pieroxy/lz-string

[58] E. Dowell. KineticJS Information Page. [Online]. Available: http://kineticjs.com/

[59] E. Drowell. (2016) Concretejs. [Online]. Available: http://www.concretejs.com/

[60] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[61] R. Stallman *et al.*, "The gnu project," 1998.

[62] GNU General Public License. Free Software Foundation. [Online]. Available: http://www.gnu.org/licenses/gpl.html

[63] GNU Affero General Public License. Free Software Foundation. [Online]. Available: https://www.gnu.org/licenses/agpl-3.0.html

[64] (2000) FreeBSD License. University of Berkeley. [Online]. Available: https://www.debian.org/misc/bsd.license

[65] (2000) MIT License. Institute of Massachusetts. [Online]. Available: https://opensource.org/licenses/mit-license.php

[66] S. R. Walli, "The posix family of standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[67] E. W. Dijkstra, "Cooperating sequential processes," in *The origin of concurrent programming*. Springer, 1968, pp. 65–138.

[68] A. B. Downey, "The little book of semaphores," *http://greenteapress.com/semaphores*, 2005. [Online]. Available: http://cs.miami.edu/~burt/learning/Csc521.121/docs/downey08semaphores.pdf

A. Glossaries

A.1 Biological Terms

DNA Deoxyribonucleic Acid. 14

ELOD Expected LOD score. 55

HMM Hidden Markov Models. 37

HWE Hardy-Weinberg Equlibrium, law that states that allele frequencies in a population remain constant in the absence of external factors. 22

IBD Identity-By-Descent, Genotypes matching in two individuals due to a common ancestor. 52

IBS Identity-By-State, Genotypes matching in two individuals by chance. 52, 156

Informative marker Marker thought to lend power to a linkage analysis. 44

LOD Logarithm of the Odds (score). 34, 35, 55, 149, 192

Morgan Genetic map unit, where 1 cM denotes 0.01 expected crossovers. 21

RNA polymerase Enzyme that produces primary transcript RNA. 16

SNP Extremely prevalent binary marker. 23, 67

STR See microsatellite. 23

UTR Untranslated Region. 17

VNTR Variable Number Tandem Repeat. 23

alleles Locus that spans across both chromosome pairs. 16, 44, 90, 147, 204

allosomal Chromosomes X and Y. 16

amino-acid Sub unit of Protein. 17

antisense See reverse. 15

autosomal Chromosomes 1 to 22. 16, 29, 49, 125, 158, 194

autozygous Homozygous alleles that arise from separate paths of descent as a result of consanguineous interbreeding. 30

base pairs Nucleobases that span across the two chromosome strands. 14

biallelic Alleles with only two variations, work better in conjunction with SNPs. 22

bit size The size of the pedigree as determined by the number of founders, non-founders, and genotyped founder couples. 28

bivalent A pair of homologous chromosomes. 19

carriers Individuals who carry the disease allele but not the phenotype. 29

centromere Central structure binding pairs of chromosomes together. 16, 65

chiasmata Point of constant contact where crossover events occur. 20

chromatin Xondensed DNA wrapped around proteins to fit in nucleus. 15

chromosomes Long DNA strands bound together. 14, 49, 114, 149, 193

co-dominance A case where neither allele dominates the other and phenotypes are independent of each other enough to both be expressed. 22, 27

coding Part of DNA known to encode for genes. 16

codons Triplet groups of nucleobases. 16

consanguineous Relating to individuals who breed within the family and cause inbreeding loops in the pedigree. 28, 107, 160

crossover interference Measure of interference between otherwise separate crossover events, where the act of one may hinder another. 21, 33

crossover Exchange of genetic material between homologous chromosomes. 20, 102

diploid Cell with two sets of chromosomes from each parent. 15, 26

dominant Heterozygous allele where phenotype manifests. 27, 29, 49, 126, 158

double helix Shape of helical DNA backbone. 14

exons Coding part of a gene. 16

forward 5' to 3' orientation. 15

founder alleles Unique alleles that contribute to a pedigree, portions of which are inherited by non-founders. 28, 179

founders Members of a pedigree who have no parents. 28, 62, 114, 160

genome Complete genetic material present in an organism. 14, 56, 149, 194

haploblock Contiguous block of haplotypes of the same phase. 31, 102, 179

haploid Xell with a single set of unpair chromosomes. 15

haplotype Phased genotype. 31, 44, 72, 168, 192

heterozygous Alleles differ. 25, 29, 50, 98, 195

homologs Pairs of chromosomes. 16

homozygous Alleles are identical. 25, 97, 183

ihaplo.out The Allegro haplotypes output file. 57

incomplete dominance Same as co-dominant except that traits are related and so phenotype is a blend of both. 27

intergenic Region between genes. 17

linkage disequilibrium Two linked loci that have a recombination frequency higher or lower than the amount expected if the loci were independent. 20

mRNA messenger RNA. 17

markers Genetic marker that denotes a known locus in a genome. 21, 41, 89, 147, 200

meiosis Process in which single-cell splits into four gametes. 18

meiotic spindle Spindle used to equally divide chromosomes and create two daughter cells. 19

metaphase plate Plane that bisects the meiotic spindle, axis in which chromosomes align during metaphase. 19

microsatellite A block of repetitive DNA, 2-5 bp. 23

minisatellite A block of repetitive DNA, 5-50 bp. 23

multi-allelic Genotypes with more that just two alleles. 27

multi-point parametric Linkage analysis that uses a map of genetic markers to reconstruct inheritance along a chromosome. LOD is calculated by comparing each marker against all other assumed unlinked loci. 35, 56, 198

non-founders Members of a pedigree who inherit genetic data from founders. 28, 139

non-recombinant Offspring with genotypes that do match a single parent at a given locus. All offspring a recombinant across an entire chromosome. 27

nucleobase Four bases: A,T,C,G. 14

nucleotides Sub-units of DNA, consists of a phosphate group and nucleobase. 14

open reading frame Uninterrupted string of codons. 17

pedigree A collection of related individuals represented in a graph diagram indicating mating and offspring. 28, 35, 43, 104, 149, 192

peeling The act of removing individuals from a pedigree in order to reduce the number of loops. 128

phased Path of descent of each allele is known. 31, 100

phenotype Observable traits. 16, 158, 206

polygenic traits Inheritance of phenotype affected by more than one gene. 27

polymorphic Alelles with multiple variations. 22, 90

recessive Heterozygous allele where phenotype does not manifest. 27, 158

recombinant Offspring with genotypes that do not match any single parent at a given locus. 27

recombination frequency Frequency of independent assortment between two loci. <50% if linked. 20

reverse 3' to 5' orientation. 15

ribosome Converts mRNA into amino acids. 16

sense See forward. 15, 43

splicing Process of cutting introns out of gene locus and stitching together exons. 17

start codon ATG. 17

stop codon TAA, TGA, TAG. 17

telomeres Caps at the ends of chromosomes that shorten every cell division. 16

transcription DNA being coped into mRNA by . 17

translated The process of converting between mRNA and Protein. 17

trio A single mother-father-offspring group. 42, 97, 192

unphased Unknown path of descent. 31

uracil The mRNA equivalent of Thymine. 17

zygote Cell formed by the fertilization of two gametes. 18

A.2 Computing Terms

API Application Programmers Interface, a well-documented list of possible function calls and services offered by a library. 78

ARM Type of low energy processor used in mobile devices with a small restricted instruction set. 196

Arch Type of Linux OS that follows a "Keep It Simple, Stupid" (KISS) ethos. 60, 190, 197

Bash Bourne-again Shell, type of terminal shell language. 76, 198

C++ Low-level language that compiles programs to machine code. Emphasis on speed and optimization. 75, 76

CPU Central Processing Unit, the "brain" of a computer. 74, 223

CUDD Colarado University Decision Diagrams, an extension of MTBDDs. 63, 193, 199

Cairo 2D graphics library. 57, 67, 85, 196

ConcreteJS Optimized general-purpose 2D Javascript framework, from the same author as KineticJS. 207

Cross-platform Transcends Operating System specifics. 80

DBI General Database Interface. 196

DNS Domain Name Server, resolves textual domain names into IP addresses. 87

DOM Document Object Model, interface to HTML/XML documents. 88

DOS Degree of Separation. 138, 195, 204

Debian Type of Linux OS, one of the oldest, very robust. 59, 197

DirectX Type of 2D/3D graphics specification with closed source implementations aimed at Windows platforms. 90

ECMAScript6 Script specification (version 6) used by Actionscript and Javascript. 88, 96

FOSS Free and Open Source Software. 86, 196, 222

GET Type of request-response protocol for accessing data bundled with HTTP headers. 206

GNU Recursive Acronym: GNU's Not Unix. Coined by Richard Stallman founder of the Free Software Movement. 48, 172, 222

GPU Graphical Processing Unit, chip dedicated to computing graphical data and drawing data to screen. 91

Gentoo Type of Linux OS aimed for performance users. 60, 197

HSV Hue-Saturation-Value, type of colour space aimed at preserving channels for common image changes. 116, 204

HTTP Hyper-Text Transfer Protocol, common internet protocol for sending ordered packets of data over the internet with handshaking error checks. 201

IDE Integrated Development Environment, a rich and full-features source code editor. 96

IP Internet Protocol. 87

JIT Just-In-Time compiling, the practice of creating sub-binaries of portions of code known to create bottlenecks. 80

JavascriptCore WebKit's Javascript Engine. 91

Java Medium-level programming language that runs in a live environment. 76

KineticJS 2D Javascript framework used extensively within thesis. 95, 207

LZMA Lempel-Ziv-Markov chain Algorithm, type of compression method. 206

Linux Free Open-Source operating system conceived by Linus Torvalds. 59, 88, 196, 224

MAKEPED Linkage pedigree format with 6 standard columns of: familyID, patientID, fatherID, motherID, gender, affectation. 128

