GATB

Workshop

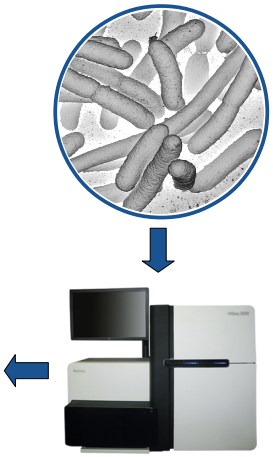
D.Lavenier

E.Drézen

- NGS technologies produce terabytes of data

- Efficient and fast algorithms are essential to analyze such data

```
>read1
ACGACGACGTAGACGACTAGC
AAAACTACGATCGACTAT
>read2
ACTACTACGATCGATGGTCGC
GCTGCTCGCTCTCTCGCT
...
>read10.000.000
TCTCCTAGCGCGGCGTATACG
CTCGCTAGCTACGTAGCT
...
```



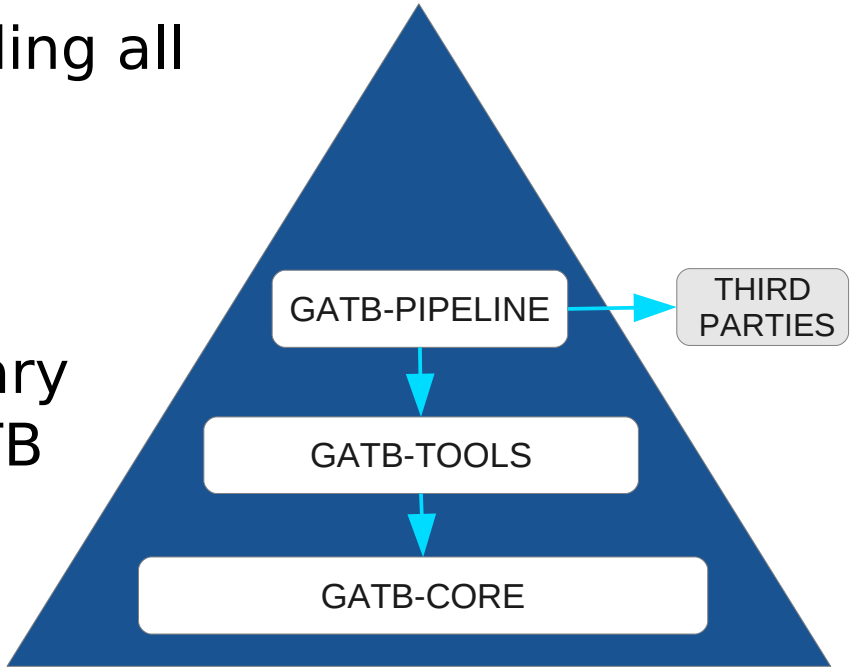
- The **Genome Assembly Tool Box** (GATB)

1. Open-source software developed by GENSCALE
2. Easy way to develop efficient and fast NGS tools
3. Based on data structure with a very low memory footprint
4. Complex genomes can be processed on desktop computers

- The **GATB** philosophy: a 3-layer construction to analyze NGS datasets

1. **GATB-CORE**: a C++ library holding all the services needed for developing software dedicated to NGS data

