MetaDBGWAS: bringing bacterial GWAS to the metagenomic scale!

Louis-Maël Guéguen

June 3, 2022

Contents

List of Figures

1 Abbreviations

- DBG: De Bruijn Graph
- GWAS: Genome Wide Association Studies
- SPSS: Spectrum Preserving String Sets
- LMM: Linear Mixed Model

2 Tools

- Lighter `https://github.com/mourisl/Lighter.git` commit fecf711
- bcalm2 `https://github.com/GATB/bcalm.git` commit 40469f3
- REINDEER `https://github.com/kamimrcht/REINDEER.git` commit b6c205a
- DBGWAS `https://gitlab.com/leoisl/dbgwas.git` commit 4ab9784f
- gatb-core `https://github.com/GATB/gatb-core.git`
- boost `https://www.boost.org/` latest
- R `https://www.r-project.org/` latest
- Gemma `https://cran.r-project.org/package=gemma2` commit a78dc77
- Bugwas `https://github.com/sgearle/bugwas.git` commit 0692b15
- CMake `https://cmake.org/` latest
- GCC `https://www.gnu.org/software/gcc/` latest
- phantomjs `https://phantomjs.org/` latest

3 Introduction

Low-cost sequencing allowed biologists to quickly accumulate genomic data to conduct large scale analysis. Among such, genome wide association studies (GWAS) emerged. GWAS is a way to use the new sequencing power to link genetic variation, like single nucleotide polymorphism (SNP), to complex phenotypes. GWAS method consists in encoding the phenotype of interest into ones and zeros to reflect its presence or absence respectively; it then applies a regression model of the SNP absence/presence pattern against phenotypes and outputs p-values (H_0: the SNP has no effect on the phenotype) for each SNP. Further analysis is then conducted on the most significant variants. Because it tests variants both in and out of gene sequences, GWAS lead to a broaden comprehension of the impact of single nucleotide polymorphism on the genome. Human GWAS led to the discovery of numerous variants than helped understand widespread diseases: cancers, obesity, neuro-degenerative diseases. If GWAS also has its limitations [1], its wide use helped improve it over the years: it is now extended to the transcriptome as transcriptome wide association studies (TWAS) and despite issues such as the clonal nature of the populations, GWAS methodology has been expanded to bacterial genomes as well.

Applied to bacteria, GWAS gives us the opportunity to discover why specific strains are resistant to certain antibiotics; or which variant allow other strains to metabolise certain molecules. DBGWAS [2] is a tool that makes these application of GWAS a reality using k-mer-based methods. k-mers are sequences of length k formed with an alphabet (here, and for the whole report A,C,G,T); they are obtained from a sequencing read by listing all consecutive sequences of length k existing in the read. Consequently, k-mer-based methods are methods that manipulate them and exploit their properties. Such techniques help avoid alignment of the reads, which is necessary to identify SNPs presence or absence within bacterial genomes but is also time consuming and memory-hungry. Moreover, alignments require a reference genome, which in the case of bacteria, is arguably not in phase with the reality of high variation found in prokaryotic genomes. Indeed, bacteria often incorporate horizontally transferred DNA, and have plasmids, which are mobile DNA elements too, yet bigger than simple genes. These elements cause challenge when biologists need to define a reference genome, because they exists in large numbers and all can be either present or absent, resulting in a great amount of combinations. In bacterial metagenomics, the scale of the problem is amplified by the presence of several species. Furthermore, both plasmids and horizontally transferred genes represent differences that are much larger than SNPs (which are by definition 1 base variation) and thus are not handled by GWAS tools. Ignoring such large parts of the bacterial genomes would be highly detrimental for a GWAS analysis. Finally, a lot of genomes in databases contain sequences from other organisms, despite sometimes being from a completely different kingdom [3]. And this contamination problem is even more crucial for metagenomics [4], because the sequencing is done in bulk with multiplexing: all the DNA is pooled in thousands of cells, with barcodes to distinguish them. Having different species pooled together might lead to contamination. Hence, k-mer-based methods grant us the possibility to overcome all these difficulties. DBGWAS showed successful at finding known SNPs or horizontally transferred genes that cause resistance to antibiotics in *P.aeruginosa*; it proved even more useful by detecting previously unknown variants linked to resistant phenotypes, paving the way to a better understanding of drug resistance mechanisms. DBGWAS is based on de Bruijn graphs, a structure build from overlaps of length k-1 of k-mers, and unitigs that result from the de Bruijn graphs compaction (an operation that simplifies non-ambiguous regions of the graph and yields more compressed yet similar information). What is tested by DBGWAS is the association between presence/absence patterns of unitigs (therefore their k-mers) and the phenotypes. Testing unitigs allow to capture both small (SNP) and large (genes, plasmids) variants.

There is now demand for bacterial GWAS at a meta-genomic scale: it would allow to conduct research in a broader range of contexts, like finding the variant that might explain why a chemical component is degraded in certain environments and not others. But bacterial meta-genomics (actually, meta-genomics in general) meets one particular difficulty: the datasets are much larger. They can be larger in number of distinct k-mers (because they are composed of different species, with different genome compositions) and potentially more reads. Thus, it requires a lot more memory and running time, and the growth in needed resources might not be linear depending on the tools and methods used. Recent progress in k-mer-based methods tackle this problem, using ingenious techniques such as sets of solid k-mers, minimizers and monotigs. Their implementation yielded large gains in processing time and memory, going from 800 GB of memory and three months to assemble the genome of *P.taeda* (22GB) [5], down to just 40GB and 2 days with Bcalm2 [6]. To furthermore bring down the quantity of data for the whole process, filtering k-mers is necessary. The filter is based on their number of occurrences, and is crucial: filtering too much might remove variants, but a weak filter will let noise pass, resulting at least in a longer running time. Very rare k-mers might be the result of a sequencing error not corrected, which we want to remove; or truly rare k-mers. There is a limit to the detection power of a statistical link between a sequence and a phenotype, thus k-mers with very low occurrence can safely be removed. But the number of occurrences of a k-mer is dependant on the quantity of data, the sequencing cover, and potentially on sequencing bias. Plus, the definition "frequent enough" is not straightforward. This subject has not been discussed yet, and needs methodological development. It is then then the perfect time to take advantage of all these progress and contribute to the fields of bioinformatics and metagenomics by creating a new program that handles the metagenomic scale for bacterial GWAS.

4 Tools and Methods

The pipeline `Metadbgwas` uses several tools, chosen for their capacity to handle large datasets. The following subsections explain their respective roles and the methods they are based on. To make reading more comfortable, figure 1 shows the summary of the pipeline : step 1 corrects the reads, step 2 finds unitigs, step 3 builds the presence / absence matrix of the unitigs in the samples, uses a linear model to get relevant statistics and the visualisation is done. The words sample and files are used interchangeably, and the dataset designates all the files/samples. "Steps" designates the execution of a tool of the pipeline as well as specific sub-parts of a tool execution.

