# Assembly improvements by read mapping and phasing

Antoine Limasset

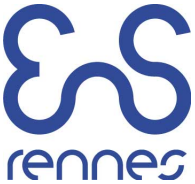

Master research Internship


Master thesis Report

# Assembly improvements by read mapping and phasing

*Author:*
Antoine Limasset

*Supervisors:*
Dominique Lavenier
Pierre Peterlongo
Genscale




**Abstract**

Massively parallel DNA sequencing technologies are revolutionizing genomics research. Billions of short reads can be generated at low costs (Illumina launched in January 2014 its HiSeq X Ten Sequencer which delivers the first $1000 human genome at 30x coverage [1]). This storm of genomic data calls for the development of scalable and accurate techniques. The difficulty of de novo assembly and limitation of sequencing techniques have led to dozens of assembly algorithms, none of which is perfect. Here we present a novel approach combining the sequence alignment and the assembly fields: a read mapping algorithm working on a De Bruijn graph instead of sequences named BGREAT (de Bruijn Graph REad mApping Tool). The main focus of this work was the scalability of the tools in both terms of memory usage and throughput. To this end we chose to study state-of-the-art low memory and efficient algorithms. This report will give a vision of the assembly and alignment fields, present the algorithm and it's actual implementation and finally study the results obtained and their implications.

# Contents

# 1 Introduction

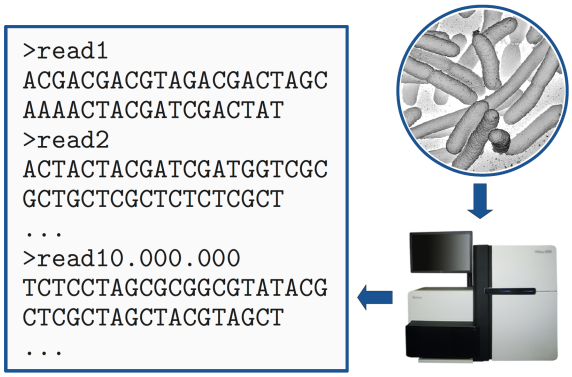



Figure 1: Sequence extract sequence of DNA called "reads" from cells

Sequencing remains at the core of genomics. The process is used to determine the sequence of nucleotide bases of an organism's genome. It has many application in fields such as diagnostic, biotechnology, forensic biology, agronomy, environment and biological systematics and has become indispensable for basic biological research. Sequencers provide millions of short sequences called reads randomly extracted from the genome (see figure 1), this information is not directly usable and is usually highly redundant in term of coverage (much more bases in the reads that in the genome) due to the needs of statistical methods. In 2008 with the next generation sequencing technologies (NGS), massively parallel sequencing platforms such as Illumina, or Solid appeared and provided unprecedented increase in DNA sequencing throughput. The Illumina sequencing technology can produce up to 3 billion reads with a read length between 50 and 300 base pairs [2] [3]. The amount of data produced is huge and is still increasing, the throughput of sequencers are growing while the costs continue to drop (see figure 2 [4]). A major milestone, the "$1000 genome" has been reached in January 2014. Illumina launched its HiSeq X Ten Sequencer which delivers the first $1000 genome at 30x coverage (including reagent costs, instruments depreciation, and sample preparation) in a day. Due to the scale of the data used, most algorithms dealing with genomic data are matter of high performance computing.

Two major technical points make difficult the "assembly", the determination of the genome of an organism. First, as it was introduced, sequencers do not give the complete genomic sequence but short extracts called reads, usually of around hundred base pairs length. Second, the reads are not perfect extracts of the original sequence, different type of errors can appear in the reads with different probabilities. The accuracy and the type of error depend of the technology used. Accuracy can go from 85% to 99.9% but is usually around 98-99%. To overcome those problems, overlap-graph based algorithms have been proposed to find a set of sequence that approximate the original sequenced material, we call these sequences "contigs". Ideally just one sequence is found that match the genome, but it never occurs in practice. Because of its combinatorial nature, the problem (which is NP-complete [5]) can not be solved perfectly, and actual algorithms are based on heuristics that treat the data in a decent time but allowing mistakes. The algorithmic challenge is double, get the best possible assembly from the available reads and do it in the most efficient way in both term of computational time and memory usage.

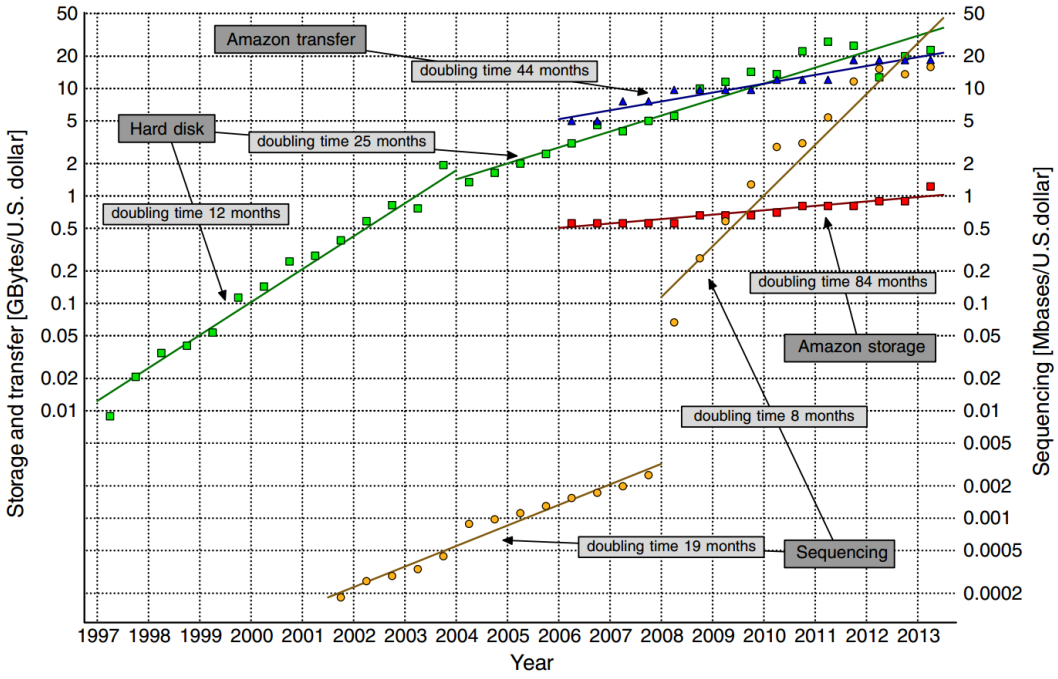



Figure 2: Trends

Another basic treatment of genomic data is to detect similarities between sequences by alignment. Region of similarity may involve functional, structural, or evolutionary relationships between the sequences. Read alignment is usually the first step of genomic data analysis and play a critical role in medical and population genetics. Once again the challenge is the scale of the data to index and to align. But if the throughput is a key factor in those algorithms, they also need to be sensitive and to allow mismatches and other error patterns.

Assembly and alignment are two distinct problems, both central in bioinformatic and well studied. The main idea at the base of this internship was to use both fields and propose a novel use of alignment techniques to improve the assembly quality. Assemblers decompose reads in shorter sequences called "kmers" and put them in a overlap graph. The information of the reads is not fully used since they forget how kmers were originally connected. A proposed solution to recover this information was to align the read on the paths of the graph used for the assembly. In brief, this algorithm will allow to recover the information contained in reads that are lost due to the data structure used, but will not be restricted to this application.

The first considered application was the "correction" of the assembly graph, where the read information is used to improve the quality of the assembly. A trivial analysis would be to remove paths in the graph that are not covered by an actual sequence (by a read). By doing this we will simplify the graph and will avoid doing some mistakes (using paths that are very unlikely to be actual sequence from the genome). More complex analysis could be performed with this kind of information and further improvements could by done using this algorithm. A second possible application is to "phase", ie to separate the genomic sequences from the different parents of the individual (the chromosomes) using the fact that a read is an extract of one of the chromosomes.

A third, unplanned application have been envisaged, the compression of reads data.

In order to introduce the necessary notions, we will focus on assembly and alignment algorithms and present the phasing problem, to see what are the main techniques used and what can be improved. Then we will present the proposed solution, the BGREAT algorithm, and discuss its actual implementation. And finally we will present the studied applications and the results obtained.

# 2 State of the art

## 2.1 Assembly

Sequencing remains one of the main challenging process of genomic and determining a complete genome sequence is still an important application. Despite the success in determining the human genome [6] DNA assembly is still a challenging process, even with high computational resources it is difficult to get good assemblies. In a perfect world an assembly would be one sequence matching the genome, but in practice the assembly is smaller and fragmented (see figure 3). We will present the main methods used and explain why those limitations appear.

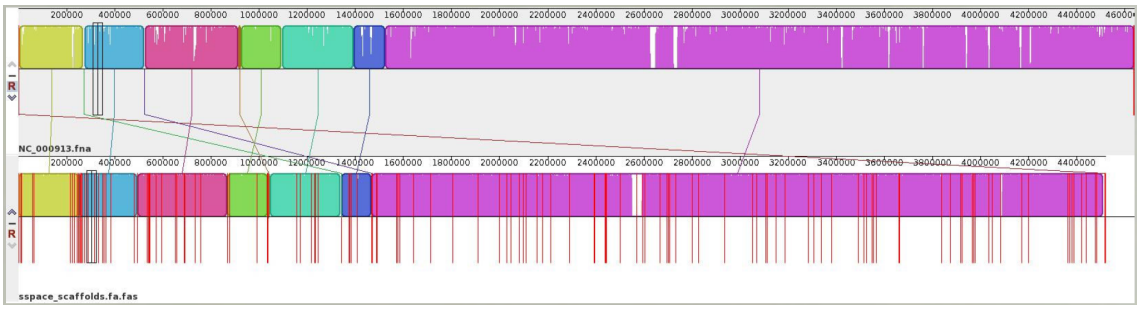



Figure 3: Example of a reference genome (top), and an assembly aligned to it (bottom). Different sequences (contigs) separated by red lines)

The main data structure used in assembly algorithms is a overlap-based string graph. The most classical and most used use a specific string graph called the De Bruijn graph. The first paper using De Bruijn was "An Eulerian path approach to DNA fragment assembly" [7].