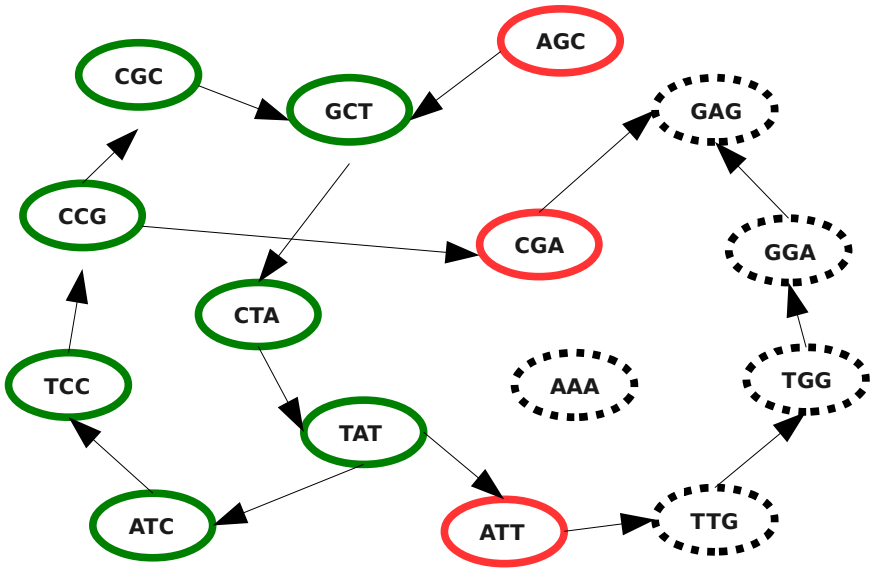
2. **GATB-TOOLS**: a set of elementary NGS tools mainly built upon the GATB library (k-mer counter, contiger, scaffolder, variant detection, etc.)



3. **GATB-PIPELINE**: a set of NGS pipeline that links together tools from the previous layer

GATB-CORE structure for NGS data

- **Compact de Bruijn graph**
- Encodes most of the information from the sequencing reads
- Graph with low memory footprint



  - Complex genomes can be processed on a desktop computer
  - A whole human genome handled with 6 GBytes
  - The raspberry genome can be assembled on a Raspberry Pi

- Each read is split into words named **kmers**
- A kmer has a fixed size K
- Example for K=11

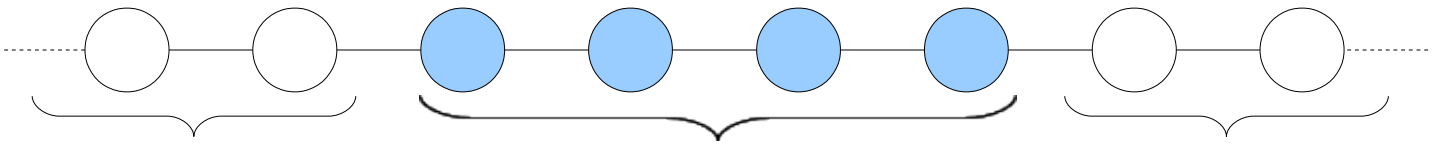
```
>read 1
ACGACGACGTAGTAAACTACGATCGACTAT
```

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

- Count kmers and keep solid kmers (i.e. present at least N times)
- Each solid kmer is inserted as a node of a **de Bruijn graph**
- Nodes A,B are connected <=> suffix(A,K-1) = prefix(B,K-1)
- Two reads sharing (K-1) overlapping will be connected in the de Bruijn graph

```
>read 1
ACGACGACGTAGTAAACTACGATCGACTAT
```

```
>read 2
CTACGATCGACTATTAGTGATGATAGATAGAT
```