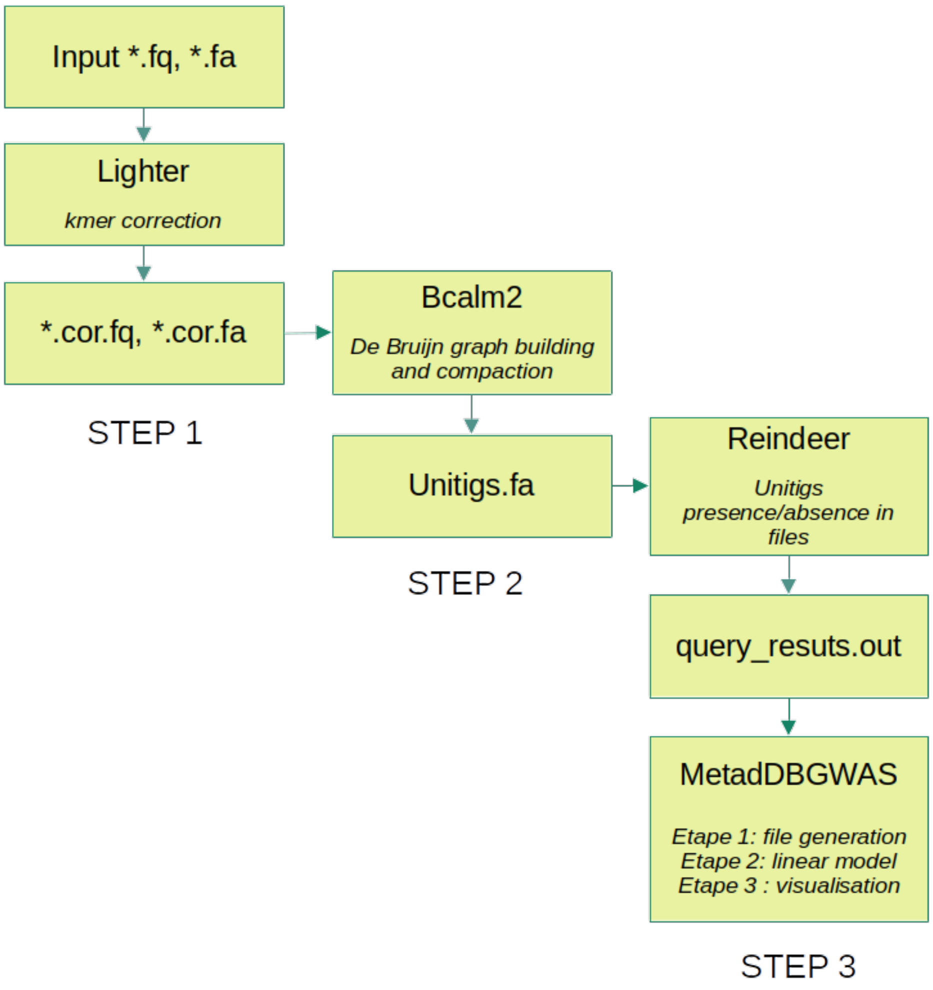


Figure 1: Overview of the `Metadbgwas` pipeline.

4.1 Lighter, solid k-mers and Bloom filters.

Lighter [7] is the first tool of the pipeline. Its goal is to correct the sequencing reads given by the user. As `Metadbgwas` relies on k-mer presence, absence and abundance in the sequences, having too much k-mers that are wrongly quantified would make the analysis results not trust-worthy. Moreover, it allows the user to feed almost-raw reads (they still should be trimmed) to the pipeline, minimising the amount of upstream work required, making it more friendly to beginners and more self-sufficient.

To correct the reads, Lighter builds from them a collection of "solid k-mers": sequences of nucleotide passing a threshold test to know if they can be considered as true (not sequencing errors). Roughly, the frequency of their appearance will determines if k-mers are "solid". The principle of the tool is to randomly pick reads, from which k-mers are extracted. Solid k-mers are found, stored, and then used to determine how a non-solid k-mer found in reads should be corrected. Randomly sampling reads allows Lighter to avoid counting all the k-mers present in the dataset, which is a memory consuming and can take a while on big datasets. Also, because incorrect k-mers (which contain sequencing errors) are less frequent than correct k-mers, as the probability to pick a read α decreases, they are less likely to be be selected. These reads and the solid k-mers resulting are stored in a memory-efficient manner using Bloom filters [8], an important asset when dealing with a big dataset. Moreover, the user can let the tool decide the probability with which reads are picked, assuring constant memory usage independently of the coverage. Figure 2 and the next paragraph explain how do Bloom filters work and why they are so efficient:

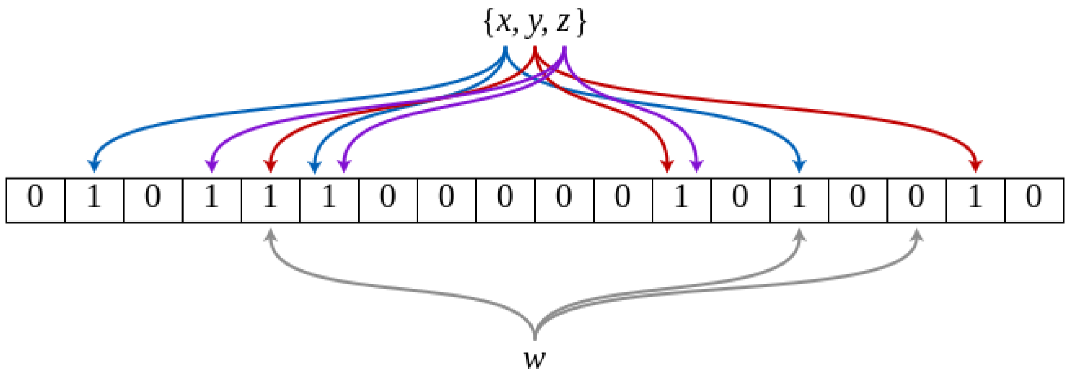


Figure 2: By David Eppstein - self-made, originally for a talk at WADS 2007, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=2609777`. A Bloom filter is here illustrated by the array of bit. Each arrow's colour represent a hash function, and each arrow represents the application of the hash functions to the elements. Elements x, y, z, are part of the set: all arrows (as functions) point to a 1. The application of the hash functions to w point to at least one 0: w is not in the set. Also notice how hashed elements can collapse to similar cells: this phenomenon creates false positives.

Bloom filters are probabilistic data structures used to store sets, that have the advantage of taking low memory space and being adaptable through several parameters. They are constituted by an array of bits all originally set to zeros. Each element added to the set is mapped to a position in the array, given by series of independent hash functions calculated to set the bit to 1 at that position. To check whether or not an element is in the set, the hash functions are applied and lead to bits set to 1 if the element is part of the array, else to a set of 0. An element that is not part of the set might still be mapped only to ones in the array; the request of an element to check if it part of the Bloom filter can yield false positives. There exist a documented trade-off between the size of the filter and the number of hash functions to balance false positives and computing time/space requirements. Overall, due to its small size in memory and adaptability (number of hash functions, size of the array), the Bloom filter is a handy tool to use memory with parsimony.

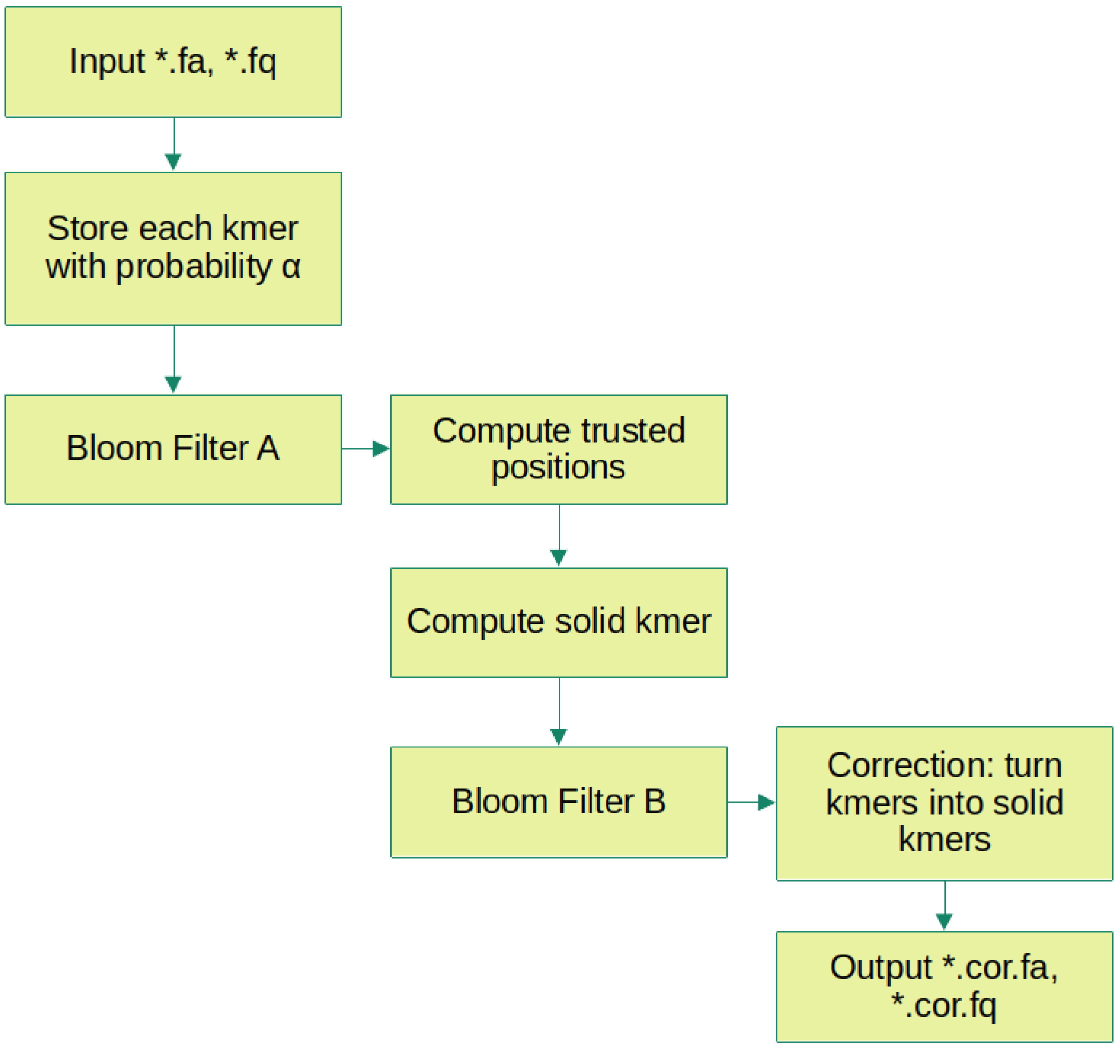

Input *.fa, *.fq
Store each kmer with probability α
Bloom Filter A
Compute trusted positions
Compute solid kmer
Bloom Filter B
Correction: turn kmers into solid kmers
Output *.cor.fa, *.cor.fq

Figure 3: Lighter workflow, simplified.

