Research Notebook

Entry 1

Meeting: 5th July 2016, 5-6pm

The first meeting dealt with a small discussion on how the final year project (FYP) is to be put into effect.

This included the fact that a research notebook (this being it) is to be kept as a recording mechanism so as to hold records detailing research ideas, notes on papers and meeting details. It was also mentioned that certain papers will need to be read in order to get familiar with the biological side of things.

For the time being, what needs to be accomplished by the next meeting is simply the reading of the previously mentioned papers which will be provided by the supervisor.

Entry 2

Book Notes: 'Artificial Intelligence and Molecular Biology' edited by Lawrence Hunter; Chapter 1 'Molecular Biology for Computer Scientists'

"Computers are to biology what mathematics is to physics" – Harold Morowitz.

Evolution has three components: inheritance is the main factor in determining an organism's structure and function as the parent organism's characteristics are passed down to the children; variation is the factor which makes child organisms different from their parents; i.e. a source of it must be found in the inheritance (ex. mutation, random changes in inherited material, sexual recombination, genetic rearrangements and even viruses); selection is the factor which favours certain organisms reproducing over others; i.e. it determines which variations will possibly persist. Natural selection is founded on an organism's reproductive fitness (deals with adaption to the environment).

Evolution can be described as a search through a precisely defined space of possible organism characteristics – a single messenger molecule called deoxyribonucleic acid or DNA. This is represented in a simple, linear, four-element code but with a translation which is complex. An organism's genetic encoding is called its genotype while its group of physical characteristics is called its phenotype.

Living things can be found in every type of environment; the same being said for the molecular level of life. Similar organisms can have different chemical compositions and genetic blueprints. All of an organism's genetic material is called its genome. A genome's size is not proportional to the corresponding organism's complexity. The size varies from 5000 elements in a simple organism to more than 10^{11} elements in certain higher plants. A human's genome has around $3x10^9$ elements.

In spite of this diversity, all organisms have a certain unity to them in that they nearly all have the same basic mechanisms – cells. Even the metabolic pathways (the reactions occurring in the cell) are similar across all organisms. The coded genetic material is written in approximately the same molecular language in every organism. Evolution is the reason for

both the unity (due to inheritance from common ancestors) and the diversity (due to variation and selection) of living things.

Recognisable types of plants and animals make up only around 20% of living things. Eucarya (eucaryotes) have cells containing nuclei – a specialised area in the cell holding the genetic material. Other cellular areas called organelles exist, examples being mitochondria (where respiration takes place; i.e. oxygen is used for a more efficient exchange of food into energy) and chloroplasts (where energy is obtained from sunlight). All multi-cellular (animals and plants) and many single-celled (protists and fungi) organisms are Eucarya.

An important claim which underlies the majority of biological theorizing is that all organisms appear to have evolved from a common ancestor. All evolutionary theories state that the variety of life resulted from inherited variance through a constant descendant line. The stronger the relation between two species, the more recent their organisms diverged (the more recent their common ancestor). Knowledge of the DNA sequences of many genes in multiple organisms allows direct estimates of the genetic divergence time.

A typical vertebrate has more than 200 different specialised cell types. However, all the cells in a multi-cellular organism have the exact same genetic code. The differences arise from differences in gene expression, a process whereby gene information is used in a gene product's synthesis.

Most cells have a lot of similar qualities such as genetic material and the ability to translate genetic messages into the protein (main type of biological molecule):

- Proteins are molecules which achieve most of the living cell functions;
- Genetic material is information which is generally stored in long strands of DNA. Note that Eucaryotic DNA is grouped into X-shaped structures; i.e. chromosomes;
- Nuclei contain the genetic material of Eucaryotic cells in the form of chromatin which has a variety of long stretches of DNA bounded by nuclear proteins.

Bio-molecules regulate networks of chemical reactions and include macromolecules (proteins, carbohydrates and lipids) along with multiple small molecules. The cell's genetic material specifies the process of protein creation and the quantity and time it will occur. The resulting proteins control the functions of the cell. The genetic material in question is known as a particular macromolecule, DNA.

The simplest cell has more than a thousand bio-molecules interacting. Humans likely have more than 100,000 types of proteins specified in their genome (present in different cells). The 'machine' language which encodes a set of instructions describing living systems' objects and processes contains four letters, with the text describing a person having around $3x10^9$ characters.

Materials found in the environment of an organism are broken down and reassembled into another organism following the instructions in the genome. The child organism will have instructions similar to the parent. The basic units of matter are proteins; the basic unit of energy is a phosphate bond in the molecule adenosine triphosphate (ATP); the units of information are four nucleotides which are combined into DNA and RNA.

Approximately 70% of any cell is water. Around 4% are small molecules like sugars (one being ATP). Proteins put together 15/20% and DNA & RNA make up 2-7%. The remaining 4-7% contains cell membranes and other similar molecules.

Proteins have many roles, one of them being of switches which control whether genes are turned on or off. They are made up of amino acids. The protein's primary structure is made up of a sequence of amino acid residues and is directly coded for in the genetic material – the individual elements of a DNA molecule form triples, unambiguously specifying an amino acid. A genetic sequence maps directly into a sequence of amino acids.

Nucleic acid control biochemical action. All genetic information is stored in deoxyribonucleic acid (DNA) and ribonucleic acid (RNA), which are polymers of four simple nucleic acid units called nucleotides. There are four nucleotides found in DNA, each one consisting of a purine (adenine A or guanine G) or a pyrimidine (cytosine C or thymine T), a sugar (deoxyribose in DNA and ribose in RNA) and one or more phosphate groups. The length of a DNA sequence is measured in kb (thousands of bases). Nucleotides are also called bases and so, since DNA consists of two complementary strands bonded together, these units are called base-pairs.

Nucleotides are normally abbreviated by their first letter and appended into sequences; example: CCTATAG. They are linked to each other in the polymer by phosphodiester bonds; the bond being directional with a strand of DNA having a head called the 5' end and a tail called the 3' end. DNA forms a double helix; two helical/spiral-shaped strands of the polypeptide running in opposite directions and held together by hydrogen bonds. Adenine binds solely with thymine (A-T) while guanine bonds solely with cytosine (G-C). Due to these bonding rules, although one strand's sequence is completely unrestricted, the complementary strand's sequence is completely determined. This property allows high consistent copying of DNA information and transcriptions into complementary strands of RNA which directs protein synthesis (with uracil U instead of T).

DNA can take other forms apart from the double helix B-DNA. It can become Z-DNA, for example, whereby it reverses its twist direction. Such alternative forms may turn certain genes on or off.

An organism's genetic information can be kept in one or more DNA molecules/chromosomes. Diploids are sexually reproducing organisms with two alike DNA molecules physically bound together (one from each parent) in their chromosomes while haploids are similar but with single DNA molecules in their chromosomes. Humans are diploid having 23 pairs of linear chromosomes. All the genetic information of an organism is referred to as its genome.

In DNA (for example the codons GCT and GCG), most codon synonyms vary in the last nucleotide only. This is referred to as the degeneracy of the code. The code is strongly conserved over evolution however there are still differences in codon to amino acid translation when comparing various organisms; in fact, there are a few systems which use a slight different code. Codons come in triples and thus the parsing of a segment of DNA (the synthesis of protein maps from codon sequences to amino acid sequences) can start at three possible places. This problem can be compared to the decoding of an asynchronous serial bit stream into bytes where each of the parsings is called a reading frame; an open reading frame (ORF) is a parsing with long string of codons having no interfering stop codons.

It is also possible to read a DNA sequence off either strand of the double helix. The second strand is the complements of the first so a sequence can be read inverted and in the opposite direction; i.e. reading from the antisense or complementary strand. This type of message can also be parsed three ways resulting in a total of six possible reading frames for every DNA sequence.

A DNA sequence which codes for a single protein usually has introns (inserted), noncoding sequences, inserted. They are removed before the sequence is mapped into amino acids. The DNA segments which do code for the protein are exons (expressed). DNA, in addition to protein coding, holds a lot more information as every cell in the body has the same DNA but each cell type generates a different set of proteins.

Every cell has the same DNA but they code for different proteins. This is due to the difference in the regulation of the genetic machinery – the regulatory mechanisms of the process of protein coding compose a parallel system with multifactorial feedback and control structure.

Genes are either expressed/not expressed (on or off). The process of production is controlled by a collection of proteins in the nucleus of eucaryotic cells which affect which genes are expressed. Histones are proteins which are tightly bound to the DNA in eucaryotic chromosomes and are some of the most conserved proteins in life despite billions of years of divergence in their evolution. Topoisomerases are proteins which rearrange and untangle DNA and are the next prevalent proteins in chromosomes. Most regulatory proteins recognise and bind to specific DNA sequences called control regions (border the protein regions of genes). Promoters are sequences occurring towards the 5' end (upstream) of the region which encourage protein production, enhancers are similar sequences occurring either downstream or relatively far upstream of the region while repressors are sequences which tend to prevent protein production.

Molecules which are related are called homologous. The following are some aspects of molecular evolution: point mutation is the change of a single nucleotide in a genetic sequence; gene duplication is a chromosomal rearrangement where additional copies of a gene are inserted into the genome; pseudo-genes are similar to actual genes but they are not expressed; crossover is a process where the DNA from the parents of a sexually reproducing organism forms a type of combination which is passed to the child organism.

Most mutations have little effect. For example, mutations in the third position of most codons have little effect at the protein level due to genetic code redundancy. Neutral mutations are the support of genetic drift – the phenomena accounting for the DNA differences for functionally identical proteins in different organisms. On the other hand, some point mutations may lead to death or diseases; for example: cystic fibrosis. It is very rare that a mutation end up being advantageous.

Some areas of biomedical research involve humans directly or by just their cells which are grown in the laboratory, but it should be noted that not many human cell types can live outside the body. Human cancer cells can and so they are an important research tool.

Nearly every aspect of human biology can be correlated in some organism. The following six are the main models in molecular biology: *escherichia coli* (bacterium) – other organisms'

genes are inserted into its genome to produce the mentioned genes in quantity; *saccharomyces cervesiae* (eucaryotes) – the yeast helps in making copies of moderate-sized pieces of DNA using yeast artificial chromosome YAS when sequencing large amounts of DNA; *arabidopsis thaliana* (common weed) – good model for higher plants with a genome with very little repetitive DNA; *caenorhabditis elegans* (worm) – one of the simplest creatures with a nervous system allowing tracing of genetic mutation effects; *drosophila melanogaster* (common fruit fly) – used in genetics research, mainly in genetic expression and control as well as genetic programs specification; *mus musculus* (basic laboratory mouse) – identical to people in terms of biochemistry with a relatively large genome.

A group of cells having identical genomes are clones. Identical child organisms are also clones but are sometimes called cell lines. Restriction enzymes are naturally produced by bacteria to attack foreign DNA; biologists use them to cut and paste specific DNA fragments into vectors. This is only effective a fraction of the time however; as cells and vectors are small and easy to grow, the process can be applied to many of them.

Hybridisation, a technique which measures the similarity between two related DNA sequences, tests how strongly the single-stranded versions of the molecules stick together; i.e. hybridise. The more easily they come apart, the larger the amount of differences there are between their sequences.

The Human Genome Project is the effort to produce a map and then the sequence of the human genome. A genetic map's purpose is the identification of the location and size of all of the genes of an organism on its chromosomes. Linkage analysis is a procedure which looks at genes' relationships (phenotypes) in large numbers of matings (crosses) to identify which are generally inherited together and thus, more likely to be near each other. It is possible to determine a medium-sized piece of DNA's sequence and thus, if a gene has been mapped its area sequence can be found and as such even the protein responsible for the genetic characteristics (this relates to inherited diseases.)

The ability to divide the genome is a requirement to determine its sequence. Thus, by taking this sequencing ability to sequence many different overlapping pieces and assembling them, the sequences of large pieces of DNA can be determined. The ordering of the pieces must be known and together, they must cover the entire genome. This process may involve polymerase chain reaction PCR. PCR makes possible the rapid production of large amounts of a specific region of DNA. It exponentially amplifies entire DNA molecules or regions of it for which bracketing primers (short pieces of DNA with a specific sequence) can be generated. A collection of short (easy to synthesise), unique (unambiguous DNA) sequences spread throughout the genome must be identified to be used as primers in order to use PCR for genome mapping and sequencing. The genome sites corresponding to these sequences are sequence tagged sites STSs. The more STSs known, the finer grained the genome map provided.

An early aim of the Human Genome Project is the production of a list of STSs spread at around 100kbp intervals over the whole human genome. Any DNA region can be identified by its two bracketing STSs. These STSs can then be stored in a database to maintain large clone collections. The project may also require sequencing of all the introns and other non-coding regions of DNA. One can target only coding regions for sequencing and thus be able to identify sequences used by a cell to produce proteins at a point in time. Thus, attention can be focused on the genome parts coded for expressed proteins.

There are several databases which maintain genetic sequences: Genbank, the European Molecular Biology Laboratory nucleotide sequences database EMBL and the DNA Database Japan DDBJ. The main protein sequence database is the Protein Identification Resource PIR. There are also several databases which maintain three dimensional structures of molecules: the Protein Data Bank PDB, BioMagRes BMR, CARBBANK, Chemical Abstracts Service (CAS) Online Registry File and Cambridge Structural Database. Genetic map databases (GDB) and a database of inherited human diseases and characteristics (OMIM) are maintained at the Welch Medical Library at Johns Hopkins University.

Entry 3

Article Summary: 'The Hidden History of the Maltese Genome', Think Magazine, Issue 16

One of the largest research groups in Malta has recently discovered that certain Maltese families have a mutation. This resulted in a master regulator which could help bone marrow alleviate haemoglobin switching and the blood disorder thalassemia. Another research area the group is looking into is heart disease – the Maltese Acute Myocardial Infraction (MAMI) study – as Malta's morality rate is higher than the European average. Its focus is finding the genetic component behind three key heart-disease related problems.

Tens of Maltese people have already been partially sequenced; the plan being to map the genomes of approximately 1% of the population (4,000 Maltese people). Accomplishment of this would lead to Malta being one of the best genetically documented countries globally. Presently, a public health genomics database, a bio-bank and the current Maltese population's origins have been implemented and become known.

The Malta Human Genome Project (MHGP) consists of the Malta Council for Science and Technology in the Health & Biotechnology sector for funds, the University of Malta for the research consortium lead and Mater Dei Hospital, Erasmus MC, Rotterdam, The Netherlands and Complete Genomics Inc., California, Silicon Valley, USA for partners.

The reading of someone's DNA shows the likeliness of someone developing a disease and their relatedness with others. The reading of a nation's DNA shows why that population is more likely to develop a disease or how that population came to exist.

The Human Genome Project's aim was to decode the sequence of human DNA in order to better understand human biology and evolution. Another result was the development of DNA sequencing technologies; the newest being known as Next Generation Sequencing which allows a small group of scientists to sequence one person's genome in a few weeks for approximately $1000 (the low price encouraging innovation).

99.9% of every human's DNA sequence is the same and only 0.1% is different, accounting for the variations or mutations of a person. As every human is genetically different, so are large groups of populations. When a particular population's genetics are studied, the current data on the human genome does not seem adequate and so many countries are initiating their own genome projects – Malta now being one of them.

The three-year Maltese Genome Project was launched in 2015 and is based on approximately 25 years of human genomic research in Malta. The end result should be an

average/representative example of the entire Maltese population which will help the diagnosis of rare diseases and the investigation of new therapies; in particular how gene variants affect the Maltese population.

A human can equate to 200-400 Gb of raw data from the DNA acquired and passed through a sequencing machine, where the DNA fragments are copied and monitored in order to determine the original sequence. This is then aligned to a reference genome to find any gene variants or mutations which are specific to the Maltese population along with information regarding Maltese origins. It is of course important to confirm the results as the machine is not 100% perfect; although errors have been minimised.