**Definition 1.** *Given a set of strings* $S = \{r_1, r_2, ..., r_n\}$ *and a minimum overlap threshold value* $k$*, the De Bruijn graph for* $S$ *is a directed graph* $BGk = (V, E)$ *where:*
$V = \{length(d) = k - 1 \mid \exists\ i$ *such that* $d$ *is a substring of* $r_i \in S\}$
$E = \{(d_i, d_j)$ *: if the prefix of length* $k - 1$ *of* $d_i$ *is a suffix of* $d_j$ $\}$

A minimal example is presented in the figure 4. The set of reads

$$\{ATTCG, GTTCGA\}$$

gives the set of 4-mers

$$\{ATTC, GTTC, TTCG, TCGA\}.$$

The main idea behind using De Bruijn graph is that paths will allow the construction of larger sequences (see figure 5). Clearly, every read $r_i \in S$ is translated into a path composed of $(|r_i|k+1)$ nodes. And because of the definition of the De Bruijn graph, overlapping sequences will form a path in the graph.

In theory, the assembly is the path of minimal length that traverse each nodes. In practice, because of ambiguities, a simple path is impossible to find. We can see in the example of the figure 6 that it can exists multiple paths of minimal length that traverse each nodes (ACTGACTGAATGCC that start with the "up" loop or ACTGAATGAGTCCC that start with the other loop). Assemblers

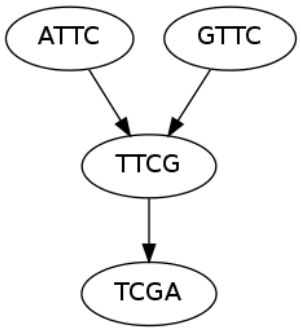



Figure 4: An example of De Bruijn graph with $k = 4$ and the set of reads $S = \{ATTCG, GTTCGA\}$

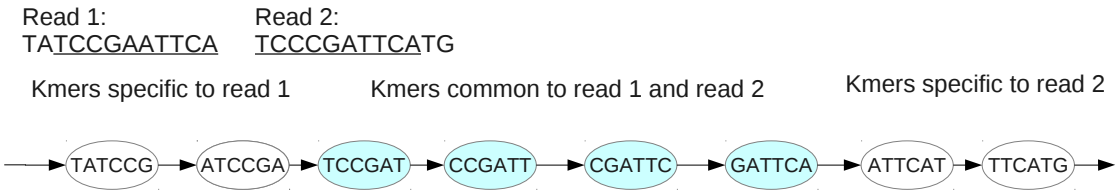



Figure 5: How a De Bruijn graph allow to create longer sequences

try to return a set of unambiguous sub-paths of maximal length. A trivial way to get such set is to compute and return the simple paths of the graph (the simple paths of the figure 6 are $\{ACTG, TGA, GAGTC, GAATG, GACC\}$). But it is often not enough because of the number of returned sequences. Sequencing error complicates the De Bruijn graph, but some errors are easily recognized by their structure in the graph. For example, errors at the end of a read usually form dead-end tips in the graph. Errors in the middle of a read create a pattern called "bubble" where alternate paths appear and terminate at the same node. De Bruijn graph assemblers search for these patterns in the graph to "clean" the graph by removing the erroneous nodes. The sequences returned after this phase of graph simplification are called contigs.

The canonical stream of an assembler is the following (Velvet [8], ABySS [9], SOAPdenovo [10], SGA [11]):

1. Read correction (corrected reads)
2. Kmer counting (kmers)
3. Simple paths computation (unitigs)
4. Graph simplification (contigs)

The first phase is optional and is intended to cope with the sequencing errors, it slightly modify the reads to "correct" them. Correctors can remove many simple sequencing errors but can not guarantee the correctness of he reads and can possibly make mistake and create new errors.

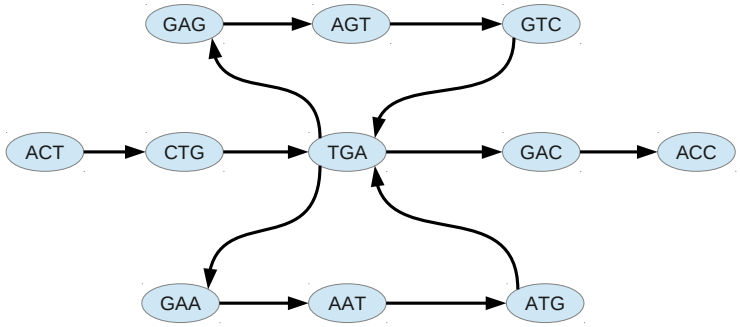



Figure 6: Example of ambiguity

The second phase extract the kmers (word of length $k$) from the reads. This phase is called kmer counting because it only keeps kmers that appear a certain number of time in the reads. Knowing the coverage of the genome offered by the set of read, the user will decide the threshold to apply on the occurrence of kmers. This principle of threshold is based on two assumptions:

1. kmers way too rare are likely sequencing errors and should not be considered
2. kmers present often are likely to represent actual sequences in the genome

The third phase consist in compacting the De Bruijn graph by computing the simple paths.

**Definition 2.** *Given a De Bruijn graph G=(V, E)*
*A simple path is a subset of V* $v_1, v_2, ..., v_n$ *such that*
$\forall i \in [1, n-1], (v_i, v_i + 1) \in E$ *implies*
$\forall j$ *such that* $(v_j, v_i + 1) \in E$ *implies that* $j = i$ *and*
$\forall j$ *such that* $(v_i, v_j) \in E$ *implies that* $j = i + 1$

An equivalent version of the De Bruijn graph is currently used, called the compacted De Bruijn graph, where nodes forming a simple path are merged. In this representation, the length of a node is arbitrary (superior to $k$) but the edges still represent $k-1$ overlaps (see figures 7). The simple paths of maximal length are called unitigs.

The last phase consists in further simplification of the graph in order to get fewer and longer nodes. Since the graph is created from actual genomic data, this implies that the graph obtained is not just a random graph, some general topology and patterns are expected. Some patterns can occurs that do not make sense in the context, for example a small island connected to nothing do not make sense because we expect very big sequences (for example chromosomes). An other example is the red pattern in the figure 8, typical of sequencing errors and can be removed from the graph.

Some biological events also create known patterns in the graph. For example a SNP (single nucleotide polymorphism) is a very recurrent phenomenon that create a "bubble" in the graph.

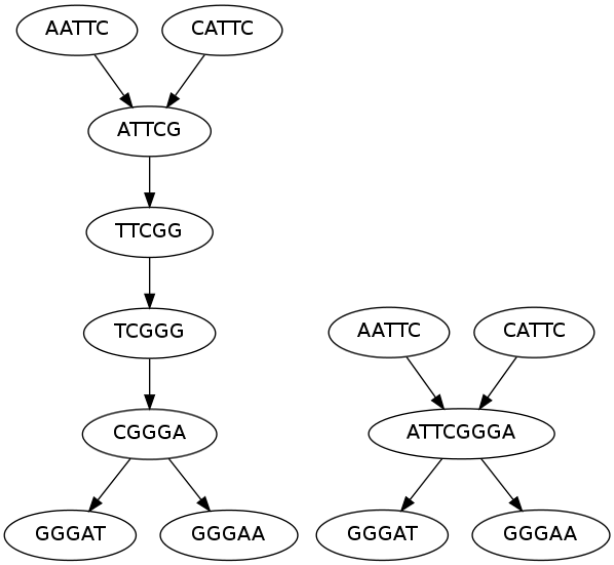



Figure 7: De Bruijn graph example with k=5 and set of nodes S=$\{AATC, CATTC, ATTCG, TTCGG, TCGGG, CGGGA, GGGAT, GGGAA\}$ and its compacted version

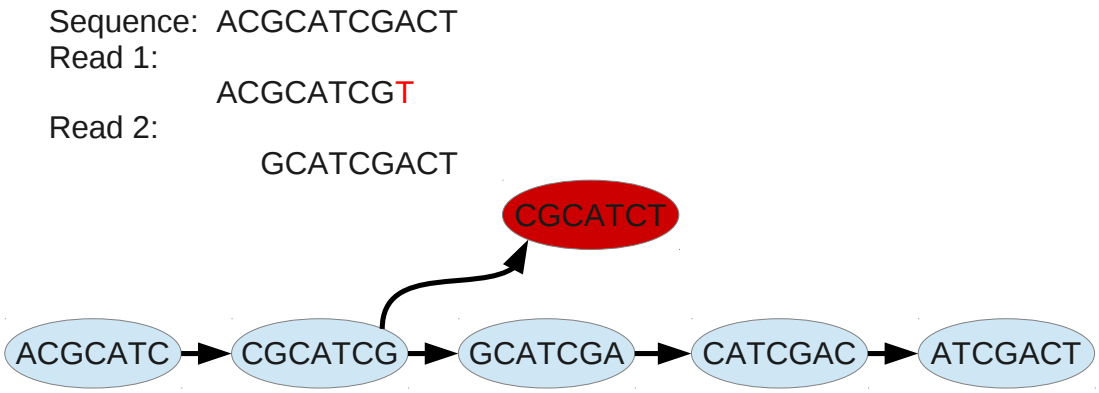



Figure 8: How a sequencing error impact the De Bruijn graph. The first read contain a sequencing error and create a dead-end in the graph

An example of bubble is shown in blue in figure 9. We can see that the pattern is almost linear, multiple paths start from a node and all of them rejoin the same node at one point. To simplify the graph, the assembler will choose a path in the bubble (ie a set of nodes that form a path from the beginning to the ending of the bubble) and discard the other nodes. By doing so, we make arbitrary choice and then can make mistakes.

As we saw, the main structure used to do the assembly is a De Bruijn graph. The main problem of this choice is that the De Bruijn graph is an extremely heavy structure and the memory usage of standard assemblers are huge (see figure 10 from [12]). The need of most assembly run will overcome the regular desktop computer capacity. Recently, some focus was made on the memory efficiency of those algorithms, trying to use efficient data structures to store the graph such as bloom filter [13] or FM-index [14].

### 2.2 Read Mapping

The problem of sequence alignment is finding regions of similarity between sequences. From two sequences and a method to quantify your notion of similarity, the problem is to determine the correspondences between substrings in the sequences such that the similarity score is maximized. The method often use a scoring function and can differ a lot with the usage. It is supposed to allow some sequence variations as:

1. Substitutions ($ACGA \rightarrow AGGA$)
2. Insertions ($ACGA \rightarrow ACCGGAGA$)
3. Deletions ($ACGGAGA \rightarrow AGA$)

The insertions and deletions will results in "gaps" in the alignment. The following example show how to handle insertions and deletions allowing to add blank characters.