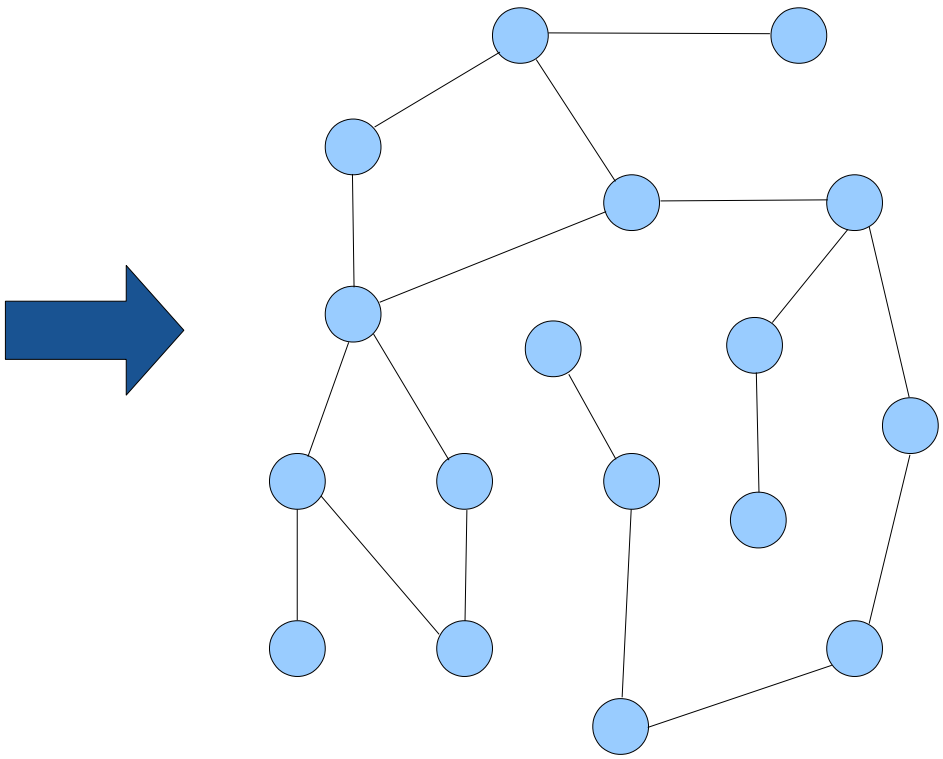
Kmers specific to read 1

Kmers common to read 1 and read 2

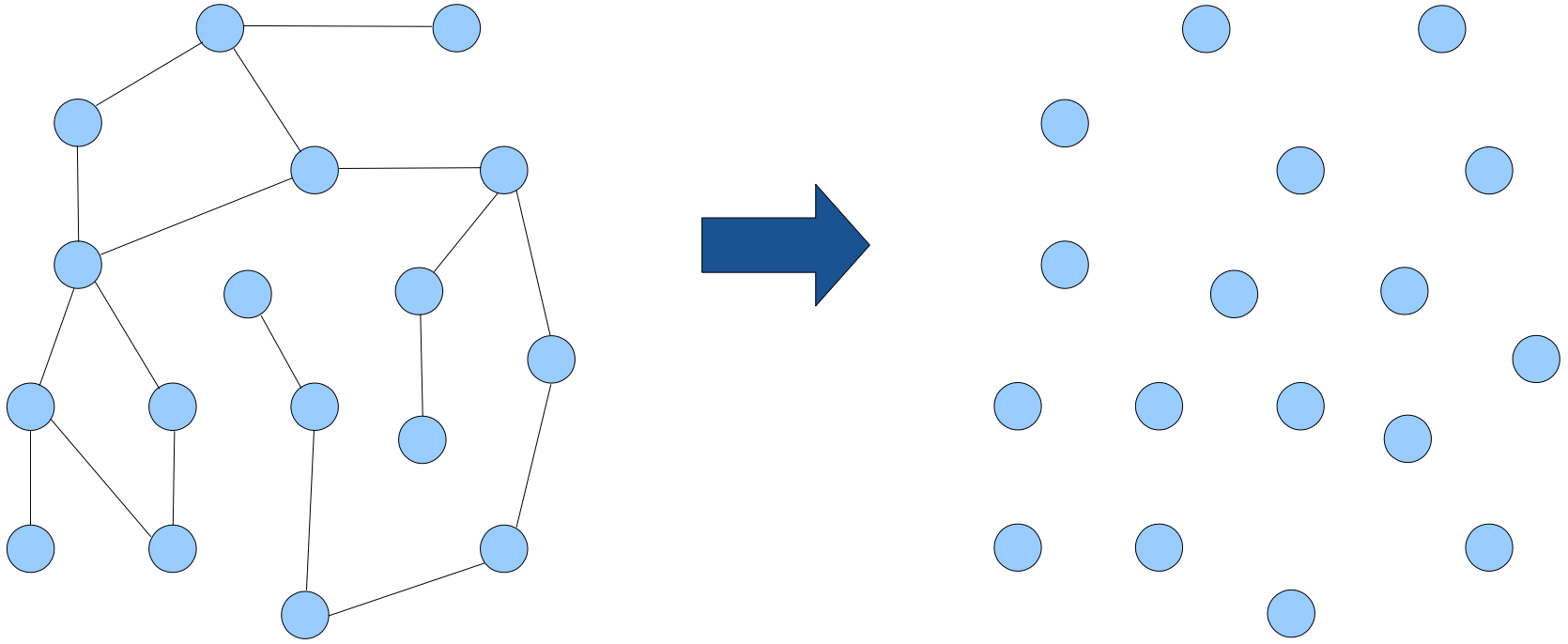
Kmers specific to read 2

- In a nutshell...
  1. Split the reads into kmers
  2. Count kmers and keep the solid ones
  3. Insert the solid kmers into a de Bruijn graph

