'Algorithms for DNA Sequencing' by Ben Langmead
Online Course (Notes)

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:

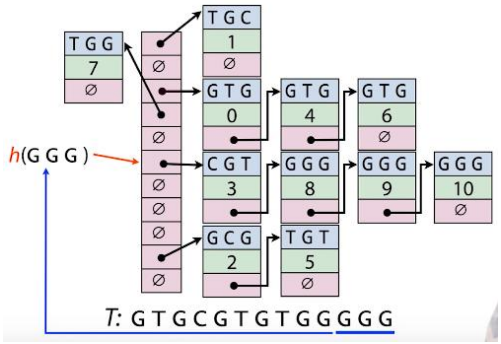
CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

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:

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

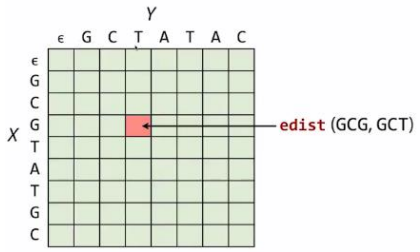
$$\delta(x, y) = 0 \text{ if } x = y, \text{ or } 1 \text{ otherwise}$$

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

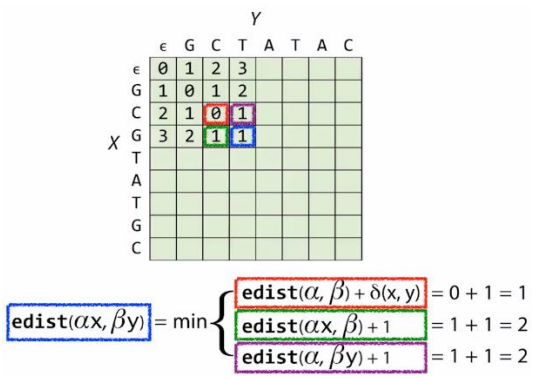
The function for the edit distance is very slow as multiple recursive calls occur when the function is run, some of them being duplicates; i.e. will return the same result – wasteful. It would be better if the answers of duplicate calls were remembered so that said call would not need to be run again.

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:

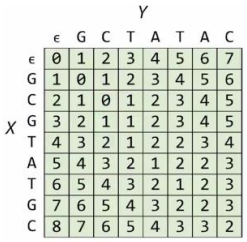


The bottom-right element is special as it will eventually contain the edit distance between both of the entire strings – the solution.

Using the expression defined previously, the following procedure is done:



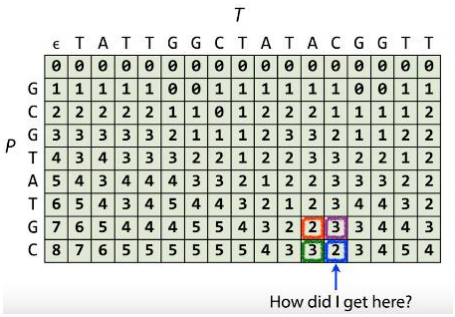
The first row and the first column correspond to the edit distance between the empty string and something which is always equal to the length of the other string – i.e. their values are ascending integers. Below is the filled in matrix:



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.

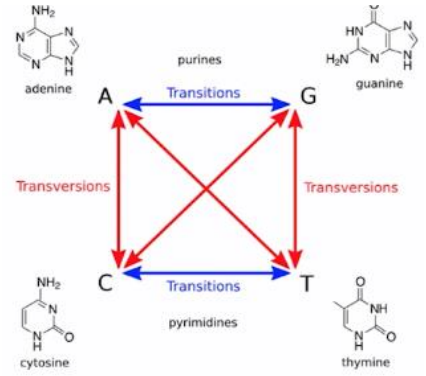
37. Meet the family: Global and Local Alignment

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).

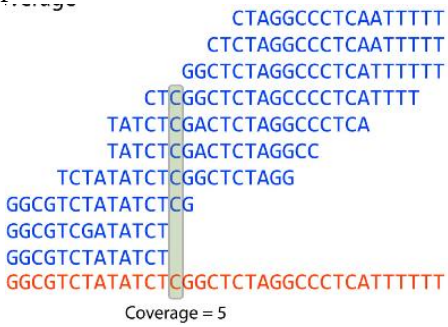
Another version of the problem is assembly (De Novo assembly or De Novo Shotgun assembly). 'De Novo' means from scratch and 'shotgun' refers to the fact that the reads are coming randomly from all over the genome. The benefit of being able to see the picture of the completed puzzle is not given; for 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.