MESA MESA is an open source software implementation of OpenGL and enables graphics to be drawn under the API using the CPU. For this reason it is not as fast as the OpenGL implemented by card vendors such as Nvidia or AMD upon their respective GPUs. 90

MIMD Multiple Instruction Multiple Data. 80

MISD Multiple Instruction Single Data. 80

MTBDD Multi Terminal Binary Decision Diagram, a compact structure used to minimize the number of unique computations. 63

MVC Model-View-Controller, Software development practice of separating graphics from data and representing data statically. 78

OAuth2 Type of authentication protocol. 205

OS Operating System, manages hardware and software. 59, 80, 195, 224

OpenGL Type of 2D/3D graphics specification with closed source and open source implementations aimed at all platforms. 90

PATH A global environment variable that allows for the execution of scripts and binaries without having to specify full pathname. 59, 225

PDF Portable Document Format, type of document container. 50, 57

PHP Scripting language used on HTTP servers to automate background system tasks. 200

POSIX Portable Operating System Interface, standards specification common across Unix and Unix-like OS's. 226

PS Post Script, vector graphics-centric document format. 98

Perl High-level general-purpose scripting language. 50, 76, 196

Python High-level scripting language that runs in a live interpreter environment. Emphasis on code readability. 46, 55, 75, 197

Qt C++, Python and Java Framework used to facilitate in cross-platform development. Has superior string handling. 77, 80

RAID Redundant Array of Independent Disks. 203

RAM Random Access Memory, fast device used to store temporary data. 63, 178, 202

SIGHUP Signal Hang Up, a signal emitted by a process when the user that spawned the process logs off. Usually kills the process at the same time, but can be overriden using **nohup**. 201

SIMD Single Instruction Multiple Data. 80

SSD Solid State Drive, a type of fast non-mechanical storage. 202

SSL Secure Socks Layer, type of encrypted communication protocol. 201

Sort Naturally Type of sorting algorithm that alternately sorts data alphabetically and numerically (but not alphanumerically). 196

SpiderMonkey Gecko's Javascript Engine. 91

Strings Type of data type used to hold textual information. 83

Three.js Core 3D library in Javascript. 91

Tk Tkinter GUI framework used heavily in application development. 67, 85, 196

Ubuntu Type of Linux OS, based upon Debian with a focus on desktop applications and general operability. 59, 197

V8 Google's Javascript Engine. Integrated more recently into WebKit engine. 91

VRAM Virtual RAM, the addition of real RAM and swap space. 63

WebGL Web standard for incorporating OpenGL bindings into Javascript. 85, 90

autocompletion The process of automatically completing a word based on a starting prefix. 96

base64 A type of 64-bit encoding. 206

compilation The process of breaking down high-level user scripts or programs into low-level machine tokens. 96

dedicated graphics Graphics from a dedicated hardware device with its own memory such as a graphics card. 190

embedded graphics Graphics from hardware that comes premade with another component, sharing resources with that component. 190

event listeners Type of input polling that performs a pre-specified action upon receiving input. 94

forking The act of splitting off a task from the main control flow and letting it perform work in parallel. 76

getters and setters Functions that retrieve or set a value, usually under an Object-Oriented context. 131

horizontal-buffering Large text that is not often delimited by newline characters can produce significant latency in text editors that try to load the entire line into memory. Some text editors support horizontal buffering which only loads a portion (usually just the visible component) of the text to overcome this issue. 66

journaling The process of actively indexing files in a filesystem. 202

macros Snippets of code defined within the IDE usually with the intention of running only for a specific platform, though they can be used whenever the cost of repeatedly calling a function has a higher cost than simply copy/pasting the code snippet. At compilation, the compiler detects the platform and moves/-pastes the appropriate macro into the normal code scope and produces a binary specific to that platform. 80

mipmap A series of small renders of otherwise large images to be loaded at points where the zoom factor would not otherwise be able to tell the difference in pixel density between the mipmap and the original large image. 195

mutex A mutually exclusive variable that cannot be used by a process if another is currently using it (blocking the process). 227

mySQL Declerative open source database for storing records, now owned by Oracle. 205

object-oriented Programming paradigm in which data is structured into classes which dictate functionality and access. 83

object Data construct that has properties bound to it. 83

overflow/underflow If a variable tries to store information that exceeds ('overflows') the upper-bound of the data type it is within, it may undesirably take on the value of the lower-bound of that data type. The reverse is also true ('underflow'). 83

paging The process of swapping portions of memory out of swap space and into the RAM for immediate processing and vice versa. 178, 201, 202

parity A single bit added to the end of a string that indicates whether the 1-bits are even. 204

primitives Types of common data forms that exist in many languages. See Appendix. 83

regex Regular Expressions, type of string used to match other strings. See Appendix for more information. 84

standard template libraries Libraries that influence many parts of the language standard (primarily C++). 76

striping The act of spreading data across multiple block devices. 203

swap space Reusable memory on disk that acts like RAM, only much slower. 63

terniary Code that consists of three parts (if,then,else), typically written on a single line. 89

thread safe A process is thread safe if multiple threads can execute the same process without them clashing or competing for variables. 58

B. Appendix

B.1 Hardware Specification

The server hardware used to incorporate the High-Throughput Linkage Analysis Pipeline was a **Dell PowerEdge R710**, with:

- **CPU** 16 × 3.02 GHz.
- **RAM** 256 GB 1066 MHz Error-Correcting.
- **Disk** 20 TB SAS (RAID-5).

The desktop hardware used to develop the HaploHTML5 pipeline was a custom rig, with:

- **CPU** AMD A10-7850K APU with Radeon R7 Graphics, 4 x 3.7 GHz
- **RAM** 24 GB 1366 MHz DDR3.
- **Disk** 16 TB 7200 rpm
- **Graphics** NVidia GTX under Linux *nouveau* open source driver

B.2 Computing Concepts

B.2.1 Colour Spaces

A colour space is a way to index colours under a certain schema. There are 4 main types used in digital processing:

1. **RGB** Red-Green-Blue; directly maps to the Red/Green/Blue channels on monitors where (for example) an increase in the R-component will produce a redder image, and likewise with the G and B components for greener and bluer images respectively.

Colours	Colour Models												
	R	**G**	**B**	**H**	**S**	**V**	**C**	**M**	**Y**	**K**	**Y**	**U**	**V**
Red	255	0	0	0	100	100	0	1	1	0	0.3	0.5	-0.17
Orange	255	128	0	30	100	100	0	0.5	1	0	0.59	0.29	-0.33
Yellow	255	255	0	60	100	100	0	0	1	0	0.89	0.08	-0.5
Light Green	64	255	0	90	100	100	0.5	0	1	0	0.74	-0.17	-0.42
Green	0	255	0	120	100	100	1	0	1	0	0.59	-0.42	-0.33
Spring Green	0	255	128	150	100	100	1	0	0.5	0	0.64	-0.46	-0.08
Cyan	0	255	255	180	100	100	1	0	0	0	0.7	-0.5	0.17
Teal	0	128	255	210	100	100	1	0.5	0	0	0.41	-0.29	0.33
Blue	0	0	255	240	100	100	1	1	0	0	0.11	-0.08	0.5
Indigo	128	0	255	270	100	100	0.5	1	0	0	0.26	0.17	0.42
Purple	255	0	255	300	100	100	0	1	0	0	0.41	0.42	0.33
Magenta	255	0	128	330	100	100	0	1	0.5	0	0.36	0.46	0.08

Table B.1: Colour Indexes for RGB (8-bit), HSV, CMYK, and YUV.

Intermediate colours such as Yellow, Purple, and Brown are created 1:1 representations of two channels (with the third remaining constant), and Black→Grey→White representations are produced when locking the values of all three channels. Darkening and lightening an image requires changing all three channels.

2. **CMYK** Cyan-Magenta-Yellow-Key; where Key represents the darkness of an image. The colour model is used primarily in printers and uses subtractive mixing to achieve different colour blends (e.g. Green is the subtraction of Yellow and Cyan). Has the same colour scaling patterns as RGB but can produce darker semitones using the K component.

3. **YUV** Luminance-ChromaU-ChromaV; where Luminance denotes the brightness and the two chroma channels denote separate colour ranges as shown by the two axes in Figure B.1. The colour model makes use of the human eye's lack of color perception compared to awareness of changes in brightness. Early engineers used this 'hack' to reduce the size of TV transmissions by reducing the U and V components of an image without any perceptible difference to the quality of the image.

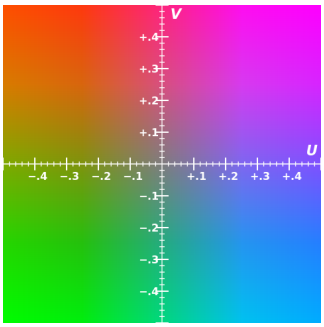

Figure B.1: YUV scale for Y = 0.5 (compressed). Image courtesy of Wikimedia Commons.

4. **HSV** Hue-Saturation-Value; where Hue denotes a shade of colour, Saturation the intensity of the colour, and Value being the brightness. This allows for changes in colour to be recorded by a single channel only and is a very convenient way to modify brightness too. Stepping through a rainbow of colours involves only modifying the H component, as shown in Figure 5.1 on page 212.

Table B.1 shows the values that each colour model takes for each colour in the table. Note the constancy in the S and V channels of the HSV model.

B.2.2 Primitives

Data generally comes in two forms:

1. **Numeric**

   (a) **decimal**, the integer value of a number represented in base-2 (or 'binary') number format, (e.g. 8 = 1000 (4-bits), 7 = 111 (3-bits), 652 = 101000110 (9-bits)). For n bits, the decimal range is $[0,2^n - 1]$[92].Decimals tend to

[92]This is only in case of positive unsigned integers. To include negative integers too, the first bit is sacrificed as an indicator of polarity. This produces signed integers and follows a more involved schema known as *two's-complement* in order to only have a single representation of the number zero. The numeric range for n bits is then [-2n-1, 2n-1 - 1].

come in fixed bit-sizes[93] so there can be some redundancy in representing a small number with a large decimal type.