Figure 3 shows the workflow of Lighter. In step 1, the tool reads the input sample files and randomly selects reads with a probability α. They are stored in Bloom filter A. At step 2, threshold is then defined using α. For a read of length r, a position i for $0 \leq i \leq r$, if a position has more k-mers from Bloom filter A overlapping it than the threshold, then the position is marked as *trusted*. It is otherwise *untrusted*. k consecutive trusted positions give rise to a solid k-mer. Solid k-mers are stored in Bloom filter B. In step 3, solid k-mers are then used to correct untrusted positions : if a k-mer in a read is not in the Bloom filter B, it must be corrected. The correction is done by finding which substitution yields the longest stretch of solid k-mers. The corrected reads are written in files containing the "cor" tag. Figure 4 shows an example of correction.

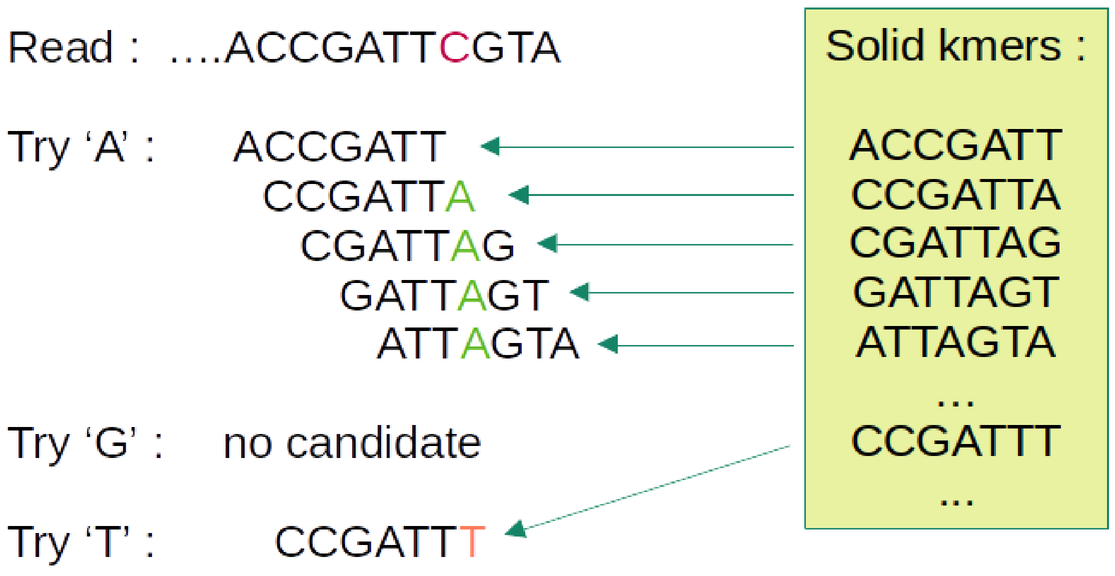


Figure 4: Correction procedure with Lighter. Nucleotide C coloured in burgundy in the read must be corrected. The longest stretch of solid k-mers (in the green box) is found for nucleotide A (light green), so we select A to replace C.

Depending on the length of the k-mers, the tools shows different trade-off between precision and recall. `Metadbgwas` keeps the value recommended by the authors.

4.2 Bcalm2 and the de Bruijn graphs

Before addressing the subject of Bcalm2 [6] itself, two notions must be explained: the de Bruin graph (DBG) and the unitigs.

We give a node-centric definition of the DBG: for a set of k-mers of length k, the DBG nodes represent all distinct k-mers of the set; and edges exist between a pair of nodes if there is an overlap of length k - 1 between the end of a k-mer of the pair and on the beginning of the other k-mer. A fork occurs when 2 k-mers map on the same end of a third one, and if the fork is resolved by two k-mers overlapping the beginning of a same third one, a bubble is created. Figure 5 illustrates a DBG with k = 4, containing a bubble. Once the DBG is built, it can be compacted: all k-mers in non-ambiguous paths (an ordered set of successive nodes connected by edges with no forks nor bubbles) are glued together to form one sequence : a maximum unitig. More precisely, since k-mers overlap of $k - 1$ bases in a path, only non-redundant nucleotides are added up to form a unitig sequence.

In the rest of the report, unitigs will refer to maximum unitigs.

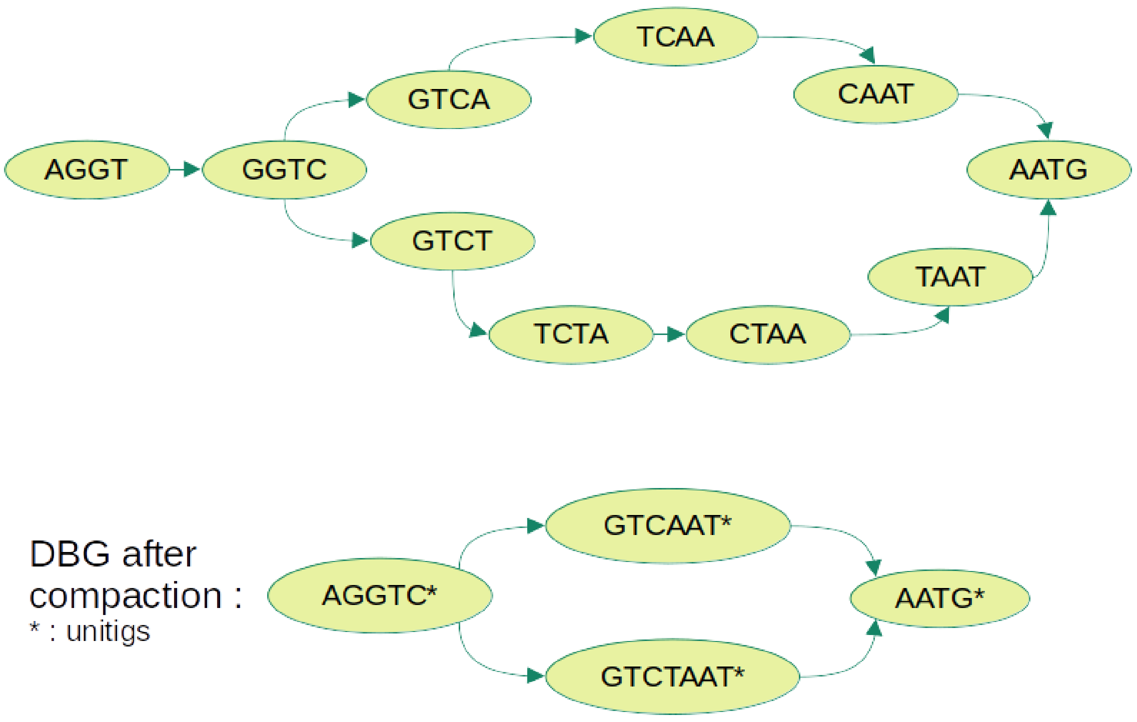


Figure 5: De Bruijn graph with k-mers of size k = 4. Each k-mer overlaps its neighbours on a length of k - 1 = 3. There is an example of an ambiguity with k-mers GGTC and AATG.

Once the reads are corrected, the next step is to compact them into unitigs. This is accomplished by Bcalm2, a tool built upon the c++ library gatb-core. It is based on the previously defined DBG. Numerous approaches have been tested to reduce the footprint of the two steps for de Bruijn graph: building and compaction. Using minimizers (defined in section 4.3), Bcalm [9] reduced the memory usage needed compared to other competing tools, and its new version Bcalm2 adds a divide and conquer approach to parallelise the workload to reduce the running time, by building smaller DBG first, then gluing them back together using duplicated k-mers. To prepare for the next step with REINDEER, we use Bcalm2 twice at different scales. The first pass builds the unitigs at sample level. These sets of unitigs contain the differences in k-mer composition between samples; they have the variants information we need. The second pass builds unitigs globally from the union DBG (DBG of all samples). When doing so, we determine which unitigs could possibly be present across all samples. Because such set of unitigs is very large, due to sequencing errors that remain uncorrected by Lighter, the presence of very rare k-mers, and simply the sheer number of potential combinations an abundance filter must be applied. This filter is detailed in the methodological results sub-section 5.1. Then, the next step is to obtain the description of the difference in unitigs presence or absence between the samples. This task is described in the next subsection.