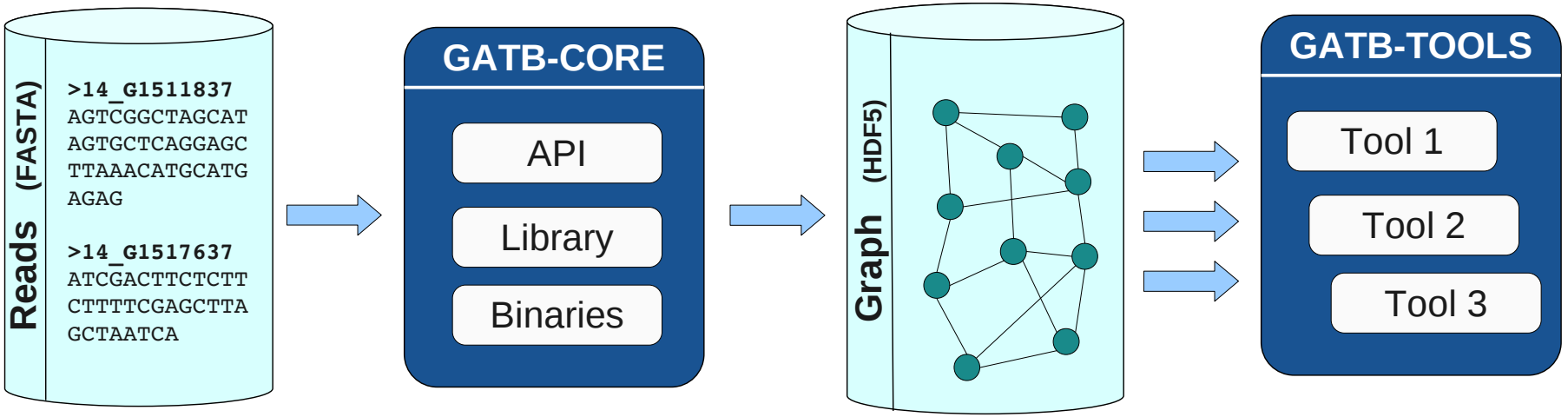
```
>read 1
ACGACGACGTAGACGACTAGCTAGCAATGCTA
GCTAGGATCAAAACTACGATCGACTAT

>read 2
ACTACTACGATCGATGGTCGAGGGCGAGCTAG
CTAGCTGACGCTGCTCGCTCTCTCGCT

...

>read 10.000.000
TCTCCTAGCGCGGCGTATACGCGCTAAGCTAG
CTCTCGCTGCTCGCTAGCTACGTAGCT

...
```