1. Substitution ($\frac{ACGA}{AGGA}$)
2. Insertion ($\frac{AC----GA}{ACCGGAGA}$)
3. Deletion ($\frac{ACGGAGA}{A----GA}$)

A simple way used to score an alignment is to use a gap penalty function, $w(k)$ that indicates the cost of a gap of length $k$ and a substitution matrix $s(a, b)$ that indicates the score for aligning character $a$ with character $b$. Then the score of the alignment

$$\frac{ACG--T}{TCGGGT}$$

would be

$$s(A,T) + s(C,C), s(G,G) + w(2) + s(T,T).$$

We can note that finding the best possible alignment is a combinatorial problem, there is $\binom{2n}{n}$possible (global) alignments for 2 sequences of length $n$. It mean that two sequences of length 100 have approximately $10^7$ possible alignments. It is not possible to be exhaustive with this kind of complexity, it implies that most aligner will use heuristics to highly reduce the number

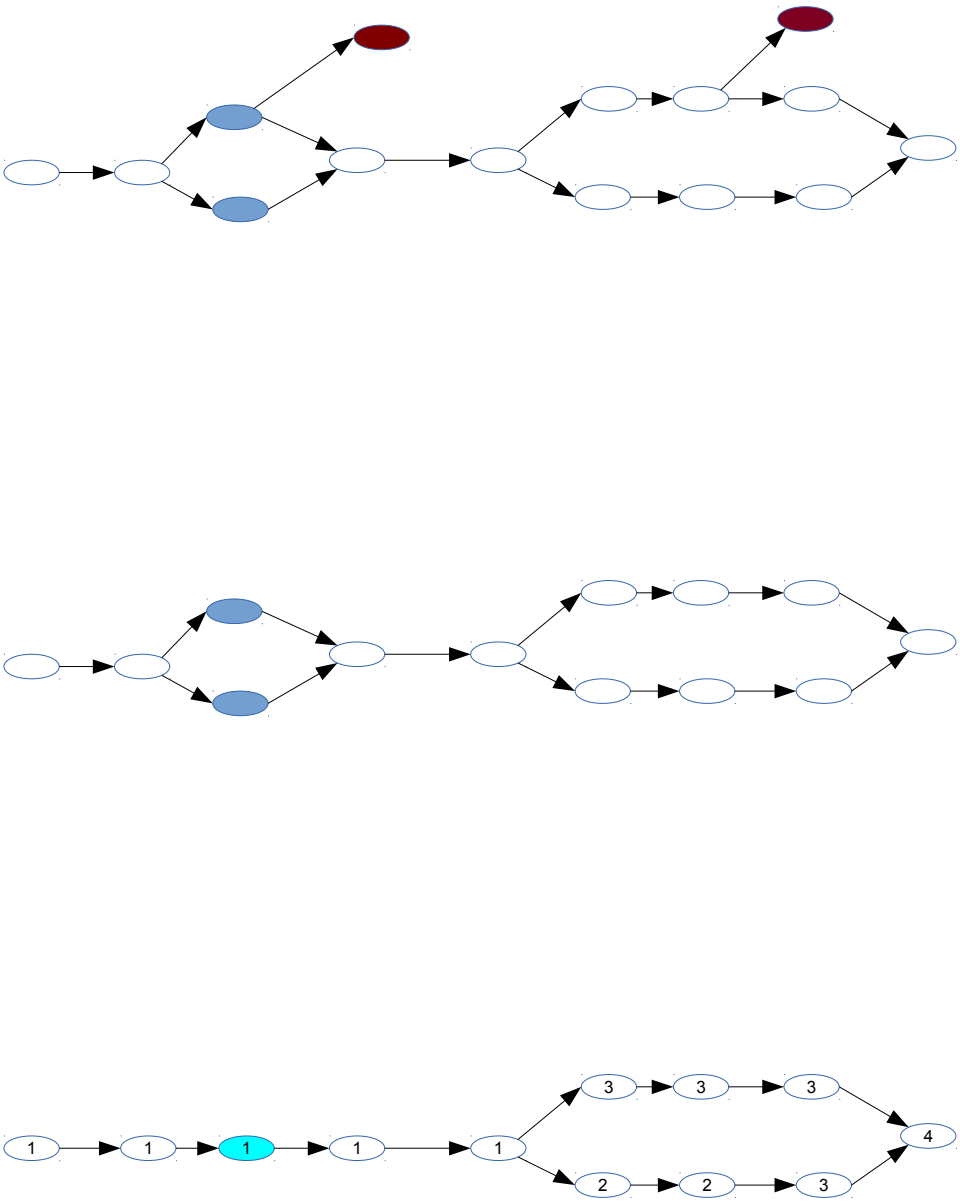



Figure 9: Pattern that is likely to be sequencing error in red, small bubble in blue and the four contigs obtained. The numbers represent the different contigs returned

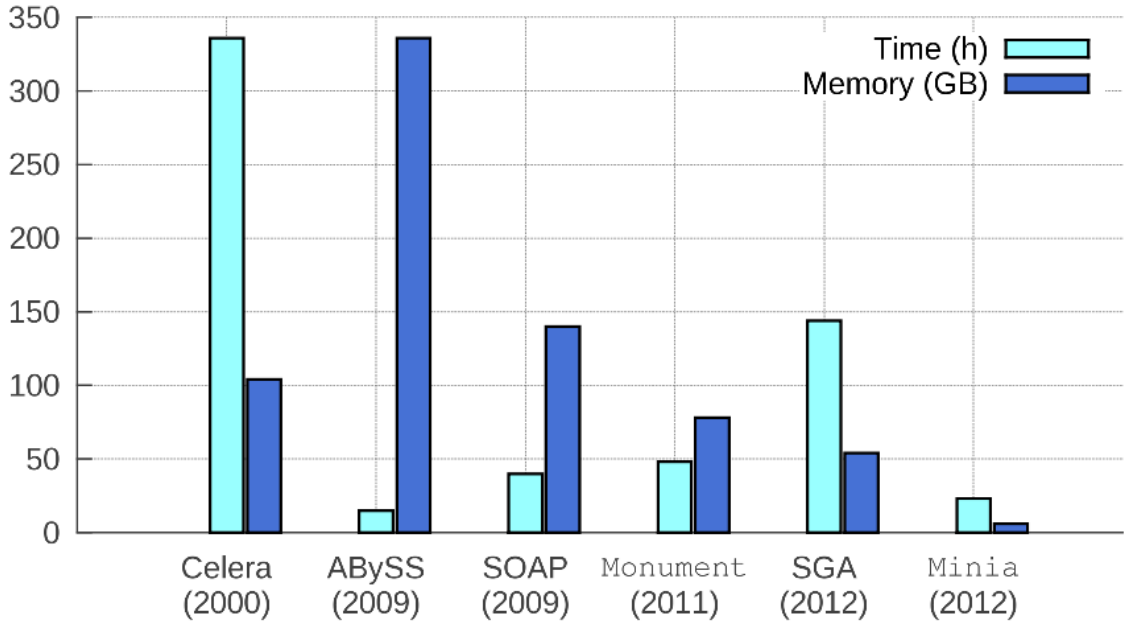



Figure 10: Comparison of different assembler needs for a human genome

of possibilities. By doing so, algorithms are able to handle large data sets but can miss good alignments.

We can also note that, depending of the application, we are not interested in the same way to align. Three different ways are widely used:

1. Global alignment: find best match of both sequences in their entirety
2. Local alignment: find best subsequences match
3. Semi-global alignment: find best match without penalizing gaps on the ends of the alignment

More commonly, we will want a local alignment, it can used for comparing protein or DNA sequences that share a common motif but differ elsewhere.

We defined the alignment between two sequences, but we are in fact often interested in multi-alignment to discover common motifs in a set of sequences. It allows to compare a query sequence with a library or a database of sequences and and find reference sequences that are close to the query sequence and the position of the query sequence on these. As it was introduced this is one of the most important problems in bio-informatics and alignment tools are among the most used tools (The original paper of Altschul presenting BLAST [15] was the most highly cited paper published in the 1990s).

We will consider three milestones in this field :

- Smith-Waterman
- BLAST

- Read mapper (FM-index)

Smith-Waterman algorithm [16] is the first milestone in this field, solving the local alignment problem between two short sequences with a time complexity of $O(n * m)$ for two sequences of length $n$ and $m$. But once again the main problem when dealing with genomic data is the scale. Sequence databases can now reach orders of magnitude above billions of base pairs in millions of sequences.

A second milestone in the alignment field is the canonical tool BLAST [15] [17]. BLAST (Basic Local Alignment Search Tool) is the first of a generation of fast algorithms allowing efficient sequence alignment. Before it, doing databases research was very time consuming because a full alignment procedure (Smith-Waterman) was practiced. The idea of BLAST is to find similar sequences by locating short matches between two sequences before to comparing them entirely. It can be summarized in three steps:

1. List words that are substrings of length $w$ of the query sequence
2. For each word listed, identify all matches with the database sequences
3. For each word match "hit", try perform larger alignments in both directions, stop when the score decrease too much from his highest value.

But if BLAST is faster, it can not guarantee that the optimal alignment is found. The trade-off between sensibility and speed can be parametrized using the notion of score, the algorithm search for pairs with an alignment score superior to a parameter. But even with good parameters we can can miss good alignment since BLAST do not explore all possibilities but the more promising.

The third generation of alignment algorithms take distance from BLAST-like algorithms and are fully designed for short read sequence alignment. This generation include widely used tools as BOWTIE [18] [19] SOAP [20] [21] [22] or BWA [23]. Read mapping focus on align short sequences (reads) on large sequences with a very high similarity when alignment can align larger sequences with eventually low similarity. They are no longer standard alignment tools, and are inefficient to map long sequences. This generation is also characterized by the usage of the FM-index [24] [25] and other Burrow-Wheeler techniques that allow very fast and very memory efficient mapping. The FM-index is a compressed self-index, which means that it compresses the data and indexes it at the same time. In this context it allow to index the reference sequences in the main memory, when it would be often impossible without compression due to the size of the references.

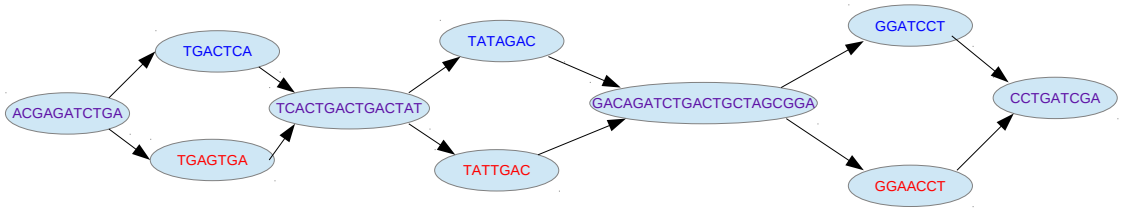
## 2.3 Phasing

**Definition 3.** *We define here the genotype of an organism as its whole genomic sequence.*

*For diploid organism (who have two copies of each chromosome), the genotype would be the sequences of each chromosome of each pair.*

A diploid organism have in its genotype two versions of each chromosome that are slightly different. The problem that appear when we try to assemble the genome of a diploid (or more generally a polyploid) organism, is that the assemblers try to recover a unique sequence from the reads. The common sequences will be merged in the De Bruijn graph and the differences will form bubbles. In figure 11 we can see the two sequences of the diploid individual and a possible assembly.