(b) **floating-point** numbers are a compact approximation of a number as deemed by the specification[60]. Numbers are represented as powers of exponents such that $5 \sim= e^{1.61}$ (where 1.61 would be the stored number in this loose example). The precision of the representation is determined by how many bits the number has. Floats have a much larger range than decimals, albeit with less accuracy

2. **Textual**

   See Strings section below.

B.2.3 Strings

Strings are a type of data that is defined as any block of text surrounded by single or double quotation marks. Strings compared to other data types are very memory intensive since it is usually unclear exactly how much memory a string is going to take.

Numbers can be assigned specific memory sizes to contain their values such as char(8-bit) , short(16-bit), int(32-bit), and long(64-bit) for decimal numbers, and float(32-bit) and double(64-bit) for floating point numbers, each with their specific range of values.

Strings are essentially an array of characters, and so if each character is 8-bit, then the phrase "Hello World!" has a size of 96 bits (12 characters x 8 bits) . When strings are created they are assigned initial sizes to contain a specific text, but they are essentially immutable past this point since concatenating two strings together requires re-computing their sizes and copying the text into a new block of memory (i.e. you cannot directly shorten or extend a string since it creates memory leaks).

[93] byte (8-bit), short/char (16-bit), int (32-bit), long (64-bit), and (in some languages) long long (128-bit).

i String Interning

Because of the costly processes of manipulating strings during program runtime, some string-processing tasks take place to reduce the redundancy in duplicate strings. A list of all unique strings declared in the code are "pooled" into static addresses at compile-time, and then are referenced every time a string variable is assigned one of their values.

When an entirely new string is required, the "new" keyword is used to tell the environment that it will not be referencing a previously pooled-string and that extra memory must be allocated.

B.2.4 RegEx

Regular Expressions (also known as RegEx or RegExp) are character patterns that are used to match text or 'strings' in a flexible manner. A relevant example would be trying to match a [chromosome,position] tuple:

```
chr01   87654321   [DETAILS] // <--file1
    1:87654321     [DETAILS] // <--file2
  01   87654321    [DETAILS] // <--file3
```

Here, all files reference the same variant (chromosome 1, position 87654321) but due to the flexibility of the VCF specification and the non-standardization of recording conventions between biologists, there is much disparity between the way they are referenced. However a good regex to match the same variant in all these files would be simply:

```
^(chr){0,1}(0){0,1}1(:|\s+)(87654321)
```

which, in pseudo-code translates to "at the beginning of the line, look for 0 to 1 mentions of 'chr' followed by 0 to 1 mentions of '0', followed by a '1', followed by either a ':' character OR('|') 1 or more ('+') mentions of a white-space ('
s') character (e.g. a tab, a space), followed by '87654321'.

i Regex String Component

Parsing strings for specific phrases is another feat as it requires tokenizing the data into characters and performing character matching upon a reference string. This is not hard if you have a specific string in mind to compare (e.g matching "Hello" in "Hello World" would complete its search after comparing only the first 5 characters), but becomes much more complex when dealing with more abstract search criteria (e.g. extracting all lines in a text file that contain full stops but NOT commas).

This is where regex takes precedence as it converts a reference string into what is known as a Finite State Automaton (FSA) which links together characters in such a way that one character can lead to the next but only in a specific pattern. For example to match any sequence that has an 'X' followed by 0 or more 'Y's followed by an 'X' again is specified by pattern: XY*X

Figure B.2 shows how such a pattern is first converted into character-specific states of 'X', 'Y' and blank intermediaries which allow a specific sequence of letters occur by taking either one of paths 'a' or 'b'. This state diagram can be further compacted and optimized into a Deterministic Finite State Automaton (DFSA) which is a simple non-character specific two-state system which allows transitions between states via 'X' and 'Y' input paths. The table below shows possible string matching scenarios using such a system and it shoud be noted that the last string input is not valid for the RegEx given hence why it ends prematurely.

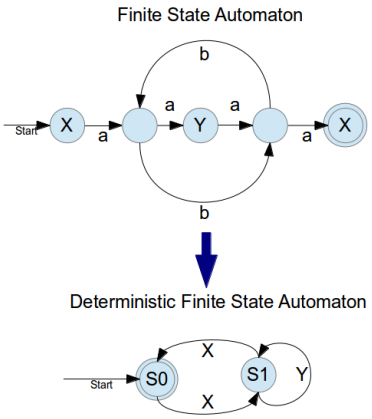

Figure B.2: Finite State Automaton.

RegExp :XY*X	
String	Path Taken
XX	a(b)a
XYX	a(aa)a
XYYYX	a(aa)b(aa)
XYXYX	a(a?END)

DFSA's can be further compacted and optimized to create fast pattern matching systems with little overhead, making them the ideal tool to deal with strings flexibly. Multiple genetic sequences can be extracted this way by using the same regular expression undoubtedly making RegEx a very powerful tool for programmers and geneticists alike.

B.2.5 Licensing

Software licences are typically split into two groups: *CopyLeft* and *Permissive*.

CopyLeft A CopyLeft license is a protective license that allows anyone to modify and/or redistribute copies of their work under the condition that they must distribute their source code and that the receiver must also be bound under

the same agreement. This ensures that software remains FOSS throughout all works that incorporate the license.

Free Software Movement guru Richard Stallman was the first to create a Copyleft license, the GNU [61] General Public License (GPL), which has gone on to make two more versions:

1. **GPLv2**
   (a) **GPLv2.1**, Enforces the copyleft obligation of the license in all works that incorporate it.
   (b) **Lesser GPL (LGPL)**, Weakens the obligation so that it can be incorporated into proprietary software, where the LGPL portions of the code must still be FOSS to the end-user. Has the extra caveat that LGPL can be relicensed under any GPL.
2. **GPLv3**, Better relicensing, protection from DRM (in theory), and no source code redistribution requirement as long as no software if redistributed either [62].
3. **Affero GPL (AGPL)**, Same as GPLv2 except it fixes a loophole caused by Application Service Providers (ASPs) who allowed users to use software, but did not distribute it themselves and thus were protected from copyleft jurisdiction. The fix involves making the source code available to any network user on an AGPL network. In the case of the a scripting web language like Javascript, this is now default behaviour [63].

Permissive A license that lets anyone do whatever they want without any restrictions on modification and redistribution, but they must credit the original author who is not held liable for any problems that occur later on.

1. **BSD**, Based upon the original licensing of FreeBSD (Berkeley Software Distribution) Operating System. Distributors are not allowed to use the original author's name in an unwanted way [64].

2. **MIT**, Same as BSD but without endorsement restrictions, freer [65].
3. **Apache**, Same as MIT, but any modifications made must be announced and the project name/trademark must differ from original unless name use is desired from original author.

B.2.6 Compiler

There are several optimizations that the compiler undertakes to optimize user-written code to be more efficiently read and processed by the CPU.

The optimizations are too numerous to list, but a few examples are provided below in order to provide some reference upon the nature of the optimizations:

Parallelize Split sequential data into separate dependency chains that can be processed in parallel.

Common Expression Elimination Calculate the value of reused static expressions only once, and cache the result.

Expand Constants Precompute expressions that involve only constants so that they don't need to be calculated at runtime.

Strength Reduction Convert difficult operations into smaller easier ones. e.g. replace an integer multiplication with a series of repeated additions instead.

Fast Pathing Split an instruction set into a "fast" path that handles the majority of input cases, and handle the infrequent edge cases separately.

Tail Recursion Elimination Convert stack-greedy recursive functions into less demanding iterative functions instead.

Loop Peeling/Splitting Remove unchanging variables within for or while loops outside of the loop and process separately (before or after the main loop body).

Loop Unrolling Copy and paste the body of a loop in repeated succession if the loop has a small body or few iterations.

Inline Code Small functions that occur repeatedly incur a cost of switching to them whenever they are executed. It is better to simply copy-and-paste them as entire code blocks at calling points so that they are part of the current instruction.

Bunch Related Data Keep related code blocks and variables close to each other in memory.

B.2.7 Linux Filesystem

Linux OS's have filesystem structure that is well-conserved across different versions (or "flavours") of Linux.

The filesystem tree structure is as follows, stemming off the main root branching point '/':

/boot Bootloader specific files are stored here and are used to load the initial RAM filesystem which in turn loads the full OS.

/dev Provides special accessors to real and virtual block devices such as terminals, hardisks, input/output buffers, random number streams, etc.

/etc Stores system-wide variables that are preserved across reboots. Noteable entries are *fstab* which stores (and automatically mounts) the mount points of the hard drives used by the OS, *mdadm* which stores the configuration of a RAID setup, and *resolv.conf* that is used as a nameserver to resolve internet addresses.

/home Location of all user data (separated into respective user-specific folders with user-specific access permissions).

/mnt Generally empty, but can be used as a mount point for anything.

/opt System-wide scripts and binaries that are not installed through the standard system package manager. Convention states that extragenous software should go here.

/proc File descriptors of active (or previously active) processes, with accessible properties such as status and system resource usage.

/root Location of the root user's home directory. Access is frowned upon.

/sys Where platform-specific device settings are retrieved and set.

/var System runtime variables populated by normal OS operation. Reset every boot.

/tmp Same as /var but with user access.

/usr Where all the system libraries, binaries, shared data resources, and include headers are stored. Most PATH locations point here.

Linux also has it's own filesystem types that have limited compatibility with the default Windows filesystems (VFAT and NTFS).

A few of the more comonly used filesystems are: ext{2,3,4}, ReiserFS, BtrFs, XFS, and JFS.