Evolutionary genetics studies rely on two genetic markers:

- Mitochondrial DNA – needed to live. It is inherited from one's mother and only passed on by daughters so researchers can trace ancestry through the female lineage due to haplogroups;
- Y chromosome – Human DNA is made up of 46 chromosomes, with each parent contributing half. Gender is determined by the X and Y chromosomes; XX = female and XY = male, the combination depending on the father. It is useful for evolutionary studies on men's origins as it also has haplogroups (specific parts of mitochondrial DNA).

From mitochondrial DNA analysis, humans left East Africa as a small group of males and females. The first Maltese humans are believed to have been Sicilian farmers. Certain revelations about Maltese origins were published in 2004 by the Annals of Human Genetics, situated in Malta, resulting in the fact that contemporary Maltese come from just a few hundred miles north. This resulted from observing Y chromosome haplogroups retrieved from around the Mediterranean and identifying common population groups. The nearly complete mitochondrial DNA data gathered for the Maltese Genome Project seems to point to the same result – that most contemporary Maltese originate from Sicily and southern Italy of about a thousand years ago.

The original Maltese population was visited by many small groups over the centuries and in effect, left behind gene variants and mutations; i.e. Founder Effects. These newly introduced Founder Mutations spread across the population as it expanded and mixed with genomes from faraway countries. The population grew exponentially.

The altering population numbers over historical events created genetic bottlenecks and impacted on genetic diversity in such a way that DNA mutation led to rare diseases; most of the Maltese genetic mutations being shared with Sicily and southern Italy. Some of these DNA mutations which lead to rare diseases are disproportionately high in the Maltese population; some being: gangliosidosis, coeliac disease and blood disorders likes thalassemia.

A 2007 study focused on a mutation in the SPR gene which leads to a rare disorder known as Segawa's Disease (a motor neuron disorder similar to Parkinson's disease). A high proportion of the Maltese population were found to have this single mutation, resulting in diagnosis and treatment at birth to prevent severe disability.

Research and diagnostics will be moving to whole genome sequencing. The Maltese Genome will determine the treatment of diseases widespread locally and help others worldwide. A complete picture of Maltese origins to today will be formed.

Entry 4

Paper Summary: 'Fast and accurate mapping of Complete Genomic reads' by Donghyuk Lee, Farhad Hormozdiari, Hongyi Xin, Faraz Hach, Onur Mutlu, Can Alkan

High throughput sequencing (HTS) sequences large groups of human genomes; the problem lying in the read mapping stage where, although tools for the most used HTS methods have been developed, open source aligners are still deficient for the Complete Genomics (CG) platform. It provides paired-end reads of 29-35 base-pairs (*bps*) and a fragment size of 400-1500 bps like other HTS technologies with the difference that a CG read does not have consecutive bases but instead it is composed of multiple sub-reads which may either overlap or have gaps between them

Burrows-Wheeler Transformation with FM-indexing (BWT-FM) is thus not practical for CG data due to its gapped nature and its inability to scale well with indels (**in**sertion or **del**etion of bases in the DNA of an organism). For this reason a sensitive read mapper sirFast was developed to align these reads to the reference assembly using a hash based seed-and-extend algorithm, supporting both SAM and DIVET file formats.

A seed-and-extend mapper first chooses the seeds and then checks their genome locations using a hash table. An extension step called verification, where the similarity score is measured between the read and reference, is taken where a positive score is dealt due to insertions, deletions or substitutions while a zero score is kept by each matching base; the smaller the score, the higher the similarity between the read and reference. The mentioned alignment algorithms (bp-granularity mapping algorithms) are from dynamic programming with the verification step complexity per location being quadratic. This complexity may be reduced to O(kn) if k is an upper bound for allowed indels.

The mappers' performance and accuracy is dependent on how the seeds are selected in the first stage and on the seed type; spaced seeds are not suitable for CG reads so consecutive seeds are used with length 10 and seed index via a hash table. The concept of combined seeds is implemented; its main benefits being the reduction of potential locations and of flexible-sized expected gaps (to decrease mapping time and overhead). The selected seeds then search for read potential locations which are verified using an alignment algorithm – Levenshtein, Smith-Waterman or their variants. The high number of expected gaps leads to the usage of Hamming distance.

Both simulated and real data sets were used to assess method performance. When mapping reads independently sirFast showed full map-ability and a potential use for duplication detection. When implementing paired-end read mapping and precision/recall tests sirFast showed high precision and recall in paired-end mode. When mapping a real dataset and comparison against the CG mapper the reference human genome assembly was used to show that the CG mapper is more error-prone when compared to sirFast.

Theoretically, full dynamic programming requires O(n^2) run time and the combined seed method takes O(kl) where k is the total gap size and l is the seed size. Also, hash tables are used for seed queries as they perform in O(1) time while suffix index binary search takes O(log n) time.

Therefore, the cost of generating CG data is relatively low. However the lack of analysis tools limit research in proprietary algorithm discovery among others. Thus, sirFast should improve the variation detection mechanism for the CG platform.

Entry 5

Notes: 'Algorithms for DNA Sequencing' by Ben Langmead from Johns Hopkins University; Online Course

1. Why Study DNA Sequencing and Computational Genomics?

The field of DNA sequencing is where computer science and life science intersect – computer science is applied to DNA sequencing data to study rare genetic diseases, human origins and evolution, cancerous tumours, microbes and bacteria or even basic genome workings.
The field of computational genomics relates algorithms to their success/failure and so understanding of what they can do is important. No large-scale project in this field does not employ computer scientists as such developed algorithms are crucial.
The algorithms and data structures studied also apply to information retrieval and natural language among others involving large quantities of text.

2. DNA Sequencing Past and Present

First generation DNA sequencing was brought forth by Fred Sanger in the 1970s using Chain Termination Sequencing (or Sanger Sequencing or Burst Generation Sequencing). It remained the predominant method along the Human Genome Project, till around 2001.
In the year 2007 another sequencing technology was brought to light, going by the name X Generation Sequencing or Second Generation Sequencing. It is massively parallel (able to sequence billions of DNA at the same time) and so it is also relatively inexpensive. Speed, accuracy and ease of use also proved to be beneficial factors in studying DNA and RNA. Generally petabytes of data is being sequenced annually.

3. DNA: the Molecule, the String

DNA is the molecule which codes your genome; i.e. all the genetic information/genes which maintain your body and which are encoded in a language of the letters A, C, G and T. These letters represent different types of molecules/bases: A – adenine, C – cytosine, G – guanine and T – thymine.
A DNA molecule is shaped like a double helix made up of the complementary base pairs A and T as well as C and G. In order to write down the sequence of bases which describe a particular molecule, a top-down approach can be taken resulting in a string. Human chromosomes are on the order of hundred millions of bases.
DNA sequencers read multiple short stretches of DNA to make sense of the whole long piece. This is done by repeatedly reading off randomly selected substrings/snippets from the middle of the input DNA called sequencing reads. Second generation sequencers produce reads of around a hundred bases long – they are many orders of magnitude shorter than the input DNA. But as multiple reads are generated, the whole genome is covered many times over; i.e. redundant information about any given base of the genome being sequenced is received.

8. How DNA is Copied

The way in which second generation sequencers work is a process called sequencing by synthesis.
When a cell in a human body divides to form two daughter cells, the genome in the cell gets copied and so almost every cell has a copy of the human's genome. The starting point is the double stranded DNA (double helix), which gets split into two single stranded,

complementary molecules of DNA; each strand still having the genome sequence written on it and acts as a kind of template to form back the double stranded DNA.
The enzyme which places the complementary base in its place is called the DNA polymerase – a mechanism which, when given one of the strand templates and a base, will construct the complementary strand resulting in a double stranded copy of the template strand.
If this is done for both of the single stranded DNA, the result is that of having two double stranded copies of the original DNA.

9. Sequencing by Synthesis

How does a sequencer eavesdrop on the copying process in order to sequence multiple templates simultaneously? Input DNA is cut into snippets, formed as billions of single stranded templates and randomly deposited on a flat surface such as a slide. These templates are the molecules to be sequenced where one sequencing read per template will be obtained.
Some DNA polymerase and bases (raw material) are then added. Each base is terminated – they have a terminator section attached to them which prevents another base from being placed on top of it in order to keep reactions in sync. Thus the polymerase will only add a single terminated complementary base for each of the template streams.
A snapshot of the top view is taken where the terminated bases are engineered to glow a particular colour, the colour corresponding to the base. The camera will pick up the light emitting from the terminators of all templates simultaneously; i.e. massively parallel. This tells us which base was added to which template strand.
Next, the terminators are removed and some DNA polymerase and terminated bases are added again. The whole process is repeated – a sequencing cycle – until the complementary strands are formed, inferring the sequence of the templates.

10. Base Calling and Sequencing Errors

Sequencers can of course make mistakes and convey uncertainty.
Before adding any bases or polymerases to the slide, the template strands are amplified by making multiple copies of them where all the clones are clustered around the original strand. The clusters of clones are needed instead of the individual templates because otherwise, when a snapshot is taken, there would not be enough light coming from a single template.
However, a problem can occur where an un-terminated base is added causing that strand to be ahead of schedule, i.e. that template is out of sync. A snapshot would display this as two colours would to be able to be seen instead of one. Further spuriously un-terminated bases could be integrated showing that more sequencing cycles tend to correlate to more out of sync strands.
A piece of software analyses these images and tends to figure out what all the bases are – a base caller. It sometimes has to deal with ambiguity, affecting its confidence for that particular image. For each base call, the base caller reports an important value called the base quality – the base caller's estimate of the probability that the base was called incorrectly.
The following equation is used to calculate the base quality $Q = -10\log_{10}p$ where p is the probability. This expression is used as it makes for easier interpretation; i.e. $Q = 10$ implies a 1 in 10 chance of the call being incorrect, $Q = 20$ implies a 1 in 100 chance, $Q = 30$ implies a 1 in 100 chance, etc…

11. Reads in FASTQ Format

FASTQ is the typical file format for sequencing reads of DNA. It contains four lines of information – the first line contains the name of the read and some further information, the second line is the sequence of bases as reported by the base caller, the third line is a placeholder and the fourth line encodes the sequence of base qualities for the corresponding

bases in the second line. A typical FASTQ file involves a set of records made up of these lines and concatenated together into one big file.
The characters in the base quality line match up with corresponding characters in the base sequence line. Each base quality is an ASCII-encoded representation of Q; the higher the value of Q, the more confidence there is that the base is correct. In other words, a character is being used to encode a number.
Phredd33 is a method of converting Q to its corresponding character by rounding the Q off into the nearest integer, adding 33 to it and then converting it to the corresponding character according to the ASCII table.

14. Sequencers Give Pieces to Genomic Puzzles

How is sequencing data analysed? This question cannot be answered just by looking at the reads as they are far too short – a single read isn't even long enough to cover a 'G' in its entirety. So the reads have to be stitched back together somehow so that the sequence of the input DNA can be inferred.
Unrelated humans have genomes that are 99.8-99.9% similar – one or two differences every thousand bases or so. Thus, each genome can be used as a template to help put together the required genome. Multiple reference genomes like this for different species exist.
The problem can be referred to as the Read Alignment Problem – given a sequencing read and a reference genome, how do we find where the read best matches/aligns most closely to the reference genome? Analogously, it can also be referred to as the Assembly Problem – if a reference genome is not available, how would the closest alignment be found?

15. Read Alignment and why it's hard

Different individuals of the same species have similar genome sequences and so this is used in the read alignment problem. A process is repeatedly gone through, once for every read in the dataset, where a sequencing read and a reference genome are compared in order to find the closest match between them. A second generation sequencer outputs on the order of billions of sequencing reads. The human reference genome is about three billion bases long and its printed version is found at the Welcome Collection in London.
Both the sequencing read and the reference genome can be thought of as strings. This is advantageous due to the large amount of data structures and algorithms available to manipulate strings – it is still an active area of research impart due to the emerging problems in genomics.

16. Naïve Exact Matching

The Exact Matching Problem is a real computational problem, used to try and solve the read alignment problem, where all the offsets of some pattern string P which occur within a longer string of text T are wanted to be found (the offset is the left-most occurrence of P within T).
Referring to the naïve algorithm implemented in Python:
Example: Let $x = |P|$ and $y = |T|$. How many alignments are possible given x and y? The answer is $y – x + 1$.
Example: Let $x = |P|$ and $y = |T|$. What's the greatest total number of possible character comparisons? The answer is $x(y – x + 1)$. This is the worst case analysis. When would this happen? The answer: when every character in P matches every character in T; every character comparison results in a match and so the inner loop is iterated over for the maximum number of times.
Example: Let $x = |P|$ and $y = |T|$. What's the least total number of possible character comparisons? The answer is $y – x + 1$ as for each character alignment, at least one character

comparison is done but the first comparison could be a mismatch. When would this happen? The answer: when the first character in P doesn't occur anywhere in T.
Example: How many character comparisons occur with P: word and T: There would have been a time for such a word? The answer is 40 mismatches + 6 matches = 46 comparisons. This is much closer to the minimum of 41 then the maximum of 164.
In practice, the number of character comparisons done is usually closer to the minimum then it is to the maximum.

19. Boyer-Moore Basics

The Boyer-Moore algorithm, also used to try and solve the read alignment problem, is similar to naïve exact matching with the difference that it skips alignments which it does not need.
Example: P: 'word' and T: 'There would have been a time for such a word'. u does not occur in P so the next two alignments can be skipped.
This can be generalised into a principle which can be used in multiple situations – will learn from the character comparisons done in order to skip alignments which probably won't result in a match. Boyer-Moore uses this principle to the maximum possible extent.
Boyer-Moore tries alignments in left-to-right order as done in naïve exact matching but tries character comparisons in right-to-left order.
Another component of Boyer-Moore is the Bad Character Rule – upon mismatch, alignments are skipped until either the mismatch becomes a match or P moves past the mismatched character. (Shifting a character by n implies skipping n-1 alignments).
The last component of Boyer-Moore is the Good Suffix Rule – let t be the substring matched by the inner loop and then skip until either there are no mismatches between P and t or P moves past t.

20. Boyer-Moore: Putting it all together

In practice, both rules of Boyer-Moore are implemented together. So when a mismatch occurs, both rules are tried; each rule returning the amount to shift P/the number of alignments to skip. The maximum of the two is taken.
Boyer-Moore is advantageous over naïve exact matching as even in the best case, the latter will never skip an alignment and so will never fail to examine each and every character. Thus, in practice, Boyer-Moore is expected to be substantially faster.
In order to use the rules, the ability to look up how far the rules tell us to skip is needed; pre-processing. Boyer-Moore implementations build some look-up tables for both rules beforehand so that every time a rule is applied, the number of alignments needed to be skipped is simply looked up in the required table. The only information needed to build these tables is P.

21. Diversion: Repetitive Elements

A human genome sequence is the results of a complex evolutionary process which tends to introduce certain patterns into the genome (which would not have been possible if generated randomly). It is extremely repetitive. Over time, the genome gets invaded and infiltrated by little pieces of DNA called transposable elements which are capable of cutting and pasting themselves inside the genome when the conditions are right. This has occurred so many times that approximately 45% of all the bases of the human genome sequence are covered by transposable elements.
There are many different kinds of transposable elements. A particularly renowned example is the alu element which occurs more than a million times in the genome – about 11% of the human genome sequence is covered by these.

These repetitive portions of the genome do effect whether algorithms for the read alignment problem and the assembly problem are well designed. This is due to the fact that they create ambiguity – there is inherit ambiguity – and so problems will be created for the algorithms.

23. Pre-processing

The cost of pre-processing the pattern P is something that can be amortised over many problems – it might be costly to do once but because it is used repetitively, the cost is made up for over time.

The idea of pre-processing the text T sounds to be a lot more work as it can be billions of letters long. However, as said, the cost can be amortised over many problems. If the situation involves having to solve many matching problems where T is the same but P can vary, it may be worthwhile to pre-process T.

An algorithm that uses a pre-processed version of T is called an offline algorithm whereas an algorithm that does not pre-process T is called an online algorithm. The definition stays the same regardless if P was pre-processed or not.

Example: Is the naïve exact matching algorithm an online or offline algorithm? Answer: online as no pre-processing is done.