ACGAGATCTGACTCACTGACTGACTATAGACAGATCTGACTGCTAGCGGATCCTGATCGA




ACGAGATCTGA***G***TCACTGACTGACTAT***T***GACAGATCTGACTGCTAGCGGA***A***CCTGATCGA

Possible output contig :
ACGAGATCTGAGTCACTGACTGACTATAGACAGATCTGACTGCTAGCGGATCCTGATCGA

Figure 11: Issue of classic assembly on diploid genome. The common parts of the blue and red sequences are merged in the De Bruijn graph in purple, but minor differences create bubbles and the assembly resolve these bubble by choosing a path. As a result, the output contig do not correspond to any original sequence

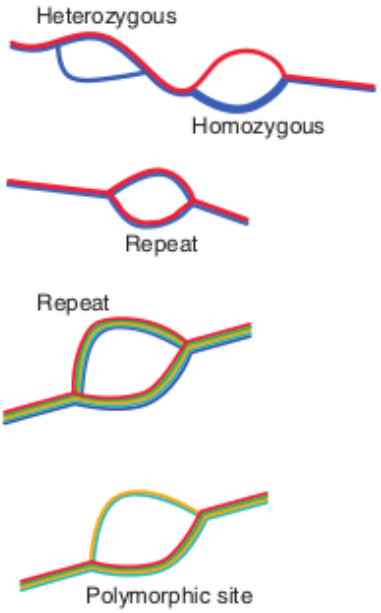



Figure 12: Colored De Bruijn graph

In fact assemblers will give as a result a sequence that is neither a version nor the other but a mix of the two. The problem of phasing is to be able to differentiate the different versions of each sequence. In our case, we are interested in identifying alleles that are co-located on the same chromosome by phasing. Phasing can be done with biological method or can be estimated by computational methods.

Nowadays there is, to our knowledge, no actual phasing tools. Some method talk about "read threading" [26] without give real details or implications. It make it difficult to know if it could achieve phasing or not. We choose to present here an inspiring method called the colored De Bruijn graph [27]. The idea is to use multiple source of reads, from different individuals, and superpose the different De Bruijn graphs obtained, which are close but with some variations. The difference sources are represented with different colors in figure 12. The different sources are used to detect scenario and improve the different assemblies. The goal here is not to phase, since this approach allows the differentiation between individuals or data sets but does not allow the phasing of a single individual, but to show that adding read information in a De Bruijn graph highly improve analysis.

# 3 Methods

## 3.1 Original idea

As we introduced, the main technique used for assembly is to split reads into kmers and to put them in a De Bruijn graph. This technique is efficient and widely used. We can highlight two drawbacks of this technique:

- Not read-ware (the full read information is not used)
- Not polyploid-aware (assembly is aimed to recompose one sequence)

The goal of this internship was to allow to map the reads on the paths of the De Bruijn graph to recover the full read information.

A first approach to allow this was BLASTGRAPH [28], following the same principle as BLAST, indexing seed and finding an anchor in the graph and extending to find alignment. The algorithm was intended to allow to check the presence of a read in a De Bruijn graph. But use BLASTGRAPH on millions reads to enrich the graph with the reads information would be way too costly. The benchmark proposed by the paper for $10^7$ nucleotide show a memory usage of 1GB and 10 second of query time. The objective of the internship was to conceive a tool able to handle with reasonable resources a typical Illumina run ie 100 millions reads, around 10 billions nucleotide. With a linear extrapolation BLASTGRAPH would present a memory usage of 1TB, an indexation time of more than 3 hours and a query time of some CPU years.

Here we propose a novel and scalable approach able to map hundreds of million reads on a huge De Bruijn graph. We implemented the algorithm in C++ in a software called BGREAT (de Bruijn Graph REad mApping Tool).

The first fixed goal of this internship was to be able to map a Illumina run (hundred of millions reads of length around 100) with reasonable memory and time on the De Bruijn graph created from the same reads. A more ambitious goal was to be able to handle an Illumina run on a regular desktop computer and the challenge was to handle a human genome on it. The second planned goal was to improve the assembly using the information obtained. The last, long-term goal was to conceive a phasing tool.

## 3.2 BGREAT Algorithm

**Definition 4.** *Given a Compacted De Bruijn graph $G_k = (V, E)$*
*$O$ is an overlap of $G \iff length(O) = k - 1$ and*

$$\exists n \in V,\ O = n[1, k-1]$$

*or*

$$\exists n \in V,\ O = n[size(n) - (k-1), size(n)]$$

Here we present the proposed solution, the BGREAT algorithm, as follow:

- Objectives
- Intuition

- Data structures
- High level vision
- Pseudo code

The main goal of the BGREAT algorithm is the following: align reads on a compacted De Bruijn graph. The secondary goal is to have have a minimal memory consumption while having a good throughput. The input will be a file containing the nodes of the compacted De Bruijn graph and a file containing the reads to be mapped on it. The output will consist in a file describing how the reads are mapped on the graph (on which nodes in which positions).

First of all, there is two main cases when you want to map a read on the paths of a De Bruijn graph. Since nodes have arbitrary lengths, some nodes will be large enough to have reads that map on them. This case is pretty simple to handle, a standard read mapper is able to map the reads on the large enough sequences. The complex part is when reads map on a path that include multiple nodes, following edges of the graph. The BGREAT algorithm focus on that case since the standard read alignment problem is well known.

To give an intuition, our approach is somewhat close to BLAST. We will index some well-chosen anchors, find those anchors in the reads, and extend the match following the possible paths of the graph. Two main problems appears:

1. Choose good anchors (Indexing much anchors will cost memory and time so we have to choose interesting ones)
2. Knowing the structure of the graph (Having the whole graph structure in memory is extremely memory expensive)

To cope with these two problems, we choose to index the overlaps between the nodes (see definition 4). This idea is the essential concept of the algorithm, we can give some facts to explain why overlaps are very good anchors.

- An overlap is of length $k-1$ (which is quite long)
- An overlap is shared by multiple unitigs (almost always)
- An overlap is shared by at most eight unitigs
- The number of overlap is "reasonable", at most the double of the number of unitig (to give an order of magnitude, a human genome can count 10 millions unitigs)
- The overlaps are easily computable (read the unitigs)

To sum it up, we have long anchors ($k$ is usually between 30 and 60), that are (almost) guaranteed to appear multiple time in the unitigs (but a bounded time), we do not have to compute anything to get them and the number of overlap is reasonable even for a human genome. But overlaps are not only good anchors, in fact the knowledge of which unitigs share each overlap is enough to know the structure of the graph (see figure 13). If you take a node from the graph, you can know the "sons" of this node by selecting the nodes that begin with the end overlap of your

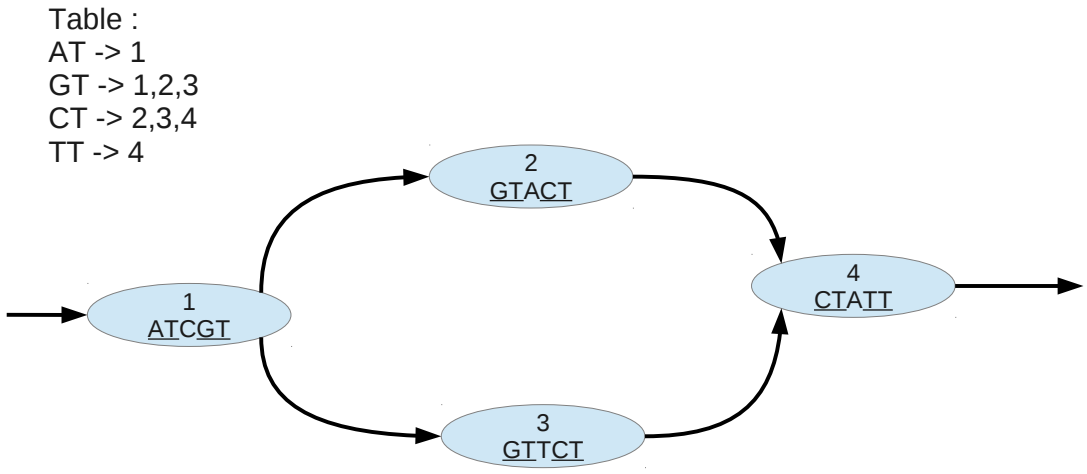



Figure 13: You can know the local structure of the graph with a query on the table. For example, the sons of 1 $(2, 3)$ can be deduced from the nodes that contains GT $(1, 2, 3)$ by looking the first characters

vertex and you can know the "fathers" by selecting the nodes that end with the begin overlap of it.

In fact, the key idea of the algorithm is not the BLAST approach, it is to use the fact that overlaps are the core sequences of a De Bruijn graph and to focus only on these sequences.

The algorithm will only keep one notable data structure, a (hash) table that take overlaps as keys and unitigs who share that overlap as values ($overlap- > unitigs$). It will serve three purposes:

1. Test if a sequence is an overlap
2. Knowing which unitigs share an overlap
3. Knowing the "sons" or the "fathers" of a unitig

We avoid here to have the complete graph structure in memory. Unitigs can be kept on disk and loaded only when needed if the available memory in not sufficient, this is the default behavior and all benchmarks have been made this way. This allow the algorithm to be very memory efficient.

The algorithm can be summarized like this:

1. Index the overlaps between the nodes (filling the table)
2. Finding an overlap in a read (test the table for each $k - 1$ substrings of the read)
3. Try to align the rest of the read on the possible paths in the graph

The two first steps are pretty simple. You read the unitigs, and fill your hash table. You check every subsequence of length $k-1$ of your reads to know if it is an overlap.

We explain know what to do when an overlap is found in a read. The method consist in two almost identical phases, map the beginning of the read and map the end of the read. To map the end of the read, we will load the unitigs ending with the found overlap and see if one the them map on the right part of the read. If the right part of the read is fully covered then it is over, if not we will continue recursively with the overlap at the end of the last used unitig. Mapping the beginning of the read is exactly the same procedure and can be deduced. A note can be made on the performance of this algorithm, the combinatorial possibility that can appear when mapping the read on the graph in "dense" region has been optimised with a memorisation of the path already tried. We can note that on complex genome like the human genome, this optimisation is mandatory. We did not details it in the pseudo code for more clarity. We present in figure 14 a minimal execution of the BGREAT algorithm.

We propose a pseudo-code version of the BGREAT algorithm.

