GATB

The Genome Assembly & Analysis Tool Box

Main Concepts

GATB

The Genome Assembly & Analysis Tool Box

- NGS technologies produce many short reads

```
>read 1
ACGACGACGTAGACGACTAGCTAGC
AATGCTAGCTAGGATCAAAACTACG
ATCGACTAT

>read 2
ACTACTACGATCGATGGTCGAGGGC
GAGCTAGCTAGCTGACGCTGCTCGC
TCTCTCGCT

...

>read 10.000.000
TCTCCTAGCGCGGCGTATACGCGCT
AAGCTAGCTCTCGCTGCTCGCTAGC
TACGTAGCT

...
```

GATB

The Genome Assembly & Analysis Tool Box

- Each read is split into words named **kmers**
- A **kmer** has a fixed size K

- Example for K=11

```
>read 1
ACGACGACGTAGTAAACTACGATCGACTAT
```

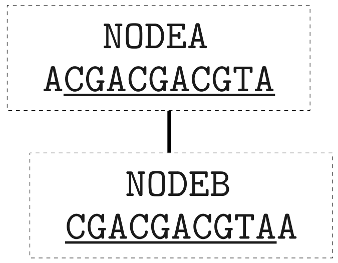
↓

```
ACGACGACGTA                                kmer  1
 CGACGACGTAG                               kmer  2
  GACGACGTAGT                              kmer  3
    CGACGTAGTA                             kmer  4
              ...                          ...
                  ACGATCGACTA              kmer 18
                   CGATCGACTAT             kmer 19
```

- Each kmer of size K is inserted as a node of a **de Bruijn graph**

- Two nodes A and B are connected
  <=>
  suffix (K-1) of A equals prefix (K-1) of B

- Example (K=11)



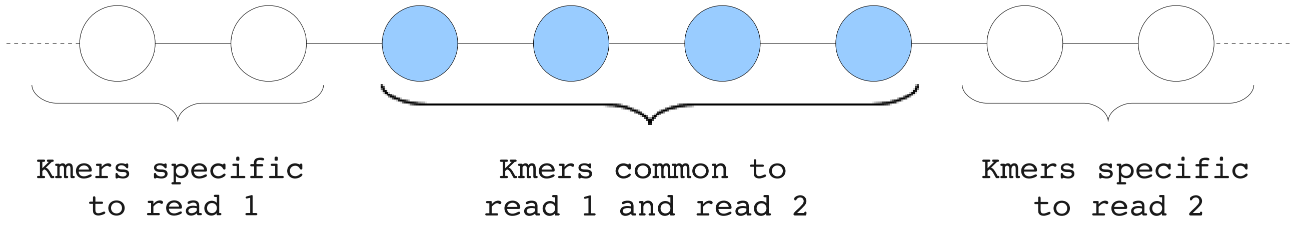
GATB

The Genome Assembly & Analysis Tool Box

- Two reads having common kmers will be connected in the de Bruijn graph
- Example (K=11)

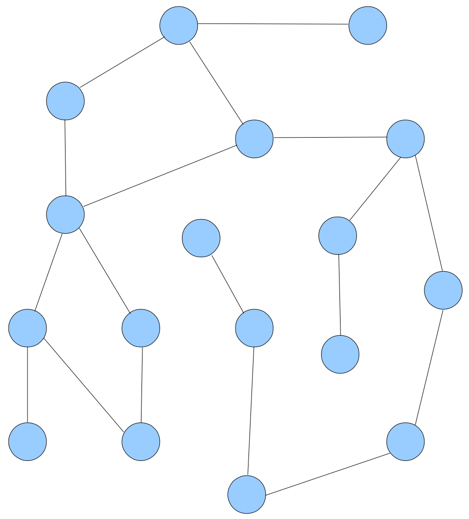
```
>read 1
ACGACGACGTAGTAAACTACGATCGACTAT
```

```
>read 2
CTACGATCGACTATTAGTGATGATAGATAGAT
```



- From reads to de Bruijn graph
  1. Split the reads into kmers
  2. Insert the kmers into a de Bruijn graph

