Regular Article

On the use of algebraic topology concepts to check the consistency of genome assembly

Jean-François Gibrat

MaIAGE, INRA, Université Paris-Saclay, Jouy-en-Josas 78350, France

Received July 1, 2019; accepted September 6, 2019

This paper presents a preliminary work consisting of two contributions. The first one is the design of a very efficient algorithm based on an “Overlap-Layout-Consensus” (OLC) graph to assemble the long reads provided by 3rd generation technologies. The second concerns the analysis of this graph using algebraic topology concepts to determine, in advance, whether the assembly of the genome will be straightforward, i.e., whether it will lead to a pseudo-Hamiltonian path or cycle, or whether the results will need to be scrutinized. In the latter case, it will be necessary to look for “loops” in the OLC assembly graph caused by unresolved repeated genomic regions, and then try to untie the “knots” created by these regions.

Key words: NGS technologies, OLC assembly graphs, genomic repetitions, homology groups, Betti numbers

The sequencing technologies

Over the past 25 years, sequencing technologies have become central to biology. Advances in this area have made it possible to significantly reduce costs while massively increasing the yield (the number of nucleotides sequenced in an experiment). These advances were compared to those in electronics (Moore’s Law), but the doubling period is 10 months for sequencing technologies while it is 18 months for electronics. Indeed, the cost of sequencing the human genome, since its first publication in 2001, has been divided by a factor of 10 million. First of all, sequencing makes it possible to determine the genetic information of an organism (its genome). The genome of the organism of interest is split into a very large number of redundant fragments, part of the sequence of which is determined by sequencing technologies (the sequenced part of a fragment is called a “read”). Due to redundancy, these reads are overlapping. This overlap is used to reconstruct the original genome (referred to as genome assembly). Even when the genome of an organism is already known, it may be necessary to sequence it, or the products of its expression, again to study, e.g., gene expression under certain conditions (RNA-seq), regulation of gene expression (ChIP-seq), small non-coding RNAs, genomic variants, epigenetic modifications, 3D chromatin conformation inside the nucleus (Hi-C), etc. For instance, in genomic medicine, the genome of tumor tissues of a cancer patient is sequenced to try determining the cause of the disease (point mutation in a gene or regulation site, chromosomal rearrangement, variation of the gene copy number, etc.). For these analyses, the reads are not assembled, but are mapped to the known genome. The mapping of reads to the genome is the prerequisite step of many further analyses performed by biologists as listed above. Assembly and mapping are the two primary analysis techniques for processing the sequencing data from which all other analyses are derived.
Hereafter, we will only be concerned with genome assembly. From 2007 onwards, 2nd generation technologies (then called NGS for “Next Generation Sequencing”) became available and set in motion the process of reducing costs and increasing yields at the same time. At present, the largest sequencing machines, operated by the major sequencing centers, can produce 20 billion short reads (of length at most 2×150 nucleotides) in one experiment, for a yield of 6 trillion sequenced nucleotides. These technologies also have the advantage of having a very low error rate (<0.5%). However, as we will discuss below, the short length of the reads is the Achilles’ heel of these technologies regarding genome assembly. Many genomes of higher organisms, for instance those of plants, are characterized by repetitions of genomic regions. The short reads do not allow these repeated regions to be spanned, causing major difficulties for assembly algorithms (resulting in fragmentation of the genome and incorrect connections between certain parts of the genome). Third generation technologies have been introduced in 2016. These technologies are characterized by a distribution of read lengths with a median around 15,000 nucleotides and whose longest reads in the distribution tail can reach a length of 100,000 nucleotides. Such read sizes allow, to a large extent, to solve or at least strongly mitigate, the problem of repetitions in genomes. Unfortunately, the error rate of these technologies is approximately 15% (on average, a sequenced nucleotide out of seven is incorrect). They have a moderate yield (between 1 and 10 billion nucleotides per experiment), but growing rapidly.

The different generations of sequencing technology, with their strengths and weaknesses, have a major impact on the algorithms used to assemble or map the reads they produced. Usually, each generation requires the development of new algorithms to be able to perform efficiently these two fundamental tasks.

**Genome assembly algorithms**

As indicated above, genome assembly is based on overlapping redundant reads. Mainly, two approaches have been proposed to tackle this problem.

