Interim Report #2 - MinCE: fast quantification of large metagenomics datasets along with species and strain abundance

Student: Thorhallur Audur Helgason

Mentor: Professor Lior Pachter

Definitions

metagenomic sample - a sample from 'the real world', such as a water sample from a lake or a fecal sample from a patient, as opposed to a sample from bacteria cultured on a petri dish. Metagenomic samples are thus expected to contain multitudes of different species of organisms.

fastq sequencing data - an unordered collection of short 'reads' from a sequenced sample, with each read being a string representing a snippet of genetic material found in the sample. For a metagenomic sample, these reads would be expected to be from multiple different species, with no readily available way to name the number of different species present or categorize reads into their respective species of origin. The method of sequencing is somewhat error prone, which has to be factored into any analysis.

k-mer - a substring of genomic sequencing data of length k. These k-mers overlap, such that for a given k-mer k_i, the subsequent k-mer, k_{i+1}, starts on the second letter of k_i. We'll call this a **k-mer overlap**.

(coloured) deBruijn graph - a directed graph created by breaking one or several genomes into its constituent k-mers. The nodes of the graph are said k-mers, with a directed edge going from k_i to k_j if k_i k-mer overlaps with k_j. For a single genome, the graph represents a compacted version of the entire genome. For multiple genomes, the graph can be coloured to distinguish from which genome each node originates. Such a coloured deBruijn graph can be used to identify variability between its different constituent genomes.

sketch - a collection of s integer values, here $s = 5000$. A sketch is produced by treating every k-mer of a genome with a specific hash function, resulting in a numeric hash value for each k-mer. The lowest s hash values resulting from this process become the sketch for said genome, functioning as a representative for the genome as a whole. This sketch is effectively a random subset of the k-mers comprising the genome and can be used as a quick substitute for comparison between genomes, the reliability of which is mathematically supported.

sketch distance - the number of hash values differing between two sketches. With sketches of size $s = 5000$, if two sketches share 4997 hash values, their sketch distance would be $5000 - 4997 = 3$.

UF cluster - a tentative structure in the clustering of sketches into cliques. These UF clusters are formed by calculating the sketch distance between every sketch in the corresponding data set. UF clusters are formed with respect to a given threshold, which at the moment is 5. If the sketch distance is less than or equal to this threshold value of 5, the sketches are merged into one cluster using a union-find approach. This approach can lead to a runaway process, where large UF clusters have a tendency to get larger, because the only criteria to merge into a cluster is having a sketch distance of 5 or less to any member within the cluster. The advantage to this approach is that every sketch is only a member of one cluster and within each cluster every sketch is at most at distance 5 from its closest sketch within the same cluster. Thus every cluster has a void surrounding it of at least width 6, making it a good first clustering step.

clique - the final step of the clustering process. Cliques are currently formed within each UF cluster with a bottoms up approach, which is described in further detail below.

A short 'Previously on...'

The following is the second interim report for my work this summer at Caltech, written as a continuation of my first interim report. This first pages serve as a quick overview of the work done so far. The field of research is Bio-informatics / Computational Biology. My project is a continuation and expansion of a software tool I started to develop last summer, when working for Professor Páll Melsted at the University of Iceland, from whom the concept originates. The working name of the method and software is MinCE and the latest version is available on https://github.com/mannaudur/MinCE-kallisto.

Mince

The past decades have seen significant advances in the field of genome-sequencing, both in terms of precision and cost-effectiveness. This has led to an exponential growth of catalogued genomes, particularly among bacteria. Most sequencers output an abundance of short reads, which form the whole genome when properly aligned. The efficacy of identifying these sequenced bacteria in real-life, metagenomic samples has been hindered by the sequencers' inability to discriminate between specific origins of reads in samples containing multiple species.

MinCE is a program for quickly identifying bacterial species and intraspecific strains in metagenomic samples, given that those species and strains are present in its MinCE-library. This library is created by first *sketching* each genome from a given dataset, which refers to treating every k-mer of length $k = 31$ within the genome with a hash function and keeping the lowest $s = 5000$ values in a list, known as a sketch. This is effectively a random subset of the genome and is used as a representative for the species and/or strain within the library. When a sequencing file is passed to MinCE, it will also be sketched in a similar manner and compared to each sketch within the library to find its closest match. For closely related species within the library, it is possible these sketches will prove identical, as the variation between them might not find its way into the lowest 5000 hash values. Thus multiple sketches might return a perfect match and for these cases, an extension of the sketches' resolution is required. By cataloguing the clusters of similar sketches within the MinCE-library, we can anticipate this situation and have the necessary data available. When each cluster is identified, a coloured deBruijn graph is created from the whole genomes which are represented by the sketches within the cluster. From this coloured deBruijn graph we can extract sequences which distinguish between the genomes within the cluster. Thus, when the comparison between sketches results in multiple best matches, these readily available sequences should be able to discern between those possibilities. MinCE currently runs on a dataset of 258,339 genomes of Eubacteria and Archaea.