Example: Is the Boyer-Moore algorithm an online or offline algorithm? Answer: online as it pre-processes only P.

Example: Is a web search engine an online or offline algorithm? Answer: offline as it needs to pre-process the WWW; otherwise every search would involve a new scan of the WWW and would slow things down.

Example: Is the read alignment problem an online or offline algorithm? Answer: offline as many sequencing reads need to find where they originate with respect to the same reference genome.

24. Indexing and k-mer indexes

To pre-process a long text T one can use an index – querying the index – in a similar manner to a book index with alphabetically ordered key terms and grocery store aisles with grouped items. These highlight the basic tools for organising data.

The building of an index for a DNA string – indexing DNA – will be more similar to the index of a book as the idea of ordering is implemented. However, keep in mind that a genome does not consist of words as it is one long string so we need to break it up into words ourselves. One way to do this is by taking every substring having a particular length of T and treating them as words.

Example – T: 'CGTGCGTGCTT' with substrings of length 5 will produce the following index with the substring's associated offset/s in the text (alphabetically ordered):

CGTGC: 0, 4
GCGTG: 3
GTGCG: 1
GTGCT: 5
TGCGT: 2
TGCTT: 6

The term k-mer is used to refer to a substring of length k. Thus, the above example is a 5-mer index.

The querying of the index makes use of a pattern string P. With the above 5-mer index and taking P: 'GCGTGC', the exact matching problem can be solved – all the offsets where P occurs in T can be found as such (finding the index hits):

GCGTG: found at offset 3

To see if the rest of P matches within T, another character comparison has to be done – verification. In this case, the verification step succeeds as C matches. Thus, it can be concluded that P occurs within T at offset 3.
A variation on this is the case that we take the second 5-mer from P (CGTGC) instead of the first. As the index contains all the 5-mers from T, querying from a different 5-mer of P should still work – it will not fail to find a match. In this case, the 5-mer occurs twice in the text as index hits are found to be at offsets 0 and 4. Thus, two different verifications need to be done. At offset 0, the verification fails as there is no character to the left within that part of T. At offset 4, the verification succeeds.
It does not matter which 5-mer from the pattern is used as any of them will lead to the correct set of matches.
Another example can be taken using P: 'GCGTGA'. The first 5-mer hits offset 3 but the verification step fails. Thus, P does not match T. Not all index hits lead to matches.
Another example can be taken using P: 'GCGTAC'. The first 5-mer does not hit any offset so the index hit step fails. Thus, P does not match T.

25. Ordered Structures for Indexing

A multimap is a map associating keys (k-mers) with values (offsets in the genome) – key-value pairs. It is called as such because a k-mer may be associated with many different offsets. In order to implement a multimap, either an ordering or a grouping data structure can be used.
The index data structure based on ordering is simply a list of k-mer offset pairs alphabetically ordered by k-mer. To query this index, a binary search is used.
Example: T: 'GTGCGTGGGGG' and P: 'GCGTGG' with the following index:

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

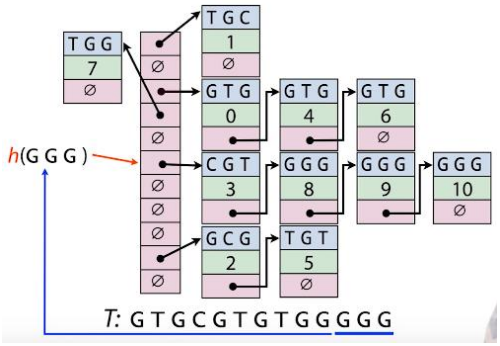
Applying a binary search on the list using the second 3-mer of P, TGG, leads to the comparison of being alphabetically greater TGG > GTG (which is in the middle). Ignoring the first bisection, the problem is divided into two. The search is applied on the remaining entries leading to TGG > TGC. This half is ignored and the search applied to remaining two entries. As there are only two, the middle is taken to be the first, resulting in TGG = TGG. This match corresponds to an index hit at offset 7.
The total number of bisections needed to be performed in order to find the key in the index is approximately equal to $\log_2(n)$ bisections per query, n being the number of keys in the index. The logarithm is of base 2 as the process repeatedly divides the problem by 2.
Python provides a set of functions related to binary search in a module (collection of functions and classed) called bisect. For example: bisect_left(a, x), where a is a sorted list in ascending order and x is an item, returns the left-most offset where x can be inserted into a while maintaining the order.
Implementing the function on the previous query example with P: 'GCGTGG', bisect_left(index, 'GTG') returns the offset of the first position where the 3-mer could be

inserted while maintaining sorted order of the list – in this case, it corresponds to offset 0. To find further positions of the pattern, the current offset must be looked up before continuing.

26. Hash Tables for Indexing

Another data structure which can be used to implement a multimap is a hash table. An empty hash table consists of an array of buckets (empty boxes/null references). As items are added, these buckets become lists. Also associated with this table is a hash function h, which maps each distinct k-mer onto one of the buckets in the array.

Example: All the 3-mers of T: 'GTGCGTGTGGGGG' are added to the table by using the hash function to randomly assign them to buckets and to append the corresponding key-value pair to the bucket (a linked list involving the 3-mer, the offset and a null reference). An already seen 3-mer will be assigned by the hash function to the same bucket as before – as there is another entry in the bucket, it will be added on to the end (or beginning) of the list which is already present. A collision occurs when different 3-mers end up being present in the same bucket – this is not surprising as there are more possible 3-mers then there are buckets – and so, the pigeonhole principle is applied. Many collisions may cause the data structure to slow down. The following is a diagram of a possible hash table:



Taking P: 'TGTGGG' and querying this hash table with the second 3-mer, the hash function of the 3-mer is applied to map it onto a bucket resulting in 'CGT' – the only bucket to look at as it's what the hash function assigned 'GGG' to. The first entry is checked, observed to not match and ignored. The next three entries in the list match resulting in index hits at offsets 8, 9 and 10.

In Python, the dictionary type is an implementation of a hash table.

28. Variations on k-mer indexes

What if not every k-mer was taken into consideration? For example, what if only the k-mers at even offsets in T were taken note of and the odd offsets ignored? An advantage of this is that the index is now smaller, takes up less memory and is faster to query.

However, the fact that there are fewer k-mers in the index may be a cause for concern with regards to index misses which would have otherwise been found if all the k-mers were kept. This is compensated for by taking more queries from both even and odd offsets with respect to P. This scheme can be generalized to not only index every other k-mer from T but also, say, every third k-mer or every n-th k-mer. The number of queries from even and odd offsets with respect to P must then be n, with the offsets being modulo n.

Another variation is that instead of building the index over substrings of T, it is built over subsequences of T. Substrings are also subsequences but subsequences are not necessarily substrings.

For example: with T: 'CGTGCGTGCTT', take the first subsequence of a particular shape. In this case 'CGTGCGTGCTT' will produce the index CGGGT at offset 0. Repeating this over subsequences of the same shape will generate the following index:

CGGGT: 0
CGGTT: 4
GCTCT: 3
GTCTG: 1
TGGGC: 2

With P: 'GCGTACT', the index is queried by extracting the same shape subsequence from P; i.e. 'GCGTACT'. This will produce an index hit at offset 3.

This subsequence method tends to increase the specificity of the filter provided by the index – when an index hit occurs it will lead it to a successful verification a larger fraction of the time than if the substring method was used.

29. Genome Indexes used in Research

The question of how to build faster, smaller and more flexible genome indexes is an active area of research. Some of the advances over the last several years that have come in read alignment have related to how the genome can be fitted into a relatively compacted index so that it can fit in memory but still be queried quickly.

The suffix index is another type of index similar to the substring index where instead of extracting every substring of a certain length from the genome, every suffix is extracted. A simple idea to perform rapid querying, like when using substrings, is to take all the suffixes, sort them in alphabetical order and use binary search to find all the suffixes that have P as a prefix. As the suffixes are in order, all of the suffixes that share some prefix are going to be consecutive.

Is this data structure too big? Consider a genome of length n. If all the different suffix lengths of the genome are added up, n(n+1)/2 characters in the index will be produced as it grows with the square of n/quadratically with n. By the time n gets to 3 billion characters (the human reference genome), the above expression will be far too large.

This issue can be sidestepped by representing a suffix of the genome by its offset into the genome – instead of representing it explicitly as a string, it can be presented by one integer which denotes the offset with respect to the beginning of the genome. The list of integers/offsets is called a suffix array. Besides this, the genome needs to be stored as well in order to keep track of each offset's corresponding sequence. The expression becomes approximately $n^2/2$ as the index of integers grows linearly (it is in fact equal to the genome length). This results in a more manageably-sized and practical data structure.

The suffix array is just one example from a family of indexes called the suffix indexes. Another type is the suffix tree which organizes the offsets on the basis of grouping instead of ordering. Another type is the FM index which is based on the Burrows-Wheeler Transform (BWT). It is very compact as it consists primarily of the BWT genome which is the same exact size as the human genome itself – it's just the genome with permuted characters.

If each type of data structure was used to build an index of the entire human reference genome, it can clearly be seen that the size varies across structures – suffix tree ≥ 45GB, suffix array ≥ 12GB and FM index ~ 1GB. The FM index is thus widely used and the basis for some popular read alignment tools in the Bowtie and BWA (Burrows-Wheeler Aligner) families.

30. Approximate Matching, Hamming and Edit Distance

The exact matching problem can be solved using naïve-exact matching online methods like Boyer-Moore and index-assisted offline methods like k-mer, subsequence and suffix index.

However, the read will not necessarily match the genome exactly in its point of origin – it is expected that there will be some differences. Differences between read and reference occur because of sequencing errors and natural variation between genomes. Exact matching algorithms are therefore not sufficient.
Algorithms for approximate matching are thus needed as they allow for differences between P and T. Some differences which may be encountered are:

- Mismatch/Substitution – differing characters in P and T;
- Insertion – an extra character in P relative to T (the space in T being referred to as a gap). It can be equivalently compared to a deletion in T with respect to P;
- Deletion – insertion in T with respect to P or vice-versa.

The distance between two strings is how much they vary from each other.
The Hamming distance is defined over two strings X and Y of the same length – it is equal to the minimum number of substitutions needed to turn one string into the other. Example:

X: 'G**A**GGTA**G**CGG**C**GTTTAAC'
Y: 'G**T**GGTA**A**CGG**G**GTTTAAC' => Hamming distance = 3

The Edit distance (Levenshtein distance) is defined over two strings X and Y – it is equal to the minimum number of edits (substitutions, insertions, deletions) needed to turn one string into the other. Examples:

X: 'TG**G**CCGCGCAAAA**A**CAGC'
Y: 'TG**A**CCGCGCAAAA**–**CAGC' => Edit distance = 2
X: 'GC**G**TATGCGGCTA**–**ACGC'
Y: 'GC**–**TATGCGGCTA**T**ACGC' => Edit distance = 2

The Hamming distance is limited to substitutions while the Edit distance has the option of using insertions and deletions as well. The former required its strings to be of equal length while the latter does not.
Exact matches are within a Hamming distance of 0. The naïve-exact matching algorithm can thus be adapted to allow for mismatches having a Hamming distance greater than 0 (to be able to look for occurrences of P within T that are within some Hamming distance).
The naïve-exact matching algorithm can easily be adapted to the approximate matching setting. It is harder to adapt the Boyer-Moore matching algorithm but there is a method which allows for this to be done in general. It allows the use of any of the exact-matching algorithms as a tool to solve the approximate matching problem.

31. Pigeonhole Principle

In order to apply exact matching algorithms to approximate matching problems, consider P being divided into two pieces/partitions labeled u and v (two non-overlapping substrings which cover P). The idea is that if P occurs in T with one difference/edit, then either u or v must appear with no edits (remains exactly matching) while the other is changed with that edit. The approximate matching problem can thus start to be solved by using any exact matching algorithm to search for occurrence of u and v within T – the 1-edit case.
The general case where more than one edit is allowed, say k edits, consider P being divided into k+1 partitions labeled $p_1 \ldots p_{k+1}$. The idea is that if P occurs in T with up to k edits, then at least one of the partitions must appear with no edits. This principle holds as if there are k edits and k+1 partitions, not all of the partitions can be changed (only up to k can be).
This is similar to the pigeonhole principle which states which states that if n items are put into m containers, with n > m, then at least one container must contain more than one item.
Example: Consider P to be divided into 5 partitions, as occurrences of P with up to 4 edits are being looked for. An exact matching algorithm is used to find exact occurrences of each of these partitions within T. Assume that the only match found is p_4 – this is a hint that there may be an approximate match of P to T in the neighbourhood of this partition. A verification

step is thus applied to determine whether the entire P occurs in the neighbourhood of that partition hit.
The pigeonhole principle is the bridge which allows for exact matching algorithms to be used to find approximate matches.

33. Solving the Edit Distance Problem

A new, flexible family of methods which use dynamic programming and the idea of edit distance, but also solve the approximate matching problem are discussed. It is used to solve many different bio sequence analysis problems including global alignment and local alignment. Dynamic programming and edit distance do not depend on any exact matching algorithm in the same way the pigeonhole principle does. It can be thought of as a separate class of algorithms than those observed so far.
Defining an algorithm to solve the Hamming distance is quite easy. Defining one to solve the edit distance is harder.
Example: If |X| = |Y|, what can be said about the relationship between hammingDistance(X, Y) and editDistance(X, Y)? Answer: editDistance(X, Y) ≤ hammingDistance(X, Y) as the edit distance also has the ability to use insertions and deletions apart from substitutions.
Example: If X and Y are of different lengths, what can be said about editDistance(X, Y)? Answer: editDistance(X, Y) ≥ ||X| - |Y|| as in order to edit X into Y, at least as many edits as is required to make them the same length need to be introduced (lower bound).
The basic idea behind the edit distance algorithm is that to solve the edit distance between two strings, knowing the edit distances between prefixes of the strings will help calculate it. Let α be a prefix followed by the base C and β another prefix followed by A:

$$\text{edist}(\alpha C, \beta A) = \min\{\ \begin{array}{l}\text{edist}(\alpha, \beta) + 1 \text{ //edit } \alpha \text{ into } \beta \text{ and substitute to turn C into A} \\ \text{edist}(\alpha C, \beta) + 1 \text{ //edit } \alpha C \text{ into } \beta \text{ and insert A at end of } \beta \\ \text{edist}(\alpha, \beta A) + 1 \text{ //edit } \alpha \text{ into } \beta A \text{ and insert C at end of } \alpha\end{array}$$

This expression can be generalized as such:

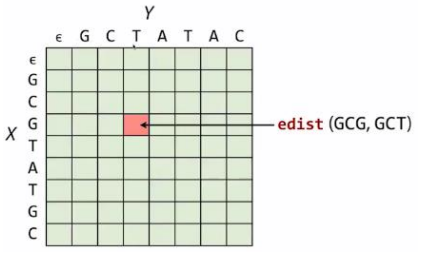
$$\text{edist}(\alpha x, \beta y) = \min\{\ \begin{array}{l}\text{edist}(\alpha, \beta) + \delta(x, y) \\ \text{edist}(\alpha x, \beta) + 1 \\ \text{edist}(\alpha, \beta y) + 1\end{array}$$

$$\delta(x, y) = 0 \text{ if } x = y, \text{ or } 1 \text{ otherwise}$$

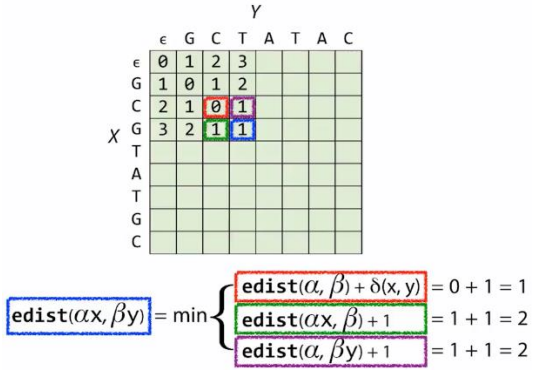
This can be considered as a recursive function. It returns the edit distance between two strings but it has a problem.