**Greedy algorithms**

The oldest one is based on greedy algorithms that iteratively join reads according to the extent of their overlaps. One such algorithm consists in finding the shortest common superstring, i.e., the shortest character string that contains all the reads. Note, however, that insisting on obtaining exactly the shortest string leads to an exponential algorithm. In practice, a heuristic algorithm is used instead. In this approach, overlapping reads that conflict with already assembled contigs are ignored. Although the earlier genome assemblers relied on such greedy algorithms [1–2], it became soon apparent that they had severe limitations for large genomes exhibiting many repetitions, leading to highly fragmented genome assemblies.

**Graph algorithms**

The second approach is based on graph algorithms: OLC (Overlap-Layout-Consensus) and de Bruijn graphs. Although OLC graph algorithms were the first to be proposed [3], the advent of 2nd generation sequencing technologies, with the considerable number of reads they yield, led bioinformaticians to switch to algorithms based on de Bruijn graphs in recent years.

**Algorithms based on de Bruijn graphs**

The vertices of a de Bruijn graph are all the k-mers of size $k$ over a particular alphabet (A, C, G, T for DNA) and directed edges connect two vertices if the $k$-1 last letters of the first vertex are identical to the $k$-1 first letters of the second vertex. Thus, 2nd generation reads are divided into k-mers overlapping by $k$-1 positions that constitute the vertices of the corresponding de Bruijn graph. Assembling the genome, then, consists in finding an Eulerian path in the de Bruijn graph, i.e., a path that goes once through all the edges [4–5]. However, note that most algorithms do not attempt to find a complete Eulerian path, but merely return unambiguous, unbranched parts of the path.

Although it may seem at first totally counterintuitive to divide the already very large number of reads into smaller pieces, de Bruijn graph-based algorithms have two main advantages. First, they do not require the cross comparison of all the vertices (reads) to define the edges as in the algorithms based on OLC graphs (as discussed hereafter) which is a costly operation when the number of reads is very large. Second, all the repetitions are collapsed in the same set of k-mers in the graph. On the downside, the number of vertices is approximately the number of nucleotides in the genome (this disregards repeated k-mers and the ends of the reads). For large eukaryotic genomes, this generates enormous graphs with a very large memory footprint. In addition, sequencing errors compound the problem by adding, for each error, $k$ spurious vertices. A number of algorithms have been developed to try alleviating this problem [6–8], but this is still an active research area.

**Algorithms based on OLC graphs**

The availability of data from the 3rd generation sequencing technologies is changing the situation again. They are characterized by very long reads (compared to 2nd generation reads) with a median around 15,000 nucleotides as mentioned above. With such data, the order of magnitude of the number of reads required for sequencing the human genome with the coverage of 30 is one million. This is a respectable number, but far from the billions of k-mers that comprise the vertices of a de Bruijn graph. Therefore, this puts the algorithms based on OLC graphs back into play. The principle of algorithms based on OLC graphs is as follows. The vertices of the OLC graph are the reads. There is an edge between two vertices if there is an overlap between the two corresponding reads. To assemble the genome, it is
generally considered necessary to find a Hamiltonian path in the graph, that is, a path that goes through all the vertices once and only once. To correctly join reads together, more information than just the simple knowledge of their overlap is needed, namely the number of overlapping nucleotides, whether this overlap occurs at the 5’ or 3’ end of the first read in a connected pair of vertices (if one of the reads is included in another, it is removed from the vertex list) and finally the mapping sense. The latter is a consequence of the fact that, experimentally, reads are generated from both the direct and complementary strands of the DNA. Therefore, it is necessary to align both the direct and complementary sequences of one of the reads of the pair with the sequence of the other. Note that the edges are oriented. Each edge shares the same weight (the number of overlapping nucleotides) and mapping sense (0 if the two reads overlap with their current sequence, 1 if the complementary sequence of one of them has been used to align them) but different directions according to which vertex of the pair (Vi, Vj) is considered first. The attribute associated with the edge direction indicates at which end of the first read of the pair occurs the overlap. For instance, considering the direction (Vi, Vj), if the j-th read overlaps the 3’ end of the i-th read, the attribute of the edge is 3’. Conversely, for the direction (Vi, Vj) the edge attribute is 5’. It thus makes a difference if the assembly path goes from Vi to Vj or takes the opposite direction.