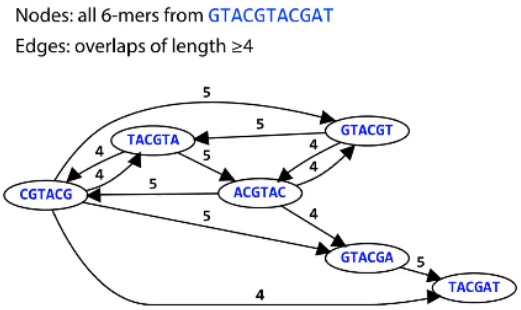
A perfect match is not always the case for this law. This is because of sequencing errors and polyploidy (ex. humans have two copies of each chromosome, and copies can differ).

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

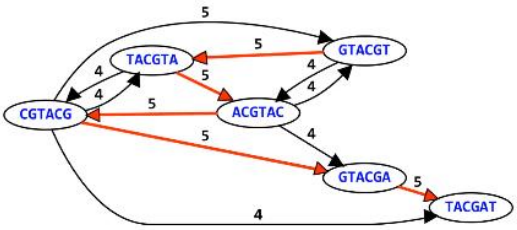
In order to represent all the overlap relationships at once, a directed graph will be built (nodes as ovals and edges as arrows) – the nodes represent objects/concepts and the edges represent relationships among them.

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:

order 1: AAA AAB ABA ABB BAA BAB BBA BBB
AAAB

order 1: AAA AAB ABA ABB BAA BAB BBA BBB
AAABA

order 1: AAA AAB ABA ABB BAA BAB BBA BBB
AAABABB

order 1: AAA AAB ABA ABB BAA BAB BBA BBB
AAABABBAABABBABBB ⟵ superstring 1

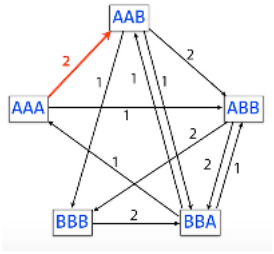
This is a candidate superstring. It is not necessarily the SCS but it might be, depending on if the right ordering was picked. Different orderings will result in different superstrings. For example:

order 2: AAA AAB ABA BAB ABB BBB BAA BBA
AAABABBBAABBA ⟵ superstring 2

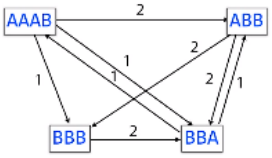
To find the absolute SCS, all the orderings needs to be tried. So for every possible way of ordering the n input strings (for every permutation) adjacent pairs of strings will be merged together according to their maximal overlap. The SCS over all the orderings will be the resultant SCS.
This is so slow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

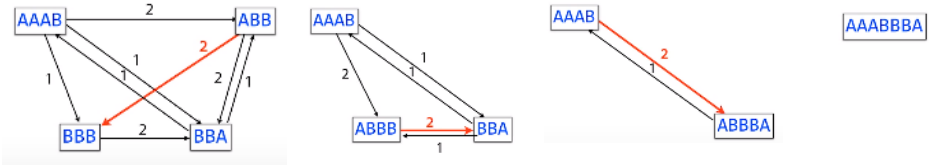
This algorithm is much faster then the previous one. It is called greedy as it makes a series of decisions and at each decision point, it will choose the option which reduces the length of the eventual superstring the most. This seems to be a good stategy however making the greedy decision at each point 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):

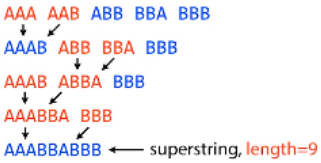


This merge has made the graph slightly simpler/smaller – one less node and at least one less edge in it. The same process is repeated until one node is left:



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.

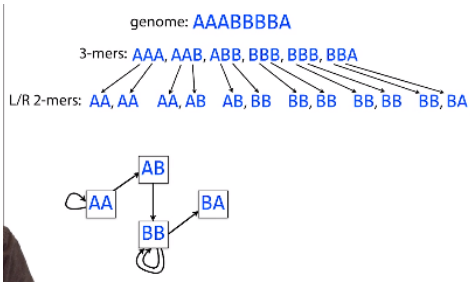
An alternative to the SCS is the De Bruijn graph and Eulerian walk method that avoids the over-collapsing problem. However it still cannot avoid the overall problem: that repeats make assembly difficult.

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

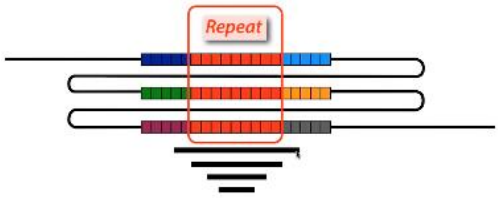
How do real software tools for solving the assembly problem work and how do they deal with repetitive genomes and other issues in practice? Although the SCS and Eulerian walk algorithms won't be used, their graphs (the overlap and the De Bruijn graphs respectively) will still be used. The graphs correspond 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…