Clustering the dataset

The main focus of my first interim report was on the problem of finding these clusters within the MinCE-library. The clusters are formed by looking at whether the sketch distance between members in the dataset goes below a certain threshold value, which we can call our *cluster-threshold*. The value of this threshold is best decided on through empirical testing, as it should approximate the general margin of reliability for sketches as representatives. Let's say we have some sequencing data from a metagenomic sample, which unbeknownst to us did contain *Piscirickettsia salmonis* strain AY3800B. For a cluster-threshold value $t = 5$ - the initial threshold value chosen this summer - we are saying that the compounded effects of sequencing errors, possible parts of the genome missing from sequencing data, and variability within the species, could result in our sketch of the sample having a distance 5 from the sketch of *Piscirickettsia salmonis* strain AY3800B in the MinCE-library. Therefore, by using this threshold as our pair of compasses, we should join *Piscirickettsia salmonis* strain AY3800B and every sketch surrounding at a distance $d \leq 5$ to a cluster, so we can discern between them by use of distinguishing sequences.

Whether $t = 5$ is a reasonable value is difficult to say without further context. A first indication that it might be too high, is the fact that the biggest cluster of perfectly identical sketches in our library contained 63 sketches. As a first pass, the sketches were clustered using a union-find approach, with a threshold value of $t = 5$. The union-find approach meant that these clusters might potentially grow quite large, as the only criterion for joining two clusters into on large cluster is that any two members of those clusters have a sketch distance of $d \leq 5$ between them. It is, therefore, a very rough first draft for a clustering. Upon examining these initial clusters, the idea of using a final clustering-threshold value of $t = 5$ was permanently discarded. One such large cluster, containing 486 sketches, turned out to have a mean inner sketch distance of 2.8, a median of 3 and a max sketch distance of 12. Though this cluster was obscenely large, it did not appear to be sparsely populated. Rather, it was an indication of the problem as a whole; for vectors of size $s = 5000$, effectively populated in $\mathbb{R}^n$, where $n = 4^{31}$, there are a lot of ways for them to have a distance of 1 between them, not to mention the possible combinations for larger distances. These distances *do* of course ultimately translate somewhat roughly to relatedness and the relatedness of species does not easily project onto a two-dimensional family tree, particularly in the bacterial world where horizontal gene transfers abound.

In the first interim report I detailed my effort to try to split up these larger UF clusters. I started out with annoy, a k-nearest neighbors algorithm, and found that required me to first map my sketch-distances onto some n-dimensional space. I attempted multi-dimensional scaling on the larger clusters but the strain output was unacceptable, owing to the compact nature of the clusters. In any case, I would need the clusters to be overlapping and did not find any documentation on annoy for such a feature. I then attempting a minimum-cut algorithm to identify weak points within the clusters and split them in an iterative process, where all sketches adjacent to the cut would join both sub-clusters. Again, these clusters proved too dense for this approach, as this would either quickly or immediately lead to one or both sub-clusters being the original one, once the cut-adjacent sketches had been added to both sub-clusters. This would not do either. I finally settled on my own implementation, which at least resulted in smaller sub-clusters than the original UF clusters. From this point, these sub-clusters are called cliques. The implementation was dubbed the 'bottoms-up approach' and starts by forming cliques from all sketches having distance 0 between them, then forms cliques out of the remaining sketches that have distance 1 between them and so on until it reaches a given threshold point, at which point it allocates the remaining sketches to their respective nearest clique(s).

The implementation of bottoms-up cliques is the only part of MinCE that is still written in Python, as I am still not convinced it is sufficiently rigorous.

Work since Interim report #1

On selecting distinguishing sequences

My first interim report ended on a cliffhanger, suggesting that my next step would be to implement an algorithm called Deterministic Column Subset Selection (DCSS) - developed and published[1] by Prof. Pachter and his students - to select distinguishing sequences from the genomes in cliques. This wound up not being entirely what was needed. To illustrate this, I will first explain in more detail the steps leading to this selection. First the genomes for every member of a clique are piped into a coloured deBruijn graph, which can be thought of as subway-map for the uninitiated. Each line in the graph represents a sequence of bases (A, C, G and T) found in the communal set of genomes comprising the deBruijn graph. The lines split and rejoin throughout the graph, where each split represents a variation between the genomes and the rejoining of said split indicates that the continuing sequences are again identical. These graphs are not formed from the continuous sequences of whole genomes, but rather by first breaking the genomes into their constituent k-mers and linking those k-mers if they k-mer overlap. Thus, it isn't really a graph with a clear beginning and end. For a collection of distantly related genomes, this graph would be a nonsensical tangle of isolated islands joining and splitting erratically with limited useful information.

[1] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6347249/

In our case, however, these graphs are formed from genomes which we know to have very nearly-identical or identical sketches, indicating they are quite similar. Thus, the resulting deBruijn graph will have long stretches of sequences which don't split and those that do split will be significant, as they describe the isolated segments of variation between these genomes. Using the program bifrost[2], these deBruijn graphs can be translated into two-dimensional matrices containing only zeros and ones, which I will refer to as *bit-matrices*. The bit-matrix columns represent the genomes in the clique and the rows every continuous segment in the deBruijn graph. Thus, for a deBruijn graph D, a cell $D[i, j] = 1$ would indicate that genome j contains sequence i, otherwise $D[i, j] = 0$. The format of the intial output graph would therefore suggest a *row selection* problem, rather than a columnar one. The file containing the sequences corresponding to each row is also easily generated through bifrost. I completed the code to translate cliques into these bit-matrices and corresponding sequence files just prior to uploading Interim report # 1.