A naive algorithm to determine the edges of the OLC graph would result in a O(N^2) time where N is the number of reads. Note that the number of reads is proportional to the genome size, G and coverage, C. In addition, if dynamic programming is used, the alignment of each pair of reads would take O(LiLj) time where Li and Lj are the read lengths. This is particularly detrimental for 3rd generation reads that are very long.

To assemble a genome, it turns out that we do not need to find a Hamiltonian path, a simple spanning tree of the graph is sufficient. Any spanning tree will do. If there is no loop in the graph, the exploitation of this spanning tree is straightforward (see below). If there are loops in the graph, more caution should be exercised. Hereafter, we call the path in the spanning tree that leads to the genome assembly a pseudo-Hamiltonian path (or cycle if the genome is circular). For this purpose, a depth first search algorithm can be used that is O(V+E) in time (where V is the number of vertices and E the number of edges). However, the OLC graph is very sparse. Assuming, for the sake of simplicity, that the reads have all the same length L, the number of reads to cover C times a genome of length G is \( N_G = GC/L \). The probability that two reads overlap is given by \( p = (2L − 2\Lambda)/G \) where \( \Lambda \) is the minimum overlap length ensuring that it is very unlikely that the overlap is due to chance. The mean number of reads overlapping a given read is, ignoring the \( \Lambda \) term, \( N_G p = 2C \). It follows that the mean number of edges of the graph is \( CV \). To assemble a genome into a single contig, it is enough to have a coverage, \( C \), of about 10–15 [9]. Thus, the depth first search algorithm applied to an OLC graph is roughly \( O(V) \) in time.

In the following, I will present an algorithm based on a hash table that creates the OLC graph in a time that is linear in the number of reads. Then, I will show how one can harness some algebraic topology concepts to check whether this graph will lead to a consistent genome assembly. I will explain what is meant by “consistent” in this context.

**Methods**

**Creating the assembly OLC-graph**

**Building the hash table (hashing)**

First, the reads are hashed (i.e., indexed). A window of size \( k \) is moved along the read sequences, one position at a time, and the sequence in this window is transformed into a hash value that is used as index in the hash table. According to the magnitude of \( k \) there are two possibilities. If \( k \) is sufficiently small for the number of possible words of \( k \)-letters with a 4-letter alphabet \{A, C, G, T\} to be moderate (for instance, there are 1,048,576 words of size 10) the size of the hash table is taken as being \( 4^k \). The hash values are computed recursively as follows. Each k-word is considered as an integer expressed in the base-4 numeral system. We define a function \( f \) that maps the 4 nucleotides to the 4 digits of this base, e.g., \( f(A) = 0, f(C) = 1, f(G) = 2, f(T) = 3 \). The hash value associated with the \( i \)-th \( k \)-word in the sequence, \( h(i) \), is thus computed as:

\[
h(i) = \sum_{t=0}^{k-1} f(\text{seq}[i+t]) \times 4^{k-t-1}
\]

where \( \text{seq}[i+t] \) is the nucleotide found at the \( (i+t) \)-th position in the read sequence (the numbering of nucleotides in the sequence starts at 0). Note that only \( h(0) \) needs to be computed this way, other hash values are computed recursively as:

\[
h(i+1) = 4 \times h(i) + f(\text{seq}[i+k]) - f(\text{seq}[i]) \times 4^{k-1}
\]

\( \text{seq}[i+k] \) is the new nucleotide that is included in the window when the latter is right-shifted by one position and \( \text{seq}[i] \) is the one leaving the window. With this definition, there is a one-to-one correspondence between a \( k \)-word and an integer in the range \([0, 4^k-1]\) and thus there is no possibility of collision in the hash table. When \( k \) is large, e.g., for \( k=25 \) there are more than \( 10^{10} \) words, the above method obviously does not work. The size of the hash table is chosen to be the power of 2 the closest to the number of different \( k \)-words existing in the reads (taking into account both the direct and complementary senses). The hash value is computed recursively as proposed in ntHash [10]. The hash value of the \( i \)-th word is given by:
leaving the byte on the left are reintroduced on its right end side. Each color-coded bit is shifted by 4 positions to the left and those 97 (lower panel). For simplicity, integers are represented on a single word. For instance, the 4-word ‘GTAC’ is coded as 10110001 to represent the cyclic binary left rotation of 4 positions that is larger than the size of the variable holding the integer (‘<<’ in the C language) whose behavior is undefined if the cyclic binary left rotation uses the bitwise left shift operator.