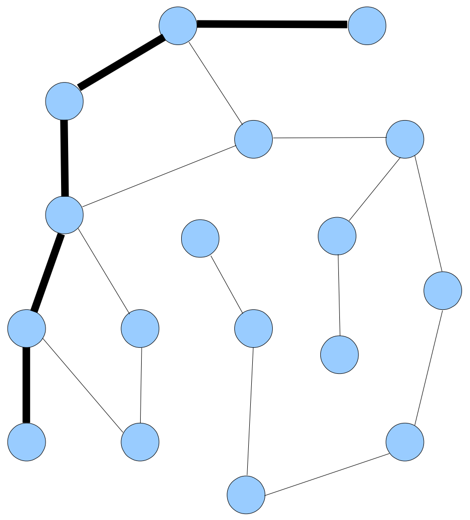
```
>read 1
ACGACGACGTAGACGACTAGCTAGC
AATGCTAGCTAGGATCAAAACTACG
ATCGACTAT

>read 2
ACTACTACGATCGATGGTCGAGGGC
GAGCTAGCTAGCTGACGCTGCTCGC
TCTCTCGCT

...

>read 10.000.000
TCTCCTAGCGCGGCGTATACGCGCT
AAGCTAGCTCTCGCTGCTCGCTAGC
TACGTAGCT

...
```



GATB

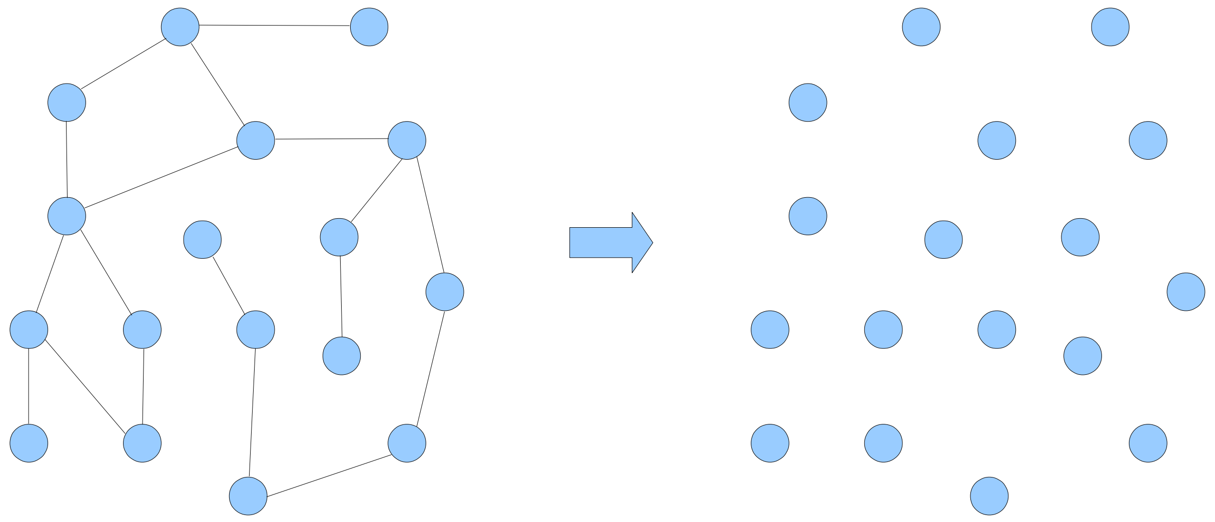
The Genome Assembly & Analysis Tool Box

- Assembly task : traverse the de Bruijn graph
- A path in the graph is an assembly sequence called **contig**
- Example :



The graph may not be resolved in a single contig !

- GATB-CORE only stores nodes of the de Bruijn graph, *the edges are computed on the fly when needed*.
- Consequence : **lower memory footprint**



GATB

The Genome Assembly & Analysis Tool Box

- GATB-CORE stores the nodes of the de Bruijn graph in a **Bloom filter**
- A **Bloom filter** is a space-efficient structure used to test whether an element is a member of a set

  False negatives are not possible

  False positives are possible
- Consequence : **lower memory footprint**
- An extra data structure named **cFP** is used to avoid the false positives drawback

GATB

The Genome Assembly & Analysis Tool Box

- Requesting whether a node belongs to the graph

  => request the Bloom filter

  1. If the answer is « no », the node doesn't belong to the graph
  2. If anwer is « yes », the cFP structure is requested