4.3 Reindeer and monotigs

REINDEER [10] is a tool designed to efficiently index and retrieve abundance of sequences in large collections of samples. It uses hash functions, spectrum preserving string sets (abbreviated SPSS) and the new concept of monotigs.

The first step with REINDEER is to build the index of the unitigs found with Bcalm2 for each file (sample = file). Next, we query the unitigs found for the whole dataset (all the files) to obtain for each unitig a vector of its presence/absence pattern across the samples. This matrix is given to `Metadbgwas` . Before explaining how the tool works, aforementioned SPSS and monotigs must be detailed.

Spectrum preserving string sets are defined as the following (Rahman and Medvedev 2020 [11]): given a set of k-mers X, a set of strings S of length $\geq$ k is a SPSS representation of X if the set of all k-mers from S is exactly X. Examples of SPSS are a set of k-mers (which represents itself), unitigs, or monotigs. From Marchet *et al* (2020), the definition of a monotig is : the sequence of a path in the union DBG in which all constituent k-mers have the same count-vector (abundance count in each sample) and the same minimizer. What must be remembered here is that monotigs represent all k-mers found in the dataset. The main difference between unitigs and monotigs is the compaction step. Monotigs are the result of the compaction of consecutive, and *same abundance*, *same sample presence pattern* k-mers. Alternatively, unitigs are the compaction of non-ambiguous (same definition), consecutive k-mers. The main asset of monotigs is to represent all k-mers with the same abundance from the dataset. It is used to get the presence / absence matrix. We picture the principle of monotigs in figure 6.

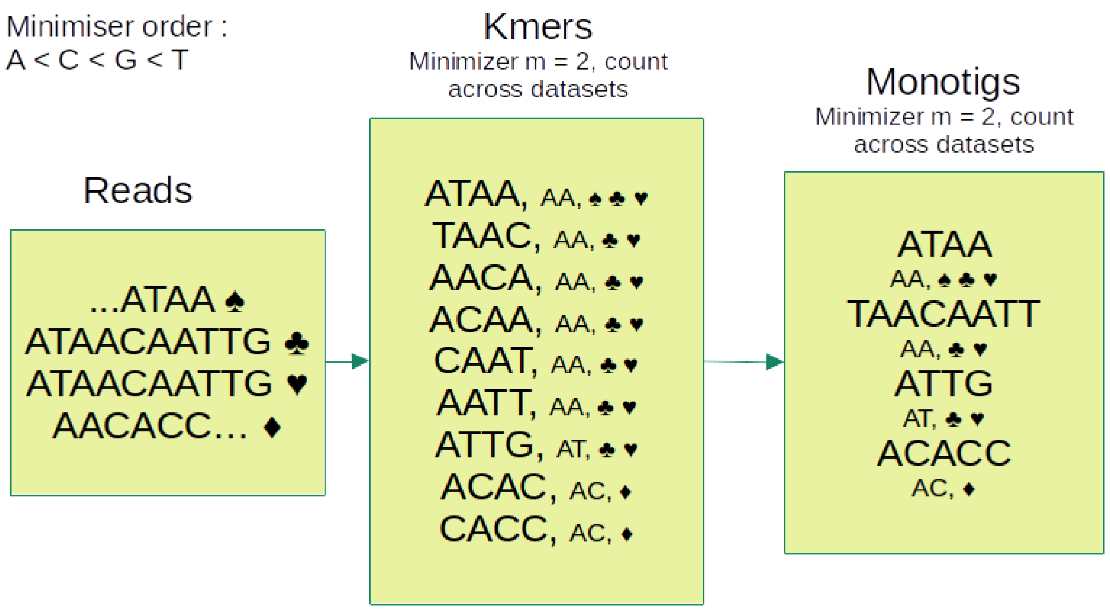


Figure 6: Building monotigs from reads. Middle rectangle shows the k-mers from the reads, with their minimizer and the dataset symbol to show their presence/absence pattern. Right rectangle shows the compaction of k-mers with the same minimizer and pattern: the monotigs.

Monotigs use the notion of union DBG that we define by: a union DBG is the DBG of a set of nodes equal to the set of nodes of a set of DBGs, with edges built from overlaps of $k-1$. More practically, it is the DBG built on all distinct k-mers of the dataset. Monotigs are built upon minimizers. The l-minimizers of a string is its smaller substring of length l according to some ordering. That order can be determined by values output by hash functions, or the lexicographic order as depicted in the figure above, etc and each k-mer has its minimizer. They are used to build the monotigs efficiently. Then, k-mers overlapping each others with the same count will be merged into monotigs.

A quick overview of the REINDEER: the first step is to read the data from the output files of Bcalm2. Next, REINDEER constructs the monotigs to index the k-mers of the union DBG built on all the samples. Finally, a matrix M is built with monotigs as rows, and samples as columns. The values in this matrix are the counts of the monotig in each sample.

The query of the unitigs from de union DBG will allow us to obtain the matrix of presence/absence that we will feed to the modified DBGWAS. The query must be performed using the index. Each query sequence (here unitigs from Bcalm2 output) is decomposed into its k-mers. Different values can be chosen to determine the percentage of k-mers to find in a file to count the query sequence as present in it: the use of a percentage comes from the reformulation of the query problem mentioned above. The choice of this value is discussed in sub-section 5.1 . The k-mers are then transformed into monotig identifiers, which allow us to retrieve the corresponding monotig in the matrix build as explained above, and find the count.

4.4 DBGWAS

Because `Metadbgwas` is a modified version of DBGWAS, the later must be explained in order to highlight the work accomplished to change it. DBGWAS, for De Bruijn Graph GWAS, is a k-mer-based tool which finds variants strongly associated with a particular phenotype. The statistics are calculated with a linear mixed model (LMM) and the results are presented through an interactive web page. The overview of the tool is presented by the figure 7:

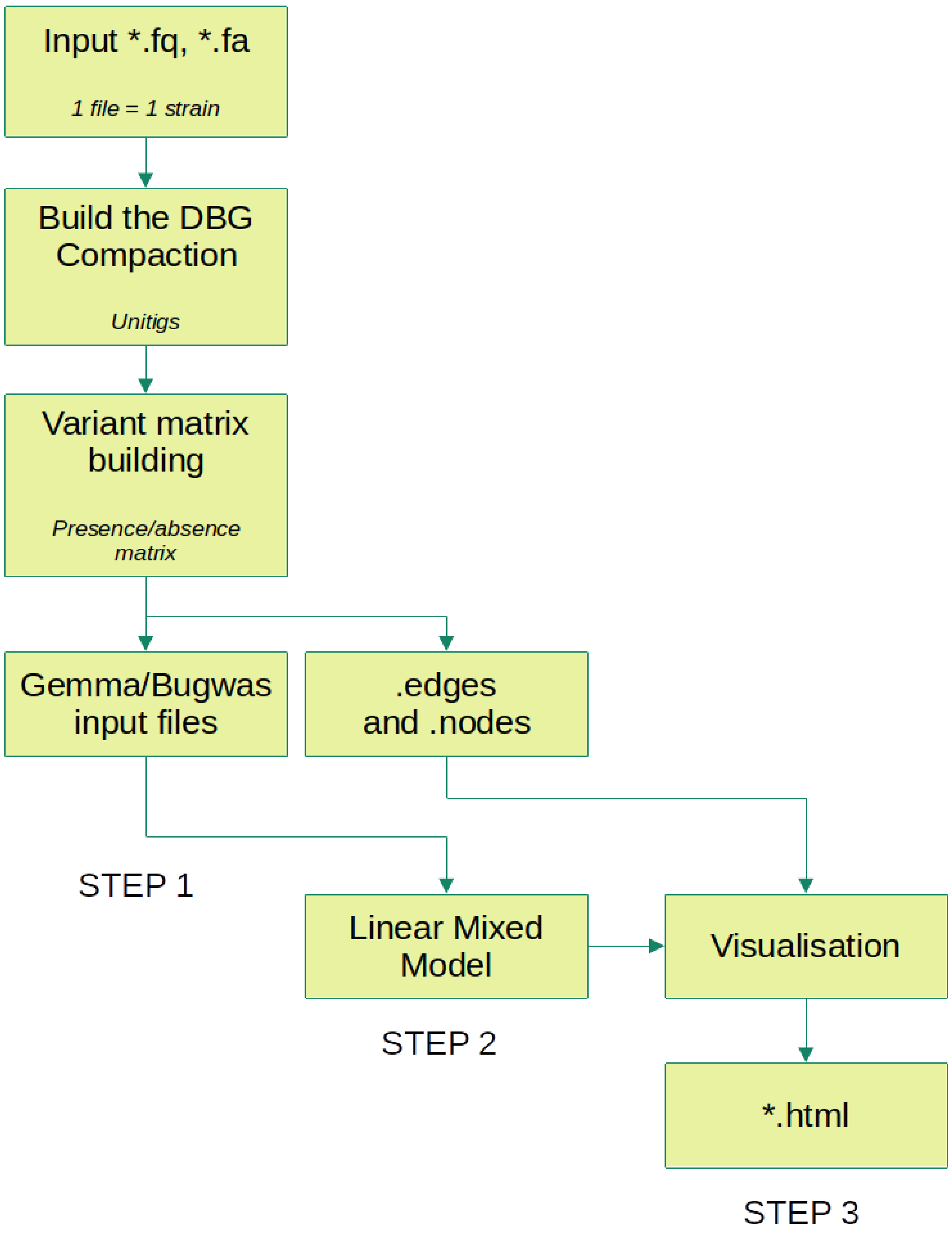


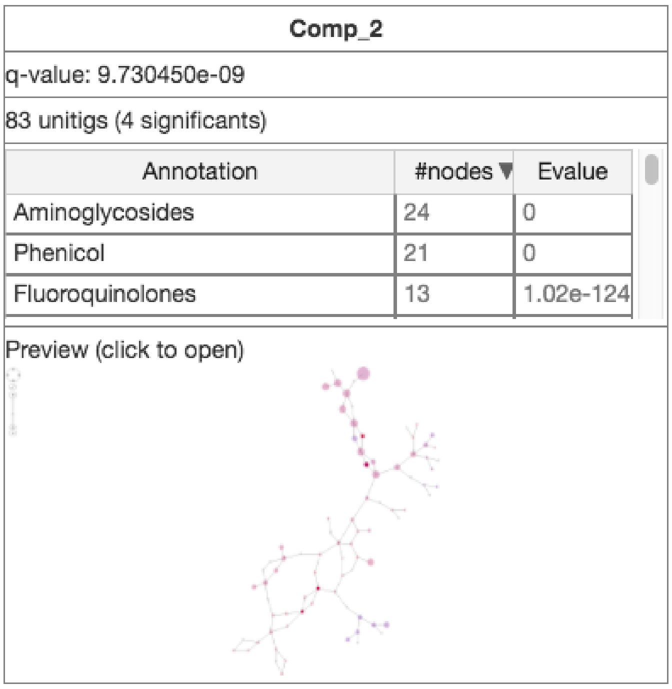
Figure 7: Workflow of DBGWAS. It is decomposed in 3 steps: step 1 finds unitigs, their absence/absence pattern across strains, and prepares input files for the LMM. Step 2 applies LMM to the data. Step 3 builds the numerous files needed for visualisation.

In step 1, a DBG is built and compacted to obtain the unitigs. Then, each unitig is marked as present or absent for each strain to get the variant matrix. From this matrix, the input files for the R packages Gemma and Bugwas are generated, as well as 2 files to depict the DBG: .edges and .nodes (self-explanatory names). The package Gemma (Genome-wide Efficient Mixed Model Association) implements the LMM, whereas Bugwas controls for the lineage effect to take the population structure into account. In step 2, the LMM finds the p-values for the association between the unitigs and phenotypes. The LMM is detailed below.

One of the issues with bacterial GWAS is the structure of the population. Because individuals are closely related due to the clonal nature of their reproduction, entire groups of the bacteria sequenced might share variants. It becomes difficult to distinguish variation *inter* and *intra* groups. One way to account for this hierarchy is to separate fixed effects and random effects, while taking into account the groups. This is what LMM does. For equation (1): Y is a $N \times 1$ column vector of phenotypes. Let X be the variant matrix, of dimensions $N \times p$: N individuals and p predictors. β is the $p \times 1$ column vector for fixed effects coefficients. Let W^T be the design matrix for random effects with N lines and $q \times J$ columns for q random effects (complements of β) and J groups. α the line vector of q random effects, which are distributed as $\alpha \sim \mathcal{N}(0, \sigma^2), \sigma^2 > 0$. α follows a normal distribution centered on 0 because the effect can be either positive or negative on the phenotype. ε the line vector of residuals :

$$Y = X_i\beta + W^T\alpha + \varepsilon_i \tag{1}$$

What is tested is the null hypothesis $H_1 : \beta = 0$ and the alternative hypothesis is $H_2 : \beta \neq 0$. p-values yield q-values once corrected with the Benjamini-Hochberg procedure (correction for the false discovery rate). Finally in the third step, the results are presented through a web page using Cytoscape and the .edge and .nodes files. The final result is described in a web interface. The significant unitigs and their neighbourhood are presented in a sub-graph, along the potential blast annotation, and q-value. The reason to look for the neighbourhood is that it provides context: if there is only one isolated unitig, it might be a SNP. If there is a long chain of significant unitigs, it might be an entire gene linked to phenotype. The sub-graph is interactive, giving the user the possibility to change it for a figure. Annotation is provided if the user gave a protein or nucleic acid database to the integrated blast suite. An example is given in figure 8.



Comp_2

q-value: 9.730450e-09

83 unitigs (4 significants)

Annotation	#nodes ▼	Evalue
Aminoglycosides	24	0
Phenicol	21	0
Fluoroquinolones	13	1.02e-124

Preview (click to open)

Figure 8: From the DBGWAS wiki documentation. Significant component of the graph, and its blast annotation. A red-ish coloration shows an predominant association with phenotype 1, blue-ish for 0. The size of the circle represents the frequency of the unitig (bigger means more frequent). A more detailed view is given when clicking on the component preview.

5 Results

This section described the results obtained. Due to an underestimation of the work required to use the output of REINDEER as an input for DBGWAS, a lot of time was unexpectedly dedicated to implement the needed modifications. It fortunately did not prevent us from developing the whole pipeline, nor to advance the methodology on the subject of k-mer filtering. Once the pipeline was ready, REINDEER showed being a bit unstable, crashing 3 run of the pipeline with unforeseen bugs. We were still able to get results such as the number of unitigs resulting from different abundance values to filter out low abundance k-mers. The tool `Metadbgwas` is available at `https://github.com/Louis-MG/Metadbgwas`.

5.1 Methodological results

In this section, several methodological questions are addressed, in order of appearance in the pipeline. We did not have any concerns about the default parameters of Lighter, and just added the option to discard unfixable reads. This is done to remove low-quality reads, and reduce erroneous k-mer presence.

In Bcalm2, the main question was how to filter the k-mers, and if the filters of the 2 steps should be the same or not. Because the statistics rely on k-mer presence or absence, it is required to keep every k-mer present in the reads to build the unitigs sample-wise. So first, the unitig obtainment is run for each file with a minimum abundance value of 1. This parameter is particularly important for `Metadbgwas` .

We can keep all k-mers of each sample because we already corrected most of the false bases. Secondly, the k-mer, even if present in low abundance, might be significantly linked to the phenotype: it might contain a SNP that is the root cause of a phenotype. With sequencing biases, or a sequencing with a lower yield, the k-mer could be in low abundance, yet important. We then compute the DBG for the whole dataset. This is necessary for the next step of presence/absence matrix generation of the unitigs in the files. Contrary to the sample-by-sample de Bruijn graph generation, we filter the k-mers at this step. All the files are used, hence k-mers should be present more often than once to be kept. There was a discussion about the threshold that should be used to filter out low-abundance k-mers. A k-mer could be considered to be present at least once per file with the phenotype of interest, but that would make the choice dependant of the dataset size, and would not account for population structure: resistance in populations might be due to different mutations, hence different k-mers. A fixed threshold has been discussed too, but as the dataset size increases, the less stringent the value will be. Moreover, the number of occurrence of a k-mer depends also on the coverage of the sequencing. Thus, the threshold was fixed as the 2.5 percent quantile of a Poisson distribution (sequencing coverage follows a Poisson distribution) with lambda parameter being the mean coverage.

In REINDEER, the indexing step is done with default parameters. At the querying step, there was more attention to bring to our choices. By default, if more than 40 percent of the query sequence's k-mers are found in the index, then the sequence is considered present. In `Metadbgwas` , as we want to know the true absence, we set the parameter to 0: if at least one of the k-mers of the sequence is present in the index, the unitig will be marked as present in the sample. The output of REINDEER is then given to the modified DBGWAS. The modifications implemented in DBGWAS are detailed in the next sub-section.