**Data**: unitigs
**Result**: HashMap(overlap → unitigs)
**for** *each unitig U* **do**
  add (U[1,k-1] → U) to HashMap;
  add (U[U.size()-(k-1),U.size()] → U) to HashMap;
**return** to HashMap

**Algorithm 1:** Fill the HashMap

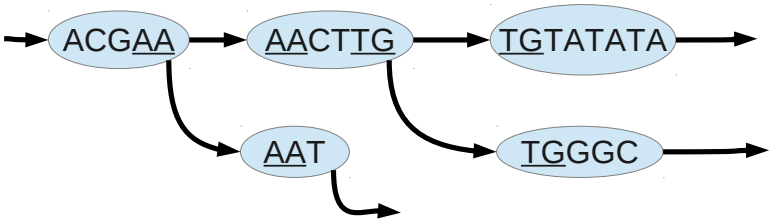
**Data**: Read,HashMap(overlap→ unitigs)
**Result**: Path in the unitigs that match the read or ∅
**for** *each substring S in the Read starting at a position p* **do**
  **if** *S is in the HashMap* **then**
    path1=FindPathMatchBegin(Read, p,HashMap);
    path2=FindPathMatchEnd(Read, p,HashMap);
    **if** $path1 \neq \emptyset$ *and* $path2 \neq \emptyset$ **then**
      **return** path1+path2
**return** ∅

**Algorithm 2:** Read mapping

### 3.3 Implementations details

We present here some details that can matter but are not necessary for a high level vision of the algorithm. In fact some reads will not have any anchor, and our algorithm will fail to map them on the graph. In the case of a big enough unitig, the read could be completely included in the unitig without have any overlap (see figure 15). Two solutions have been envisaged to capture this case that can be the most represented scenario on simple genome:

- Use a standard aligner to map the read on the big unitigs
- Index sequences in the middle of big unitigs




Read:
CGAACTTGTATA

Overlap found:
CGAACTTGTATA

Unitig matching the left part found:
CGAACTTGTATA
ACGAA

Unitig match the right part found:
CGAACTTGTATA
ACGAA
AACTTG

Unitig match the right part found:
CGAACTTGTATA
ACGAA
AACTTG
TGTATATA

Read mapped on unitigs:
{ACGAA, AACTTG, TGTATATA}

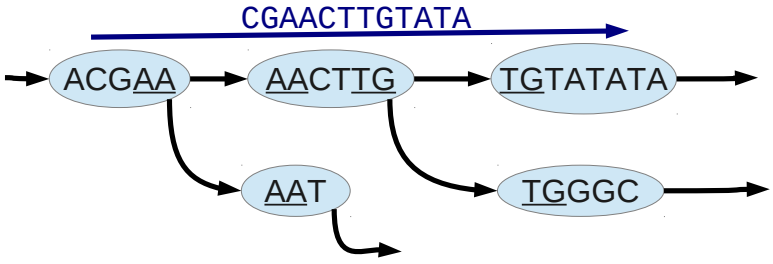



Figure 14: Example of execution of the BGREAT algorithm

```
Data: Read,p,HashMap(overlap→ unitigs)
Result: Path in the unitigs that match the begin of a read or ∅
for each unitigs U ending by the overlap Read[position, position+k-1] do
    if U.size() ≥ position then
        if match(U[U.size()-p,U.size()],Read[0,p]) then
            return (U.number(), U.size() − p)
    else
        if match(U[0,U.size()],Read[p-U.size(),p]) then
            path=FindPathMatchBegin(Read,p-U.size(),HashMap);
            if path ≠ ∅ then
                return (U.number(),U.size()-p)+path;
return ∅
```

**Algorithm 3:** FindPathMatchBegin

```
Data: Read,p,HashMap(overlap→ unitigs)
Result: Path in the unitigs that match the endn of a read or ∅
for each unitigs U begining by the overlap Read[position, position+k-1] do
    if U.size() ≥ Read.size() − p then
        if match(U[0,Read.size()-p],Read[p,Read.size()]) then
            return (U.number(), 0)
    else
        if match(U[0,U.size()],Read[p,p+U.size()]) then
            path=FindPathMatchEnd(Read,p+U.size(),HashMap);
            if path ≠ ∅ then
                return (U.number(),0)+path;
return ∅
```

**Algorithm 4:** FindPathMatchEnd

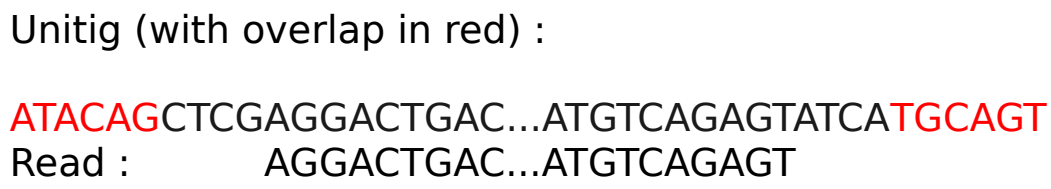



Figure 15: Case of a read matching the center of a large unitig, the read does not share any overlap

The first solution works very well in practice, with the advantage to not rise the amount of memory used.

We did not highlight the problem of mismatches in the main algorithm. If the compacted De Bruijn graph is created from all kmers from a set of reads, then all those reads will match perfectly on the graph. But in practice, all kmers are not used to create the De Bruijn graph. Rare kmers, with low occurrence in the reads, are not conserved because are very likely irrelevant (see kmer counting in assembly section). If a read includes a sequencing error, some of its kmers will not be in the De Bruijn graph and the read will not map perfectly on the graph.

In practice, with Illumina read with 1% of erroneous nucleotide and a read length of 100 base pairs, only around few percents (around 2%) of reads can be perfectly mapped simply because most reads have at least one sequencing error. Three solutions have been considered to cope with this problem:

- Accept low percentage of mapped reads (eventually use more reads)
- Correct reads
- Allow mismatches

The first is the simplest and the fastest, here we consider that a low percentage of reads mapped is sufficient for our application. The second is to run a read correction algorithm on our input reads (before the creation of the graph). This way most sequencing errors will be removed and the percentage of mapped read will be much higher. In practice, the read correction software "Bloocoo", have shown the best results, reaching 98% of mapped read on E.coli dataset. This solution is very efficient if we can accept to modify our reads. The third solution is to allows mismatches in the algorithm. It is very easy to allows mismatches in the extension phase we just have to use a custom function instead of the strict equality. But some mismatches can not be recovered this way. In fact if a sequencing error is present in an overlap, the anchor will not be found. Solution to allow mismatches in overlap have been tried but are really expensive in both terms of time and memory so we choose no to allow it. But we considered to allow some mismatches on "non-overlap" part using a custom function to compare sequences. Allowing mismatches have two drawbacks:

- Time Performance
- A read can match on multiple paths of the graph

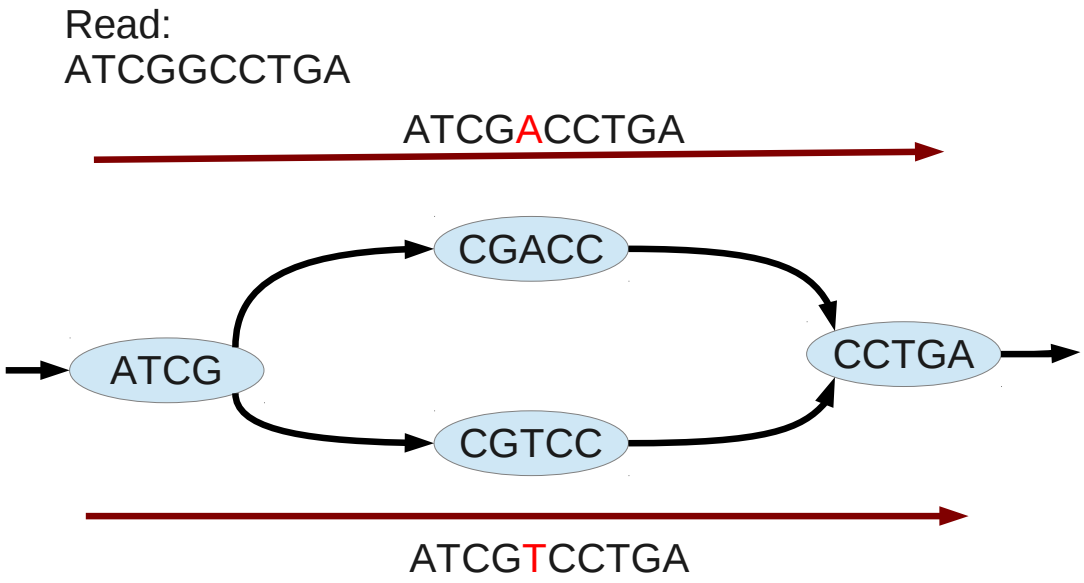



Figure 16: Allowing a mismatch allow the read to match on two "equivalent" paths of the graph

In practice it is more expensive to uses custom function instead of equality. When a read can map on multiple paths, we can choose the path with the minimum number of mismatches but often, (for example on a bubble as in the figure 16) all paths will include the same number of mismatch.

In such a case we have three choices:

- Not map the read
- Make an arbitrary choice
- Map the read on each possible path

The 3 solutions have their advantages and their drawbacks. By not mapping the reads we lose information but we do not do any mistakes. By making an arbitrary choice we can do mistake but will map more read, this is not the right choice for phasing but can be a good choice to assemble if the coverage is too low. The last solution is theoretical and could be very time expensive due to the possible combinatorial explosion and we did not thought of an application favoring this solution.

Concretely we implemented a fast method to map reads without any mismatch, that is very efficient for corrected reads. We also implemented a method allowing mismatch in "non overlap" part, mapping on the first solution encountered, which is equivalent to the arbitrary choice. We are interested, for phasing for example, to avoid to map in such a situation and it may be part of a future work.

### 3.4 Performances

To give an idea of the memory usage of BGREAT we choose to run it on three well studied data sets:

- Escherichia coli: 1 Illumina run, 5.4M reads, 526.5M bases `http://www.ncbi.nlm.nih.gov/sra/?term=SRR959239`
- Caenorhabditis elegans: 1 Illumina run, 67.6M reads, 6.8G bases `http://www.ncbi.nlm.nih.gov/sra/?term=SRR065390`
- Homo sapiens 2 Illumina runs, 602.4M spots, 150.6G bases `http://www.ncbi.nlm.nih.gov/sra/?term=SRR1291041`

We corrected the reads with Bloocoo and computed the unitigs of the De Bruijn graph with k=63 and a with a kmer occurrence filter of 3. Then we mapped the reads on the unitigs obtained with BGREAT an measured the memory usage. The pure performance of BGREAT are really promising:

- The memory usage for the E.Coli data set was less than 5MB
- The memory usage for the C.elegans data set was 75MB
- The memory usage for the human genome data set was only 2GB

When the unitigs are kept on disk, the table ($overlap- > unitigs$) is the only structure using noticeable amount of memory. The memory usage of BGREAT will be proportional to the complexity of the De Bruijn graph, here the number of unitigs/overlaps. The results obtained are very impressive, very few tools can deal with a genome as complex as the human one with so few memory. It will allow this tool to run on desktop computers and will not require a cluster to be used.