The nucleotides of the reads are first encoded on 2 bits (A=00, C=01, G=10, T=11), allowing the packing of 4 nucleotides per byte. The tag is the integer represented by the nucleotide

\[ h(i) = L^{k-1}g(seq[i]) \oplus L^{k-2}g(seq[i+1]) \oplus \ldots \]

\[ \oplus L^i g(seq[i+k-2]) \oplus g(seq[i+k-1]) \] (3)

where \( \oplus \) is the bit-wise exclusive OR operator, \( g() \) is a function mapping the 4 alphabet letters to random integers. \( L^p \) represents the cyclic binary left rotation of \( p \) positions that is applied to the binary representation of the random integer assigned to each nucleotide by the function \( g \) (see Fig. 1). As above, only the first hash value needs to be calculated in this way. All the others are computed using the following recursive formula:

\[ h(i+1) = L^h(i) \oplus L^i g(seq[i]) \oplus g(seq[k+i]) \] (4)

Note that this method is applicable up to \( k=64 \) since the cyclic binary left rotation uses the bitwise left shift operator (‘<<‘ in the C language) whose behavior is undefined if the shift is larger than the size of the variable holding the integer (at most uint64_t in C).

This computation of the hash values can generate collisions in the hash table (when different words turn out to have the same hash value). Therefore, k-words must have a tag to distinguish them in case of collision. To compute this tag, the nucleotides of the reads are first encoded on 2 bits (A=00, C=01, G=10, T=11), allowing the packing of 4 nucleotides per byte. The tag is the integer represented by the nucleotides of the k-word encoded this way. It is unique for a given word. For instance, the 4-word ‘GTAC’ is coded as 10110001 which is 177 in binary. This procedure is straightforward up to \( k=32 \), since an encoded 32-word needs 64 bits to be stored (in a uint64_t variable in C). If \( k \) is larger than 32, one must compute the tag integer modulo some large prime number. This can be done piecewise, thus avoiding overflows. The drawback of using the modulo is that i) it cannot be excluded that two different k-words have the same hash value and the same tag after taking the modulo, even though this is not very likely and ii) the computation of the modulo slows down the computations.

The value of \( k \) that is used for hashing the reads depends on which type of data needs to be assembled. If the data are raw, uncorrected 3rd generation reads, then this value must be small (as mentioned above, 3rd generation reads have an error rate of about 15%, i.e., on average 1 in 7 nucleotides is false). Thus, a value of \( k \) of circa 6–9 is required. If the 3rd generation reads have previously been corrected, then \( k \) may be greater, although a larger value of \( k \), in practice, provides only a modest benefit in terms of computation.

If there is no collision, each row of the hash table (a row corresponds to a given hash value and, hence, to a unique k-mer) contains a list of doublets (read index, position of the k-mer in this read). If there are possible collisions, each row of the hash table contains a list of triplets (read index, k-mer tag, position of the tagged k-mer in this read). As mentioned above, the tag is used to distinguish different k-mers with the same hash value.

**Using the hash table to find overlapping reads (mapping)**

The reads are once again hashed, but this time using non-overlapping windows. Each window is considered first in the direct sense, i.e., the actual sequence in the read, and then in its complementary sense for the reason explained above.

For each read, \( i \), this provides a list of quadruplets \( \{R_i, k_i, pos, sns\} \), where \( R_i \) is the index of another read, \( k_i \) is the \( j^{th} \) k-mer of \( R_i \) that maps read \( i \) at position \( pos \) with sense \( sns \). This is called the mapping phase. The process is schematically shown in Figure 2a.

**Determining overlap characteristics needed for assembling the reads (aligning)**

This step consists of an analysis of the list associated with a given query read, which results from the previous step, to extract all reads that overlap with this query read along with the characteristics of the overlap (the number of overlapping nucleotides, the mapping sense, the 5’ or 3’ end of the read at which the overlap occurs). Figure 2b illustrates this process.

This procedure allows the creation of the OLC graph in a time linear in the number of reads.