5.2 Modified DBGWAS

Modified DBGWAS is the final tool of the pipeline. It was developed by modifying DBGWAS: the step 1, and slightly modifying the step 3, to parse different files to load the graph with Boost. Step 2 remains untouched as well as the visualisation output. The tool is summarised if figure 9 :

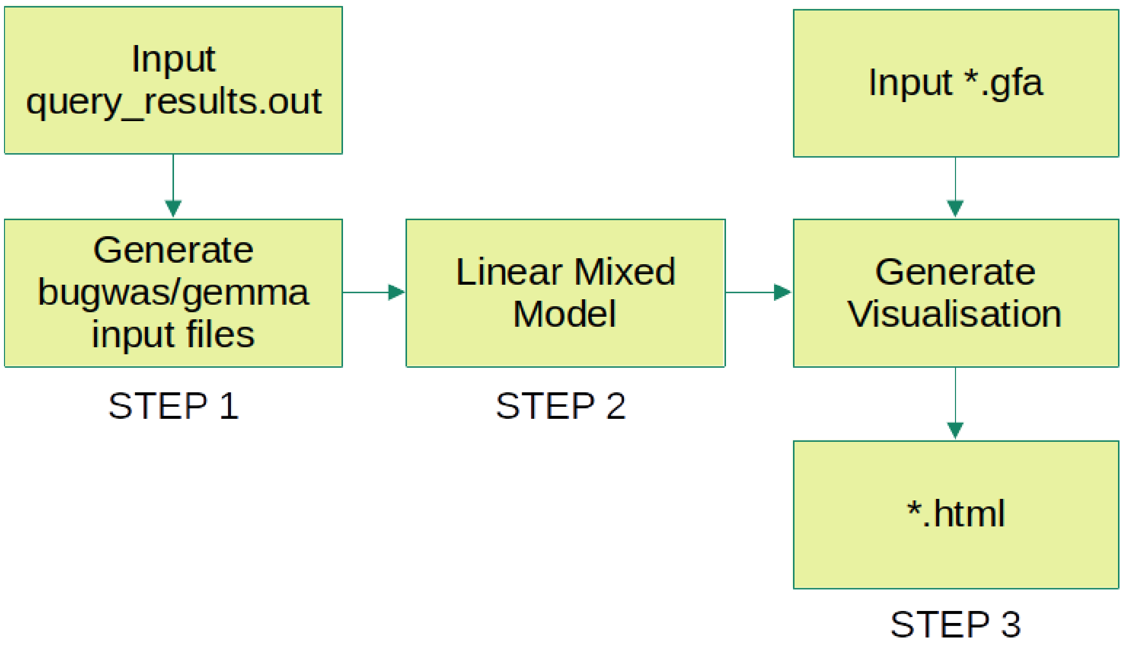


Figure 9: Workflow of modified DBGWAS. First step generates input files for the LMM (using C++), step 2 is still the LMM (Rscripts), and step 3 uses the .gfa to generate visualisation (C++ and Javascript).

Finally, building the pipeline also involved heavy modifications of the tool DBGWAS, which gave rise to a new version of it. Step 1 was building the de Bruijn graph, compacting it into unitigs, and creating the presence/absence matrix input files for the R libraries Gemma [12] and Bugwas [13] at step 2. It was replaced by the previously described tools, and the parsing of the output file of REINDEER. The query output file has the following structure: a header, with indexed file's names as column headers. Each line corresponds to a unitig queried. And for each of these queried unitigs, its abundance in each file is reported in the adequate column. The modified DBGWAS uses the following strategy to convert that file into Bugwas and Gemma input: each line is stored in a structure retaining the ID and vector of abundance pattern. Then each abundance vector is transformed into a presence/absence vector: any value superior or equal to 1 is changed into a 1, and 0 are still 0. To conform to the minor allele description (definition needed), vectors are eventually "reversed": if there are more ones than zeros in the vector, we change the ones by zeros and vice-versa. If the pattern was reversed or not is kept in one of the Gemma/Bugwas input file. Once the final form of the pattern obtained, we check if this pattern has been seen before. This is done by storing each previously unmet pattern in "unique patterns" vector. The name "unique" reflects the fact that the vector contains only distinct patterns, hence each pattern in the vector is unique in the vector. The correspondence between the pattern and the unitig id is then kept in a key-value structure (a dictionary), and updated for each id, whether the pattern was previously seen or not. Finally, the unique patterns, the association between unique patterns and the unitig IDs, the number of unitig each unique pattern represents, whether or not the pattern was changed for the minor allele description, and the correspondence between files and phenotypes are stored into distinct files. These files are the Bugwas/Gemma input. The process is shown in figure 10.

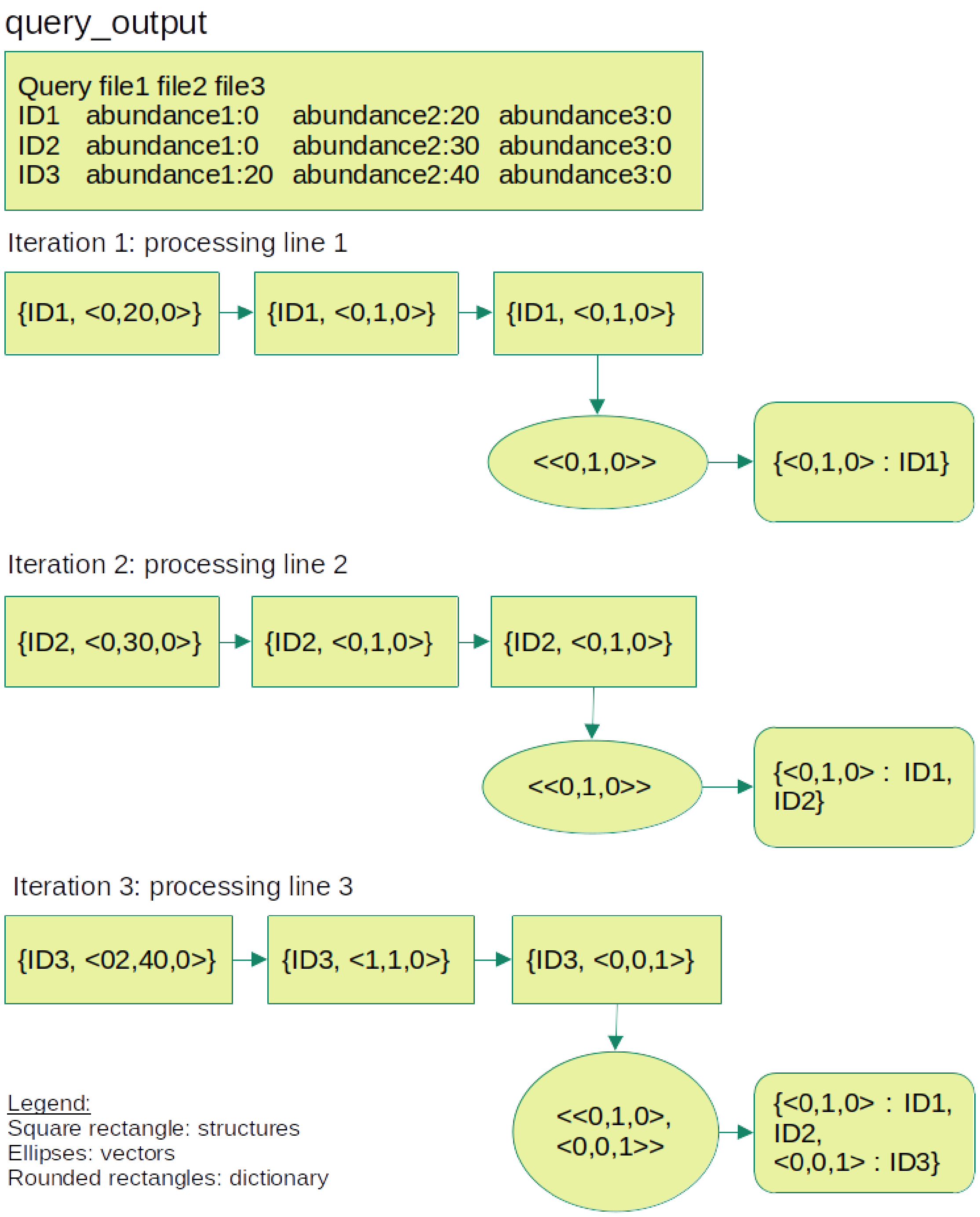

query_output
Query file1 file2 file3
ID1 abundance1:0 abundance2:20 abundance3:0
ID2 abundance1:0 abundance2:30 abundance3:0
ID3 abundance1:20 abundance2:40 abundance3:0
Iteration 1: processing line 1
{ID1, <0,20,0>}
{ID1, <0,1,0>}
{ID1, <0,1,0>}
<<0,1,0>>
{<0,1,0> : ID1}
Iteration 2: processing line 2
{ID2, <0,30,0>}
{ID2, <0,1,0>}
{ID2, <0,1,0>}
<<0,1,0>>
{<0,1,0> : ID1, ID2}
Iteration 3: processing line 3
{ID3, <02,40,0>}
{ID3, <1,1,0>}
{ID3, <0,0,1>}
<<0,1,0>, <0,0,1>>
{<0,1,0> : ID1, ID2, <0,0,1> : ID3}
Legend:
Square rectangle: structures
Ellipses: vectors
Rounded rectangles: dictionary

Figure 10: Example of a query output file processing by modified DBGWAS.