Now the aim of the algorithm for selecting sequences should be a bit clearer: to select rows from the bit-matrix that can distinguish between the members of the clique. We will want some redundancy in the number of these sequences, as we have to account for variation between members of the same species and strain, so I chose the arbitrary *not-to-high not-to-low* value of 10, meaning that each member of the clique would ideally have 10 sequences chosen which they contain. A good sequence would be one that only appears in a given member, because that would distinguish that member from every other one. Such a sequence would have a corresponding row in the bit-matrix having only a single 1 and all other cells having value 0. From this point on, I will refer to the number of 1's in a row-vector as the row-vector's *degree*. A row-vector of degree 2 will appear in two genomes in the clique, distinguishing the pair from every other member within the genome but not distinguishing between the two members of the pair. A logical result is that row-vectors having low degree are preferable to high-degree vectors.

Attempted Deterministic Column Subset Selection

The algorithm for DCSS was only available through a Github[3] linked in the NCBI article. This Github account contained data and algorithms to run the entire experiment detailed in the original article, which required me to sift through the code to find something akin to the article's pseudo-code. I found what I surmised to be the required code snippets and pasted them into a Jupyter Notebook for experimentation. Although this code did result in a subset of columns, the set did not align with my needs as it contained multiple high-degree vectors. Another difficulty posed by the approach was defining the number of columns selected. As far as I could gather, due to its step involving singular value decomposition, the algorithm could not exclude any number of columns. A test case of a 6×795 bit-matrix - formed from a deBruijn containing six members and 795 continuous segments - could not be reduced further than to a 6×733 matrix.

As this was all based on code lifted from the Github which I only suspected I understood, I decided to write my own implementation based on the article's pseudocode and amend it to specify precisely how many columns to retain, and thus how many to discard. This required removing a part of the algorithm, which I will now explain in further detail. Given an original $n \times d$ matrix A, where n are genomes in the graph and d are segments in the deBruijn graph, the algorithm calculates a rank-k subspace leverage score for each of the i columns:

$$\tau_i(A_k) = a_i^T (A_k A_k^T)^+ a_i$$

Here a_i is the i^{th} $n \times 1$ column vector of A, A_k is the rank-k singular-value decomposition approximation to A and the subscript $^+$ refers to the Moore-Penrose pseudo-inverse of $(A_k A_k^T)$. This rank-k subspace leverage score is stored for each column in the original matrix and for each selected column, the score is added to a running sum. The columns are selected in order of largest rank-k subspace leverage score. The algorithm accepts the original matrix A and two parameters, a positive integer k (referring to the "rank-k") and an error term ϵ. Defining Θ as the set containing indices of selected columns, the first pass to select columns terminates when

[2] https://github.com/pmelsted/bifrost
[3] https://github.com/srmcc/dcss_single_cell

$$\sum_{i \in \Theta} \tau_i(A_k) > k - \epsilon.$$

If, after this criteria is fulfilled, the number of selected columns is still $|\Theta| \leq k$, the algorithm continues the process of selecting the column a_i having the largest $\tau_i(A_k)$ until $|\Theta| > k$. My modification was simply to skip the step involving the comparison of accrued rank-k subspace leverage scores against $k - \epsilon$ and go straight into the second step of awaiting the number of selected columns to exceed k. My understanding of the algorithm, admittedly lacking in the implications of using a Moore-Penrose inverse and only superficial with respect to singular-value decomposition, was that this removal didn't break the entire thing.

With this amended version of the algorithm ready to go, I tested it out with an iterative envelope. Starting with $k = d$, the row-sum of the resulting sub-matrix was evaluated for each iteration and if any row summed to less than 10, the value for k was increased by 1 and the algorithm run again. Sadly, this didn't amount to much. Running this amended DCSS algorithm on several samples of cliques (bit-matrices transposed to fit the columnar nature) still resulted in the algorithm discarding most or all of vectors having degree 1. It approached doing the opposite of what I wanted, which could be useful, but strayed far enough from it to remain impractical.

Greedy Sequence Selection

I therefore decided to return to my implementation from last summer. That code was written during a week long stay in the countryside and had never gotten any overhaul or critical review. It was entirely written in Python and I suspected it could be simplified quite a lot, with several redundant sections being the result of its origins as a perpetual stew[4] during my holiday. I rewrote the whole thing in C++, simplifying it along the way and optimizing what I could. The following section describes the functionality of the algorithm and closes by addressing its shortcomings. To attach some name to this algorithm, I'll call it the *Greedy Sequence Selection* algorithm or GSS for short. To recap, this algorithm receives a bit-matrix describing which members contain which sequences in the deBruijn graph, as well as a fasta-file of sequences, where the indices in the fasta file correspond to the indices of the rows in the bit-matrix. We receive the proper names of the clique's genomes through the bit-matrix but in the following text, we will simply refer to genomes as members or by the number of their corresponding column index.