**Assembling the reads**

A depth first search algorithm is then used to find a spanning tree that contains a pseudo-Hamiltonian path (if the genome is circular one finds a pseudo-Hamiltonian cycle, i.e., the path comes back to the vertex that started the path). The genome is assembled along this path. This is just a matter of “bookkeeping” for each new vertex included in the path, the corresponding read is joined to the growing contig according to the mapping sense, the number of overlapping nucleotides and the 5’ or 3’ end at which the overlap occurs.

As shown in Figure 3, if there are unresolved repetitions in the genome, it is not possible to find a pseudo-Hamiltonian path. The graph exhibits loops that cause the path to go several times through the same vertices. This can result in wrong connections between different parts of the genome or in fragmented contigs. A question then arises: can we predict in advance whether the OLC graph contains loops that will complicate the genome assembly? This question can be
addressed using certain concepts of algebraic topology.

**Algebraic topology in a nutshell**

To give even a simple overview of algebraic topology is far beyond the scope of this article. Algebraic topology is increasingly used in the domain of topological data analysis (TDA). The interested reader is referred to a presentation of M. Wright who does a fine job at explaining the use of algebraic topology for TDA to a layman audience (www.youtube.com/watch?v=h0bnG1Wavag). For a more knowledgeable audience, A. Zomorodian [11] provides an excellent overview of the algebraic topology concepts underlying TDA.

According to Wikipedia, “Topology is concerned with the properties of space that are preserved under continuous deformation such as stretching, twisting, crumpling and bending, but not tearing or gluing”. A classic illustration of topology is provided by the continuous deformation of a mug into a donut. Algebraic topology derives algebraic objects (typically groups) from topological spaces to help determine when two spaces are similar. It also allows us to compute quantities such as the number of pieces the space has, and the number and type of “holes” [12]. The latter is typically what we are interested in in this article. More precisely, we want to compute Betti numbers. These numbers are used to distinguish topological spaces based on the connectivity of n-dimensional simplicial complexes. A simplicial complex is a set composed of points (0-simplices), segments (1-simplices), triangles (2-simplices), tetrahedrons (3-simplices) and their n-dimensional counterparts (n-simplices). It is a way of representing topological spaces that is particularly well suited to calculations. Technically, Betti numbers are the rank of the homology groups associated with a simplicial complex. In this paper, we consider a very simple topological space, namely the OLC graph. To represent this topological space, it is sufficient to examine simplicial complexes of dimension 2 (i.e., up to 2-simplices). Therefore, there are only two Betti numbers of interest, $\beta_0$ and $\beta_1$, which correspond respectively to the number of connected components of the graph and the number of one-dimensional holes in the graph (i.e., loops). We use the python

---

**Figure 2** This figure shows how the hash table is used to determine which reads overlap. Panel a) mapping stage: A read (here the x<sup>b</sup> read) is divided into consecutive, non-overlapping k-mers of size k (here k=4). Each 4-mer, m, is transformed into a hash value (as explained in the text). This hash value indexes a row of the hash table (HASH) that contains a list of all other reads in which the current 4-mer has been found, together with its position in these reads. This generates for other reads, here the y<sup>b</sup> read is shown, a list of quadruplets [read number (x), index of the k-mer in x, position in y at which this k-mer maps, sense]. In the figure, the 1st k-mer of read x maps read y at position 115 in the direct sense (0), the 2nd k-mer of x maps read y at position 119, up to the 27th k-mer of x that maps read y at position 219. Panel b) alignment stage: After all reads have been processed in this way, one analyzes the resulting list for each read to find large consistent paths. For a given read, the list contains the matching positions on that read of all 4-mers of other reads, in both senses. If the reads have been corrected previously, a consistent path must involve consecutive 4-mers of the same read that maps the query read at positions distant of 4, with the same sense. The figure shows that 27 4-mer of read x maps consecutively read y at positions distant of 4 nucleotides, starting at position 115 of read y (schematically shown in red). Thus, read x overlaps read y by 27*4=r nucleotides (r is the number of nucleotides at the 3′ end of read y that are too few to constitute a k-mer: r<k). Read x overlaps read y at the 3′ end in the direct sense (0). The program checks that results are consistent for the symmetrical pairs (x, y) and (y, x), i.e., that when read y is processed it does overlap read x by the same number of nucleotides, in the same sense, but at the 5′ end of read x. This procedure can easily accommodate occasional errors in reads. However, if the raw, uncorrected reads are used, the procedure must be modified. In this case, fewer k-mers of read x maps read y due to the sequence errors in both reads (shown in green). One must find the longest path in which i) the k-mers are in ascending order, ii) they map to the same position, and iii) the distance between their mapping positions is within a certain tolerance. This tolerance is taken as being 3 times the standard deviation, $\sigma$, of the lengths. For instance, let us consider the 5th k-mer and the 8th k-mer (k=4) of read x (in green). Without sequencing errors, they should map positions distant of 16 nucleotides in read y. One must thus have: $|\Delta pos|\leq16\leq3\sigma$ where $\Delta pos$ is the actual difference of mapping positions. This standard deviation is computed from the known error rates (probability of a mismatch, of a deletion, of an insertion) of the 3rd generation sequencing technology used. This longest path is efficiently computed recursively.
interface of the Gudhi library (version 2.3.0) [13] to compute the Betti numbers for our OLC graphs.