Each implement partitions and file storage in different ways that lend relative performance boosts in different task-specific ways.

i Inodes

Most file systems store files using inodes which are unique descriptors for accessing a file. The reason why filenames are not used as the descriptors themselves is due to the dynamic and changing nature of filenames, wherein a file may be renamed but the file itself remains unchanged.

It is the inode that assigns the file the new filename without changing the file's location within memory.

Inodes are also useful when linking files via hard and soft symbolic links, where a hard link would point to the actual inode which would remain valid even if the original file was renamed. A softlink would point only to the original file (which in turn would point to the inode), and would be subject to any changes made to the original file.

ii POSIX Permissions

Most Linux and Unix systems are said to be POSIX-compliant if they adhere to the principles set by the standard [66].

One of the most prominent features of the standard is the set of permissions attributed to each file, where file access is restricted to certain users (or certain groups of users) who can perform certain functions upon it.

File functions are independently limited to: Reading, Writing, and Executing. A file can be read and written to, or written and executable, or just executable, or any variation of three options. Further, these operations can be limited to a triad of users: the owner, the group, and other users.

This essentially prevents the average user from making dangerous edits to system files, where only the root user has access to the file system, and the root user is used sparingly. Typically an average user would have to invoke root privileges in order to make a system edit, which would require knowledge of the root password. More common implementations make use of the **sudo** utility which temporarily elevates the current user's privileges without having to directly invoke root.

B.2.8 Semaphores

A semaphore is tool that aids in the synchronization in multiple parallel processes/threads such that they can operate concurrently with one another with clashing when accessing the same shared variables.

First invented by Edgar Dijkstra [67], they are defined as follows:

1. A semaphore can be set to any value upon initialization, but can only be incremented/decremented thereafter.

2. Each thread that decrements the semaphore must block itself from performing further processing and wait if the semaphore becomes negative. It can only resume processing when another thread increments the semaphore.

3. Each thread that increments the semaphore unblocks another waiting thread.

That is; the value of the semaphore (unknown to all threads) represents the number of active threads when positive, and the number of waiting threads when negative [68].

Semaphores serve as a means for threads to signal each other when to stop and go, as well being mutex variables such that no two threads access the same resource at the same time.

B.3 Poisson Distribution

A discrete probability distribution that describes events that are expected to occur within fixed time intervals, known to recur at an average rate that is independent of the last time interval.

The probability of such events are given by the equation:

$$P(k) = \frac{\lambda^k e^{-k}}{k!} \tag{B.1}$$

where:

$k =$ number of expected events in the time interval

$\lambda =$ the average frequency of said events in the time interval

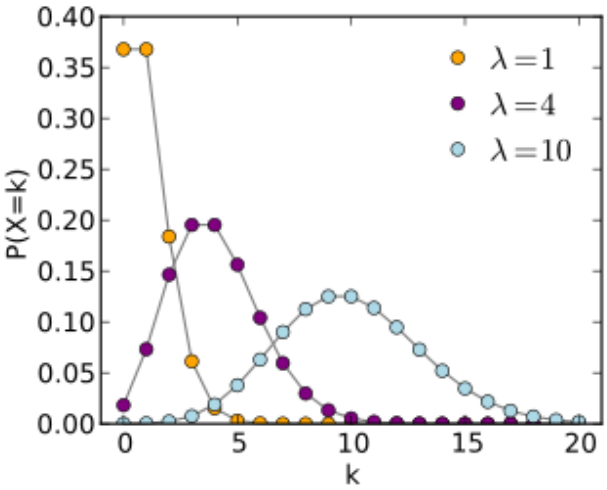

Figure B.3: Poisson Distribution for three different λ values for increasing values of k.

B.4 Results Data

B.4.1 LOD Scores for 21-bit Linkage

Scenario	Expected Max. LOD
(Base) 0	3.31
1	3.72
2	3.64
3	3.75
4	3.44
5	3.63
6	3.72
7	3.45
Mean of 7 scenarios	3.62