The original bit-matrix output by the bifrost code is much larger than is needed. For a clique containing n members, the number of unique *vector-forms* possible is 2^n and it's very unlikely we'll see all of them. By *vector-form* I refer to the pattern exhibited by the vector. For a clique with three members, we have 8 vector-forms:

$$[0,0,0] \quad [0,0,1] \quad [0,1,0] \quad [0,1,1] \quad [1,0,0] \quad [1,0,1] \quad [1,1,0] \quad [1,1,1]$$

The first thing the GSS algorithm does is strip the original matrix down to its constituent vector-forms and log how many times each one appears. The results of this is a *reduced* bit-matrix only containing one copy of each vector-form and a list containing references to each copy's associated sequences. This is done by reading the .tsv file describing the original matrix and logging the results in a set. The reduced matrix is then formed from the vector-forms in the set and sorted row-wise by increasing degree, which will be important later. Along the way, GSS also logs two kinds of relations. The first maps the index of each row-vector to every sequence that was associated with that form. This simplifies our selection process. If we have five sequences appearing in members "1" and "3", we don't care which of them we choose for our final sequences. All we care about is choosing a sequence that has the corresponding vector-form $[1, 0, 1]$. Thus, we store the indices of all 5 sequences in a list, which is paired to the vector $[1, 0, 1]$ in our reduced matrix. The second kind of relation is pairing members of the clique to vector forms. If this supposed vector-form $[1, 0, 1]$ is situated in the 4th row of our reduced matrix,

[4] https://en.wikipedia.org/wiki/Perpetual_stew

both members 1 and 3 will have index 4 stored in their corresponding *relation-log*. This log is used as a quick way to find the vector-forms available to each member later on in the algorithm. Originally these relations were represented by way of identifying the vector-form by its corresponding binary number. Once the cliques turned out to exceed the size of 64, my 64-bit short-hand proved insufficient and a total reworking of the algorithm was needed. This section describes only the current, updated version of the algorithm and won't go into details about the many missteps along the way, nor their resulting debugging eras.

The second step of GSS is to select as many as 10 copies of unique sequences for each member in the clique. This is the same as iterating over our list of degree-1 vector-forms and picking as much as 10 associated sequences, if we have enough. This is easy, as our reduced bit-matrix is sorted in order of increasing degree. If each member within the clique has a corresponding vector of degree-1 with 10 or more associated sequences, we can exit and the algorithm has succeeded. This seems to be the case quite often, as many of these cliques are formed with a likely more-than-generous threshold-value, so finding unique sequences is often trivial. In the cases where we haven't reached our goal, we step into the third step of the algorithm.

This step requires an explanation of the connection-matrix, C. This is simply an $n \times n$ matrix which logs how often members have been *paired* with each other. For the earlier example of choosing one copy of $[1, 0, 1]$, we would log member 1 as having been paired with member 3 once and likewise log member 3 as having been paired with member 1 once. This corresponds to adding the chosen vector, $[1, 0, 1]$, to the rows of the connection matrix in which the chosen vector takes value 1 - i.e. rows 1 and 3. The selected degree-1 vectors in the second step are the first ones to be logged into this connection matrix.

The third step is iterative and is as follows. GSS first identifies the member being furthest away from reaching their goal. In most instances, this will be the member with fewest associated sequences, but sometimes the number of total available sequences will be lower than 10. The goal for each member thus is computed as the minimum of 10 and the number of sequences available. Looking at our $n \times n$ connection-matrix, the value in cell $C[i, i]$ corresponds to how many sequences are selected that appear in member i. The largest difference between goal and number of sequences already selected determines the *focus-member* of this iteration. Let's say it's member 3. To find the best sequence to raise member 3's score, we fetch the indices of row-vectors related to member 3. We vector-multiply each related row-vector by row $C[i, :]$ and pick the one whose product yields the lowest score. This way we choose the vector who pairs member 3 with the members it's been paired least with in the past. If several vectors are tied for the lowest product, we will choose the lowest degree of vector available. This results from our reduced matrix being sorted by increasing row-vector degree, so the first vector to reach lowest product value will remain the chosen vector. We choose the vector, log it appropriately into our connections-matrix C and return to the first step of this function.

The iterative third step concludes when each member has reached their goal. The reason the second step involving degree-1 vectors is not merged into this step, is because it could result in some unique sequences being left out, believe it or not. This greedy implementation is "good enough" for most instances, but it has one big flaw. The focus-member is selected by looking at the member furthest away from their goal and the next row-vector is chosen based solely on the impact to that focus-member. As a result, members may reach 10 associated sequences without having been considered through the connections-matrix. This may result in the chosen sequences never distinguishing between the two. I patched this with a band-aid which iterates over the final connections-matrix and examines whether the value of $C[i, i]$ appears twice in the vector $C[i, :]$. If $C[i, j] = C[i, i]$, GSS finds the symmetric difference of member i and j's *relation-log* and finds at most two vector forms distinguishing them. It then adds up to three sequences having those vector forms to the chosen sequences. This correction is not without its faults, as it may create the same situation elsewhere and also does not examine instances where the difference between $C[i, j]$ and $C[i, i]$ is for instance 1, which is also a problematic situation. For the time being, it delivers results which are viable for testing, so it will be called good enough. It will later be improved, with the current version serving as a benchmark.

Finally, the chosen sequences are logged to a file detailing their connections to genomes within the clique and the final form of the connection matrix is logged to another file as a quick reference for the inter-relations of the clique.