From my own experience, I am well aware that algebraic topology can be very disconcerting for newcomers to this field. What the reader who is not interested in the technicalities should remember is the following. For a graph, it is possible to compute two topological numbers (the so-called Betti numbers, $\beta_0$ and $\beta_1$). We are only interested in $\beta_1$ that provides the number of loops in the graph.

Results and Discussion

Although I have described in detail an algorithm to assemble genomes, I do not want to stress too much this part of the work in the discussion. Indeed, to be comprehensive, it would require a thorough comparison with other, state-of-the-art programs that perform the same task. This work will be presented elsewhere. In what follows, I will instead insist on the problems caused by repetitions in genomes with regard to their assembly and how topological analysis based on Betti numbers can help to diagnose the situation.

**Genome assembly performance**

Table 1 presents results of the program’s running times for different “mini-genomes” of increasing size. These mini-genomes are obtained by randomly extracting a piece of DNA of a given length from the human chromosome IV. For each mini-genome size, a number of reads are randomly generated from this piece of DNA according to a normal distribution with mean 300 nucleotides (nt) and standard deviation 25 nt such that the coverage of the mini-genome is 15. The mini-genomes can be linear or circular, hence the generation of reads takes this fact into account. As described above, assembling a genome consists of two stages. The first one builds the graph in 3 steps (hashing, mapping, aligning) and the second assemble the genome by first searching for a pseudo-Hamiltonian path (or cycle) in the graph then by gradually assembling the reads along this path. Table 1 shows the times, in milliseconds, for these different steps.

![Image](image.png)

**Figure 3** Panel a shows a simplistic representation of a genome. Let us assume that regions colored in red are highly repetitive sequences, for instance GC islands. The genome is fragmented into 17 reads. Panel b shows the resulting graph. Vertices corresponding to single-color reads are represented as circles and colored accordingly. The bicolor reads are represented as triangles and are colored with the “complementary color” to red. There is an edge between two vertices if the corresponding reads overlap. Panel c shows a “high-level” view of the graph in panel b that disregards all the petty details. The graph consists of a green stem, a red “knot”, two loops in dark and light blue and a final stem in purple.

| Genome size | 10 K | 20 K | 40 K | 80 K | 160 K | 320 K |
|-------------|------|------|------|------|-------|-------|
| # nucleotides | 150,178 | 300,216 | 600,101 | 1,200,288 | 2,400,222 | 4,800,054 |
| # reads | 499 | 998 | 2001 | 4006 | 8012 | 16022 |
| # vertices | 298 | 594 | 1194 | 2387 | 4791 | 9586 |
| T graph creation | 159 | 254 | 451 | 941 | 1938 | 5003 |
| T hashing | 92 | 114 | 162 | 252 | 459 | 884 |
| T mapping | 11 | 25 | 60 | 158 | 439 | 1701 |
| T aligning | 56 | 115 | 228 | 531 | 1040 | 2418 |
| T genome assembly | 1.5 | 3.0 | 7.5 | 25.5 | 68.3 | 229.2 |
| T depth 1st search | 0.1 | 0.2 | 0.4 | 1.1 | 3.5 | 7.3 |
| T read connection | 0.9 | 2.0 | 5.3 | 20.0 | 53.6 | 201.4 |

The first row is the size of the mini-genome in kbp (thousands of nucleotides). The 2nd row is the number of nucleotides in the reads, the 3rd row is the number of reads, the 4th row is the number of vertices in the graph. Other rows give the time in milliseconds for the different steps of the program. Note that the sum of the times spent in the depth 1st search and read connection steps is less than the time spent in the genome assembly stage for a reason that is explained in the text.
another read and therefore omitted from the set of vertices. This percentage could be even lower for reads from 3rd generation technologies which are characterized by a distribution of lengths with a heavy tail towards long lengths.