Table B.2: Maximum Estimated LOD Scores for 21-bit linkage

B.4.2 Chromosome plots for 21-bit Linkage

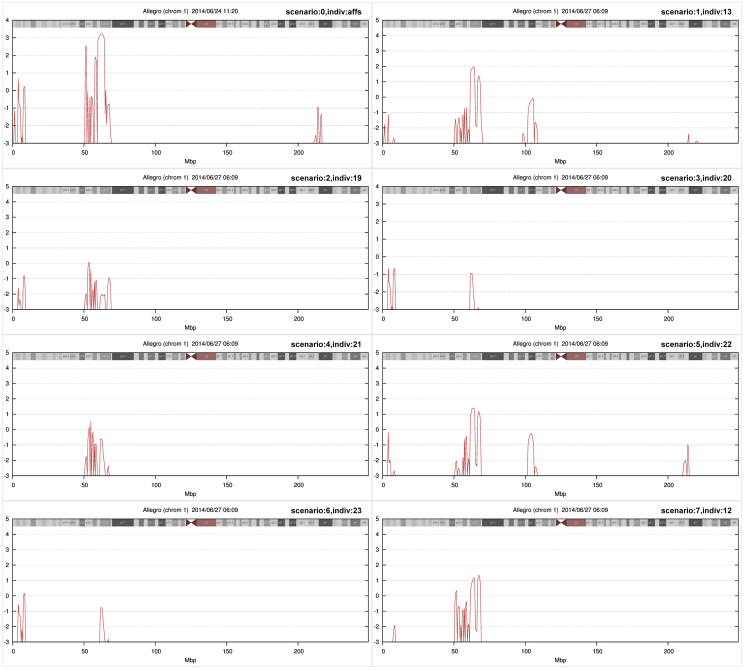

Figure B.4: Linkage plots of chromosome 1 for each of the eight analyses

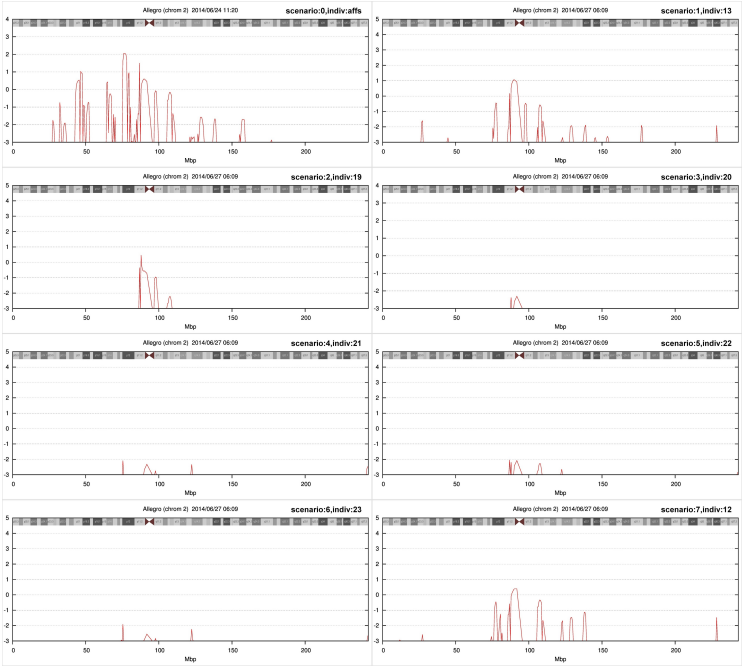

Figure B.5: Linkage plots of chromosome 2 for each of the eight analyses

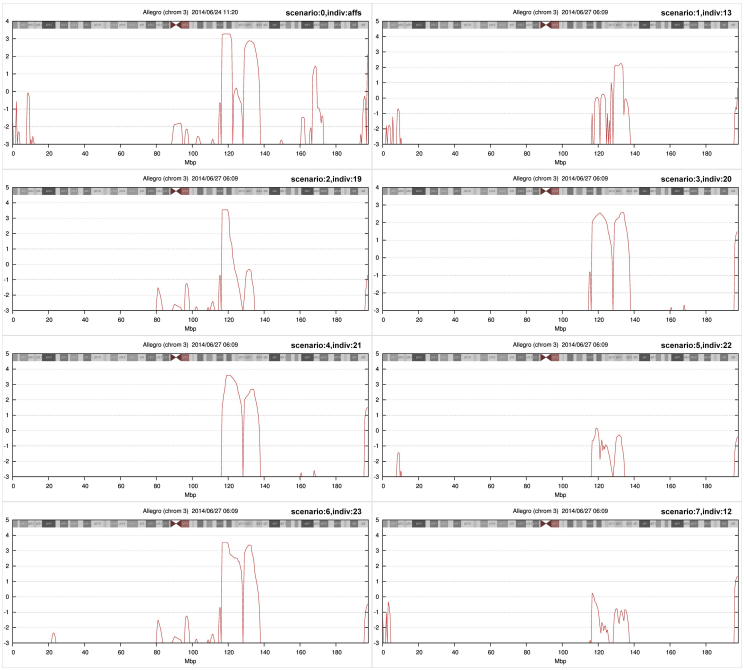

Figure B.6: Linkage plots of chromosome 3 for each of the eight analyses

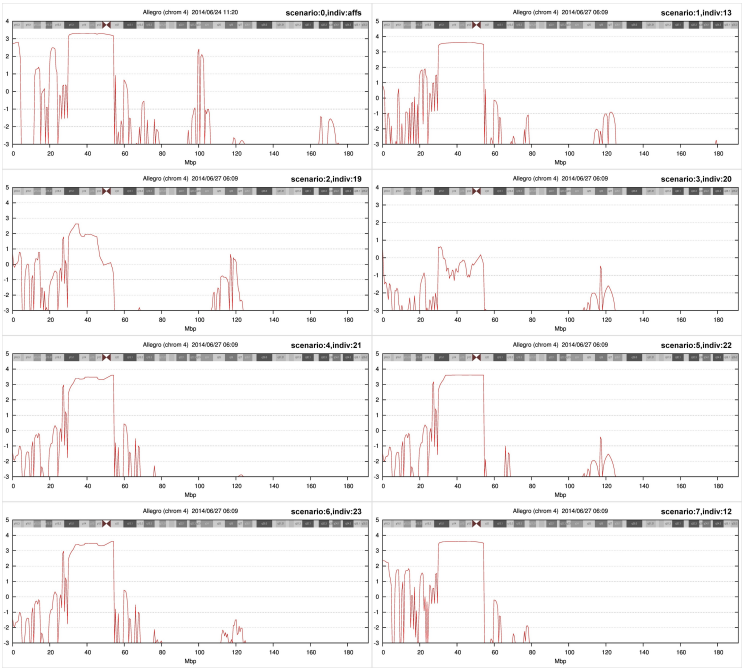

Figure B.7: Linkage plots of chromosome 4 for each of the eight analyses

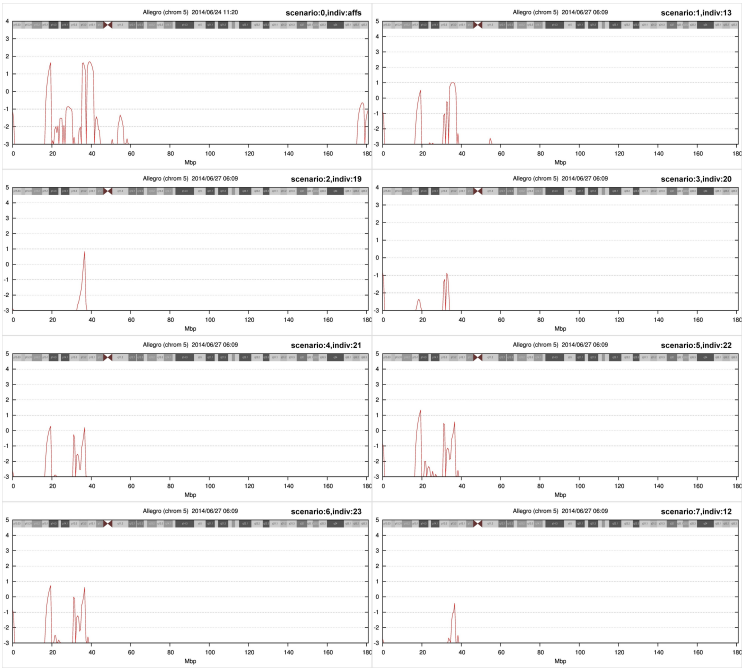

Figure B.8: Linkage plots of chromosome 5 for each of the eight analyses

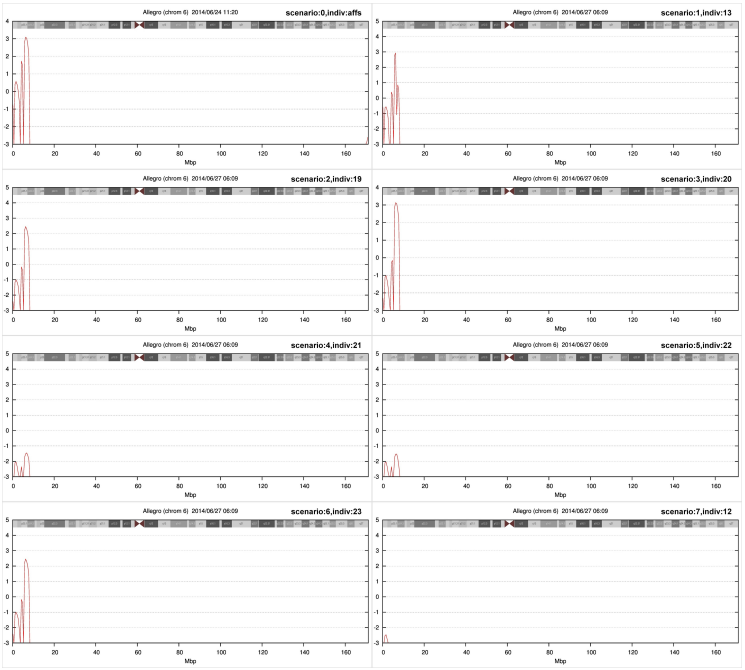

Figure B.9: Linkage plots of chromosome 6 for each of the eight analyses

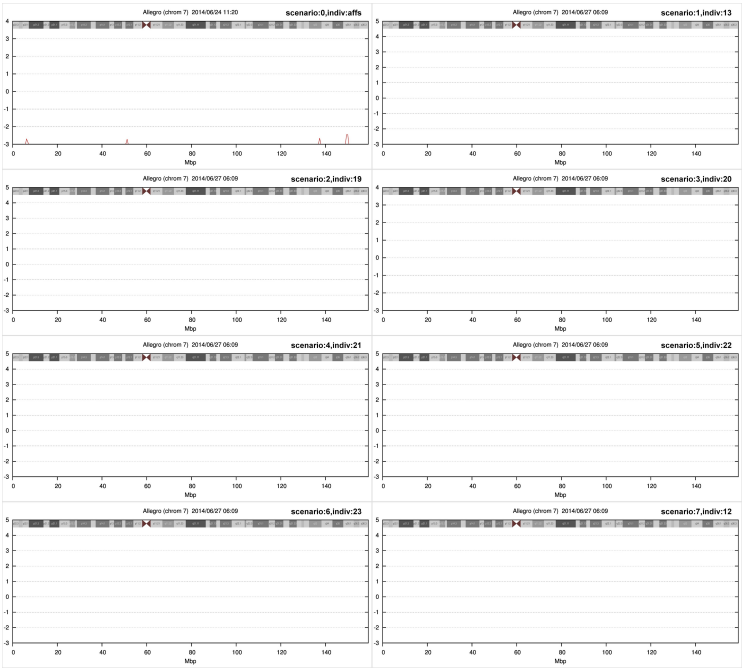

Figure B.10: Linkage plots of chromosome 7 for each of the eight analyses

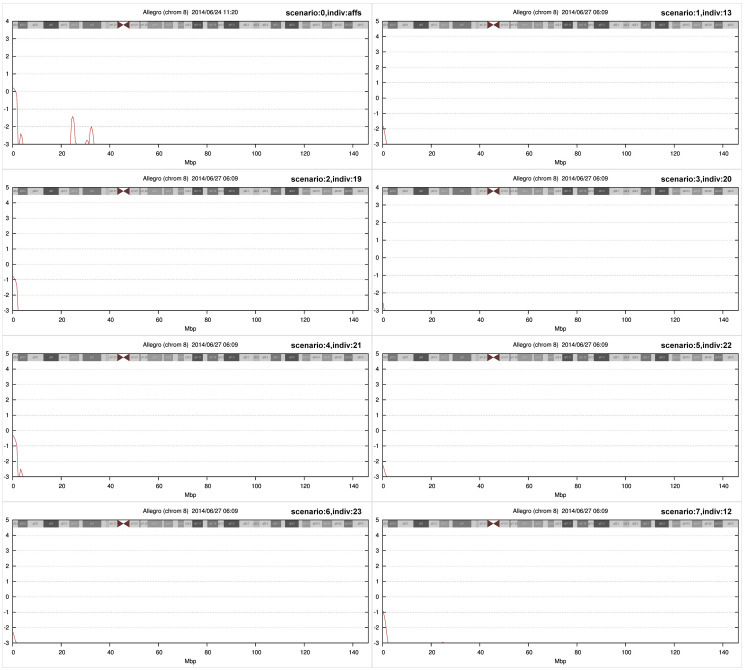

Figure B.11: Linkage plots of chromosome 8 for each of the eight analyses

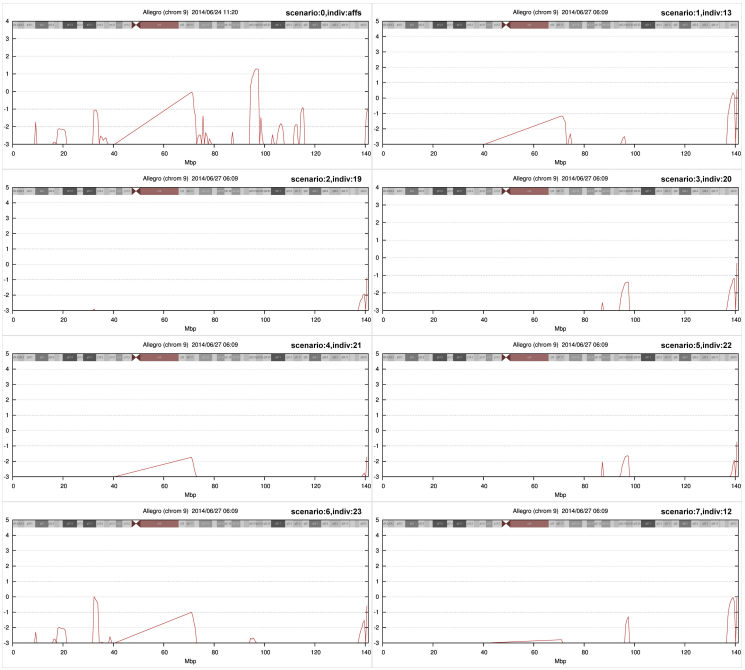

Figure B.12: Linkage plots of chromosome 9 for each of the eight analyses

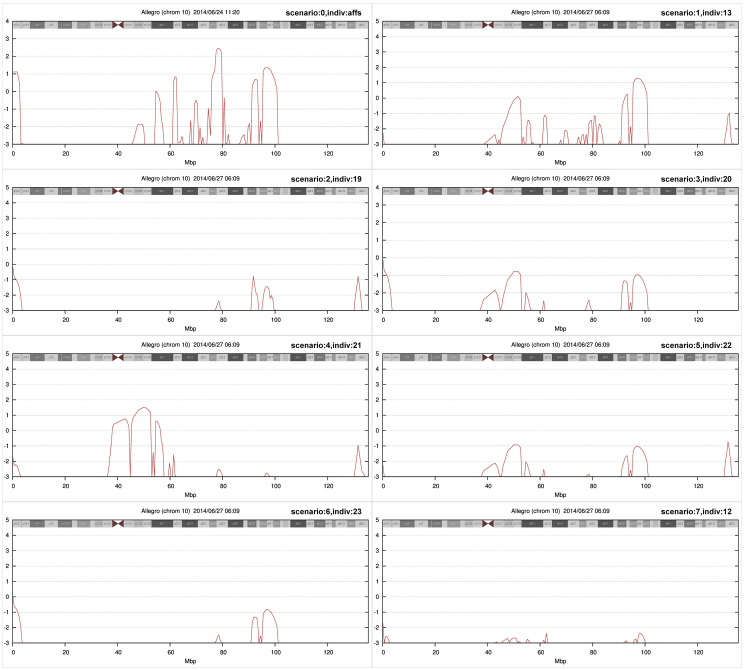

Figure B.13: Linkage plots of chromosome 10 for each of the eight analyses

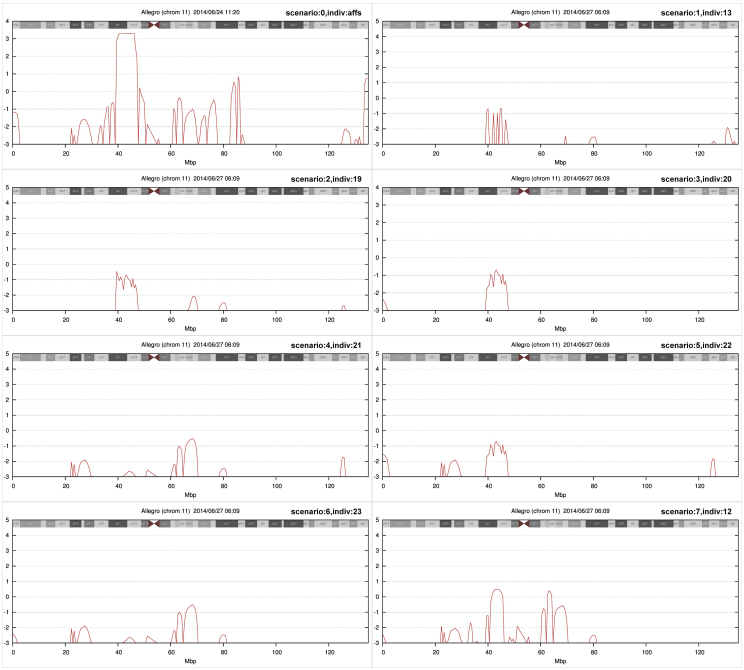

Figure B.14: Linkage plots of chromosome 11 for each of the eight analyses

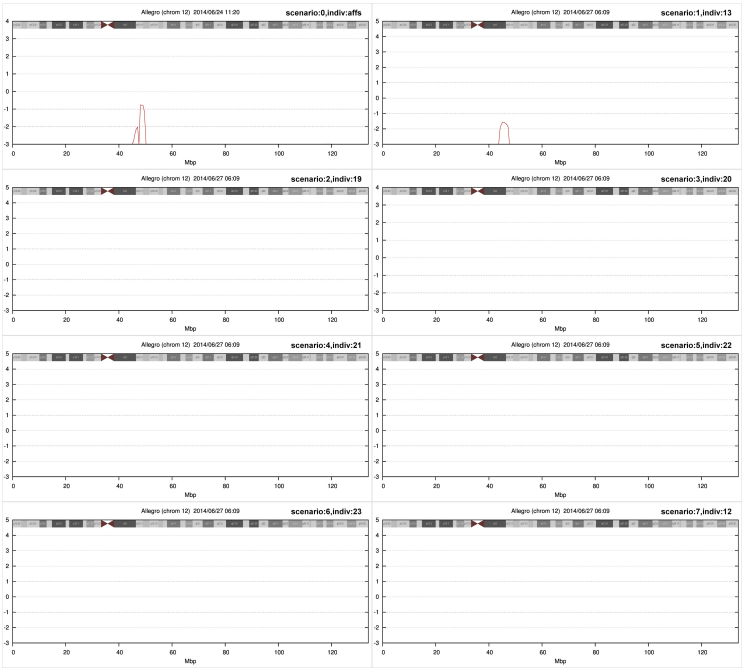

Figure B.15: Linkage plots of chromosome 12 for each of the eight analyses

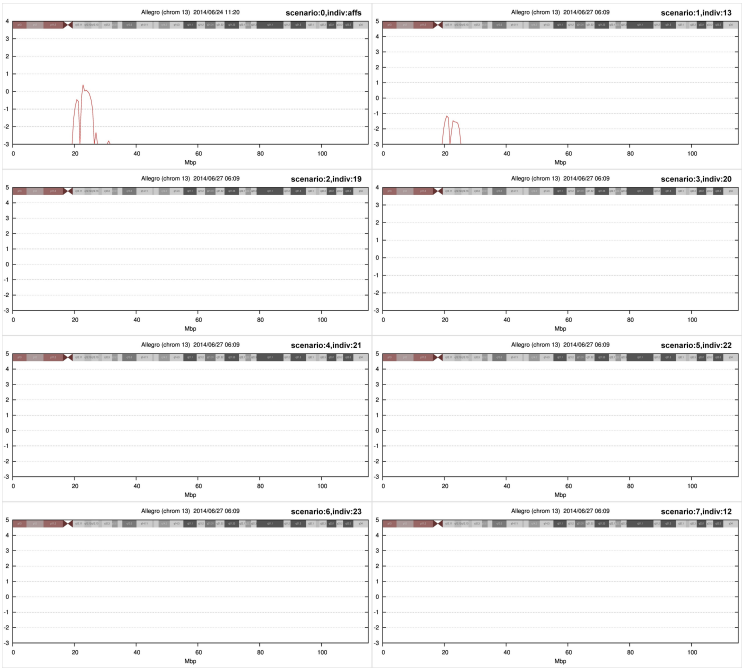

Figure B.16: Linkage plots of chromosome 13 for each of the eight analyses

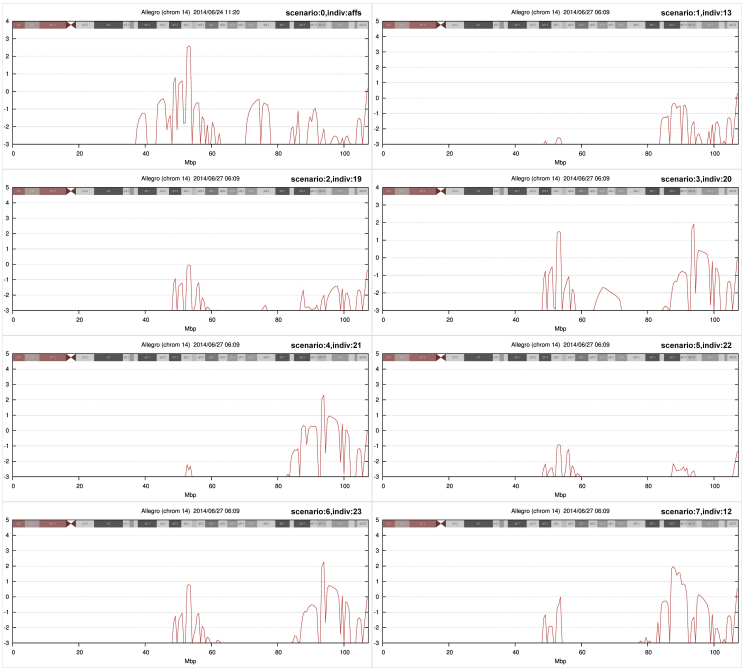

Figure B.17: Linkage plots of chromosome 14 for each of the eight analyses

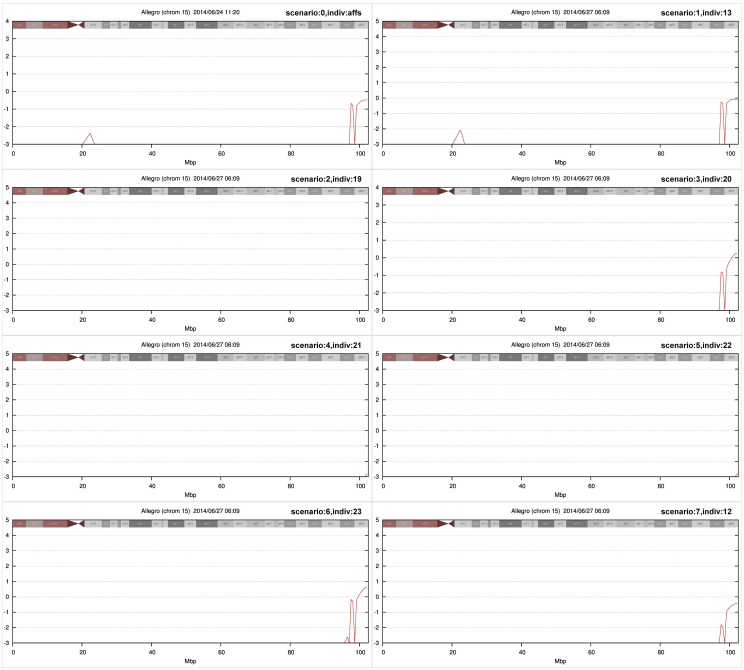

Figure B.18: Linkage plots of chromosome 15 for each of the eight analyses

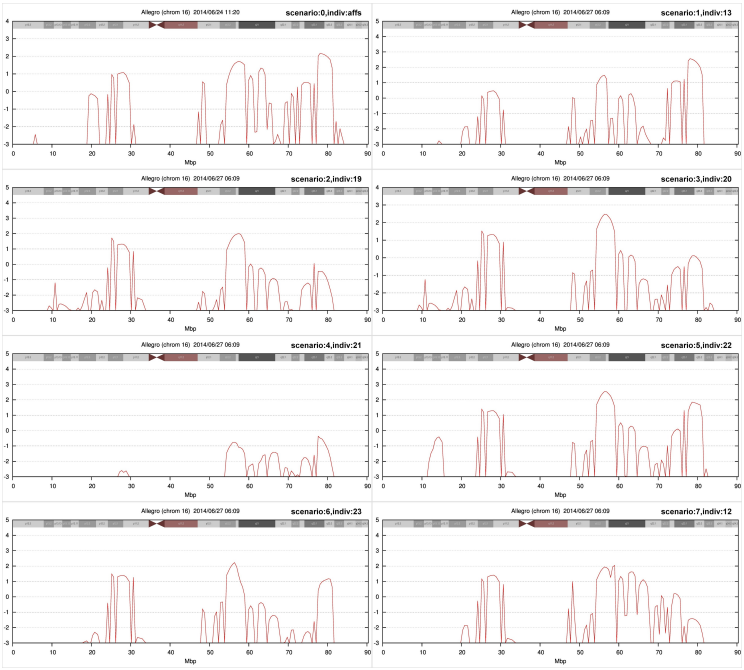

Figure B.19: Linkage plots of chromosome 16 for each of the eight analyses