The throughput of the algorithm is also quite fast. It usually map more then 10 000 reads by second on a desktop computer. It is quite comparable to the throughput of fast read mapper as BOWTIE2 that can align 30 millions per CPU hour. We can also note that the algorithm can be easily parallelized. The main table ($overlap- > unitigs$) is not modified and can be accessed concurrently by any number of threads. Each thread is responsible of a set of thread. In practice the multithreading worked very well and the throughput almost grow linearly with the number of cores as we can see in the figure 17.

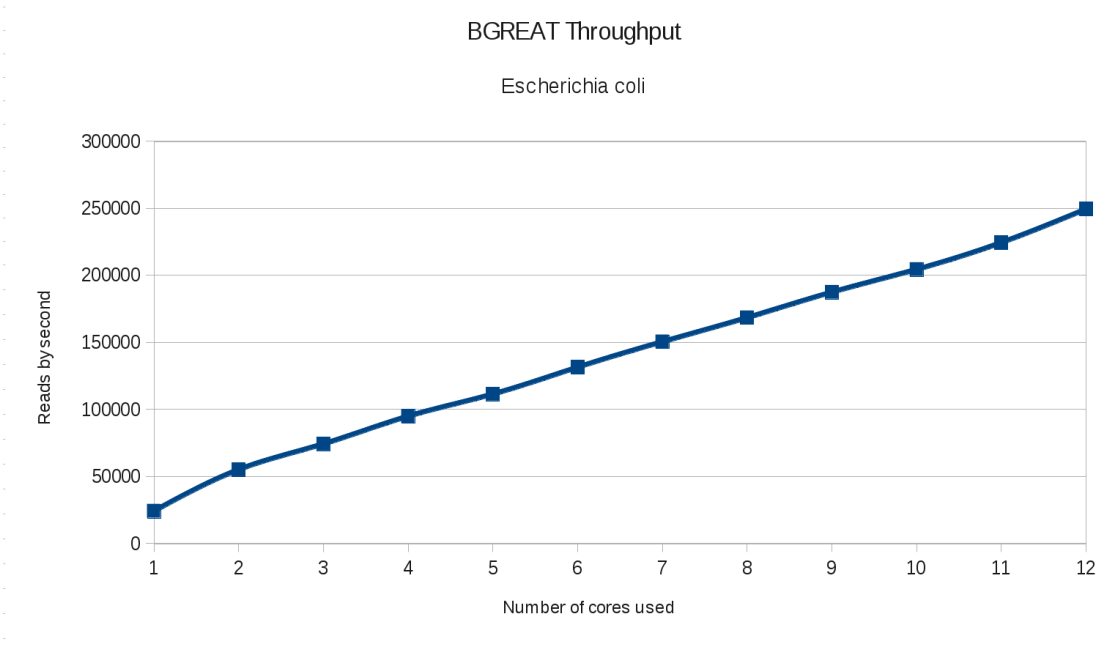



Figure 17: Multithreading of BGREAT on Escherichia coli (SRR959239) with k=60

# 4 Applications and Results

Mapping reads on a De Bruijn graph can be useful in a large set of applications. The initial purpose of the internship was to use the information to improve the assembly and for phasing. We will present here the applications that were studied, their results and what is left to do.

## 4.1 Assembly

The original idea was to map the reads on the graph in order to improve the quality of the assembly. As we explained, in order to get contigs, assembler simplify the graph, smashing bubble, removing some patterns etc ... The idea was to use the read information to improve this phase. Due to the nature of the De Bruijn graph, some sequences could be connected in the graph but not sequential in the genome. In the figure 18 the arrows between nodes are the edges of the De Bruijn graph, the two paths of black arrows correspond to actual sequences but not the red arrows. We recall that the information of mapped read could then be used to simplify the graph by removing edges that are not validated by actual reads (red arrows).

To go further, we can actually avoid to compute the edges of the De Bruijn graph and consider that two nodes are connected if and only if it exist a read where they are sequential. The interest is double, first we compute a "better" graph, closer to the reference, and we avoid a very time and memory consuming process. Here we propose fast and memory-efficient assembly chain:

- Bloocoo (read correction, not yet published)
- DSK [29] (kmer counting)

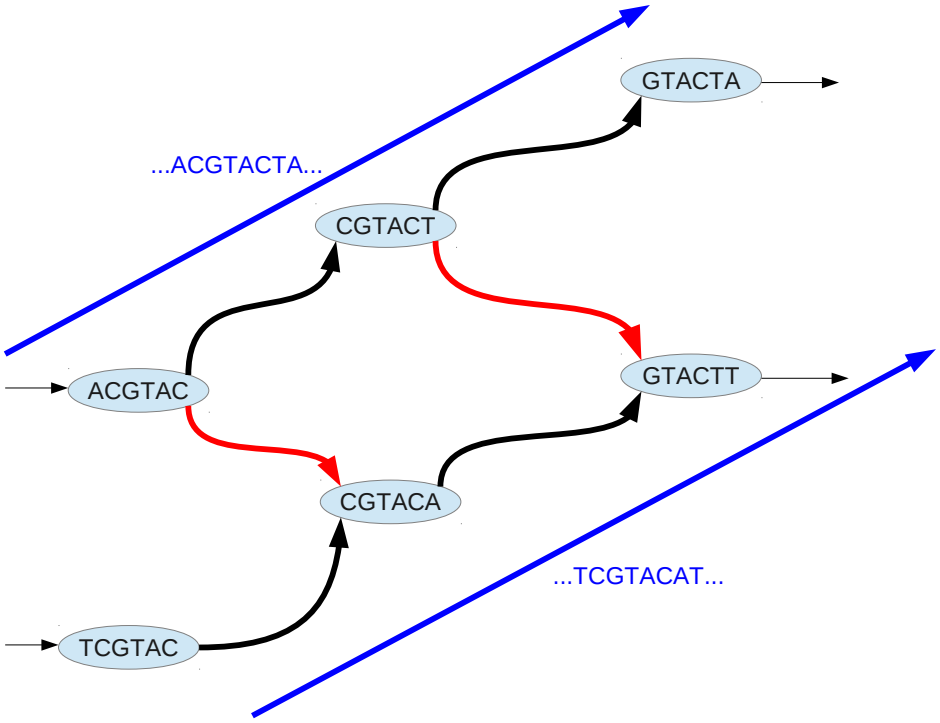



Figure 18: In such a case the information of mapped reads allow to highly simplify the graph. Black arrow are validated. Read arrows are not and could be remove to simplify the graph

- BCALM [14] (unitigs computation)
- BGREAT (map read on unitigs)
- SGA assemble [30] (compute contigs from the graph of unitigs)

The first phase is the optional read correction that is recommended to be able to map a higher percentage of the reads. The proposed software Bloocoo, to our knowledge, allows the best correction in constant memory and faster than his concurrents such as BLESS [31] or MUSKET [32]. The second phase is the kmer counting kmer phase with DSK [29], like Bloocoo the choice is motivated by the constant memory usage and good performances. The two previous tools achieve to use constant memory usage using a bloom filter. This data structure allow the insertion an arbitrary number of kmer [33]. Third phase compute the unitigs (simple paths) of the De Bruijn graph with BCALM [14], once again the choice is motivated by the low memory usage. The fourth phase maps the reads on the unitigs with BGREAT. It produce a ASQG format file describing the graph of unitigs with the connection computed by mapping the reads (ASQG is the format used by SGA to describe a string graph). The Fifth phase compute the contigs from the ASQG files with SGA assemble which is the contigs creation algorithm of SGA (String graph assembler). This is the fastest phase because the graph is already computed.

We can note that, in this assembly framework, we will not map the reads that match the core of big unitigs. In fact, for this application we do not need this information because it will not give any information on the edges of the graph.

We present here some results obtained on real data with this chain and compare it to classical assembly software. We choose to run our test on a real data set of 65 millions of Illumina reads of length 100 from C. Elegans which is a well studied organism with a genome of estimated size 100 millions bases pairs. The run chosen is SRR065390 and can be accessed at `www.ncbi.nlm.nih.gov/sra/SRX026594`. We first present a detailed run of the assembly chain and then compare it to two mainstream assemblers SAOPdenovo [34] and Velvet [8].

- DSK
  - Wallclock time: 552 seconds
  - CPU time: 2600 seconds
  - Memory usage: 4GB
- BCALM
  - Wallclock time: 2000 seconds
  - CPU time: 2000 seconds
  - Memory usage: 14MB
- BGREAT
  - Wallclock time: 882 seconds
  - CPU time: 8853 seconds
  - Memory usage: 75MB

- SGA ASSEMBLE
  - Wallclock time: 60 seconds
  - CPU time: 60 seconds
  - Memory usage: 175MB

As we can see in figure 19, the memory usage of our assembly chain is low and can be run on a regular desktop computer where other assembler require huge amount of RAM and need a cluster. We can also see that we used less CPU time than SOAPdenovo and Velvet, which are two of the most used assemblers. This performance is really promising since the main goal was to improve the quality. It is very interesting to see that to compute our version of the graph is actually efficient.

We now will present results in term of quality of the assemblies obtained. A high number of metrics is used to measure the quality of an assembly, since it is difficult to define a good assembly and even more complex be able to compare two different assemblies. In order to provide a complete point of view we used QUAST [35]. QUAST is a script combining a set of tools that compute most of the main metrics used in the different fields.

To comment these results, we will define and focus of most used metrics. The number of contigs longer than a given number of bases (500, 1000) or the length of the longest contig give an idea of the contigs length but is not precise enough to see how the lengths are distributed. In the assembly field the most used metric is the N50. The N50 is similar to a mean or median, but has greater weight given to the longer contigs. Given a set of contigs, the N50 length is defined as the length for which the collection of all contigs of that length or longer contains at least half of the total of the lengths of the contigs. The L50 is the length of the smallest contig considered by the N50. Many variants of those metrics are used:

- The NG50 who does not use the total length of contigs but the reference genome length
- The NA50 that only consider contigs that can be aligned on a reference
- The NGA50 that used aligned contigs and the length of the reference
- The N75 NG75 NGA75 that require considered contigs to cover 75 percents instead of 50 percents
- and their corresponding LG50 LGA50 L75 LG75 LGA75 ...

We choose to focus on the N50 (and NA50) values but to present all available metrics that can interest. The N50 of the assembly chain based on BGREAT is superior to the other results on both data set, which is a good thing since the main goal of an assembly is to return long contigs. We can also not that the NA50 is also slightly superior, it mean that our long contigs map well on the reference. An other interesting statistic is the number of misassemblies, that are basically assembler errors. It happen when sequences that are not consecutive in the genome are merged by the assembler. We are satisfied to see that our assembly chain do very few mistakes. Another important metric is the genome fraction, the amount of the genome covered by the contigs. The genome fraction return is between the result of SOAPdenovo and Velvet. Our results are quite close and often better than the results of SOAPdenovo and Velvet, showing that our approach make sense and could really bring a stone in the assembly field.