In Table 1, the sum of the times spent in the “depth first search” and “read connection” steps is less than the time spent in the genome assembly stage. The reason for this is that one step has been omitted in Table 1. This step consists in sorting the adjacency lists describing the overlaps between reads according to the number of nucleotides in the overlaps (the weights of the edges). Thus, the “depth first search” algorithm favors edges with high weights, i.e., it preferentially follows the connections between the most overlapping reads. This step is an “unnecessary refinement” for the sake of genome assembly, and has therefore been left out in Table 1.

Unexpectedly, the last “read connection” step, in which reads along the path are gradually linked with the growing contig, does not appear to be linear in terms of the number of vertices (the F-statistics for the best regression line is 76). A quick inspection in R points to a running time that is approximately $O(V^{3/2})$ (figure not shown). This is confirmed by the “nls” function of R (non-linear least square) that provides a value of 1.58 for the power of $V$. One possible explanation is that an assembly validation test has been added to the program, i.e., the program systematically checks the consistency of the alignment when a new read is joined to the growing “contig” (a process that is linear in the read length). However, it is not immediately clear why this would lead to a running time that is $O(V^{3/2})$ for this step. This is a bit annoying because otherwise the complete genome assembly algorithm would be $O(N)$, $N$ being the number of reads (remember that $V = \alpha N$ with $\alpha < 1$). This requires further investigation of the code to try identifying the component that is responsible for the $O(V^{3/2})$ running time of this last step.

Topological analysis of the OLC graph

The expected Betti numbers for the OLC graph of a linear genome are $\beta_0 = 1$ and $\beta_1 = 0$, i.e., the graph is connected and there is no loop. For the graph of a circular genome, the expected Betti numbers are $\beta_0 = 1$ and $\beta_1 = 1$, the graph is connected and there is one loop. Any other result points to a potential problem with the OLC graph that needs to be investigated.

We systematically computed the Betti numbers for the OLC graphs resulting from the executions of the program to produce Table 1, both for linear and circular mini-genomes. The calculation of Betti numbers, as implemented in the GUDHI library, is very fast. For example, it takes only 14 seconds for processing the 320 K mini-genome graphs (comprising about 10,000 vertices). We always obtained the expected Betti numbers, except for the 320 K circular mini-genome. For this mini-genome, $\beta_1$ was 18. Indeed, the safety procedure of the “read connect” step (the systematic test of the validity of the alignment of each new read with the contig under construction mentioned above) revealed that the resulting genomic assembly was not correct. As alluded to above, the problems occurring during genome assembly are due to “unresolved” repetitions. Unresolved repetitions means that the read lengths do not allow them to completely encompass these regions. Figure 3 sketches the issue occurring with genomic repetitions. It shows an extremely simplified representation of a genome in which the regions colored in red are assumed to be repetitive regions, for instance, GC islands. This genome is fragmented into 17 reads as shown in Figure 3a. Figure 3b displays the corresponding OLC graph. Even such a tiny graph is not so easy to interpret visu-
ally. As shown in Figure 3c, the general features of this graph consist of a green stem, a red “knot” containing the repetitive regions, a light blue loop, a dark blue loop and a purple stem. The Betti numbers for this graph are $\beta_0 = 1$ and $\beta_1 = 2$, as expected. The “knot” region prevents the depth first search algorithm to find a pseudo-Hamiltonian path, since the path must go more than once through the same vertices. It is important to gain an overview of the topology of the graph as shown in Figure 3c. This allows a better control of the assembly procedure. For instance, here we know that there are 2 possibilities for the genome assembly: i) the green stem, a repeated region, the dark blue loop, a repeated region, the light blue loop, a repeated region and the purple stem or ii) the same configuration, except for the order of the light and dark blue loops that is inverted. Note that one cannot say anything about the actual length of the different repeated regions in the genome assembly. The two alternative assemblies can be distinguished using experimental techniques such as optical mapping which provides information over long genomic distances. The $\beta_1$ Betti number provides the number of loops in the OLC graph. Unfortunately, GUDHI does not allow us to know which vertices are involved in the loops (the homology basis, in technical terms), since GUDHI, for efficiency reasons, uses the co-homology in which bases are not very illuminating. This piece of information is necessary to fully exploit the topological knowledge acquired on the OLC graph in order to help us to assemble the genome in an optimal way.