Sifting input files for sequences

The GSS algorithm creates 2.4 sequence-files per minute on average. Doing so for the roughly 11,000 cliques thus takes around three days. While the algorithm was left to crunch the library's fasta files on a server in Iceland, I started work on the algorithm to search for these sequences in an input file from a user. The idea was to fetch the sequences for clique j if the sketched input file had a sketch distance less than some threshold t to any sketch in said clique. For metagenomic samples, we would expect to have multiple good matches in distant sketches. Therefore, this algorithm should be modular, so as to accept links to the sequences of every clique within distance t of our input's sketch and search all of them simultaneously. As a first draft, I quickly implemented this using - again - the deBruijn graphs from bifrost, in a similar fashion as the bit-matrix algorithm. By creating a deBruijn graph from our input file, multiple sequence files could be passed as *query files*, asking bifrost to search the deBruijn graph for all of them with one command. The result would be a single .tsv file containing one column vector of zeros and ones, in this case describing which sequences were found within the input file.

This was probably not feasible for actual samples, as creating a deBruijn graph is very time consuming for large files. It was, however, a good way to test the functionality of the algorithm on smaller files and took no real time to whip up. I integrated this search algorithm into the main function of MinCE, combining the results of both sequence search and sketch comparison into one function to print the results in a pleasing manner to the console. I expected to run some tests on this deBruijn implementation as a proof-of-concept and as a quick way to scan for bugs in the process. Before I started that work, however, I had a meeting with Prof. Pachter.

MinCE's triple bypass surgery

The meeting was incisive in the sense that Prof. Pachter took little time to get oriented with the current state of the project, but also in the sense that said state left the meeting with much of the extraneous fat surgically removed. Prof. Pachter suggested a simplification of the whole process, which - while rendering much of my recent work obsolete - was sure to yield considerably faster run-times for large input files.

Instead of keeping sequence files for every single clique, only to be loaded in the case of favourable results from the sketch comparison step, the entire set of sequences could be treated with the same hash function as the sketches and stored in a file similar to the `hash_locator` file described in the last interim report. That file is simply a text file mapping every hash in the database to the sketches it appears in. When a new sketch is input from a user, the `hash_locator` file is loaded and the hashes from the input file are mapped to the corresponding sketches. Hence, the new sketch can be compared to every other sketch in the database through a single file and the sketches themselves are redundant for the run-time application of MinCE. This can of course also be applied to these distinguishing sequences but they would need a separate file, so as not to be mistaken for ordinary sketch values. We can search this file in the same step we feed the input file's hashes to our `hash_locator`, eliminating the need to search the input file twice. As we don't know which sketches will compare favourably to our input file, we simply feed all hashes from our input file into this sequence-`hash_locator` and log corresponding matches if they're contained in the file.

As we now have two files of this nature, the old `hash_locator` file was renamed `sketch.hashmap` and the new sequence variant was created under the name `seq.hashmap`. Creating this file required reading all of the sequence files created by GSS and hashing them to `uint64_t` values. These values aren't mapped directly to genome names, but rather through a short-hand of MinCE-IDs stored in a file called `MinCE_to_NCBI.index`. These IDs were assigned at the very start of the MinCE library's creation and a reverse-mapping was not readily available, as it was not deemed necessary at the time. Hence, some backtracking was required.

I spent a few days pondering the ramifications of this new approach. Firstly, whether the architecture could be arranged so as to bypass the backtracking stage involved with creating `seq.hashmap` and streamline the whole process. The general architecture of those `.hashmap` files and the accompanying `.index` file seemed simplistic and just begging to be optimized. Therefore, I concluded that any small optimization in the building of a MinCE

library at this stage would likely be immaterial to the finished product. Once the prototype is up and running and testing has led to a refining and solidification of its individual features, a general overhaul of the underlying architecture is warranted. My second subject of ponder was whether this changed the role of cliques at all. I'll come back to that subject in the final section of the report, detailing next steps and possible solutions to the current shortcomings of MinCE.

The current state of MinCE

MinCE is currently up and running. The main function now searches for every distinguishing sequence along with the sketch hash values and thus the only files needed to run it, aside from the executable itself, are `sketch.hashmap` (13Gb), `seq.hashmap` (5.6Mb) and `MinCE_to_NCBI.index` (9.7Mb). The load time required before *mincing* can begin is mainly tied up in loading `sketch.hashmap` and ranges from around 6 minutes on my 16Gb RAM M1 MacBook Air to 9 minutes on Mimir, a 32Gb RAM server located at the University of Iceland running Linux CentOS 6.3, last updated in 2012. For test runs mincing normal fasta files previously used to create the MinCE-library, the subsequent processing time was around 1 second long on both machines. Currently, however, the RAM demand for larger fastq files (test cases of 9.2Gb) seems to overwhelm my laptop and bog the runtime down to unacceptable levels. The Mimir server meanwhile takes around 15 minutes to finish one such run. This clearly warrants further analysis of the algorithmic complexity of the steps involved in mincing a file. I will return to this subject in the last section of this report.