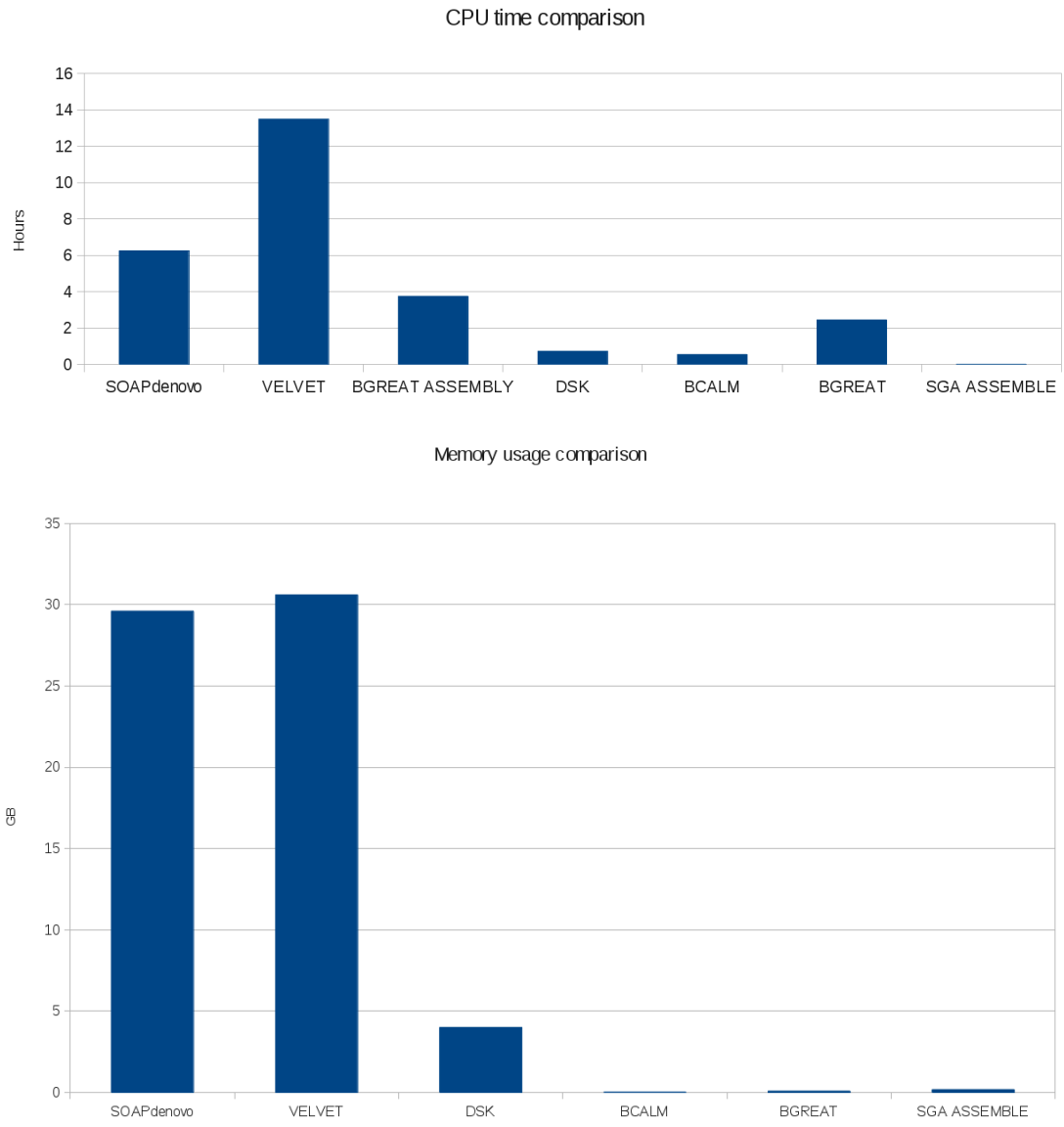



Figure 19: Comparaison of CPU time and memory used for the assembly of C.elegans data. The CPU time used by the tools of the pipeline are summed in the column BGREAT ASSEMBLY

| k=31 | SOAPdenovo | Velvet | Bgreat assembly chain |
|---|---|---|---|
| Contigs (>= 0 bp) | 372 268 | 91 128 | 463 920 |
| Contigs (>= 1000 bp) | 21 599 | 20 478 | 19 258 |
| Total length (>= 0 bp) | 109 121 230 | 97 745 289 | 112 196 945 |
| Total length (>= 1000 bp)) | 78 292 122 | 81 836 344 | 80 510 057 |
| Contigs | 33 455 | 30 204 | 29 405 |
| Largest contig | 73 163 | 62 653 | 76 628 |
| Total length | 86 830 284 | 88 857 916 | 87 827 347 |
| Reference length | 100 286 401 | 100 286 401 | 100 286 401 |
| GC (%) | 35.81 | 35.82 | 36.05 |
| Reference GC (%) | 35.44 | 35.44 | 35.44 |
| N50 | 4 545 | 5 464 | 5 980 |
| NG50 | 3 530 | 4 427 | 4 680 |
| N75 | 1 929 | 2 273 | 2 282 |
| NG75 | 1 190 | 1 491 | 1 396 |
| L50 | 4 662 | 3 986 | 3 497 |
| LG50 | 6 340 | 5 154 | 4 676 |
| L75 | 12 113 | 10 414 | 9 550 |
| LG75 | 18 783 | 15 060 | 14 776 |
| Misassemblies | 18 | 1 023 | 9 |
| Misassembled contigs | 18 | 634 | 8 |
| Misassembled contigs length | 75 484 | 3 391 054 | 41 336 |
| Local misassemblies | 12 | 322 | 395 |
| Unaligned contigs | 1 780 + 39 part | 1 553 + 225 part | 471 + 114 part |
| Unaligned length | 4 202 764 | 4 271 709 | 4 508 077 |
| Genome fraction (%) | 82.206 | 84.186 | 82.958 |
| Duplication ratio | 1.002 | 1.002 | 1.002 |
| N's per 100 kbp | 0.00 | 11.81 | 0.00 |
| Mismatches per 100 kbp | 0.85 | 10.90 | 6.77 |
| Indels per 100 kbp | 0.78 | 4.38 | 1.72 |
| Largest alignment | 73 163 | 62 653 | 75 229 |
| NA50 | 4 280 | 4 915 | 5 114 |
| NGA50 | 3 243 | 3 919 | 3 910 |
| NA75 | 1 668 | 1 901 | 1 843 |
| NGA75 | 952 | 1 170 | 1 057 |
| LA50 | 4 809 | 4 256 | 3 980 |
| LGA50 | 6 620 | 5 559 | 5 373 |
| LA75 | 13 120 | 11 655 | 11 334 |
| LGA75 | 21 130 | 17 416 | 18 046 |

Table 1: QUAST metrics comparison for C.elegans ($k = 31$). All statistics are based on contigs of size >= $500bp$, unless otherwise noted.

| k=31 | SOAPdenovo | Velvet | Bgreat assembly chain |
|---|---|---|---|
| Contigs (>= 0 bp) | 3 345 | 1 613 | 2 492 |
| Contigs (>= 1000 bp) | 1 002 | 885 | 878 |
| Total length (>= 0 bp) | 4 373 297 | 4 063 513 | 4 281 821 |
| Total length (>= 1000 bp)) | 3 951 459 | 3 804 315 | 4 021 359 |
| Contigs | 1 248 | 1 096 | 1 066 |
| Largest contig | 35 243 | 26 427 | 35 258 |
| Total length | 4 130 108 | 3 959 998 | 4 155 300 |
| Reference length | 4 639 675 | 4 639 675 | 4 639 675 |
| GC (%) | 51.76 | 52.13 | 51.68 |
| Reference GC (%) | 50.79 | 50.79 | 50.79 |
| N50 | 5 022 | 5 533 | 6 381 |
| NG50 | 4 472 | 4 776 | 5 608 |
| N75 | 2 770 | 3 099 | 3 418 |
| NG75 | 2 073 | 1 972 | 2 522 |
| L50 | 251 | 220 | 210 |
| LG50 | 304 | 286 | 250 |
| L75 | 523 | 454 | 430 |
| LG75 | 681 | 659 | 554 |
| Misassemblies | 4 | 10 | 4 |
| Misassembled contigs | 4 | 10 | 4 |
| Misassembled contigs length | 17 691 | 100 347 | 19 569 |
| Local misassemblies | 3 | 5 | 5 |
| Unaligned contigs | 11 + 18 part | 12 + 30 part | 10 + 13 part |
| Unaligned length | 38 156 | 32 982 | 38 667 |
| Genome fraction (%) | 88.077 | 84.602 | 88.635 |
| Duplication ratio | 1.001 | 1.001 | 1.001 |
| N's per 100 kbp | 0.00 | 0.76 | 0.00 |
| Mismatches per 100 kbp | 3.87 | 14.24 | 15.95 |
| Indels per 100 kbp | 0.17 | 0.82 | 0.32 |
| Largest alignment | 35 243 | 26 427 | 35 258 |
| NA50 | 4 982 | 5 456 | 6 280 |
| NGA50 | 4 429 | 4 656 | 5 452 |
| NA75 | 2 731 | 3 032 | 3 310 |
| NGA75 | 2 008 | 1 880 | 2 462 |
| LA50 | 252 | 223 | 211 |
| LGA50 | 307 | 290 | 252 |
| LA75 | 530 | 464 | 435 |
| LGA75 | 692 | 676 | 563 |

Table 2: QUAST metrics comparison for E.coli ($k = 31$). All statistics are based on contigs of size $>= 500bp$, unless otherwise noted.

**Future work for improving assembly using BGREAT** We also tried to assemble a human genome, but if the the memory usage was very low (2GB), the runtime was however quite long due the high number of disk access. We project to highly reduce the number of disk access in the near future and this will probably allow to efficiently assemble a human genome. An other optimisation that will be required to make the soft highly competitive is the binarisation of sequences. In BGREAT, sequence are manipulated in ASCII, it mean that each base pair use a full byte where theoretically, only 2 bit is necessary. It will allow faster computation, will divide the disk usage and then highly improve the runtime. It will also make interesting the option to keep unitigs in memory. We expected the cost to put in memory the unitigs on the human genome to 1GB. As it was presented this allow very fast execution and could make BGREAT a really efficient tool.