If the Betti numbers of the OLC graph are those expected for a linear or circular genome, one can be confident in the quality of the genome assembly.

Conclusion

Third generation sequencing technologies have revived genome assembly algorithms based on OLC graphs. In this paper, which is a preliminary report, I presented an efficient algorithm to assemble the reads generated by these technologies that is based on an OLC graph. Clearly, more tests of this algorithm and comparisons with existing methods must be done before releasing the program to the life science community. The second contribution of this paper is to show how one can harness algebraic topology concepts to determine, beforehand, whether the assembly procedure will be straightforward or whether one will need to carefully investigate the results. Here also, further work is needed to determine which vertices/reads are involved in the different elements of the OLC graph (the stems, loops and knot of Fig. 3c) when unresolved repeated regions prevent the algorithm to find a pseudo-Hamiltonian path or cycle. However, with read lengths up to 100 kbp (thousands of nucleotides), and still increasing, provided by 3rd generation sequencing technologies, unresolved repeated regions are likely to become few and far between.

Acknowledgement

The author would like to thank F. Chazal for a helpful discussion on the use of the GUDHI library. The author would also like to warmly thank Professor Go to whom this special issue is dedicated on the occasion of his 80th birthday. The postdoctoral fellowship I did in his laboratory at Kyoto University 30 years ago (as time goes by!) remains an excellent memory.

Conflict of interest

None

Author Contribution

JFG conceived and designed the work, wrote the computer programs, ran them, analyzed and interpreted the results and wrote the article. The author (obviously) approves the manuscript.

References

[1] Huang, X. & Madan, A. CAP3: a DNA sequence assembly program. Genome Res. 9, 868–877 (1999).
[2] Sutton, G. G., White, O., Adams, M. D. & Kerlavage, A. R. TIGR Assembler: a new tool for assembling large shotgun sequencing projects. Genome Sci. Technol. 1, 9–19 (1995).
[3] Kececioglu, J. D. & Myers, E. W. Combinatorial algorithms for DNA sequence assembly. Algorithmica 13, 7–51 (1995).
[4] Pevzner, P. A., Tang, H. & Waterman, M. S. An Eulerian path approach to DNA fragment assembly. Proc. Natl. Acad. Sci. USA 98, 9748–9753 (2001).
[5] Zerbino, D. R. & Birney, E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. 18, 821–829 (2008).
[6] Simpson, J. T. & Durbin, R. Efficient de novo assembly of large genomes using compressed data structures. Genome Res. 22, 549–556 (2012).
[7] Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J. & Birol, I. ABySS: a parallel assembler for short read sequence data. Genome Res. 19, 1117–1123 (2009).
[8] Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yang, J. K., et al. SOAPdenovo2: an empirically improved memory efficient short-read de novo assembler. GigaScience 1, 18 (2012).
[9] Lander, E. S. & Waterman, M. S. Genomic mapping by fingerprinting random clones: a mathematical analysis. Genomics 2, 231–239 (1998).
[10] Mohamadi, H., Chu, J., Vandervalk, B. P. & Birol, I. nthHash: recursive nucleotide hashing. Bioinformatics 32, 3492–3494 (2016).
[11] Zomorodian, A. Topological data analysis. In Proceedings of symposia in applied Mathematics. vol. 70 (The D. E. Shaw Group, New York, NY, Editor, 2012)
[12] Robins, V. Algebraic topology. In Mathematical tools for physicists, 2nd edition. (Grinfeld M, ed.). Chapter 5. (Wiley, New-York, 2014)
[13] Rouveure, V. Cython interface. In GUDHI, user and reference manual. (GUDHI Editorial Board, 2016), https://gudhi.gforge.inria.fr/python/latest/