The parsing is done through single-threaded code as multi-threaded parsing methods are complicated and might not be worth it performance-wise. Another important modification that was brought is the change from a custom format (.nodes and .edges) to describe the DBG to a standard format, the Graphical Format Assembly (.gfa). Defining standard formats is one of the struggles of bioinformatics: as the techniques, types of data and tools develop at high rates, the community develops in parallel different ways to store information. Some file format become standard by force of use, others by decisions, but even then already existing programs have to be modified to adjust. Hence the need to align the format to the current standard for de Bruijn graphs. This modification led to slightly modify the step 3 to adapt the graph construction. Egde and Vertex (another word for node) structures were added, as well as functions to populate them before loading the information in the Boost graph.

5.3 Data

Genomes are taken from van Belkum A *et al* [14] and the phenotypes are taken from Jaillard M *et al.* [15]. The genome are stored as SRA files by the NCBI, available through an accession list and the SRAToolkit. Each file contains illumina raw reads of 250bp, with around 2.5M reads per file on average for a total of 282 files and 130GB, 180Gbp. The files were converted to a FASTQ format, then trimmed using TrimGalore [16] to remove adapters and bad quality ends of the reads, and checked thereafter with FASTQC [17]. All files were kept, even the one with much shorter reads (72bp). The phenotypes encoded are the resistance to amikacin and levofloxacin two antibiotics. The pipeline was executed several times with the same set of default parameters, except for the minimum abundance for k-mers for Bcalm2 and for Lighter, which requires some user input that cannot be defaulted to a value. Lighter parameter K was set to "17 6000000", 17 for the length of k-mers used (recommended parameter) and 6000000 is the approximate size of the genome of *P.aeruginosa*, 6 mega base. Different values were tested for abundance-min for Bcalm2: 5, 10, 20, 30, trying two ideas at the same time. The first one is to see how a fixed threshold would perform; the second is to use a 2.5% quantile value of a Poisson distribution with the parameter λ set to the mean coverage of the sequencing. That value for the dataset used was around 30, which yields a 2.5% quantile value of 20. Higher values, here 30, should help us determine if using a high threshold removes too much information.

5.4 The pipeline and its performance

`Metadbgwas` is a pipeline constituted by tools specifically designed to be as efficient as possible, with low memory consumption. They are all written in C++ for a fast execution (and a bit of javascript and R for DBGWAS), including the new REINDEER-modified DBGWAS interface, and they are organised around a shell script. It is published under the open-source Zlib licence, to make it widely available without restriction and to keep the licence compatible with the AGPLv3, under which DBGWAS is licensed.

Due to crashes and bugs corrected during execution of the pipeline, it is not possible to give a precise graphic of the running time in function of filter value chosen, nor in function of the number of threads. The only execution that went further than REINDEER, which crashed repeatedly, is the execution with the parameter abundance-min set to 10. It used 40 threads and 60GB of memory. We detail the running times of the different steps: correction step can be approximated to have taken 5 hours; around 15-20 hours for Bcalm2; 1 to 2 days for REINDEER indexing and querying; and the modified DBGWAS was 10 around hours into the execution, with a complete generation of all Bugwas and Gemma input, plus hopefully a nearly finished gigantic file for visualisation (685GB when it stopped). Considering the sheer size of the files, an option was added to allow the user to compress them after execution.

DBGWAS was partially re-written, to use a different input. Its parsing of the REINDEER query results output file takes advantage of pointers to avoid object copy in the functions, thus minimising memory usage, and stores data in vectors for a compact structure in memory. It also went through a run of Valgrind to look for memory leaks, and none were found. The reliance on third-party libraries was reduced to the bare minimum, and the general structure prior and after compilation was simplified, going from a Superbuild to a simple build. Compilation instructions are generated with a simple CMake: there is no standard when it comes to compilation is C++, and CMake is one of the widespread program that makes it as standard and easy to debug as possible. Where DBGWAS was bringing local versions of the libraries boost and gatb-core within its package, burdening it, `Metadbgwas` uses the packaged version for Ubuntu, installed with the package manager apt.

Being a pipeline, `Metadbgwas` is composed of several tools. As seen in figure 1, it can be seen divided in multiple steps. We take advantage of this to ensure that users do not have to re-run everything in case of a crash, and three options allow to restart the pipeline form where it left: it is possible to skip Lighter (step 1), Lighter and Bcalm2 (steps 1 and 2), or REINDEER and all the precedents (steps 1, 2, and part of step 3). All parameters are explained, with their potential default values in the help message available through the command line interface or documentation on github. User's command is saved in a text file in the output directory to keep track of what was done.

The tool was also made available for singularity and docker, through the cloud image-hosting service of docker hub. This is important several aspect of modern science. It ensures perfect reproductibility, as the whole environment needed to run `Metadbgwas` can be reproduced and as the history of the tool is kept traceable (similar to other versioning tools, like git). Moreover, as a the usage of clusters and cloud computing takes more and more importance in their daily practice, scientists find their rights on the computing environment limited, in the sens that they cannot install tools, or libraries at will to avoid compatibility issues and more disastrous mistakes. Thus, the use of docker or singularity provides the possibility to use the tool in these contexts. There are aspects of the pipeline that can still be enhanced, described below.

Potential improvements are the re-encoding of the Bcalm2 python script responsible for the .gfa production in C++, diminishing the execution time. The assignment of one core to take care of that task while reindeer is indexing and querying the unitigs (.gfa is only needed at step 3 of `Metadbgwas`) could also save time, and REINDEER will hopefully run with more than 1 core: in its actual state, the tool crashes when multi-threaded at the query step. This problem takes a huge toll on the pipeline performance. More disk space could also be saved by deleting number of intermediary files, and one particular file mentioned earlier is so big that a solution needs to be chosen to reduce its size. A cleaning option is already existing but is not fully implemented. In general, the options deserve a more detailed version of their role; it could be achieved by adding a man option and wiki page. Lighter might be better used if the user is allowed to provide a file with the genome sizes corresponding to each file; right now, only one can be given, gene if approximate, which can accommodate different close sizes. A specific flag in which the user decides the pick-rate α is recommended to by-pass this, but it would still be more convenient. To continue what was done with the boost and gatb-core libraries, the tool Bcalm2 is also available through apt, and could be installed this way rather than with a git sub-module. It might simplify a bit more the installation process and make the tool more available for other tasks, as it wont be needed to reach its path deep in `Metadbgwas` . Looking at the containers; if the pipeline is available through Docker and Singularity, it is not yet ready for Kubernetes, which is becoming popular. Conda will also be considered for the diffusion of the pipeline. Finally, the code might need, or present the opportunity to be improved in the future. Whether or not this work is done by its original developer or anyone else, it is important to prepare this eventuality and make it as comfortable as possible. Linters are great assets to make the code more readable, and thus one should be applied to the C++ code of the modified DBGWAS, and if feasible to the shell script of `Metadbgwas` . For less experienced users, an interactive mod could be added to the tool. Instead of risking to mix up arguments, they would be prompted one by one by the tool, with reminder of the argument represents.

5.5 Biological results

The number of unitigs was obtained for all filtering values. For the parameter abundance-min with values 5, 10, 20, 30, they were respectively 130M, 55M, 24M and 13M unitigs. Each time the filtering value is doubled, the number of unitigs decreases by a factor of approximately 2. Unfortunately, all the executions of the pipeline stopped at some point due to unexpected errors when using REINDEER (for minimum abundance of 5, 20 and 30), or because of storage shortage (minimum abundance of 10) caused by the cluster default space available to a user filled by the gargantuan DBGWAS file. That particular issue should be solved once an appropriate threshold value is found.