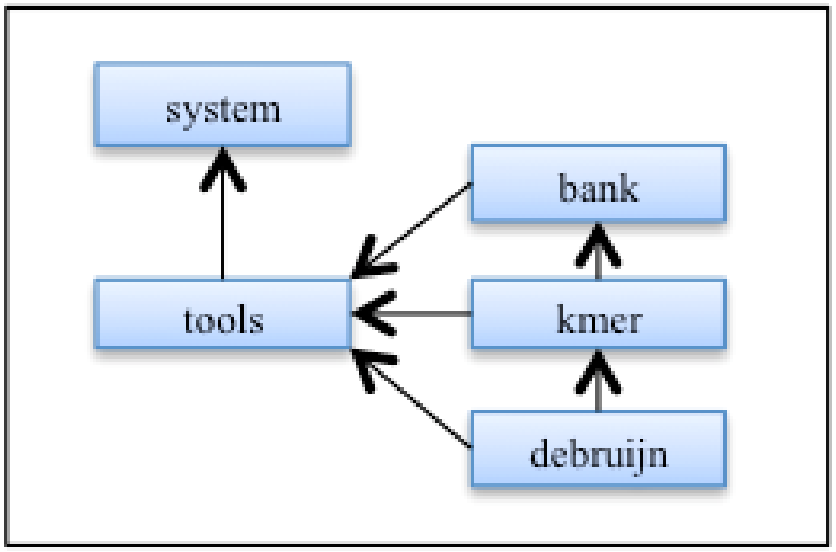
- **GATB-CORE** only stores nodes of the de Bruijn graph, *the edges can be computed on the fly when needed*.
- The nodes are stored in a Bloom filter, a space-efficient structure
- **Two reasons for a very low memory footprint**



- So, **GATB-CORE** transforms a set of reads into a compact de Bruijn graph
- The graph is saved in HDF5 format
- Such a graph can be used by tools developed with the **GATB-CORE** C++ Library



- **GATB-CORE** C++ library
  - Fast development of efficient tools
  - Easy and fast learning curve (full documentation with numerous code samples)
- From developers point of view
  - don't have to bother with the de Bruijn graph construction
  - focus on their own algorithms
- **GATB-CORE** high level API



- 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

- **Details on the graph API** (class Graph)
  - Iterate all the nodes of a graph
  - Iterate branching nodes of a graph
  - Get neighbors nodes/edges from a node
  - Get in/out degree from a node
  - Depth First Search from a node
  - Breadth First Search from a node

- **Two use cases of de Bruijn graph navigation**
  - Assembly
  - Detection of patterns in the graph

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

int main (int argc, char* argv[])
{
    // We load the graph (in HDF5 format) from the given 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;
}
```

Now, it's time to see

GATB-CORE in action

- **GCC (> v4.4) and CMake (> v2.6) must be available**
- **Download the HelloWorld project**

```
wget http://gatb-core.gforge.inria.fr/versions/bin/HelloWorld.tar.gz
```

- **Untar the archive and go to the root directory**

```
tar xvf HelloWorld.tar.gz
cd HelloWorld
```

- **Build the project** **(cf. README.md)**

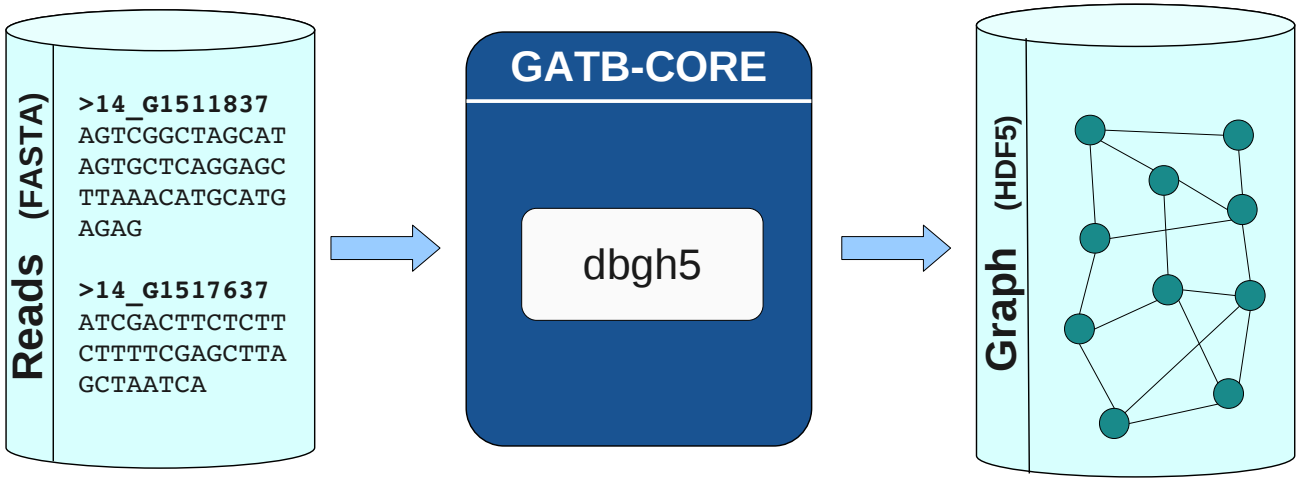
```
mkdir build;  cd build;  cmake ..;  make
```

- **A binary using GATB is generated !**