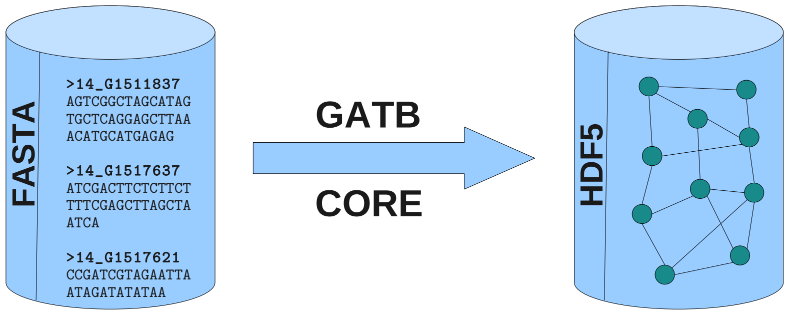
- As a result, we have a deterministic low memory footprint structure

- The **HDF5** file format is used for storing the whole graph information (suffix .h5)

GATB

The Genome Assembly & Analysis Tool Box

- **So, GATB-CORE transforms a set of reads into a de Bruijn graph**



- The transformation can be done with the **dbgh5** binary provided by the GATB-CORE component

```
dbgh5 -in myreads.fa -kmer-size 31 -out graph.h5
```

- **GATB-CORE provides a C++ library for reading the de Bruijn graph in HDF5 format**

GATB

The Genome Assembly & Analysis Tool Box

- GATB-TOOLS provides sotfware based on the GATB-CORE C++ library

- In particular, some of them process information by traversing the de Bruijn graph

- A classical pipeline for such a tool is :

```
dbgh5 -in myreads.fa -kmer-size 31 -out graph.h5

SomeTool -in graph.h5 -arg1 1 -arg2 7
```

- Advantage : GATB-TOOLS developpers have just to focus on their own algorithms and don't have to bother with the de Bruijn graph construction

Some tools from GATB-TOOLS

- **Minia**

  short-read assembler based on a de Bruijn graph.The output of Minia is a set of contigs. Minia produces results of similar contiguity and accuracy to other de Bruijn assemblers (e.g. Velvet).

- **DiscoSNP**

  discover Single Nucleotide Polymorphism (SNP) from non-assembled reads

- **TakeABreak**

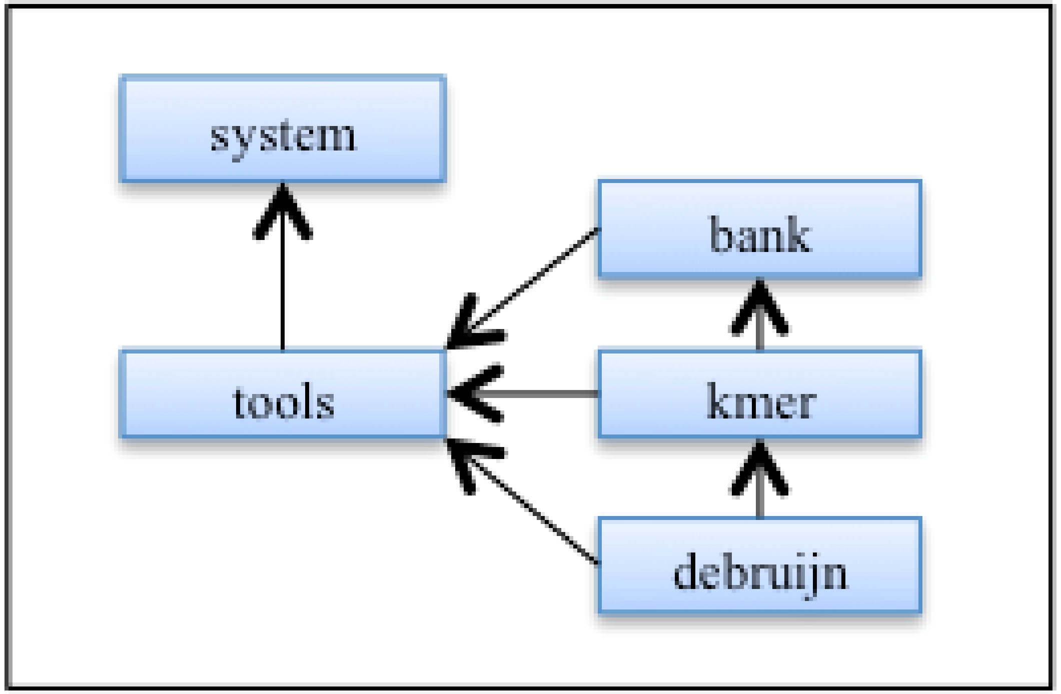
  detects inversion breakpoints without a reference genome by looking for fixed size topological patterns in the de Bruijn graph

- **Bloocoo**

  k-mer spectrum-based read error corrector, designed to correct large datasets with a very low memory footprint.