Testing on simulated metagenomic datasets

Having verified the functionality of MinCE, using fasta files known to exist within the library as inputs, the next step was to run metagenomic sequencing data on the software. Verifying such results is difficult, as the object of MinCE is to solve the difficulty of identifying genomes in metagenomic samples - a classic catch-22. However, Prof. Pachter supplied a *simulated* metagenomic dataset, generated from a predetermined set of 10, 100 and 400 genomes. The datasets and lists detailing their supposed contents were available online, so I downloaded datasets for each size of group, simulated as results from Illumina sequencers.[5]

10 genome simulated fastq file

Figure 1 shows the top results from mincing two fastq files, each 4.6Gb, generated from 10 different genomes. Of the top 11 results, 10 were present in the sample. Result 7, red as it should not appear in the sample, refers to *Methanococcus maripaludis* strain C8. Result 5 refers to *Methanococcus maripaludis* strain C7, green as it should appear in the sample. Neither genome has any distinguishing sequences in the MinCE-library, due to their set of k-mers being identical and thus no way to distinguish between them through deBruijn graphs.

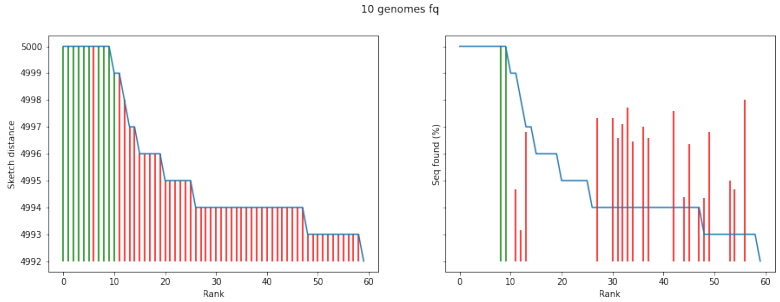

Figure 1: Green bars represent genomes present in sample, red results should not be present. **Left:** Blue line and bar height indicate number of shared sketch values. **Right:** Bar height indicates ratio of found distinguishing sequences, with range 0% at bottom to 100% at top, overlayed on blue line. 0% could also mean no sequences available in MinCE-library.

[5] https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0031386

The 11th result should be present in the graph but only had 4999 of its MinCE representative's 5000 sketch values. The fastq files were minced with a threshold value of $c = 2$, indicating that each k-mer would need to be observed twice to be eligible for consideration. This is done to counter the error rate of sequencing machines, as random errors in single nucleotides might generate new k-mers not representative of genomes in the sample. As the number of possible different 31-mers is 4^{31}, seeing the same erroneous k-mer twice was considered sufficiently improbable. The threshold value is accepted as a parameter from the user.

These results granted some optimism, as both the sketch values and the corresponding distinguishing sequences clearly delimit the true positives against the rest, aside from one false positive from a different strain of a species present in the sample.

100 genome simulated fastq file

Inspired with tentative confidence, I moved onto the next largest set of genomes, this one containing 100 genomes. Put mildly, these results were underwhelming. Figure 2 shows the results of mincing with the threshold value set to $c = 2$. On the right side, we can see that the majority of genomes which should be identified within the sequenced data come in around places 13,000-22,000. Ahead of them we have more than 10,000 genomes supposedly not found within the sample. Additionally, the right graph shows we have multiple erroneous matches finding 100% of their distinguishing sequences. This representation does not reveal the number of those sequences, whether 10 as is preferred or only 1 or 2 in cases of cliques formed from near-identical genomes. In any case, this is case for concern.

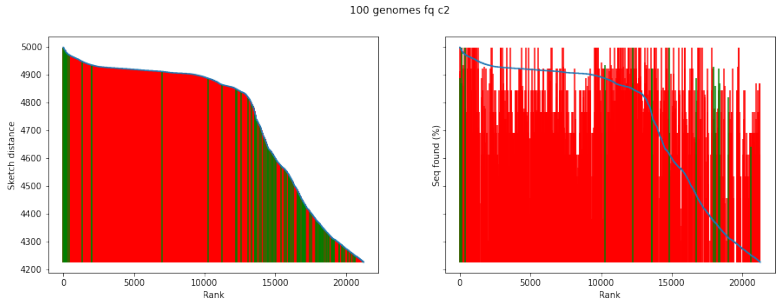

Figure 2: Results from mincing a simulated data from 100 genomes with threshold value $c = 2$.

The graphs in Figure 2 don't even tell the whole story, as one match is omitted for the sake of clarity. Figures 3 and 4 show information about the worst match of all, the genome of a megaplasmid found in the genome *Ralstonia eutropha H16* - a heterotypic synonym of *Cupriavidus necator H16*, which is the name used in the MinCE-library. This result was the 85,858th highest ranked result in the total list of matches, with only 279 hashes shared with its MinCE-library representative.

Now, this could stem from our choice of threshold, as we might have missed crucial k-mers found in this genome simply because their coverage was only 1 - i.e. the other 4721 sketch values associated with *Cupriavidus necator H16* only appeared once and thus were not considered reliable. This is of course very improbable, but I ran MinCE again, this time with $c = 1$. *Ralstonia eutropha H16* still ranked 85,858 out of all matches, but this time with 316/5000 sketch value matches. To boot, none of the distinguishing sequences were found this time either. The other genomes measured slightly better, with the majority of green results clustered between rank 10,000 and 14,000 having between 4800 and 4900 sketch values in common with the 5000 of the representative sketch.