34. Using Dynamic Programming for Edit Distance

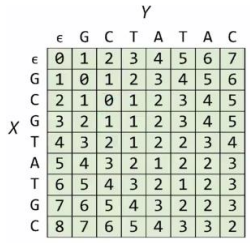
The function for the edit distance is very slow as multiple recursive calls occur when the function is run, some of them being duplicates; i.e. will return the same result – wasteful. It would be better if the answers of duplicate calls were remembered so that said call would not need to be run again.
To avoid this redundant work, the function can be re-written in terms of a matrix where each of its elements corresponds to a particular prefix of the strings (all prefixes including the empty prefix of length 0 are considered). Every element is filled in with the edit distance between the corresponding prefixes from top-left to bottom-right, for example:



The bottom-right element is special as it will eventually contain the edit distance between both of the entire strings – the solution.
Using the expression defined previously, the following procedure is done:



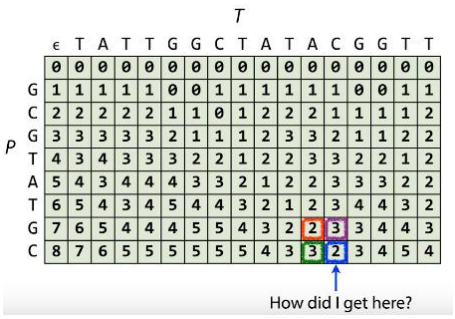
The first row and the first column correspond to the edit distance between the empty string and something which is always equal to the length of the other string – i.e. their values are ascending integers. Below is the filled in matrix:



Thus, the edit distance between the two strings is 2.
This new algorithm is very fast – the run time is a fraction of a second (the recursive function took over 30 seconds) as for any pair of prefixes X and Y, the edit distance is calculated once. This kind of problem is decomposed into smaller problems while also avoiding redundancy (recalculation of the smaller problems). It is called a dynamic programming algorithm.
Dynamic programming is a paradigm which is also used in other bio sequence analysis applications.

36. Edit Distance for Approximate Matching

Consider a new kind of matrix where the rows are labeled by the characters from P and the columns are labeled by the characters from T. An alteration to the edit distance algorithm is that the matrix will be initialized differently – the first row will all be filled with 0s (as we don't know where P will occur in T so every offset is equally likely) while the first column will be as before. The rest of the matrix is filled in exactly the same way as done in the edit distance problem. Where are the approximate matches of P within T?
In the following case, there must be a 2 edit occurrence of P within T (the occurrence with the fewest edits). This is shown by the minimal value in the final row:



This shows that P matches somewhere with 2 edits, but it doesn't show where exactly the substring in T is that it matches. How did the 2 result? The path must have been diagonal (from the red cell) as otherwise it would have been 4. Observing this value in the same manner, it too must have had a diagonal path. This procedure is repeated to show that the values go cell by cell backwards recreating the way the values were produced until:

			T														
		ϵ	T	A	T	T	G	G	C	T	A	T	A	C	G	G	T	T
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	G	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0	1	1
	C	2	2	2	2	2	1	1	0	1	2	2	2	1	1	1	1	2
P	G	3	3	3	3	3	2	1	1	1	2	3	3	2	1	1	2	2
	T	4	3	4	3	3	3	2	2	1	2	2	3	3	2	2	1	2
	A	5	4	3	4	4	4	3	3	2	1	2	2	3	3	3	2	2
	T	6	5	4	3	4	5	4	4	3	2	1	2	3	4	4	3	2
	G	7	6	5	4	4	4	5	5	4	3	2	2	3	3	4	4	3
	C	8	7	6	5	5	5	5	5	5	4	3	3	2	3	4	5	4

Instead of concluding a diagonal path, this time it's a vertical one. The procedure is repeated along a diagonal path till the top row. This path was thus followed:

			T														
		ϵ	T	A	T	T	G	G	C	T	A	T	A	C	G	G	T	T
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	G	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0	1	1
	C	2	2	2	2	2	1	1	0	1	2	2	2	1	1	1	1	2
P	G	3	3	3	3	3	2	1	1	1	2	3	3	2	1	1	2	2
	T	4	3	4	3	3	3	2	2	1	2	2	3	3	2	2	1	2
	A	5	4	3	4	4	4	3	3	2	1	2	2	3	3	3	2	2
	T	6	5	4	3	4	5	4	4	3	2	1	2	3	4	4	3	2
	G	7	6	5	4	4	4	5	5	4	3	2	2	3	3	4	4	3
	C	8	7	6	5	5	5	5	5	5	4	3	3	2	3	4	5	4

This path is called the trace back and the procedure done to find it is called tracing back. The substring of T which matches with P with 2 edits is thus known – it's the string which begins at offset 5 with respect to T. The shape of the alignment is also observed – specifically where the differences are:

T: 'GC–TAT**A**C'
P: 'GC**G**TAT**G**C'

The vertical step in the trace back corresponds to the only gap in the alignment. Thus the overall shape of the alignment follows the overall shape of the trace back.

These sorts of methods can be quite slow – the amount of work done is proportional to the number of elements in the matrix which in turn is proportional to the number of characters in P time the number of characters in T.

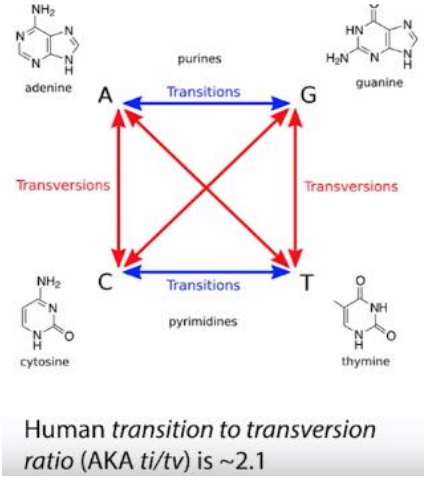
It is thus potentially impractical to use on its own. In practice it is used in collaboration with other techniques using the index and the pigeonhole principle. By combining them, the flexibility of the edit distance algorithm is gained, which naturally handles both mismatches and gaps. This can be combined with the speed which comes from using an index or the pigeonhole principle to quickly filter out a large number of alignments that won't end up with a good edit distance.

37. Meet the family: Global and Local Alignment

A couple of variations of the theme for dynamic programming of edit distance will be used to solve global and local alignment problems – both relevant for read alignment ad genome assembly.

Global alignment: The edit distance algorithm penalizes substitutions, insertions and deletions by the same amount of 1. This does not always make sense as in practice gaps could turn out to be less frequent then substitutions or substitutions of certain bases could be more likely than others. In fact, both of these are the case for DNA – some DNA substitutions are more likely than others.

All possible DNA substitutions can be divided into transitions and transversions. The bases A and G both belong to a category called purines while C and T belong to one called pyrimidines. Substitutions which change a purine to another purine or a pyrimidine to another pyrimidine are called transitions while those which change a purine to a pyrimidine or vice-versa are called transversions. If the possibilities are enumerated, it can be observed that there are twice as many types of transversions then there are types of transitions:



Human *transition to transversion ratio* (AKA *ti/tv*) is ~2.1

It can be thought that transversions are thus twice as frequent – in reality transitions are twice as frequent as transversions when comparing two human's genomes. So in the penalty scheme, the transversions might need to be penalized more than the transitions are.

If the genomes of two unrelated humans are taken, the number of substitutions versus the number of insertions and deletions amounts to the following: the human substitution rate is around 1 in 1000 bases while the in-del (small-gap) rate is around 1 in 3000 bases. In-dels are thus less frequent than substitutions and should be penalized more.

Moving beyond edit distance, different penalties to different kinds of mutations should be assigned using a penalty matrix. This matrix has an element for every penalty which may be incurred in an approximate match – some correspond to substitutions and others to insertions & deletions. A key point is that the elements can be set in whatever way is appropriate for the biological context – if working with DNA, transitions receive a lower penalty then transversions and transitions & transversions receive a lower penalty then gaps:

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

2 *Transitions* (A↔G, C↔T)
4 *Transversions*
8 *Gaps*

The edit distance algorithm barely needs to be changed in order to accommodate this penalty matrix. The value, which gets added on when the three contributions (diagonal, horizontal and vertical) are calculated, just needs to be changed:

$$\text{galign}(\alpha x, \beta y) = \min\begin{cases} \text{galign}(\alpha, \beta) + p(x, y) \\ \text{galign}(\alpha x, \beta) + p(x, -) \\ \text{galign}(\alpha, \beta y) + p(-, y) \end{cases}$$

p(x, y) is the value looked up from the penalty matrix. The rest of the algorithm is identical to the edit distance one. Global alignment is powerful as it gives the user the ability to set the different penalties according to the biological problem in hand.

Local alignment: A different kind of problem is being solved but the solution is quite similar to the solution for edit distance. The problem to solve is: given two strings identify the substring of X and the substring of Y which are most similar to each other – find the most

similar pair of substrings from X and Y. The number of possible pairs is related to the product of the squares of the lengths of X and Y but the amount of work to be done will turn out to be no different than a global alignment problem of the same size. The following is the recurrence for local alignment:

$$\text{lalign}(\alpha x, \beta y) = \max \begin{cases} \text{lalign}(\alpha, \beta) + s(x, y) \\ \text{lalign}(\alpha x, \beta) + s(x, -) \\ \text{lalign}(\alpha, \beta y) + s(-, y) \\ 0 \end{cases}$$

A scoring matrix will be used instead of a penalty one. An important difference is that the scoring matrix gives a positive bonus for a match and a negative penalty for all the different kinds of differences.

Using the recurrence and the scoring matrix to fill in a dynamic programing matrix, it will be observed that many of the entries are 0. Intuitively, this is because the goal of local alignment is to find parts of X and Y which match closely enough that they sort of pop out from the background of 0s/dissimilarities.

First, the element with the maximal value is found – corresponding to the optimal local alignment. To know which substrings are involved in that alignment (are most similar) as well as its shape, the usual trace back procedure is done. The only way the typical trace back is altered is that the procedure will stop once an element of value 0 is reached.

39. Read Alignment in the Field

Indexing and dynamic programming are complementary – they work well together.

The index allows rapid homing in on a small set of candidate locations in the genome where the pattern may have a good approximate match. The index depicts a list of places in the genome which share a substring with the pattern – it acts almost like a filter.

If there was no index (nothing acting as a filter), then dynamic programming alignment would be done to solve the read alignment problem. It allows the finding of approximate occurrences to the pattern within the text – P = read and T = reference genome. The dynamic programming matrix, which is really large as the read is around 100 bases long and the (human) genome about 3 billion, would be filled in with rows labelled with characters from a read and columns from the genome. Every read would thus take years to analyze, the data sequencer having taken a week to produce the dataset.

This is the reason that an index is needed – rapid homing in on just those portions of the reference genome, where dynamic programming/verification step is applied, is required.

Dynamic programming is needed as indexes do not deal very well with mismatches and gaps; they are good at finding exact matches. Global and local alignment algorithms are ideal as they are very flexible when it comes to substitutions, insertions and deletions. Dynamic programming is thus the ideal way to verify whether a particular index hit corresponds to an approximate match of a read within a genome.

The index is very fast and good at narrowing down the spaces to look but it doesn't handle mismatches and gaps naturally. On the other hand, dynamic programming does this very naturally but is very slow. Both are thus needed for read alignment tools.

40. Assembly: Working from scratch

Read alignment is analogous to the situation where a puzzle needs to be put together (the puzzle pieces being the reads), but with the help of a picture of the completed puzzle (the reference genome).

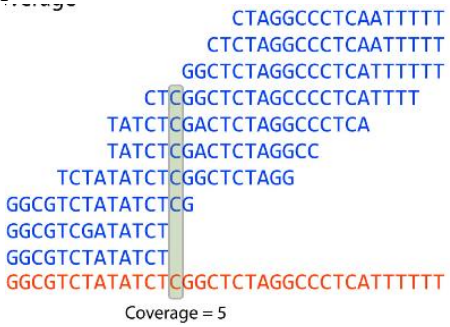
Another version of the problem is assembly (De Novo assembly or De Novo Shotgun assembly). 'De Novo' means from scratch and 'shotgun' refers to the fact that the reads are coming randomly from all over the genome. The benefit of being able to see the picture of

the completed puzzle is not given; for example: the study of a species which has never been sequenced before. It is thus more computationally work intensive then the corresponding read alignment problem. It is also fundamentally difficult but it has profound ideas which will form the basis of the modern tools for solving these problems.

41. First and Second Laws of Assembly

The assembly problem states that: given many random sequencing reads (derived from the genome to assemble), reconstruct the genome sequence from them.

Coverage at a particular position in the genome refers to the amount of redundant information about the genome. For example:



In another sense, it also shows that the reads are giving 5 distinct pieces of evidence for which base appears at that position in the genome. Moving two positions to the right in the above example, this position also has a coverage of 5 but the reads covering it don't all agree on which base appears there.

Overall coverage, the coverage average to overall positions in the genome, is calculated by taking the total length of all the reads and dividing it by the total length of the genome. For example: if there is a total of 177 bases for all the reads and 35 bases for the genome, then the average coverage is 5-fold.

The genome sequence can be pieced together due to hints given by matches between a read's suffix and another read's prefix (with slight differences) which indicate that the two reads may have originated from overlapping portions of the genome.

The first law of assembly states that if a suffix of read A is similar to a prefix of read B, then A and B might overlap in the genome from which they came from. It gives a hint that they can be glued together in order to get a larger piece of the genome.

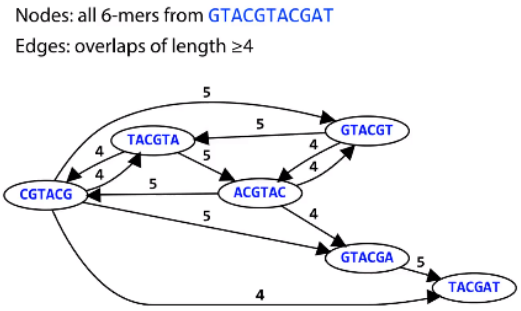
A perfect match is not always the case for this law. This is because of sequencing errors and polyploidy (ex. humans have two copies of each chromosome, and copies can differ).

The second law of assembly states that more coverage leads to more and longer overlaps; i.e. the greater the sequencing depth, the more the overlaps and the longer the overlaps will be.

42. Overlap Graphs

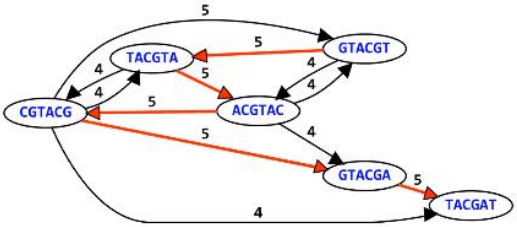
In order to represent all the overlap relationships at once, a directed graph will be built (nodes as ovals and edges as arrows) – the nodes represent objects/concepts and the edges represent relationships among them.

All the overlap relationships in a dataset of sequencing reads can be represented by making a directed graph whose nodes correspond to the reads in the dataset and whose edges correspond to the overlaps between pairs of reads. The edge will be directed from the node that has the suffix to the one that has the prefix involved in the overlap. For example:



Some overlaps are less convincing then others; for example an overlap of length 1 could just be a coincidence so it's better not to consider it in the graph. It makes sense to have some kind of threshold which states that as long as an overlap is more convincing then the threshold, it will be counted as an overlap and the corresponding edge drawn. In the above example, the threshold says that the suffix prefix match should be an exact match and have at least a length of 4 for simplicity. Given the threshold, the entire overlap graph can be written out as shown above.

The sequence of the original genome is going to correspond to a particular way of walking along the graph. In the above example, one particular path is special:



It corresponds to the sequence of the original genome, with each generated 6-mer after each other (each having a length 5 overlap) forming a walk. Every 6-mer from the original genome sequence was walked along to infer the sequence.