GATB-CORE C++ library quick overview (1)

- High level packages



GATB-CORE C++ library quick overview (2)

- The **system** package holds all the operations related to the operating system: file management, memory management and thread management.
- The **tools** package offers generic operations used throughout user applicative code, but not specific to genomic area.
- The **bank** package provides operations related to standard genomic sequence dataset management. Using this package allows to write algorithms independently of the input format.
- The **kmer** package is dedicated to fine-grained manipulation of k-mers.
- The **debruijn** package provides high-level functions to manipulate the de Bruijn graph data structure

GATB-CORE C++ library quick overview (3)

- A de Bruijn graph is represented by an object of the class **Graph**
- A **Graph** object can be :
  - built from a set of reads (a FASTA file for instance)
  - saved in a HDF5 file
  - loaded from a HDF5 file

GATB-CORE C++ library quick overview (4)

- A **Node** object is a node in the bi-directional de Bruijn graph.

```
struct Node
{
     Node::Value  kmer;
     Strand      strand;
     u_int16_t    abundance
};
```

- 'kmer' is the minimum value of the two kmers (one per strand) of a node in the bidirectional de Bruijn graph.
- 'strand' tells with which strand the 'kmer' value has to be interpreted.
- 'abundance' is the occurrences number of the kmer in the initial set of reads.

GATB-CORE C++ library quick overview (5)

- A **Edge** object is a transition between two nodes.

```
struct Edge
{
    Node       from;
    Node      to;
    Nucleotide nt;
    Direction  dir;
};
```

- A **Graph** object is immutable
  - Built once from a set of reads
  - Can not be modified
  - **Node** and **Edge** objects are retrieved from the graph

GATB-CORE C++ library quick overview (6)

- **Avalaible operations on a graph**
  - Iteration of all the nodes of a graph
  - Iteration of the branching nodes of a graph
  - Get neighbors nodes (from a node)
  - Get neighbors edges (from a node)
  - Get in/out degree (from a node)
  - Tells whether a node is branching or not
  - Many more features

GATB-CORE C++ library quick overview (7)

- **Create a graph from reads**

```
// We include what we need for the test
#include <gatb/gatb_core.hpp>

int main (int argc, char* argv[])
{
    // We get a command line parser for graphs available options.
    OptionsParser parser = Graph::getOptionsParser();

    // We use a try/catch block in case we have some command line parsing issue.
    try  {
        // We parse the user options.
        parser.parse (argc, argv);

        // We create the graph with the provided options.
        Graph graph = Graph::create (parser.getProperties());

        // We dump some information about the graph.
        std::cout << graph.getInfo() << std::endl;
    }
    catch (OptionFailure& e)
    {
        e.getParser().displayErrors (stdout);
        e.getParser().displayHelp   (stdout);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

GATB-CORE C++ library quick overview (8)

- **Load a graph and iterate its nodes**

```
// We include what we need for the test
#include <gatb/gatb_core.hpp>

int main (int argc, char* argv[])
{
    // We check that the user provides a graph URL (supposed to be in HDF5 format).
    if (argc < 2)
    {
        std::cerr << "You must provide a HDF5 file." << std::endl;
        return EXIT_FAILURE;
    }

    // We load the graph from the provided argument
    Graph graph = Graph::load (argv[1]);

    // We get an iterator for all nodes of the graph.
    Graph::Iterator<Node> it = graph.iterator<Node> ();

    // We loop each node. Note the structure of the for loop.
    for (it.first(); !it.isDone(); it.next())
    {
        // The currently iterated node is available with it.item()
        // We dump an ascii representation of the current node.
        std::cout << graph.toString (it.item()) << std::endl;
    }

    return EXIT_SUCCESS;
}
```