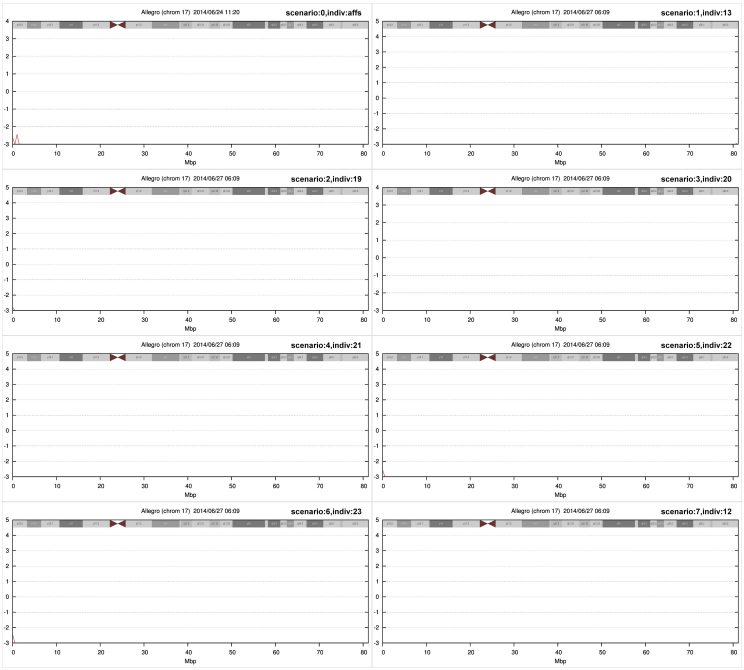

Figure B.20: Linkage plots of chromosome 17 for each of the eight analyses

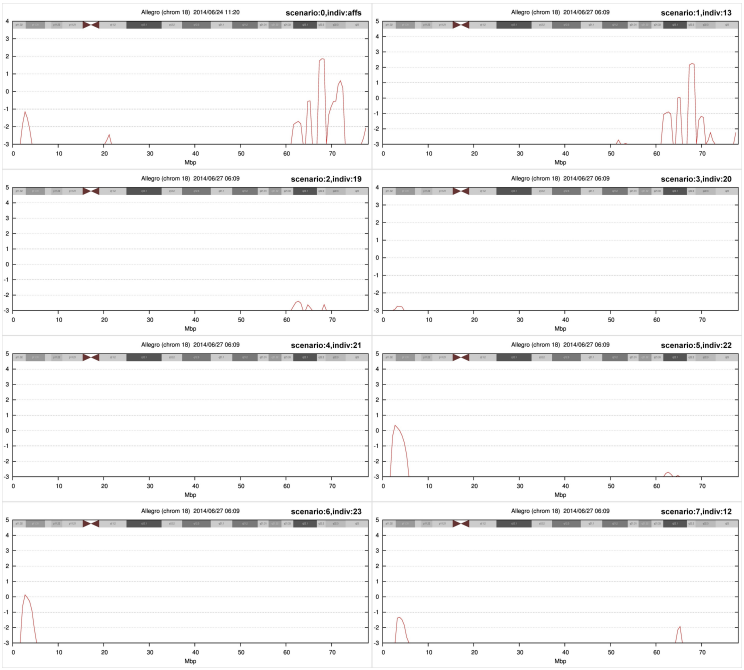

Figure B.21: Linkage plots of chromosome 18 for each of the eight analyses

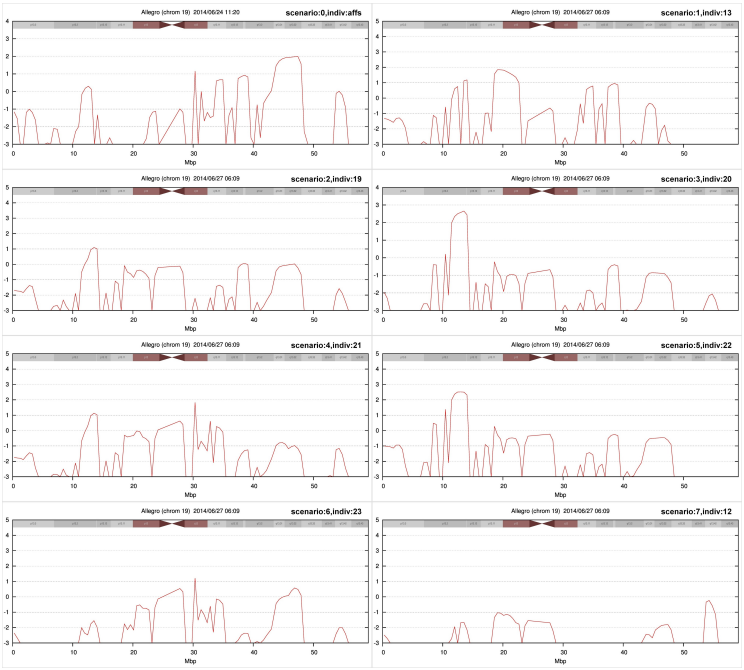

Figure B.22: Linkage plots of chromosome 19 for each of the eight analyses

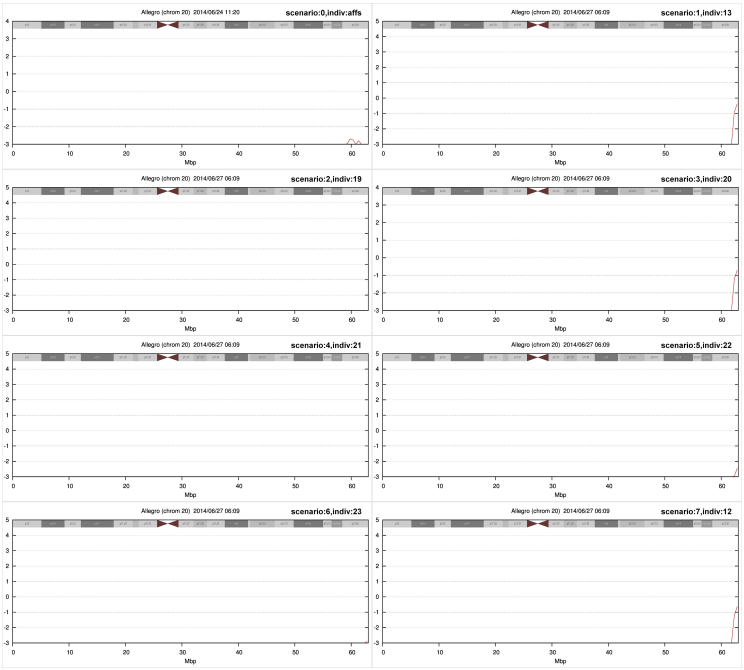

Figure B.23: Linkage plots of chromosome 20 for each of the eight analyses

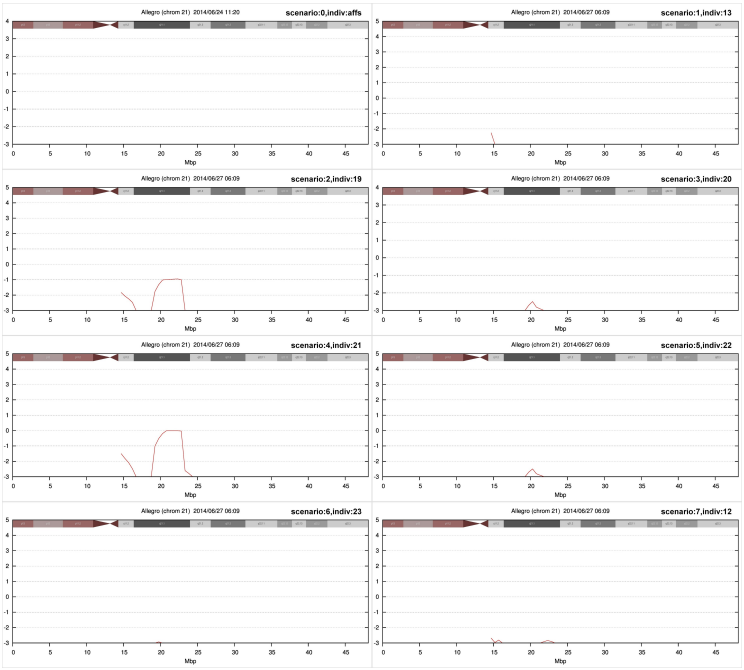

Figure B.24: Linkage plots of chromosome 21 for each of the eight analyses

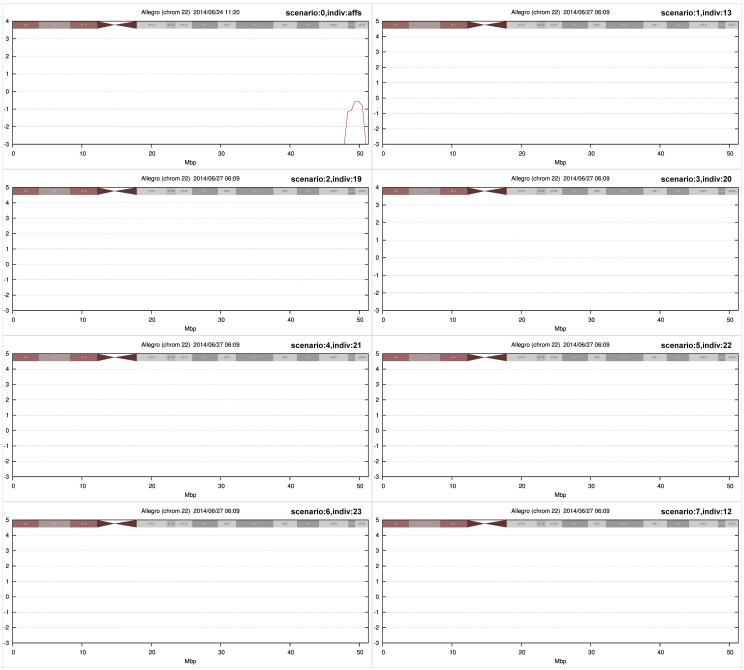

Figure B.25: Linkage plots of chromosome 22 for each of the eight analyses

B.4.3 Allegro Single-Core Individual Runtimes

B	T	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12
3	1	2.85	4.24	3.76	5.40	3.93	5.14	2.46	4.43	2.81	3.94	3.41	2.31
	2	2.86	4.24	3.61	3.66	3.07	4.77	3.76	3.21	3.12	3.25	3.23	4.57
	3	5.05	5.10	3.95	2.97	2.64	2.87	4.59	3.93	2.86	3.23	3.66	5.19
	4	5.11	4.44	5.23	2.73	3.91	2.98	5.25	3.16	2.66	5.20	4.23	2.33
5	1	4.57	2.57	3.71	2.61	2.76	4.66	3.24	4.53	3.26	5.25	4.41	4.57
	2	3.26	4.15	3.17	5.26	2.68	4.67	3.93	2.40	2.91	2.54	5.22	4.81
	3	4.42	5.22	2.73	2.97	4.28	4.92	4.46	3.41	4.30	2.33	2.38	5.22
	4	3.82	4.12	4.92	4.28	5.29	4.64	4.03	4.14	3.09	3.68	4.08	3.63
7	1	11.93	10.09	8.63	8.05	7.34	6.87	6.13	5.85	5.56	5.46	5.25	5.18
	2	13.68	9.96	8.87	7.70	6.93	6.52	6.32	5.81	5.70	5.54	5.48	5.37
	3	12.12	9.90	8.70	8.13	7.49	6.54	6.02	5.81	5.78	5.57	5.29	5.38
	4	13.79	11.11	9.18	7.85	7.48	6.59	6.30	5.83	5.52	5.46	5.40	5.24
9	1	48.27	39.97	30.97	25.45	20.35	16.73	14.16	12.62	12.16	10.50	10.33	10.00
	2	50.70	39.54	29.32	24.52	20.09	17.19	14.55	12.39	11.42	10.45	10.38	9.65
	3	53.85	39.36	29.54	24.94	19.29	16.30	14.64	12.45	11.22	10.78	9.95	9.78
	4	49.85	41.79	29.92	23.71	19.04	16.89	14.08	12.43	11.26	10.98	10.54	9.84
15	1	148.34	105.13	68.43	47.95	36.78	30.35	23.75	20.61	16.81	17.53	14.23	16.52