There are several ways to explain these catastrophic results. Firstly, the genome in place 85,858 is that of a megaplasmid found in *Cupriavidus necator H16*, not its entire genome. NCBI houses sequence data on two additional chromosomes present in *C. necator H16*, which together with the plasmid comprise the fasta file for *C. necator H16* used to build the sketch representative in MinCE's library. The megaplasmid only contributes around 6% of the total genome, which approximates the ratio of both 279/5000 and 316/5000.

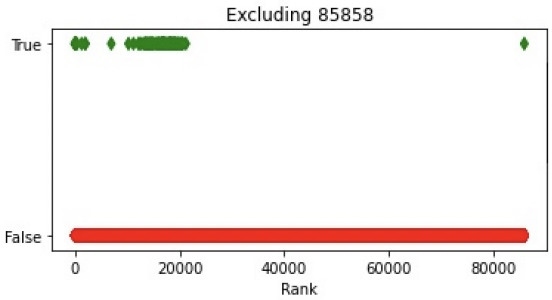

Figure 3: The distribution of genomes present in the simulated sequence-data (green) over the x-axis' ranking of all results. About 60,000 genomes not present in the sample (red) had a higher score than the last green results.

	NC_005241 Ralstonia eutropha H16
NCBI_id	GCF_000009285.1_ASM928v2
Sketch	279
Seq_found	0
Seq_total	10
In sample	True
Rank	85858
Ratio	0.0

Figure 4: The 85,858th result, a megaplasmid found in *Cupriavidus necator H16*, designated by its heterotypic synonym *Ralstonia eutropha H16* in the info file accompanying the simulated fastq data. Only 279/5000 sketch values were found and no distinguishing sequences were found.

This applies to some other members within the simulated dataset, as they are constructed from parts of the genome and not the whole. This does not suffice to explain the extremely low matches for all members of the dataset. Most red matches preceding the actual genomes present in the data have sketch values around 4900, which ordinarily would them exclude from further examination. Thus, the problem isn't false positives, but false negatives. This could possibly be explained by the fact that this data was generated in 2012, based on data available at that time. The number of sequenced genomes has increased drastically since then and so has the depth of each genome's sequencing. The data used to construct MinCE's library is based on the latest sequencing data available for each genome as of July 2021 and so might include large sections of genomic information not available to the researchers in 2012. If this were a real metagenomic sample from 2012, MinCE might have no trouble identifying these genomes as their *actual* genome has not changed in any real way since 2012.

400 genome simulated fastq file

Continuing the trend, the results from the simulated dataset of 400 genomes were abysmal. MinCE found indications of 399/400 genomes present, only missing *Candidatus Carsonella ruddii PV* which is not a member of MinCE's library. As the treshold $c = 2$ had been suspected of muddying the waters when mincing the 100 genome dataset, this was run with $c = 1$. Figure 5 shows these results, with the highest sketch match being 4837/5000 and most green matches situated between 3000 and 3500. It is important to note that each simulated dataset, regardless of size - 10 genomes, 100 genomes or 400 genomes - is stored in two 4.6Gb fastq files. Thus the 10 genome sample should have an estimated 10x coverage compared to the 100 genome sample and 40x relative coverage to the 400 genome one. Therefore, diminishing results are to be expeected. With so many unknowns possibly contributing to the lacking nature of these results, some other measures of testing need to be taken.

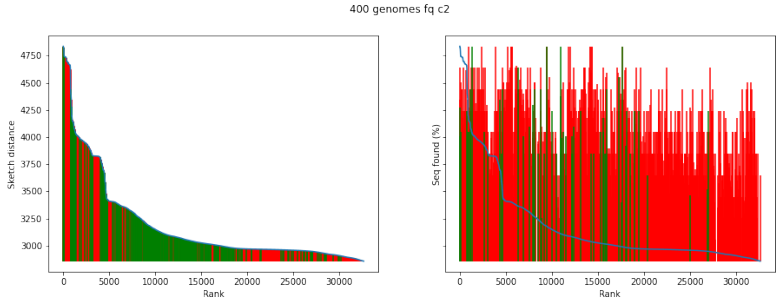

Figure 5: Results from mincing a simulated data from 400 genomes with threshold value $c = 1$. Most genomes supposedly present in the sample have between 3000 and 3500 matching sketch values out of 5000.

Next steps

1. Further testing

I'd like to use the time I have left to further test out the accuracy of MinCE. The results from the simulated metagenomic datasets can still be useful in this endeavour. Although we should expect them to be erroneous, the results from MinCE give some idea of what genomes are supposedly in the samples. By loading the corresponding index of each genome, I can estimate abundances for each putative genome in kallisto[6]. A first pass could load the index of the genomes reported to be in the sample. If kallisto agrees with MinCE that these are merely subsets of the entire genome, that will at least reassure me that the code is working as intended. Whether this sort of sparse sequencing data is representative of real-life samples, I cannot say. A second pass could load the index for false positives with high sketch matches, the results one would expect to take as legitimate if this were any other run of MinCE. There is considerable overlap between genomes in the bacterial kingdom, so there is the possibility that some of these high-ranking false positive genomes could indeed have an appreciable portion of their actual genomes within the sample. This could inform us about the practical threshold of MinCE results, both with respect to sketch and sequence matches.

Prof. Pachter also suggested I take a look at Kraken[7], a program for assigning taxonomic labels to metagenomic DNA sequences, to verify or counter the results of MinCE. Another possibility is simulating reads using specialized programs such as wgsim[8] to simulate new metagenomic datasets based on current data. I will admit, I am ignorant of how complicated that might be.