To conclude, we can say that the goal have been achieved, we produced an more "read-qware" assembly chain. The assembly quality obtained is very satisfying, with very few misassemblies and long contigs. The focus made on data structure and memory used have been rewarding since we are able to assemble a human genome on a desktop computer. Even the runtime performances are good enough to be enlightened. This algorithm still need work to be user-friendly and further improvements can be made be. It could be a very interesting alternative to the standard vision on the assembly and on the De Bruijn graph.

## 4.2 Reads Compression

We present here an application that whas not initially planned but that appeared to be a nice extention of the proposed algorithm. We thus checked at a glance to the compression applications offered by BGREAT. Presented results are encouraging and this application would deserve further investigations in future works.

The compression of sequencing data is a well studied problem that is becoming more and more important since the throughput of sequencers is raising faster than the storage available (see figure 2 page 2). In theory, the compression rate can be very high since the information is often very redundant. But in practice the compression rate can be high when using a reference, but without it the compression rate is very low (state of the art algorithms [36] reach a compression rate of 0.2 on some dataset while the trivial code 2 bit/bases offer a rate of 0.25).

We can note that the information returned by BGREAT allow to recompose the reads from the unitigs. You know on which unitigs and in which positions on them the reads map, you can then recompose them easily with only disk access. With this we though of a chain allowing the compression of the genomic part of a read file:

- DSK (kmer counting)
- BCALM (unitigs computation)
- BOWTIE2 (read mapper on big unitigs)
- BGREAT (map read on graph of unitigs)
- 7ZIP (standard compressor)

The two first phases to compute the unitigs are the same as the assembly chain . In this application we need to know where the reads map even if they do not have any overlap. In order to get this information we run a standard tool to map the reads on the big unitigs (we chose BOWTIE2

for his very high throughput and low memory usage). The reads not mapped by BOWTIE are then mapped on the unitig graph by BGREAT. The read that can not be mapped by BGREAT are just wrote as it. We got then 3 possibility:

- The read is mapped on a unitig by BOWTIE: just keep the number of the unitig and where it maps.
- The read is mapped on a path of unitigs by BGREAT: just keep the number of the unitigs and where it maps.
- The read is not mapped: we have to write it completely

The last phase is to compress the four different ASCII files (the unitigs file and the three files containing the three possible case) in an efficient way using a standard compressor. Once again, read correction is highly recommended, since way more reads will be mapped by BOWTIE2 and BGREAT and the compression rate will be much higher. The compression rate achieved on simulated data have been extremely impressive, on an extract of the human genome with a 100X coverage, a 0.02 compression rate have been achieved. This kind of compression rate have never been achieved without reference and could even be superior to the compression rate with reference. An other huge advantage of this method is that the decompression speed could hardly be faster, you just read right parts on the disk, no computation is needed.

The proposed chain has, for now, major drawbacks and potential solutions:

- Work best with corrected reads (some will not want to modify their input reads, a de-corrector have been envisaged)
- Do not keep the order of the reads (the multi-threading have to be improved to allow an order-safe option, but we can note that, most of the time, reads order does not matter)
- Do not keep the header part (the chain will have to be coupled with a header compressor)
- Memory efficient but very slow compression (almost an assembly, way longer that standard read compressor )

This application is really promising but will need further work to be a proper tool.

### 4.3 Phasing

The last goal of this internship was to phase using the information of mapped reads. The work on phasing have been way more theoretical than the two others and have no concrete results or method yet. We started the reflexion on potential solutions based on BGREAT and on the problems that we will face. The main problem is that there is a lack of polyploid model in the assembly field, as we said assemblers make the assumption that there is one original sequence and try to approximate it. If we produced a polyploid assembler returning phased contigs, regular tools would not be able to handle them. The theoretical questions are how to use phasing, who want to use it and what they concretely want. We still have to know how to represent our results, how to use them or how to adapt contig based tools before any concrete implementation.

Nevertheless we worked on some promising solutions to allow some phasing. We have listed some local patterns in the graph where phasing can be made in the diploid case. The figure 20

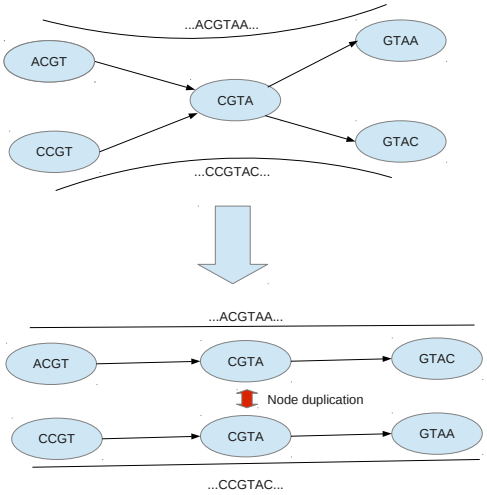



Figure 20: Pattern where node duplication allow to simplify the graph

show the basic example of "an intersection" where only two out of four paths are covered by reads. We could duplicate the center node to remove the ambiguity. The interest of such pattern is double, it simplify the graph (A classic assembler will not know that the path $ACGT \rightarrow CGTA \rightarrow GTAC$ does not exist and will consider it) and is able phase some patterns like the "bubble" as we can see on the figure 21.

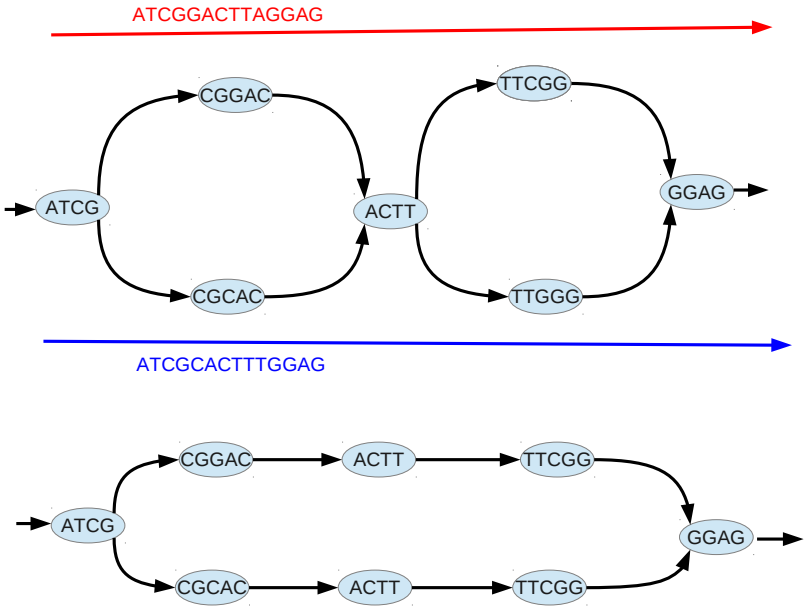



Figure 21: How patterns can phase bubble in the De Bruijn graph

# 5 Conclusion

This internship "Assembly improvements by read mapping and phasing" has multiple possible focuses and ways to explore. The proposed goals were:

- Allowing the mapping of reads on the De Bruijn graph
- Explore the possibilities of such an algorithm on assembly
- Try to phase using the read mapping

We choose to conceive a whole new algorithm to map read on a De Bruijn graph. The results showed that the proposed solution is efficient in term of throughput and reach a higher level challenge, be able to work on complex genome as the human one on a desktop computer. But we are very interested to see how much we can expect from this approach in term of efficiency. The second goal was to improve the assembly quality by using the information of mapped reads. At the beginning we wanted to improve an assembler, and we realized that we were able to conceive our own efficient assembly chain using this information. We conceived a complete assembly chain that achieved to be fast and extremely memory efficient while being more "reads aware". Once again there is still work to be done to be able to use the read information. We used a standard contigs creation algorithm, it would be very interesting to conceive a contigs creation algorithm able to make full use of the mapped read information. The goal of phasing was the most theoretically challenging, we made some advance and start to integrate some phasing in our assembler. Still we faced more fundamental problems, what do we expect from an assembly and what to do with phased contigs. This part will need more focus to give concrete results. We have also studied unexpected applications like reads compression which gave very promising results. In the near future, we want to pack properly those applications to get a distributable software for the read mapping itself, assembly, read compression and maybe phasing as an experimental feature of our assembly. Allowing read mapping on De Bruijn graph has shown very powerful and straightforward applications in bio-informatics and is very promising in term of improvement of existing software but also in term on impact on the view on the De Bruijn graph and how it is used in bio-informatics. We shown that compute a "better" (partial) De Bruijn graph was not necessarily more expensive (probably less in fact).

Personally I learned a lot during this internship, I get to study the whole assembly chain and other algorithms on the treatment of NGS data such as reads correction or sequencing data compression. My internship went very well, the subject was very large and I enjoyed to be free to focus on the part of my choice. I also get to talk and work with very interesting persons in the laboratory that gave me ideas and support in my internship. I am quite satisfied of my work and my results, I get to use some results and tools I previously obtained in my other internships which is quite fullfilling. There is still work to do to pursue this project and I really hope to be able to stay in touch with it. For all those reason I enjoyed this internship and consider it as a sucess.

## References


[1] 1000 dollars genome annonce. http://www.illumina.com/systems/hiseq-x-sequencing-system.ilmn.

[2] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. *BioMed Research International*, 2012, 2012.

[3] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three next generation sequencing platforms: comparison of ion torrent, pacific biosciences and illumina miseq sequencers. *BMC genomics*, 13(1):341, 2012.

[4] Sebastian Deorowicz and Szymon Grabowski. Compression of dna sequence reads in fastq format. *Bioinformatics*, 27(6):860–862, 2011.

[5] Niranjan Nagarajan and Mihai Pop. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology*, 16(7):897–908, 2009.

[6] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[7] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[8] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[9] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and İnanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[10] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

[11] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[12] Evomics workshop on genomics, 2014, de novo assembly.

[13] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. In *Algorithms in Bioinformatics*, pages 236–248. Springer, 2012.

[14] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared Simpson, and Paul Medvedev. On the representation of de bruijn graphs. *RECOMB*, 2014.

[15] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[16] Temple F Smith and Michael S Waterman. Comparison of biosequences. *Advances in Applied Mathematics*, 2(4):482–489, 1981.

[17] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[18] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[19] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[20] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[21] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[22] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[23] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[24] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[25] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[26] Jason R Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[27] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.

[28] Guillaume Holley and Pierre Peterlongo. Blastgraph: Intensive approximate pattern matching in sequence graphs and de-bruijn graphs. In *Stringology*, pages 53–63, 2012.

[29] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[30] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[31] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. Bless: Bloom-filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[32] Yongchao Liu, Jan Schröder, and Bertil Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

[33] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[34] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, et al. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience*, 1(1):18, 2012.

[35] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[36] James K Bonfield and Matthew V Mahoney. Compression of fastq and sam format sequencing data. *PloS one*, 8(3):e59190, 2013.