	2	122.82	104.02	69.36	45.25	33.41	26.57	22.20	22.77	18.15	14.90	14.02	15.68
	3	135.25	92.53	68.35	50.32	35.27	28.39	25.73	20.84	18.92	18.05	15.54	14.17
	4	149.16	94.61	67.91	51.79	36.27	30.59	22.39	21.87	18.88	15.21	16.07	14.51
18	1	560.07	431.28	329.35	241.50	176.33	121.75	102.94	71.74	57.31	48.92	36.20	29.11
	2	627.89	446.07	321.99	224.61	178.18	131.22	102.54	71.22	61.32	50.90	40.56	26.92
	3	615.53	423.45	312.42	234.67	175.54	133.25	97.77	79.71	63.03	45.70	34.87	34.10
	4	577.05	433.55	327.85	230.07	178.88	128.23	95.09	74.16	54.74	44.97	40.81	33.48
21	1	85836.90	85398.51	75700.43	68761.02	64279.13	59391.45	53158.33	48071.74	44633.30	40062.11	36070.67	32189.62
	2	88900.95	80324.77	77500.61	71111.27	63622.33	59746.89	52943.02	48547.76	43530.08	39342.43	35462.19	31780.26
	3	88373.64	85133.93	76832.29	70567.49	64006.13	59564.23	53673.61	48310.10	44602.27	40304.06	35896.18	32041.00
	4	88285.59	84942.15	74659.81	69737.53	64224.72	59093.24	53185.51	49198.28	44563.57	39727.78	35436.96	32172.32
23	1	827976.09	695959.31	589773.77	559135.45	413385.45	479936.78	363794.64	369337.23	280110.95	264597.30	258115.68	200159.73
	2	633775.06	574194.65	741624.34	641498.31	573560.96	442469.95	454363.44	388569.60	281608.17	285781.24	260206.20	233987.01
29	1												

B	T	c13	c14	c15	c16	c17	c18	c19	c20	c21	c22	cX	TOTAL
3	**1**	2.76	2.78	2.71	2.82	4.07	4.24	3.11	4.78	3.68	3.36	2.63	81.63
	2	4.56	3.20	2.20	3.28	2.65	3.05	2.63	2.59	4.06	3.53	5.03	80.12
	3	4.18	4.02	2.26	2.52	4.99	3.04	3.18	3.70	2.11	3.05	3.57	82.66
	4	2.44	2.34	2.36	4.15	3.36	4.60	3.36	2.17	4.99	4.23	2.87	84.10
5	**1**	3.01	3.42	2.79	3.79	4.99	3.49	3.77	2.80	2.83	4.35	5.43	86.84
	2	3.63	3.06	2.79	3.60	4.23	4.79	3.39	2.97	2.60	3.02	4.47	83.55
	3	2.44	3.83	4.56	3.22	4.60	2.85	2.21	2.67	3.60	3.27	3.67	83.57
	4	4.38	2.53	2.22	3.09	3.45	3.64	4.38	4.46	2.60	3.38	3.24	87.07
7	**1**	5.25	5.18	5.13	5.13	5.04	5.01	5.09	5.04	5.08	4.97	9.45	146.72
	2	5.16	5.05	5.05	5.03	5.12	5.05	5.11	4.96	5.09	4.98	9.68	148.15