2. Re-evaluate aspects of MinCE

The results from these tests will hopefully be informative about the specific shortcomings of MinCE in its current state. Already, I am troubled by how many 100% sequence matches are seen in results with low sketch matches from the simulated datasets. This comes back to the question, whether the role of cliques should be changed. It is possible that the desired number of sequences should be increased and so should the number of genomes having associated distinguishing sequences. The formation of cliques is not done with much rigour in the current version of MinCE, as detailed in my last interim report (see section "Bottoms up approach"). If the sketch matches of genomes present in the samples are anticipataed to be far from 99.9% of 5000 - as indicated by preliminary results from simulated datasets - it is possible that these distinguishing sequences might need to play a larger role than previously thought. However, given how many erroneous results had 100% sequence matches, it is fair to ask whether this approach is even feasible.

There are a few ways to enhance the resolution of MinCE. The first is to increase the size of sketches to $s = 7000$ or even $s = 10,000$. This would of course correspond to an increase in the size of `sketch.hashmap` file, which is not favourable. It also feels antithetical to the underlying philosophy of MinCE, if such a thing exists, where sketches are intended as a *quick* comparison tool. They are assumed to be rough estimates and imprecise and therefore we have another part of the architecture to carry that role of extended precision, namely the distinguishing sequences.

Another possibility is to increase the k-mer size to the next logical value, that of $k = 63$. This would also incur a dramatic increase in the size of `sketch.hashmap` and probably increase the runtime of each mince. However, this increase would also improve the precision of sketches considerably, presumably much more than a doubling of sketch size. Both of these possibilities would likely increase the number of cliques while decreasing the average size of each one.

[6] https://github.com/pachterlab/metakallisto
[7] https://genomebiology.biomedcentral.com/articles/10.1186/gb-2014-15-3-r46
[8] https://github.com/lh3/wgsim

3. Optimizations of current code

The main function of MinCE is quite simple in broad strokes: it accepts the input file, loads the necessary data required to mince it and then iterates through the entire file, checking whether hash values match those in our library. The architecture used to be more convoluted but has changed quite a lot the last few weeks, particularly with the simplification suggested by Prof. Pachter. Although this approach reduces the number of search-passes over the input genome file to just one, it does present some additional problems. The maximum value in our `sketch.hashmap` file is `max_hash = 6725168124769237367`, a large number to be sure but only around a third of the maximum hash value possible for k-mers of size 31, 2^{64}. When iterating over the input file, we can compare the current hash value to this `max_hash` value and if it's above it, we know it isn't present in our `sketch.hashmap` file. This means that, on average, we only have to search for every third hash value in our `sketch.hashmap` file, speeding up the processing time. The original implementation, where hash values were only fed to `sketch.hashmap`, was faster due to this reason. For the new `seq.hashmap` file we have no specific range to compare to. In fact, some identifying sequences hash to values *lower* than the `max_hash` value in `sketch.hashmap`. Therefore we need to pass every single hash from the input file to `seq.hashmap`. The algorithm ensures we never look the same value up twice, but this still grinds the process to a halt in its current form. The span of the values in `seq.hashmap` ranges from $\sim 0.0004\%$ of 2^{64} to $\sim 99.9995\%$ of 2^{64}. Therefore, checking whether any given hash is within the scope of our `seq.hashmap` file is useless.

The current implementation uses a hash table from klib called khash.[9] This was chosen last summer when work on MinCE began and has not been reviewed since. Both `.hashmap` files are stored as plain text and then read into a khash hash table, mapping hash values to the MinCE index of associated genomes. When iterating over a new file, each hash value from the input is fed into these hash tables as a key. If the key is present, the value will be a vector of the indices of associated genomes. These indices are logged as keys in their own khash hash table, where the value is a vector of two integers. The first one tallies how many sketch matches have been found, while the second one keeps a record of sequence matches. I suspect this whole thing could be optimized quite a bit, though I have not formulated any concrete ideas. The final version of MinCE has often been suggested as a server which runs either locally on computers or as a single, remote one. Whatever it ends up being, some clever memory optimizations are needed along the way given how much memory is currently required for a single mince of large fastq files.

4. Future add-ons

The current MinCE-library is based on release 202 from the Genome Taxonomy Database.[10]. In April this year, GTDB updated their database with release 207. To keep the MinCE up to date - as well as to formally design the framework around creating a new MinCE-library from scratch - I would like to create a new library based on this release. This would hopefully be generated on a local Caltech server, allowing members of Pachter Lab easier access to the continuing development of the software tool. There is also one large group of arguable organism omitted from MinCE's library as of yet, that of viruses. I am not familiar with any comprehensive databases of virus genomes, but I am certain they exist. It would be refreshing to add this set of genomes to MinCE before the summer is over, perhaps as a stand-alone package so as not to overwhelm the average runtime.

I'll conclude this report here. There's a lot of new areas open for research at the moment and I assume the following week will shape the nature of my remaining days here at Caltech. As of the time of writing, I am not entirely sure what they will entail.

Thorhallur Audur Helgason

Pasadena, California, USA

July 27th 2022

[9] https://github.com/attractivechaos/klib

[10] https://gtdb.ecogenomic.org/