Taking a closer look at the amount of unitig per filtering abundance value to filter, we find that the 2.5% value of a Poisson law with parameter λ set to the mean coverage value gives quite good results. The number of unitigs found is still manageable by all the tools, but more tests needs to be done to evaluate the time needed. The number of unitigs found with minimum abundance of 30 is half the precedent value, which is then also handled by the tools in reasonable time (estimation from the run with abundance-min parameter at 10 gives around 2 - 3 days), but the lack of DBGWAS output refrain us from drawing a conclusion about such value being to high or not. We cannot either predict if such value would filter out the significant unitigs

6 Conclusion

At the beginning of the project, the goal was to link several tools together to build a pipeline, and then to compare its result to previous findings. But the amount of work necessary to use the output of REINDEER as an input for DBGWAS was highly underestimated: the two cannot just be connected, using an output of the first as an input of the second. A deep dive into the C++ code was necessary to adapt DBGWAS; the transformation was successful, but took the majority of the time for the internship. Moreover, diverse bugs were encountered during several execution at the REINDEER step, stopping us from getting biological results beyond the number of unitigs after filtration.

We still have a major result; `Metadbgwas` , a pipeline that uses recent, state-of-the-art tools and k-mer-based methods to bring GWAS principles to bacterial metagenomes. Every step is designed to handle large dataset by taking advantage of memory-wise efficient structures like Bloom filters. Lighter uses solid k-mers to correct the reads before the manipulation of the k-mers, making the tool more self-sufficient. Bcalm2 brings down the complexity and time to build and compact de Bruijn graphs. At this step, we also bring down the quantity of data of the whole process, filtering k-mers based on their frequency has been discussed and implemented dataset-wise. It aims to eliminate low-abundance k-mers that could either be too rare or sequencing errors that were not corrected by the pipeline. We chose a statistical threshold determined by the Poisson's law (2.5%), better than a fixed threshold as it varies with the coverage and thus the number of reads. We assessed the effect of this threshold percentage on the number of unitigs retained, but could not yet verify its impact on biological results, due to several material obstacles. This work will be completed very soon. Then REINDEER indexes the large dataset in a short time and finds the presence/absence patterns of the unitigs across the samples. Finally, the modified DBGWAS applies a Linear Mixed Model that is adapted to the clonal structure of bacteria populations.

The result is available through a user-friendly web page, and interactive, graphical information to help the exploitation of the findings. The backbone of the tool is a shell script with named arguments. Several level of verbosity are possible, and the command used in save in the output directory to help people to not forget the parameters they used. DBGWAS had to modified: and the generation of the Gemma and Bugwas input files was rebuilt from scratch in C++. Using CMake, the DBGWAS Superbuild installation and compilation was simplified, local libraries were removed in favour of the apt packaged versions. This makes the whole installation lighter and is part of the good practices for bioinformatics. Moreover, a custom file format used to describe a de Bruijn graph was replaced by the more standard Graphical Format Assembly, or gfa. This change is also a part of an effort to streamline the pipeline. Furthermore, to make `Metadbgwas` a modern tool, we followed the containerisation movement and made the pipeline available to docker and singularity through docker hub. There is even the possibility for the user to build its own image locally using the dockerfile at disposition in the github repository. Containerisation is also an asset for reproductibility, which is a pillar of all sciences.

This internship has been the opportunity to learn the programming language C++, how does compilation and linking works; how to use tools like CMake to simplify the process; the difficulties and good practice of development and deployment. The pipeline was a good justification to start learning and using container, Docker and Singularity, as well as the cluster manager slurm. It also ha been an introduction to novel structures and techniques to manipulate kmers, like monotigs, or minimisers.

For future works, the pipeline can be brought to yet another rising container, Kubernetes, or the well-known environment control tool Conda. There are multiple ways to make the pipeline run faster, like by re-writing a python script in C++, and potentially replacing BLight (a component of REINDEER) with sshash [18], a new tool implementing k-mer dictionaries. Several quality of life improvements are also on the list. Above all, we must obtain the biological results, and discuss them, to understand what the pipeline outputs and how it compares to DBGWAS.

7 Thanks.

Many thanks to Laurent Jacob, Camille Marchet for their kind and enthusiastic supervision. They provided the road map to create `Metadbgwas` , helped me to understand the tools, and the methods behind them. Thank you to Francois Gindraud for his explanations of the C++ language, which made the understanding, writing and debugging of the code way less painful. I would like to acknowledge the help of Leandro Lima about DBGWAS code. This work was performed using the computing facilities of the CC LBBE/PRABI.

References

[1] V. Tam, P. Nikunj, M. Turcotte, B. Yohan, P. Guillaume, and D. Meyre, "Benefits and limitations of genome-wide association studies.," *Nat Rev Genet.*, pp. 467–484, 2019.

[2] M. Jaillard, L. Lima, M. Tournoud, P. Mahé, A. van Belkum, V. Lacroix, and L. Jacob, "A fast and agnostic method for bacterial genome-wide association studies: Bridging the gap between k-mers and genetic events," *PLOS Genetics*, vol. 14, pp. 1–28, 11 2018.

[3] C. M. Francois, F. Durand, E. Figuet, and N. Galtier, "Prevalence and implications of contamination in public genomic resources: a case study of 43 reference arthropod assemblies," *G3: Genes, Genomes, Genetics*, vol. 10, no. 2, pp. 721–730, 2020.

[4] P. Kirstahler, S. S. Bjerrum, A. Friis-Møller, M. la Cour, F. M. Aarestrup, H. Westh, and S. J. Pamp, "Genomics-based identification of microorganisms in human ocular body fluid," *Scientific reports*, vol. 8, no. 1, pp. 1–14, 2018.

[5] A. Zimin, K. A. Stevens, M. W. Crepeau, A. Holtz-Morris, M. Koriabine, G. Marçais, D. Puiu, M. Roberts, J. L. Wegrzyn, P. J. de Jong, *et al.*, "Sequencing and assembly of the 22-gb loblolly pine genome," *Genetics*, vol. 196, no. 3, pp. 875–890, 2014.

[6] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016.

[7] L. Song, L. Florea, and B. Langmead, "Lighter: fast and memory-efficient sequencing error correction without counting," *Genome biology*, vol. 15, no. 11, pp. 1–13, 2014.

[8]

[9] R. Chikhi, A. Limasset, S. Jackman, J. Simpson, and P. Medvedev, "Research in computational molecular biology," in *Proceedings of the 18th Annual International Conference, RECOMB 2014, Pittsburgh, PA, 2 to 5 April 2014. Lecture notes in computer science 8394*, pp. 35–55, SpringerVerlag Cham, Switzerland, 2014.

[10] C. Marchet, Z. Iqbal, D. Gautheret, M. Salson, and R. Chikhi, "Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets," *Bioinformatics*, vol. 36, no. Supplement_1, pp. i177–i185, 2020.

[11] A. Rahman and P. Medvedev, "Representation of k -mer sets using spectrum-preserving string sets," in *International Conference on Research in Computational Molecular Biology*, pp. 152–168, Springer, 2020.

[12] X. Zhou and M. Stephens, "Efficient multivariate linear mixed model algorithms for genome-wide association studies," *Nature methods*, vol. 11, no. 4, pp. 407–409, 2014.

[13] S. G. Earle, C.-H. Wu, J. Charlesworth, N. Stoesser, N. C. Gordon, T. M. Walker, C. C. Spencer, Z. Iqbal, D. A. Clifton, K. L. Hopkins, *et al.*, "Identifying lineage effects when controlling for population structure improves power in bacterial association studies," *Nature microbiology*, vol. 1, no. 5, pp. 1–8, 2016.

[14] A. van Belkum, L. B. Soriaga, M. C. LaFave, S. Akella, J.-B. Veyrieras, E. M. Barbu, D. Shortridge, B. Blanc, G. Hannum, G. Zambardi, *et al.*, "Phylogenetic distribution of crispr-cas systems in antibiotic-resistant pseudomonas aeruginosa," *MBio*, vol. 6, no. 6, pp. e01796–15, 2015.

[15] M. Jaillard, A. van Belkum, K. C. Cady, D. Creely, D. Shortridge, B. Blanc, E. M. Barbu, W. M. Dunne Jr, G. Zambardi, M. Enright, *et al.*, "Correlation between phenotypic antibiotic susceptibility and the resistome in pseudomonas aeruginosa," *International journal of antimicrobial agents*, vol. 50, no. 2, pp. 210–218, 2017.

[16] F. Krueger, F. James, P. Ewels, E. Afyounian, and B. Schuster-Boeckler, "Felixkrueger/trimgalore: v0.6.7 - doi via zenodo," July 2021.

[17] S. Andrew, "Fastqc."

[18] G. Ermano, "Sparse and skew hashing of k-mers,"