This example was idealized. In reality, the data in the resulting overlap graph will be a lot messier due to sequencing errors and polyploidy issues.

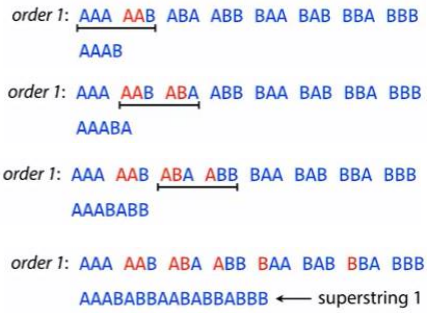
45. The Shortest Common Superstring Problem

Now, the formulation of a computational problem will be done that when solved, a genome assembly problem will in turn also be solved. This first formulation will be a good starting point for the discussion on the genome assembly problem, kind of in the same way that the naïve exact matching problem was a good starting point for the read alignment problem. The computational problem is called the shortest common superstring (SCS) problem.

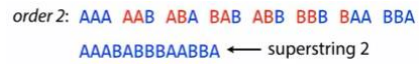
The SCS problem states that given a set of input strings S, the shortest string containing the strings in S need to be found as substrings. For example S: BAA AAB BBA ABA ABB BBB AAA BAB. The shortest common superstring, the shortest string which contains each of these strings as a substring, needs to be found. If the requirement of it being the shortest wasn't there, then all the strings of S could simply be concatenated together. An algorithm for finding the SCS exists – if the sequencing reads were given to the algorithm, then the solution to the problem, the SCS problem, would also be an assembly of the genome (a reconstruction of the original genome sequence).

The SCS has some considerable downsides. It is intractable (presently, there are no efficient algorithms for large inputs); i.e. it is NP-Complete. This does not mean it cannot be solved, just that it won't be very fast. As the number of input strings grows, the algorithm gets slower (brute force).

The idea behind the algorithm is that the strings in S will be ordered in some way and for every adjacent pair of strings, the longest overlap between them will be found followed by them being merged together. For example, given an ordering:



This is a candidate superstring. It is not necessarily the SCS but it might be, depending on if the right ordering was picked. Different orderings will result in different superstrings. For example:



To find the absolute SCS, all the orderings needs to be tried. So for every possible way of ordering the n input strings (for every permutation) adjacent pairs of strings will be merged together according to their maximal overlap. The SCS over all the orderings will be the resultant SCS.

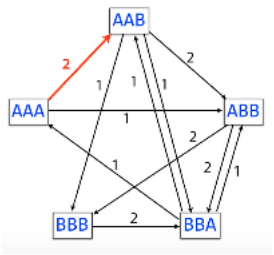
This is so slow as: if S contains n strings, n! orderings are possible – as n grows, the number of permutations needing to be tried grows rapidly.

47. Greedy Shortest Common Superstring

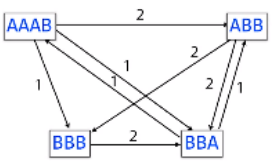
This algorithm is much faster then the previous one. It is called greedy as it makes a series of decisions and at each decision point, it will choose the option which reduces the length of the eventual superstring the most. This seems to be a good stategy however making the greedy decision at each point in the algorithm does not necessarily mean that an optimal solution will be reached.

The algorithm can be visualised using an overlap graph. The principle behind it is that it procedes in rounds and in each round, the edge which represents the longest remaining overlap (edge with the greatest number as its label) is chosen. The nodes on either side of that edge are then merged. The longer the overlaps between the strings, the shorter the final string will be.

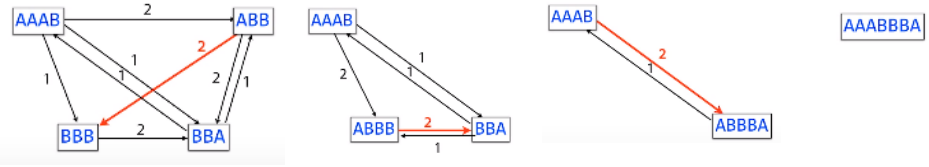
To apply the principle on an overlap graph, the edge which corresponds to the longest overlap must first be found. In the case where there are several overlaps of the same length, one of the edges is just picked randomly. For example:



The two nodes on either side of the chosen edge are then merged together by replacing them with one new node which corresponds to the two labels of the original nodes glued together according to their overlap (formation of a superstring):

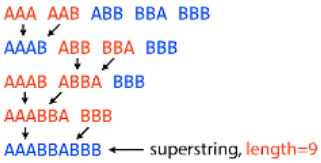


This merge has made the graph slightly simpler/smaller – one less node and at least one less edge in it. The same process is repeated until one node is left:



The last node's label is the superstring obtained from the greedy SCS algorithm. If there happen to be multiple nodes at the end of this merging process (have run out of edges but there are still multiple nodes), the superstring is created by concatenating the labels of those nodes together.

The amount of work done by this process is greatly reduced. This speed comes at a cost – the greedy algorithm doesn't always find the correct answer as the result is not necessarily the shortest. For example:



As edges are picked randomly, the resultant superstring ended up being different then the above one. The graph ended up with two separate nodes so they had to be concatenated together to form the final superstring. This is not the SCS – the one gotten in the previous run through is the correct answer.

49. Third Law of Assembly: Repeats are bad

A downside to both of the SCS algorithms is that when the genome is repetitive, the SCS of the reads is not going to be the correct answer. Finding the SCS isn't really what is wanted in this case because of the repetitive nature of the genome – the algorithm will tend to take the repetitive portions (the transposable elements for example) of the genome and collapse them down to fewer copies then should really be there. The SCS algorithm will always go with the fewest copies required to explain the reads.

This is probably the single-most important issue which makes the assembly problem difficult in practice. Repetitive sequences can make it difficult and even impossible to correctly assemble the original genome – they cause ambiguity.

The third law of assembly states that repeats make assembly difficult. When the genome is repetitive, any of the attempts to assemble it will fail in some way, shape or from. The way in which it will fail depends on the particular algorithm used to solve the assembly problem. Other algorithms, apart from the two SCS ones, make different kinds of mistakes.

About half the genome is covered by repetitive DNA sequences. This makes the assembly problem very difficult for genomes similar to the human genome.

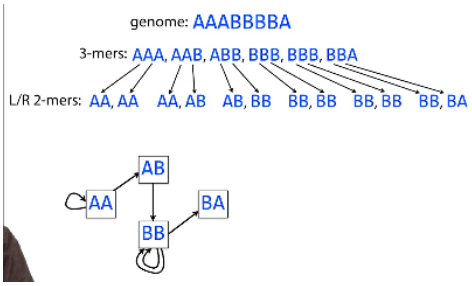
An alternative to the SCS is the De Bruijn graph and Eulerian walk method that avoids the over-collapsing problem. However it still cannot avoid the overall problem: that repeats make assembly difficult.

50. De Bruijn Graphs and Eulerian Walks

An alternative algorithm uses a De Bruijn graph, instead of an overlap one. It is a directed graph, as is the overlap graph, and it can have more than one edge pointing from one node to another (between a pair of nodes) – it is a multi-graph.

To build a De Bruijn graph, the sequencing reads are assumed to consist of each of the substrings of the genome of length k – k-mers – where each k-mer is sequenced once (note

that the assumption is not a good one to make). For every offset in the genome, that k-mer needs to be extracted and the sequencing output will be a list of all the k-mers.
The extraction for every k-mer consists of its left and right (k-1)-mers. To update the De Bruijn graph, a node for the left (k-1)-mer and another one for the right will be added. Then a directed edge will be added which points from the left to the right (k-1)-mer. If the (k-1)-mers are both the same, only one node will be added and a directed edge will point from the node to itself. For example:



Each k-mer corresponds to one edge and each distinct k-1-mer in the genome corresponds to one node in the De Bruijn graph.
Assume the genome sequence was not known and that only the De Bruijn graph was shown. Could the original genome sequence be reconstructed from this graph? The answer is yes. A reconstruction of the original genome corresponds to a walk through the graph, respecting the direction of each edge. The walk crosses each edge exactly once – it uses each k-mer exactly once – and is called an Eulerian walk.
Not every graph has an Eulerian walk but the one that does is called an Eulerian graph. Using the assumptions made about the sequencer, that each k-mer is gotten exactly once, if the graph is reconstructed with this procedure then the graph will always be Eulerian.
This provides a new algorithm for solving the assembly problem: the corresponding Bruijn graph is built from the given reads, which is Eulerian given that there is one read for every k-mer, and an Eulerian walk through the graph will give a reconstruction of the original genome.

52. When Eulerian Walks go wrong

The Eulerian gives the reconstructed genome sequence back again and it does not over-collapse repetitive portions, which is an improvement over the SCS formulation of the problem. However, it still cannot escape the third law of assembly.
A Bruijn graph can have more than one way on how to walk through it – more than one Eulerian walk/path. This leads to ambiguity and so the third law of assembly has not been escaped from.
The Bruijn graph for repetitive genomes will thus, in general, have many different Eulerian walks with only one of the walks actually corresponding to the original genome sequence. All the rest of the walks correspond to incorrect reshufflings of the genome where portions of the genome that occur between repetitive sequences are going to end up in an incorrect order. Repeats made assembly difficult again.
By decreasing the k-mer length, the chance of being effected by repeats is increased since the smaller k means there's more likely to be multiple occurrences of any given k-mer in the genome which is going to increase the chance that multiple different Eulerian walks are possible for the same graph. With G = DeBruijnGraph([st], 3): to_every_turn_turn_thing _turn_there_is_a_season. Thus, in the case of the De Bruijn graph, the repetitive genome results in a spurious reshuffling of portions of the genome.
The assumption made about the sequencing data, that the sequencer gives exactly one read per k-mer without accidently leaving out any k-mers or giving redundant k-mers or without

sequencing errors, is not actually true in practice. Typical values of k are around 30 to 50 or so whereas sequencing reads are around 100 to 300 or so bases long. Given this reality, the procedure of building the De Bruijn graph can still be used, but the result will be a graph which is not necessarily Eulerian – it will actually never be Eulerian in practice. Finding Eulerian walks will thus not solve the assembly problem.
However the De Bruijn graph representation is a common and useful way to represent assemblies – to represent the assembly problem. In fact, many modern software tools for assembly use De Bruijn graphs for the internal representation of the relationships between the sequencing reads.

53. Assemblers in practice

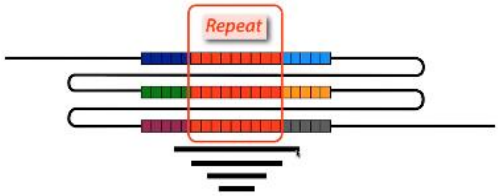
How do real software tools for solving the assembly problem work and how do they deal with repetitive genomes and other issues in practice? Although the SCS and Eulerian walk algorithms won't be used, their graphs (the overlap and the De Bruijn graphs respectively) will still be used. The graphs correspond to two different categories of assembly programs. Assembly programs which use the overlap graph are called Overlap-Layout-Consensus (OLC) programs while those which use the De Bruijn graph are called De Bruijn Graph Based (DBG) programs. In both cases the first step is to build the graph.
The obtained graph will be big and messy because of the following reasons:

- Sequencing errors which can introduce spurious diversions in the graphs – dead ends which should be gotten rid of or ignored;
- Specific to the overlap graph, it can contain edges that while not incorrect don't actually provide anything not already known – some edges are transitively inferable from others;
- Polyploidy leads to the fact that for some bases/positions in the genome (for each chromosome, two copies are inherited), the maternal inherited version differs from the paternal and so the De Bruijn graph ends up with a bubble shape where these bases differ (one side of the bubble will contain k-mers which are only present in the maternal copy while the other side will contains those only present in the paternal). One way to deal with this is collapse the graph to get a straight line walk but this must be kept note of;
- Repeats, repetitive DNA, will create ambiguity in the graph. This is dealt with by dividing the assembly into pieces as there will always be pieces in the graph where there is no ambiguity so portions can still be put together. These partial reconstructions that can be put together unambiguously are called contigs – contiguous sequences. This is a principle which real world assemblers use when the overall graph is ambiguous (which is always the case in the human genome). An assembler will report a set of contigs instead of reporting a single assembled genome sequence – every assembler is fragmented (even the human reference genome).

54. The future is long?

In theory, something can be done to counterattack the issue of repetitive DNA – the reads can be made longer due to the fact that the longer the reads are, the more likely it is to get a read which anchors some repetitive sequence that glues it with some surrounding non-repetitive sequence. This denotes where the repetitive sequence should go in assembly.
Some analogies are larger puzzle pieces which show more the relationship between repetitive and non-repetitive parts and longer lengths of k in the case of k-mers which are more likely to avoid over-collapsing due to the better chance of observing the relationship between repetitive and non-repetitive sequences.

If the reads are long enough to extend all the way through the repetitive sequence and overlap the non-repetitive sequence on either side, then that is what will allow the recreation of the genome sequence unambiguously. So how long would the read need to be to do this? Some candidate reads need to be taken into account. For example:



In this case, only the top-most horizontal line is long enough to span the entire repeat and the unambiguous sequence on either side. Thus, the reads need to span the entire repeat for unambiguous assembly.

How are long enough reads acquired? This is a hard technological question. DNA sequencers are very good at collecting lots of short substrings sampled from the reference genome. There hasn't been a technology invented yet that can read longer stretches of DNA, says in tens or hundreds of thousands of bases long, very accurately. But there are technologies that have gotten a bit closer – two of them will be discussed.

Paired-end sequencing is one of these technologies, which returns more information per DNA template sequence when compared with second generation sequencing technologies. Instead of sequencing just one end of a template due to the available number of sequencing cycles, both ends can be sequenced with paired-end sequencing. Depending on how long the template is, the ends might meet in the middle and so a read which is twice as long and which covers the entire template molecule is acquired. However meeting in the middle won't necessarily be the case as the template molecule might be longer than twice the read – a read at both ends with some kind of gap in between will be acquired. The same sorts of methods used previously can still be applied in order to either align these reads to a genome or assemble them into a genome. The amount of missing sequence just needs to be dealt with, which although not trivial, can be done. Paired-end sequencing is extremely common in practice – it's a way to get about twice as many bases out of every template strand sequenced without sacrificing much in terms of accuracy versus speed. It returns a factor of 2 in terms of improvement in read length.

Some recent technologies can return 1- or 2-order magnitude improvement in read length, though at the expense of speed and accuracy. These types of technologies sequence one molecule at a time, in contrast to the sequencing by synthesis methods. Single molecule sequencers like these are capable of generating reads that are on the order of tens or hundreds of thousands of bases long. This is a great advantage as thus, there are few stretches of DNA that cannot be resolved/anchored to a nearby unique sequence with reads that are that long. Unfortunately, that length comes at a cost as the reads end up being very error prone. On the order of 10 to 15% of the time, the sequencer will make a mistake in reading a base and thus, tools like real aligners and assemblers have to be exceptionally flexible to deal with these kinds of mismatches and gaps.

55. Computer Science and Life Science

How do computer scientists contribute to fields similar to life science? Life science requires research in computational genomics, data structures, algorithms and computers in general. Many computer scientists are needed in projects like the Human Genome Project for the following reasons: computer and life scientists deal with a lot of the same abstractions (ways of looking at the world; for example strands & strings, trees and graphs/networks) and the two fields are connected through the technology (data bottlenecks).

One way of tracking how much data is accumulating over the various projects in the world that use DNA sequencers is by tracking the growth of an archive (a sequence read archive – WWW for sequencing read data). To demonstrate, its size is currently in petabytes. As a result, life scientists need to use a lot of computation and solve a lot of computational problems in order to understand this data. This is where computer scientists come in.
These fields are also joined through the nature of the data. High throughput data is never straight forward due to errors, uncertainties, bias and ambiguity. To understand the data and draw robust conclusions from it, the skills of a data scientist are required – machine learning, statistics, applied mathematics…

Entry 6

Meeting: 26th September 2016, 3:30-4pm

Since the last meeting, research from a book, a paper, some articles and an online course was done as summarized in the above entries 2-7.