	3	5.10	5.16	5.18	5.02	5.15	5.05	5.01	5.00	5.08	4.97	10.07	147.51
	4	5.16	5.25	5.15	5.07	5.03	4.97	5.10	4.99	4.99	5.05	10.27	150.77
9	**1**	9.42	9.51	9.51	9.01	8.94	8.87	9.15	8.89	9.08	8.89	35.91	378.72
	2	9.40	9.51	9.53	9.41	9.21	9.06	9.33	9.22	8.95	9.04	37.11	379.96
	3	9.57	9.51	9.51	8.92	8.91	8.99	9.26	8.95	9.16	8.93	34.87	378.67
	4	10.02	9.74	9.18	9.35	9.20	9.22	9.30	8.98	9.26	9.14	33.44	377.17
15	**1**	15.57	13.42	14.54	13.32	15.23	15.04	14.17	13.43	14.63	13.39	86.36	775.54
	2	13.53	13.66	15.55	13.04	14.24	14.98	13.10	14.86	13.47	14.09	85.42	735.08
	3	15.52	14.56	15.56	13.35	13.71	14.54	14.33	14.52	13.66	13.32	75.28	741.71
	4	15.93	14.85	14.99	13.19	14.65	13.16	13.47	14.28	14.97	14.04	85.31	768.11
18	**1**	23.65	26.15	17.29	20.31	19.03	16.03	16.89	15.35	16.08	15.43	386.86	2779.57
	2	25.87	21.45	22.21	15.73	17.73	15.18	15.64	15.45	14.32	12.92	349.63	2809.55
	3	28.62	20.45	19.68	16.09	15.83	14.19	17.27	15.66	17.13	12.98	367.17	2795.13
	4	26.86	25.95	17.15	19.59	19.23	18.23	14.74	14.87	16.98	15.53	350.11	2758.11
21	**1**	28166.11	24758.90	21924.95	19496.51	16871.08	14691.65	12995.20	11064.64	9794.76	8952.61	81621.62	943891.22
	2	28158.18	25376.88	21855.75	19653.58	16835.85	14560.69	12993.42	11369.10	10253.55	9291.82	77823.25	940984.64
	3	28343.88	24658.88	21936.36	19119.59	16844.67	14678.22	12900.36	11165.43	10099.20	9248.76	79814.43	948114.72
	4	28494.78	24802.63	21944.49	19144.36	17181.57	15054.14	12931.26	11466.80	10049.77	8770.75	81447.60	946515.60
23	**1**	169701.00	142038.72	105292.00	78758.72	75864.94	76461.72	27940.45	18866.21	17671.06	37634.77	478507.45	6531019.41
	2	164529.46	154871.93	115034.70	74170.34	87491.93	56257.71	39683.83	20012.55	13424.57	32901.19	510920.27	6780937.40
29	**1**				315615								

Table B.3: Allegro Single-Core Runtimes (4 trials per bit-size per chromosome). B is bit-size, and T is trial number. A plot of these points can be found in Figure 4.24 on page 182.