This meeting dealt with a discussion on what was researched and thus learnt. An overview of DNA sequencing was done detailing nucleotides, sequencing reads, chromosomal inheritance from parents and comparison of a person's DNA with the human genome. Matches from this comparison occur according to the length of the reads taken, notwithstanding the errors which the sequencer will most definitely make due to the large genome size (approximately 3 billion characters long). The general format of DNA files and how each read is divided into four lines of differing detail was also discussed.

By the next meeting, a Python program which aligns a file of sequencing reads with the human genome needs to be written. The given DNA file and the results should be analyzed in order to get used to the format implemented in genomics.

Entry 7

Paper Summary: 'Fast and accurate short read alignment with Burrows–Wheeler transform' by Heng Li and Richard Durbin

The Illumina sequencing technology led to new alignment programs, which map many short reads to a genome. They work either by hashing the sequencing reads and scanning through the genome, by hashing the genome, by merge-sorting the reads and the genome or by using Burrows-Wheeler Transform (BWT).

The aligner discussed, BWA, is developed on the string matching theory using BWT. Its implementation resembles that of a top-down traversal on the genome's prefix trie with exact and inexact matches having their distinct sampling methods. A prefix trie of a string can be used to test a query W against a string for an exact substring with the node representing W being found in O(|W|) time. A suffix array (SA) is constructed taking *n* bits of space; i.e. <1GB memory at peak time of the human genome, followed by the generation of BWT. The search for SA intervals of matches can be equalled to sequence alignment.

Both exact and inexact matching can be done by a procedure which tests whether a string W is a substring of another string X. Exact matching is done using backward search which can be compared with exact string matching on the prefix trie without directly putting the trie in memory. On the other hand, inexact matching is done using bounded traversal/backtracking

which allows for no more than *z* differences while using backward search to sample different substrings from the genome, similar to a DFS on the prefix trie.

Illumina reads may be problematic due to their ambiguous bases, the support for paired-end mapping (Smith-Waterman alignment), the allowed maximum number of mismatches or gaps being calculated according to the length of the read in question and the generation of mapping quality scores being calculated using the Phredd-scaled probability of the alignment being inaccurate while assuming that the true hit can always be found (may lead to overestimation). SOLiD reads are mapped in a colour space, generated by converting the reference genome to dinucleotide 'colour' sequence and constructing the BWT index for said colour genome. Its paired-end mapping is correct for certain cases and the Smith-Waterman alignment is also applied in the colour space. Dynamic programming is implemented to convert the colour read sequences to their nucleotide counterparts with the help of the Phredd-scaled probabilities to approximate base qualities.

BWA features gapped alignment for single-end reads, paired-end mapping, mapping quality and multiple hits with SAM being the default format for output alignment, having standard tools being available for use by the users. It was evaluated against three alignment programs: MAQ, Bowtie and Soapv2 on both simulated and real data. The simulated reads' results showed BWA to have: similar alignment accuracy to MAQ, more confidence in the mapped reads and error rates than Bowtie and Soapv2, more speed than MAQ and less problems in terms of memory on modern servers due to the support of multithreading. The real reads' results showed BWA to be faster than MAQ with the same alignment accuracy and functionality but slower than Soapv2, and Bowtie to have a smaller alignment error rate with the exception of BWA if its mapping quality threshold is increased.

When the sequencing error rate is high, BWA's performance decreases as it always needs the alignment of the whole read. Longer reads are more prone to interruptions by variations or mis-assemblies in the genome making BWA crash. A possible solution would be to divide the long read into smaller reads, align them and then join to result in the full read alignment.

Entry 8

Meeting: 3rd October 2016, 1:30-2pm

Since the last meeting, a program which aligns some sequencing reads with the human genome was written along with a paper summary (recorded in entry 9).

This meeting dealt with a discussion on how the program with its accompanying functions was implemented and how it was tested on a phi X virus. The approach was deemed to be correct, although running it on larger files led to a long wait time – this could be due to the fact that the laptop being worked on not having a large enough RAM to go through the whole human genome along the large volume of sequencing reads.

By the next meeting, a copy of the code will be sent to the supervisor in order to be tested on a PC having a larger RAM. A function to develop a visualization of the matched reads against the genome needs to be written so as to be able to better see where matches occur or do not occur.

Entry 9

Meeting: 10th October, 2-2:30pm

Since the last meeting, a function to generate a data visualization jpeg file of the matched reads against the genome was completed.

This meeting dealt with a discussion on how the function works, its implementation being deemed correct. Ways on how the program could be optimized was also discussed due to the long wait time when running the functions on relatively large files. Possible methods include parallelism and/or using inbuilt Python generators with the accompanying yield function.

By the next meeting, three tasks should be completed: improving the data visualization function such that the output does not remain static (i.e. clicking/hovering over a read would display the nucleotide match in question and tell the user where the read occurs); optimizing the program overall in order to be able to run it on large files efficiently; reading the first chapter of the book 'Introduction to Genomics' by Arthur M. Lesk.

Entry 10

Book Notes: 'Introduction to Genomics' by Arthur M. Lesk; Chapter 1 'Introduction to Genomics'

A human genome's base pairs, around the size of 3.2×10^9, are distributed among 22 paired chromosomes – XX in female and XY in male. A human is defined as follows: phenotype = genotype + environment + life history + epigenetics, where the phenotype is a collection of observable traits (hereditary); the genotype is the nuclear and mitochondrial DNA sequence (leads to pharmacogenomics – personal medicine); the environment is the living surroundings and nutrition; the life history is the integrated total of experiences and the epigenetic factors are the small amount of cells having different DNA.

A common method of generalising genetic effects from those of surroundings and experience is by implementing controlled experiments with genetically identical organisms. Human sibling and twin studies have been the basis for many important decisions however it is still difficult to measure the intelligent quotient IQ of adults due to bias (IQ remains constant in early childhood).

A human genome contains: about 2-3% protein-coding genes (transcribed into messenger RNA), about 3000 non-protein coding genes (exclusive of m-RNAs and in control of gene expression), binding sites for ligands (regulation of transcription) and 10-20% repetitive elements of unknown function.

Functionally, dispersed gene families and tandem gene family arrays have moderately repetitive DNA. This could also be said for short and long interspersed elements (SINEs and LINEs) and pseudogenes – degenerate genes which have mutated beyond functionality – in the case of repetitive elements with no known function. Highly repetitive DNA belongs to minisatellites, microsatellites and telomeres.

Note that in protein synthesis, a synonymous mutation is one which changes a codon to another codon for the same amino acid. The 20 standard amino acids in proteins are as follows: non-polar including glycine G, alanine A, proline P, valine V, isoleucine I, leucine L, phenylalanine F and methionine M; polar including serine S, cysteine C, threonine T, asparagine N, glutamine Q, tyrosine Y and tryptophan W; charged including aspartic acid D, glutamic acid E, lysine K, arginine R and histidine H.

The human genome is believed to have 23000 protein-coding genes containing exons (translatable regions) interrupted by introns (spliced out before translation), the genes appearing on both strands. A gene neighbourhood includes a set of closely linked related genes due to gene duplication (mechanism of evolution) followed by divergence.

Ideally, following a genome sequence determination, the corresponding proteome (the amino-acid sequences of the proteins expressed) can be inferred. However variety is introduced through alternative splicing and RNA editing which describe the relationship between genome sequences and proteins potentially encoded in them.

Sequences are logically one-dimensional. They must adopt three-dimensional structures to perform certain activities due to their native state, reversible denaturation and renature. The following paradigm is used: DNA sequence determines protein sequence, protein sequence determines protein structure and protein structure determines protein function. Predictions of protein structures can thus be written as programs using structure prediction methods, allowing for the creation of library structures of the encoded proteins in any genome.

Cells, included in the biosphere, are classified as prokaryotes if they: do not have a nucleus, have a size of 10μm, have no organised subcellular structure as their internal differentiation, have a fission cell division and have most of their DNA in the form of a single circular molecules with few proteins permanently attached; and as eukaryotes if they: do have a nucleus, have a size of -0.1mm, have their internal differentiation in the form of nuclei, mitochondria, chloroplasts, cytoskeleton, endoplasmic reticulum and Golgi apparatus, have a mitosis cell division and most of their DNA sequestered in the nucleus by being complexed with histones to form chromosomes. Both cell types deal with packaging and cell division (after DNA replication) problems. Plasmids are found in yeast cells (eukaryotic) – yeast artificial chromosomes YACs – which are largely used in genome sequencing.

Humans contain 46 chromosomes (22 pairs) with an additional two X chromosomes for females and an X and a Y for males. Deviations from the normal chromosome complement have unwelcome results.

Genomes are long strings of As, Ts, Gs and Cs, having functional regions such as protein-coding ones, non-protein-coding RNAs and targets of regulatory interactions. Gene identification is easier in prokaryotes as they are smaller and have fewer genes than eukaryotes which are contiguous. There are two basic approaches to identification: priori methods which recognise sequenced patterns within expressed genes and their regions; and ‘been there, seen that’ methods recognise regions corresponding to previously known genes. Combined approaches are possible. Characteristics to identify eukaryotic genes: the initial 5’ exon, internal exons, the final 3’ exon and the non-random sequences of all coding regions.

Transposable elements are skittish segments of DNA which move around the genome. Alternative mechanisms of transposition, which are displayed by different types of elements, are retro-transposons (class I) and transposons (class II). Long and short interspersed elements (LINEs and SINEs) are found in the human genome, having the most common LINE L1 appearing about 20,000 times in it as well as having around 300,000 copies of the Alu element – the most common SINE with size 280kb. The total amount of L1 + Alu is 7% of the human genome. Transposable elements’ biological effects include sequence

broadcasting, altering properties of genes, being an important engine of evolution, causing chromosomal rearrangements and leakage if epigenetic modification.

Genome sequencing projects of non-human species help in research about evolution and the human genome functions. If a region if conserved over the years, then it must have been conserved foo a reason. The projects also have direct application to human welfare and history. Most genome projects target individual species as a major component of public DNA repositories comes from metagenomic data (sequences derived from environmental samples without isolating individual organisms). The first genome to be sequenced was the single-stranded DNA virus, bacteriophage X-174.

High-throughput sequencing deals with the generation of raw data and assembly. Most methods sequence DNA molecules by fragmenting them and partially sequencing the pieces which have a typical length – the read length. De novo sequencing deals with the determination of the complete sequence of the first genome from a species by fragmenting them and partially sequencing them using one/single-end or both/paired-ends with the number of bases reported being the read length. Assembling the genome comes next from the sequences of overlapping fragments. A partial assembly of these into a contiguous sequence is a contig. The fragments must cover the entire genome with enough replicates to detect errors. The data set's coverage is the ratio of the total number of bases sequenced to the genome length. After the first genome – the reference genome – is available, other genome sequences of the species can be determined with the assembly step being replaced by mapping fragments onto the reference; i.e. re-sequencing.

Exome sequencing deals with the sequencing of the protein-coding regions – the exons – making up about 1% of the entire genome. RNA sequencing deals with the conversion of RNA to complementary DNA and sequencing the results. ChIP sequencing deals with the sequencing of DNA fragments to which certain proteins are known to bind. Methylation-pattern determination deals with the comparison of native DNA sequences.

Humans' genomic sequences differ at around 0.1% of the positions (except for identical siblings). A lot of these variations come from isolated base substitutions/single-nucleotide polymorphisms (SNPs). Clusters of these are called haplotypes. Furthermore, human genome sequencing is done by a number of international organisations such as the International HapMap project and its extension, the 1000-genome project. Several companies even offer personal genome sequencing.

Genomics and proteomics have made great contributions to medicine and surgery such as: prevention of disease, detection and precise diagnosis, discovery and implementation of effective treatment and health care delivery. Analysis of a patient's genes and proteins permits selection of drugs and dosages optimal for individual patients (i.e. pharmacogenomics).

Databases archive the information generated by high-throughput sequencing and present it in a useful form. Sources of biological data include high-throughput streams such as systematic genome sequencing, protein expression patterns, metabolic pathways, protein interaction patterns and regulatory networks as well as the scientific literature/bibliographical databases. The earliest databanks were the Protein Sequence and the Protein Data banks; todays are the National Centre for Biotechnology Information NCBI and the European Bioinformatics

Institute. Databanks, apart from archiving and curating, have been active in information retrieval and analysis.

A genome browser is a type of database which deals with full-genome sequences and related information. It is a project designed to organise and annotate genome information, and to present it via web pages together with links to related data. It is similar to an encyclopaedia. It also provides tools for searching and analysis. Two major genome browsers are Ensembl and Santa Cruz Genome Browser.

The divergence of sequences and structures within and between species is referred to as protein evolution. The basic tool for investigating sequence divergence is the multiple sequence alignment and the basic tool for investigating structural divergence is superposition.

DNA sequencing has its ethical, legal and social issues. Questions have arisen about privacy issues, the data that should be included and access. There are two major national repositories of human genome information in the UK, the National DNA Databank NDNAD (for law enforcement and forensics) and the UK BioBank (for medicine). In the US, there is the Combined DNA Index System and the National DNA Index System (CODIS/NDIS for forensics).

Entry 11

Meeting: 17th October, 1:30-2pm

Since the last meeting, the previously mentioned chapter was read. As for the functions, some problems were encountered on optimizing the program and improving the data visualization.

This meeting dealt with a discussion on how these problems could be solved. The optimization method taken was correct in terms of generator usage however further additions to the other functions need to be taken in order to properly implement the needed efficiency. The data visualization method was not updated due to uncertainty on how it could be done. It was advised that a Python canvas to implement a GUI should be used, such as Tkinter (Python's standard GUI package).

By the next meeting, the problems encountered should be fixed as much as possible and, hopefully, completed.

Entry 12

Meeting: 20th October, 10:30-11am

Since the last meeting, further additions with regards to optimization were taken but problems with efficiency were still encountered. A canvas, with regards to data visualization, was implemented accordingly and seems to work as needed.

This meeting dealt with a discussion on how the efficiency could be enhanced. An additional update on generator usage needs to be constructed for the time being in order to optimize the program as much as possible. Further possible updates will be discussed in the next meeting so as to try and complete this section.

By the next meeting, optimization should be continued and a summary of a given paper (shown in the next entry) needs to be written.

Entry 13

Paper Summary: 'Computational solutions for omics data' by Bonnie Berger, Jian Peng, and Mona Singh

In the area of high-throughput sequencing, data generation and computing power are being increasingly diverged over time such that omics data analyses pose certain difficulties.

One such difficulty is the algorithmic efficiency needed to process large datasets. Genome assembly deals with the generation of a reference genome to which sequences can be analysed against and thus needs to be accurate, fast and have efficient storage methods. Most efficient assemblers are built on the de Bruijn graph as it reduces fragment assembly to the Eulerian path while others use FM-indexing. Both graph-theoretical methods still encounter problems to accurately assemble a large genome such as the human one. Read mapping in next-generation sequencing deals with the mapping of reads to a reference genome and thus needs to incur low running times and computational costs. Most efficient short-read aligners use FM-indexing to pre-process the genome compactly. Hardware accelerated algorithms use parallel dynamic programming, multicore CPUs or cache-oblivious algorithms to increase the software's speed. Large-scale genome sequencing deals with reducing the size of sequencing data for storage and analysis (sequence search). Compression algorithms include reference-based methods for re-sequencing and non-reference-based methods for repetitive DNA segments. Compressive genomics, used since search algorithms are becoming too slow, compresses data such that analysis can be implemented without decompression needing to be done by making use of genomic redundancy.

Another difficulty is data mining for transcriptomics – transcriptome quantification by RNA sequencing (RNA-seq). Identifying cell-specific expression signals within tissue profiles is done using linear algebraic methods which need measured cell type proportions to weight a linear mixed model. When proportions are not available, matrix factorisation or differential geometry methods are used to estimate. Identifying regulatory and phenotypic genes and modules (gene expression analysis) is done using: statistical methods, probabilistic graphical models, and sparse learning. Identifying gene expression alterations in disease (ex. comparisons between tumour cells and normal cells) is done using software which either extends the graphical model or uses a Bayesian network to construct pathways. Methods are not standardized so large-scale application is lacking.

The other difficulty is integrative interactomics whose analyses involve modularity – interactomes/networks being represented as graphs. Analysis of heterogeneous genomic data sets can be done using sub-networks and local clustering to uncover specific modules of interest. Network flow can be used to propagate biological processes or to identify proteins. Cellular networks can also be used for module and pathway analysis though random walk-based approaches; for example the Isorank algorithm. Interactome analysis of disease data sets are due to mutations and variations identified by sequencing certain individuals. They show that the genes may differ but the pathways are generally shared amongst the afflicted. Genome-wide association studies (GWASs) identify these leading to disease gene prioritisation and uncovering of related pathways. The modularity of such genes can be tested using permutation-based approaches. Also, the problem of set cover has been found to be beneficial when considering this heterogeneity.

Genomes, transcriptomes, proteomes, interactomes and methylomes are laboratory generated. Omics analyses' algorithms lead to the correct usage of these in biological areas. All the above areas should be researched extensively as high-throughput technologies are continuously advancing.

Entry 14

Meeting: 24th October, 11:30am-12:30pm

Since the last meeting, the mentioned summary was written as depicted above. Further optimizations were taken, ensuring efficient parsing of the reference genome and the sequencing reads.

This meeting firstly dealt with slight modifications to the parsing functions so that the best possible implementation could be used (as multiple methods were constructed). Furthermore, a discussion on how different methods of alignment could be applied to the reference genome and the accompanying reads took place; for example: edit distance, suffix arrays and Boyer-Moore to name a few.

By the next meeting, these different methods should be constructed so as to compare them between each other in order to analyse which would be best. Papers on the evaluation of aligners should also be found.

Entry 15

Meeting: 31st October, 11am-12pm

Since the last meeting, different alignment methods were constructed; namely: naïve approximate matching using edit distance, k-mer indexing and FM indexing using Burrows Wheeler Transform and suffix arrays.

This meeting firstly dealt with a discussion on how the efficiency of the alignment methods should be compared together by means of a graph for easier analysis. The problem of parsing the genome – that it takes too long on my personal laptop as compared to the supervisor's PC – can possibly be fixed by reading the genome into an *n*-bit integer instead of a string (thus achieving compression); i.e. saving the data as binary. As the possible nucleotides are A, C, G and T, they can each be assigned 2 bits; for example: 00, 01, 10 and 11.

By the next meeting, a comparison method between the alignment functions should be constructed as well as a compression technique for the DNA in the reference genome.

Entry 16

Paper Summary: 'Efficient storage of high throughput DNA sequencing data using reference-based compression' by Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, Ewan Birney

Analysis of DNA sequencing data has a growing need of handling its storage in an efficient manner. It is the first molecular data for which storage costs make up a large portion of the overall analysis costs. Three proposed approaches are to add storage, throw away some data or compress the stored data. Adding storage seems unlikely and human samples are non-

renewable so permanent electronic archives are preferred; i.e. the most feasible approach is to compress the stored data.

DNA's natural representation is a string of characters so it can be compressed using generic methods. Its compressibility can benefit from biological components such as repeat content and relationship to existing sequence. Two such works are DNACompress and DNAzip. A new and more efficient method is that of reference-based compression. It is primarily based on compressing whole genome information due to the progressive area of study/research.

The lossless reference-based compression method is based on the idea of efficiently storing identical or near-identical input sequences. The reference genome is used as a compression framework where new sequences which are identical to the reference have little impact on storage. Most reads match the sequence perfectly or near-perfectly so a mapping of the reads against the sequence is taken with the aid of Golomb codes to store relative encoded read positions and Huffman coding to compress length of reads. This technique's efficiency increases proportionally to the read length irrespective of the coverage and it increases proportionally with the coverage as well. It compares well with bzip2 compression and is more efficient than BAM-based storage.

Unmapped reads cannot be stored in the same manner so the idea of a 'useful' compression framework is implemented. These unmapped reads are pooled together to serve as said secondary framework by use of a De Bruijn graph framework. Base quality scores are stored relative to if their positions show variation (if not, they are not stored) along with a user-defined percentage, both being compressed using Huffman coding. The lowest base qualities which are identical to the reference are stored to ensure higher compression at longer read lengths.

All positions are relative encoded and stored as a Golomb code. As for size, exact read positions (encoded using strand and match flags) take up 1 bit each. Inexact matches are stored using a list of variations (read position, variation type and other information) with the variation type taking up 1 or 2 bits. Given any base, a substitution takes up 2 bits. Inserted quality scores take up 2 or 3 bits.

Hence, the compression method described constructs an explicit balance between storage cost and data precision. It provides efficient lossless compression for read alignments with little or no differences to the reference genome at approximately 0.02 bits/base. As for unmapped reads, it is still uncertain whether they should be stored or removed due to storage constraints (the decision being based on the data set being analysed). The implementation is still a prototype for BAM-based input however it can be improved according to some complex details with the necessary effort.

Entry 17

Paper Summary: 'A Compression Algorithm for DNA Sequences and Its Applications in Genome Comparison' by Xin Chen, Sam Kwong, Ming Li

DNA sequences only contain 4 nucleotide bases so 2 bits are enough to store each one. The compressibility of DNA sequences is supported by the fact that they contain multiple approximate repeats, even though this is often shadowed by mutation, cross-over, translocation, reversal events and sequencing errors.

GenCompress is a lossless DNA compression algorithm which is based on approximate matching. It even aids in comparing two sequences using a relatedness measure. Two other lossless compression algorithms are Biocompress-2 and Cfact, both being based on exact matching.

The approximate matching algorithm implemented considers three standard edit operations: replace, insert and delete. It can be easily deduced that an infinite number of edit sequences can be derived from a single transformation – the list of edit operations being referred to as the Edit Transcription $\lambda(u, v)$. Thus, by knowing u and $\lambda(u, v)$, v can be derived/encoded using any of these four methods: two bits encoding, exact matching, approximate matching and approximate matching using edit operation sequence. The third case returns the minimal number of bits, with the first case coming in at a close second.

GenCompress generalises the dictionary based, Lempel-Ziv compression algorithms for approximate matching. It is a one-pass algorithm. In order to limit the search, a certain condition C is used as a constraint; i.e. only approximate matches which satisfy $C = (k, b)$ are searched for – experimentally (12, 3) is best. A compression gain function G is also defined to check if a specific repeat leads to encoding benefits. Thus, with both C and G, an optimal prefix can be deduced using a parsing procedure. By analysing G, it was found that no approximate match in the database to help save bits exists. However, the approximate reverse palindrome can be detected.

The algorithm's main aim is to acquire an accurate compression ratio for DNA sequences; time complexity being considered afterwards. Exhaustively searching for optimal prefixes is too time consuming so some observations are made: an optimal prefix always ends right before a mismatch and the optimal edit operation sequence is reflective in a sense. Using both of these, the optimal prefix and palindrome searches can be constructed.

Two versions of GenCompress were tested – one with replacement operations only and the other with all the edit operations – against BioCompress-2 and Cfact. It was concluded that approximate matching achieves the best compression ratio and is best for finding common parts in DNA sequences. If however there are not enough approximate repeats in the sequence, then BioCompress-2 performs better.

Furthermore, the relatedness or mutual information between two DNA sequences can be calculated using the 'distance' between their pairs; close sequences being close on the evolutionary tree. Minimum alignment scores, genome rearrangement distance and reversal distance are usually used for closely related sequences. For not closely related sequences, a symmetric distance is defined using Kolmogorov complexity and it can also be used to build evolutionary trees from un-alignable DNA sequences such as complete genomes. Note that for this simple alignment distance, GenCompress has been converted into conditional compression.

Entry 18

Meeting: 7th November, 11am-12pm

Since the last meeting, the comparison method for the alignment functions was built and seems to work correctly. Also, the Huffman coding technique was implemented in order to try and compress the genome. This however proved to be inefficient as the file remained the same size.

This meeting dealt with a discussion on how the genome could be correctly compressed. The genome should be stored as integers instead of a string; i.e. storing the genome as an array of integers or using the inbuilt Python function bitarray. Correct implementation of this would lead to the need of updating all the other functions in that they read in integers and not a string. The progress report should also be started according to the given LaTex template. For now, completing the introduction and bibliography should be enough.

By the next meeting, a conversion tool should be built for compressing the genome into an integer along with the resultant required updates. The progress report should also be started.

Entry 19

Paper Summary: 'Evaluation and Comparison of Multiple Aligners for Next-Generation Sequencing Data Analysis' by Jing Shang, Fei Zhu, Wanwipa Vongsangnak, Yifei Tang, Wenyu Zhang, Bairong Shen

Next-generation sequencing (NGS) technology is used for areas pertaining to genome evolution and genomic variation among others. The generated data is aligned and mapped on the reference genome using different alignment algorithms, all of which are evaluated according to their feature, performance and accuracy hereunder.

An algorithm-based classification of multiple aligners was designed for three NGS platforms – Roche454, Illumina and ABI SOLiD – to adapt them to high-throughput data. The two main classifiers were based on the hash table-based algorithm and the Burrows-Wheeler Transform (BWT) based backtracking algorithm. The former employs the seed-and-extend strategy with k-mer indexing to improve high-throughput short reads as well as a dynamic programming algorithm (Smith-Waterman or Needleman-Wunsch) to extend the alignment; also being able to align multiple-error reads. The latter employs the prefix/suffix tree and suffix array data structures with FM-indexing for fast read searching and to solve alignment to genome copies as well as a reversible data compression algorithm (BWT) to decrease the memory usage of the previously mentioned data structures.

With regards to application-specific features, most of the aligners were found to support paired-end alignment for repetitive areas and a few of the aligners did not have the function for SNPs and structural variation discovery. The most important support needed was deemed to be gapped alignment, paired-end alignment, trimming alignment and bisulfite alignment. With regards to computational performance, computation time, maximum memory usage and mapped read counts were analysed as described below.

Most of the aligners exhibited a linear relationship between the computation time and the reference genome size. Also, most read counts effected computation time in that most aligners seemed to depend more of these counts than on the genome size. Overall, BWT-based backtracking algorithms (Illumina) had a faster computation time when compared with the others regardless of size and reads. It was also found that if multiple threads are utilised, computation time either increases or decreases depending on the aligners being implemented.

Variation of maximum memory usage was analysed by comparing the aligners against the server's memory usage percentage. It was showed that said usage was quite low and independent of the genome size so even a low RAM could run them. In the case of the

human genome, memory usage increased dramatically. Also, utilisation of multiple threads again showed either an increase or a decrease of usage due to the differing algorithms.

Mapped read counts can generally lead to evaluation of read density. Most aligners showed similar results for short-read datasets but only a handful showed this for long-read datasets. That being said, an accurate deduction for aligner capability and sensitivity could not be made as real-life data's true alignment locations are unknown. To compensate, the accuracy was instead evaluated with *in silico* data. Most aligners showed a high sensitivity as well as precision increase with multi-mapped reads. Datasets having differing error rates, indel sizes and read lengths showed high percentages of total and corrected multi-mapped reads.

In conclusion, this study should aid the user (mainly biologists) in choosing which aligner is most suitable for their research area with regards to NGS data; i.e. an appropriate selection of an aligner for the application in question can be made.

Entry 20

Paper Summary: 'Short read alignment with populations of genomes' by Lin Huangy, Victoria Popicy, Serafim Batzoglou

The increasing amount of genomic data poses problems to: short DNA sequence (read) alignment to reference genomes and variation discovery of newly sequenced genomes with respect to those previously sequenced. Already constructed short-read alignment programs use BWT as it returns linear time and a small memory requirement. The BWT of the reference genome is pre-run so later on, the reads can be mapped to it using BWT backwards search. This leads to biases and lower accuracy according to the reference chosen.

Read mapping of a newly sequenced human genome to a collection of genomes has not yet been accomplished. One attempt is GenomeMapper (Schneeberger et al.) which uses hash-based data structures and k-mer indexing whereby identical regions (due to the redundancy of sequenced genomes) are stored only once. Another (by Siren et al.) uses prefix-sorted finite automaton and BWT-based indexing. Both have a large memory requirement. To handle genetic variants and avoid any bias, BWBBLE (a BWT-based read alignment algorithm) was constructed to provide little memory consumption and relatively efficient computation time.

A reference genome can be augmented with genomic variant data from a collection of genomes; the augmented reference being the reference multi-genome. SNPs are handled by extended the reference's alphabet from 4 to 16 letters for the IUPAC nucleotide code. Thus, when aligning a read to the reference, a nucleotide can match more than one character leading to multiple separate SA intervals. These need to be minimised by the four-bit Gray code (reflected binary code), theoretically the optimal order for such a problem. Other than SNPs, differing types of genomic variations are common; i.e. insertions, deletions, inversions and translocations. These generate multiple branches (bubbles) with the SA. While these augmentations aid in handling alignment, a higher memory overhead is produced. It is however possible to reduce memory consumption by filtering out branches – lower accuracy.

The first method implemented for the BWBBLE program is that of exact matching of a read to the reference multi-genome, an extension of the BWT-based backwards search algorithm. This was then further extended to allow mismatches and gaps; i.e. inexact matching, an extension of the inexact search algorithm employed by BWA with reduced search space and improved performance. As for the memory required by the program, to reduce it the BWT

string is compressed and only a subset of the SA arrays is stored. Further reduction would unfortunately lead to increased time consumption.

The BWBBLE aligner's performance was evaluated against BWA (a greatly known BWT-based single-genome aligner) resulting in BWBBLE performing better on reads which cover a larger number of indels with a high SNP count. It does have a slower running time due to the larger amount of SA intervals but running on a multi-genome as opposed to a single-genome will lead to less running time so it compensates for itself. Furthermore, most of the same reads are aligned by both. It was also evaluated against GCSA (executes short read alignment with multiple sequences) resulting in BWBBLE taking a lot less time to construct its index.

Thus, a disadvantage of this aligner is its use of read length being dependent on padding to gather variants implying construction of multiple indices for each length. An advantage is its efficiency in aligning a multi-genome when compared to aligning each genome separately.

Entry 21

Paper Summary: 'Comparative analysis of algorithms for next-generation sequencing read alignment' by Matthew Ruffalo, Thomas LaFramboise, Mehmet Koyutürk

Next-generation sequencing techniques support many applications which produce a large amount of data, leading to many difficulties. Re-sequencing is a process whereby short reads from an individual's genome is compared/mapped against an already sequenced reference genome. Multiple factors must be taken into account for this process to occur, namely: sequencing error, short read length and volume of reads. A lot of short read alignment algorithms are available and so, a simulation and evaluation suite SEAL which simulates short read sequencing and analyses the performance of the algorithm was constructed.

Some of the read alignment programs already available are discussed. Bowtie uses a BWT index having small memory consumption but with no assurance of a high quality read mapping if no exact match exists and if it is configured for maximum speed. BWA also uses a BWT index (unlike its predecessor MAQ which uses a hash-based index) having fast searching and reliable quality scores. The mr- and mrs-Fast tools report all mappings of a read rather than only the best one, thus aiding in structural variant detection. They use a seed-and-extend method along with a hash-based index, i.e. a kmer index. Novoalign uses a hash-based index similar to MAQ with a high accuracy rate. SHRiMP uses q-gram filters, spaced seeds and a faster Smith-Waterman algorithm. SOAPv2 uses a BWT hash-based index having a fast alignment speed and a larger memory than its predecessor SOAPv1.

SEAL evaluates the programs' performance by first simulating some reads from a reference genome, with the user having the option of altering certain parameters such as: read length, sequencing error rate (current platforms reporting it at 1%), indel rate, indel length and coverage. It should be kept in mind that coverage does not directly affect accuracy but it should still be realistic due to performance. As most tools report a mapping quality score using Phred scores, the evaluation considers only those reads whose score is greater than a certain value implying a high quality – and thus high accuracy and performance rates. Also, two evaluation methods are taken as some tools report all matching positions while others report only the best matches. In fact, the accuracy of mappings is defined to be either correctly mapped, strict incorrectly mapped, relaxed incorrectly mapped or unmapped. Thus, the strict and relaxed reads provide an accuracy interval if all positions are reported.

Accuracy results were computed according to varying error rate, varying indel sizes and varying indel frequencies. The former shows that Bowtie, BWA and Novoalign are the most sensitive to mapping quality threshold at high error rates and SOAP has a high accuracy even at the lowest possible threshold. The second shows SOAP failing to align reads as it is not quite suited for indel calling, Bowtie, BWA and Novoalign have low accuracy when the threshold is low but have many incorrect mappings with low scores and mr/s-Fast are better with longer indels. The latter shows that all programs' accuracy depends on indel rate – they are inversely proportional. Runtime results were computed according to indexing time and alignment time. Most of the programs exhibit a linear relationship between genome length and index construction time. It can also be observed that most of them have a trade-off between the runtimes (speed versus accuracy) in order to optimise variation detection.

These results should prove beneficial for genomic researchers, keeping in mind that not all experimental scenarios and hardware characteristics could be simulated.

Entry 22

Meeting: 14th November, 11am-12pm

Since the last meeting, two conversion tools for compressing the genome into a bitarray as well as an integer were completed. However, the actual compression was deemed too little to be satisfactory.

This meeting dealt with a discussion on how the implementations should be tested to see how much compression is occurring with respect to the genome size (1MB, 100MB, 500MB…). A table detailing this data should be constructed. Also, the original genome parsing function should be tested on a Linux OS as opposed to a Windows OS to see if the problem could be solved in that way.

By the next meeting, the above mentioned table should be constructed and the implementation tested on a Linux OS.

Entry 23

Meeting: 22nd November, 2-2:30pm

Since the last meeting, a table detailing how the compression methods ran with respect to the genome size was constructed. It was found that the compression achieved by both methods is in fact quite satisfactory (previous observations were deemed incorrect). The following analyses took place:

Original File Size (KB)	Compressed File Size(KB) - Int	Compression % - Int	Compressed File Size(KB) - BitArray	Compression % - BitArray
1,024	285	72.17	256	75.00
10,231	2,842	72.22	2,558	75.00
100,152	27,820	72.22	25,038	75.00
500,759	139,100	72.22	125,190	75.00

Compression % = $(1 - \frac{size\ after\ compression}{size\ before\ compression})\ x\ 100$

For .gz zipped human genome:

Original File Size (KB)	Compressed File Size (KB) - Int	Compression % - Int	Compressed File Size(KB) - BitArray	Compression % - BitArray
960,605	952,915	N/A	774,241	N/A

The compression does not apply as this original file size is its zipped version.

For unzipped human genome:

Original File Size (KB)	Compressed File Size (KB) - Int	Compression % - Int	Compressed File Size(KB) - BitArray	Compression % - BitArray
3,196,759	952,515	70.20	774,241	75.78

This shows that the integer implementation (compression of genome into an integer) achieves an overall 70% compression rate while the bit array implementation (compression of genome into a bit array) achieves an overall 75% compression rate.

Additionally, the original genome parsing function was tested on a Linux OS as opposed to a Windows OS leading to it running efficiently. However it still halted on further on. Further coding reads the compressed genome efficiently but a problem arose where the program halted in the reads parsing function – the generator was not working as supposed to.

The Hamming distance matching function was also updated so that it supports integers.

This meeting dealt with a discussion on why the reads parsing function was halting. It was deduced that the integer was overflowing and hence the program was killing itself. It should be altered similar to the integer parsing genome function. Also, further amendments to the matching functions should be applied in order to support integers instead of strings.

By the next meeting, the above mentioned updates should be continued along with updates to the visualization tool to support integers. The progress report should also be continued.

Entry 24

Meeting: 28th November, 11-11:30am

Since the last meeting, the read parsing function was updated but it still seemed to halt at a certain point. One of the matching functions (Hamming distance) was updated to support integer input and the visualization tool was deduced to not need said support – only further refinement is required. The first draft of the progress report was also completed.

This meeting dealt with a discussion on why the sequencing reads generator was halting. After analyzing the function and the human reads file, it was seen that said file is formatted differently than the PhiX file which was being tested on. As such, the function needs to be converted to read the human file accordingly.

By the next meeting, i.e. tomorrow, the function needs to be converted so as to continue the rest of the analyses. The progress report draft should also be sent to my supervisor for feedback.

Entry 25

Meeting: 29th November, 4:30-5pm

Since the last meeting, the read parsing function was converted accordingly and was verified to work as needed.

This meeting dealt with a discussion on the alignment functions and how they could be converted to support integers. The reasoning behind the Hamming distance algorithm was deduced to be sound however the inconsistencies produced by the bounds was noticed; i.e. an exclusive or operation producing 11 should be considered as one mismatch, not two, since 11 corresponds to the nucleotide 'T'. The Burrows-Wheeler Transform algorithm could be tackled by first taking note of any repetitive integers in the compressed genome as well as any recurring patterns of integers – pairs, triples, etc…

By the next meeting (after the exams), the Hamming distance function should be updated. Histograms detailing the integer redundancy in the genome should also be constructed.

Entry 26

Meeting: December

Drafts of the progress report were discussed with my supervisor until a final version was submitted.

Entry 27

Paper Summary: 'Long Read Alignment with Parallel MapReduce Cloud Platform' by Ahmed Abdulhakim Al-Absi1, Dae-Ki Kang

The existing sequence aligners mostly support short read genomic sequences and lack support in cloud environments. They exhibit deficiencies in the alignment of long sequence genomic data that are currently generated using NGS technologies. The existing long read aligners that adopt the cloud platform for computation suffer from certain disadvantages. Thus, a cloud infrastructure and the MapReduce framework are combined together as a solution to support long read sequence alignment.

A dual phase execution is used in the MapReduce model; in the first phase, input data is split into fragments (associated with a mapper providing key-value pairs as outputs) whereby reduce workers are provided – key-sorted(value) pairs – to store the results in the Hadoop files system. The workers are usually virtual machines in public cloud environments.

The Burrows-Wheeler Aligner's Smith-Waterman Alignment on Parallel MapReduce (BWASW-PMR) cloud platform for long sequence alignment is implemented to solve the problem of serial execution for the map/reduce phases. The main developments for this long sequence alignment strategy are: the optimisation of SW in the BWA-SW alignment, a custom MapReduce framework to support the required computations, a parallel map and reduce workers execution strategy and a parallel execution of the map and reduce functions at worker nodes.

BWA-SW Alignment relies on the SW algorithm to align the seed matches of sequences using a similarity matrix and backtracking algorithm. The BWA-SW algorithm constructs a full-text index using FM-indexing of the query and reference sequences whereby a prefix directed acyclic word graph and a prefix trie are built respectively with the aid of suffix arrays and their intervals along with a dynamic programming mechanism. To optimize these computations, a reverse post-order traversal scheme is implemented.

The BWASW-PMR uses the MapReduce computation model for cloud computation whereby map and reduce worker nodes are deployed on a cloud cluster consisting of VMs. It considers the genomic sequence alignment in dual phases – map and reduce. In Hadoop, the map phase is executed and then the reduce phase is initiated. A parallel execution strategy of the two phases is considered to overcome Hadoop's disadvantages; i.e. phases are modelled to run in parallel utilizing all programming cores available in the worker VMs.

The optimization of SW is achieved using a wave front parallelization technique. Execution time of the optimised SW is significantly lower than the standard SW. Comparing BWASW-PMR Cloud and Bwasw-Cloud single computing node, the BWASW-PMR aligner showcases a significant speed-up. Comparing BWASW-PMR Cloud and Bwasw-Cloud on Azure, BWASW-PMR exhibits lower makespan time and long sequence alignment is faster.

The results obtained indicate significant improvement and is thus of use to the genomic community to support the required computations for long sequence alignment efficiently. The parallel executions of the map and reduce phases along with SW optimization are the main contributing factors for these experimental results.

Future undertakings include: optimisation of the BWA-SW algorithm as it uses a lot of memory; accelerating the BWA-MEM algorithm of Burrows Wheeler aligner on different platforms.

Entry 28

Paper Summary: 'MapReduce: Simplified Data Processing on Large Clusters' by Jeffrey Dean, Sanjay Ghemawat

MapReduce is a programming model for processing and generating large data sets which involves: a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values (passed from *map* via an iterator) associated with the same intermediate key. Programs written in this style are automatically parallelised, run on a large cluster of distributed commodity machines and highly scalable (processes terabytes of data on thousands of machines). Some examples include Distributed Grep, Count of URL Access Frequency, Reverse Web-Link Graph, Term-Vector per Host, Inverted Index and Distributed Sort.

Different implementations of the MapReduce in interface are possible depending on the environment. The implementation involving large clusters of commodity PCs connected together with switched Ethernet is considered here. The *Map* input data is distributed into partitions/splits across multiple machines using parallelisation while the *Reduce* data is distributed by partitioning the intermediate key space.

The MapReduce library must incorporate fault tolerance mechanisms as large amounts of data are being processed. Faults that may occur are master (special copy of program on a specific machines) failure, worker (the other copies on the rest of the cluster of machines) failure and semantics in the presence of failures. A straggler – a machine which takes an unusually long time to complete one of the last few map or reduce tasks in the computation – can lengthen the total execution time but this can be bypassed by the master scheduling backup executions of the remaining in-progress tasks when the MapReduce operation is close to completion.

Network bandwidth is conserved by storing the input data on the local disks of the cluster of machines – most input data is read locally and consumes no network bandwidth when large MapReduce operations are run on a majority of the workers in a cluster. Task granularity is achieved by subdividing both phases into a number of pieces (having practical bounds), ideally larger than the number of workers.

The writing of the map and reduce functions is generally sufficient for most needs but some extensions may prove to be useful. Examples of these include: the partitioning function, ordering guarantees, the combiner function, input and output types, side-effects, skipping bad records, local execution, status information and counters. The performance of MapReduce was measured over cluster configuration, Grep, sort, effect of backup tasks and machine failures.

The MapReduce library has been used in multiple domains including: machine-learning, clustering, extraction of data and graph computations. Its most significant use is for large-scale indexing.

With MapReduce, a simple program may be written and run efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. It is easy to use as it hides the details of parallelization, fault-tolerance, locality optimization and load balancing. A large variety of problems are easily expressible as MapReduce computations. It also scales to large clusters of machines comprising thousands of machines.

Related work includes: Bulk Synchronous Programming, MPI primitives, active disks, the eager scheduling mechanism used in the Charlotte System, the cluster management system used in Condor, the sorting facility used in NOW-Sort, the communication system in River implemented by sending data over distributed queues and TACC.

Entry 29

Paper Summary: 'Rapid Parallel Genome Indexing with MapReduce' by Rohith K. Menon, Goutham P. Bhat, Michael C. Schatz

The MapReduce parallel programming model is improved by accelerating the suffix array (SA) and the Burrows-Wheeler Transform (BWT) constructions of a DNA sequence for alignment purposes. The former index leads to rapid binary search algorithms for matching query sequences of any length while the latter one reduces the space requirements of the SA – from 12GB to 3GB – by recording as an index a (reversible) permutation of the string based on the ordering of the SA. The MapReduce algorithm uses the data processing capabilities of MapReduce to divide the suffix array construction into multiple independent ranges that are then independently solved (in parallel) in order to reduce computation time.

The suffix array is an index consisting of the lexicographically sorted list of all suffixes in a genome sequence. It enables fast, variable-length lookups for any substring in the reference genome thus accelerating sequence alignment computations. It also supports inexact alignment algorithms that allow for some differences between the reference and query sequences by using a seed-and-extend technique which finds relatively short exact matches using the suffix array to anchor the search for longer potentially in-exact matches. In practice the suffix array records only the list of suffix offsets for lookup as opposed to explicitly storing each suffix as otherwise the memory requirement would be intractable. The closely related BWT index structure reduces the space requirement more by using a permutation of the sequence as an index. It is the last column of the Burrows-Wheeler Matrix (BWM), a lexicographically sorted matrix of all of the cyclic permutations of the string.

MapReduce can distribute computation across a cluster with hundreds or thousands of computers, each analysing a portion of the dataset stored locally on the compute node. After an initial round of independent parallel computation, the machines efficiently exchange intermediate results, from which the final results are computed in parallel. The parallel computation is split in three phases – map, shuffle, and reduce; map scans the input dataset and emits key-value pairs representing some relevant information of the data tagged by the key, shuffle distributes and shuffles all values associated with a given key to collect them in a single list (a parallel distributed merge sort of the key-values pairs) while reduce processes these lists to compute the final results.

The MapReduce implementation of suffix array and BWT partitions the suffixes into non-overlapping batches with lexicographically similar values, and then sorts the batches on different machines across the cluster. The suffixes are independently assigned to partitions in parallel according to their prefix and the batches can be independently sorted.

In order to improve load balance among the reducers, a partitioner which samples the genome to select the boundaries of the batches based on the true sequence distribution was implemented. In order to improve the batch sorting performance, an optimized recursive bucket sort that pre-computes and determines problematic repeats on-the-fly was also implemented. Optimizing single and multiple character repeats accelerates the recursive sort. Also, by increasing the number of reducers, the number of suffixes per reducer is decreased along with the peak memory usage and the runtime is improved.

The algorithm was substantially limited by the Hadoop overhead (approximately half the runtime is used by it alone). There is an approximately linear relationship between the genome size and the runtime meaning the algorithm performs well in the presence of complicated repeats.

Entry 30

Meeting: 8th February, 11-11:30am

Since the last meeting, analysis of the integer redundancy in the genome was done. Research on the MapReduce method was also completed.

This meeting dealt with a discussion on ways on how to improve the genome compression for Burrows-Wheeler Transform. It was decided that results should be gathered on how often certain words occur (both 4-letter and 8-letter combinations of A, C, G and T; i.e. int and

long respectively). The implementation of this should be written using Hadoop and Python in order to get used to the MapReduce system - it will be similar to the basic WordCount method generally introduced by MapReduce.

By the next meeting, analysis on the frequency of words in the genome should be gathered, preferably using Hadoop.

Entry 31

Meeting: 13th February, 10-10:30am

Since the last meeting, the Hadoop system was configured and confirmed to work correctly by implementing a rudimentary mapper and reducer. As for the frequency of words, the implementation seems to have some bugs which need fixing.

This meeting dealt with a discussion on what analysis should be done on the human genome. Firstly, the word count function should be remedied (removed of bugs). A sliding window of one step should be taken for accurate readings of both int and long patterns. A comparison graph should be constructed detailing the size of the uncompressed genome, the compressed genome using inbuilt zip, using int (32-bit) and using long (64-bit). All of these may be constructed using some form of map-reduce.

After deducing which compression method is most useful in the next meeting, BWA will be developed to work on the chosen method; after which MapReduce will be used to facilitate said process.

By the next meeting, the present bug should be taken care of, the sliding window approach should be implemented and a comparison graph should also be constructed.

Entry 32

Meeting: 20th February, 10-11am

Since the last meeting, the sliding window approach was taken in order to generate a word count of both int and long patterns. The results were documented. This process was also implemented with MapReduce, however it could not be tested as Hadoop failed to start the namenode. A comparison graph detailing the compression methods taken was also plotted; leading to the conclusion that the integer compression performs best.

This meeting dealt with a discussion on what further steps could be taken. The compressed reference genome should be decoded back to its original uncompressed version to ensure correctness. The compression methods should also be tested against known techniques such as LZ77, LZ78 and LZW. The Hadoop configuration should be fixed in order to run the MapReduce word count on it.

Furthermore, the reference genome should be read in overlapping regions for alignment (the size of the overlap being equivalent to the size of the read). This can be done by either reading and aligning the uncompressed version or by reading and aligning the integer compressed version. Both sets of results should be compared. The mentioned alignment should be done using BWT and MapReduce. Another aligner should also be installed to compare the results.

Entry 33

Meeting: 27th February, 10-11am