```
Located in 'tools/tool1'
```

- **Before launching the binary, we need to create a graph**
- **GATB provides the 'dbgh5' command**

```
Built here : ext/gatb-core/bin/dbgh5
```



- **Example of use**

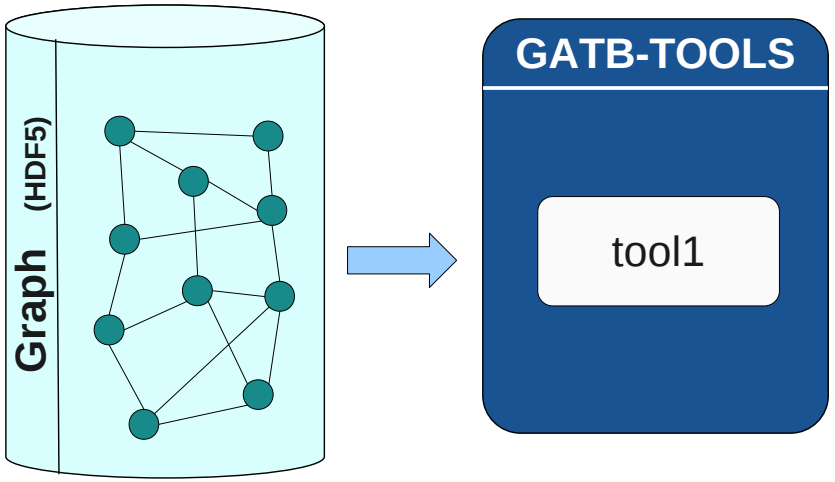
```
ext/gatb-core/bin/dbgh5 -in ../db/somereads.fasta.gz
```

- **Extract of 'dbgh5' output**

```
graph
    dsk
        config
            kmer_size                                 : 31
            abundance                                 : 3
            sequence_number                           : 395135
            kmers_number                              : 149422570
            nb_passes                                 : 1
            nb_partitions                             : 40
            nb_bits_per_kmer                          : 64
            nb_cores                                  : 8
        stats
            kmers_nb_solid                            : 3873602
    bloom
        stats
            nb_hash                                   : 4
            nbits_per_kmer                            : 6.034370
    debloom
        stats
            nbits_per_kmer                            : 8.684544
    branching
        stats
            nb_branching                              : 56601
            percentage                                : 1.5
```

- **Now, we can launch our 'tool1' binary**

```
tools/tool1 somereads.fasta.h5
```



- **Our tool uses the graph as input**
- **We have no more access to the reads !**

- **We can check the number of nodes in the graph**

```
tools/tool1 somereads.fasta.h5 | wc -l
```

- **The graph file is in HDF5 format**
  - HDF5 is used as a container
- **The graph file holds several information sources**
  - Solid kmers with abundances
  - Bloom filter, debloom information, etc...
- **We can use HDF5 tools to dump contents of the file**

```
./ext/gatb-core/bin/h5dump -H somereads.fasta.h5
```

- **BUT... we don't need to know the graph file content**

  We directly use the GATB-CORE C++ API

TESTING THE SAMPLE PROGRAM

- **So far so good, our mighty 'tool1' dumps all the nodes of the graph !**
- **Now, we can pick a node and check that it is indeed in one (or several) reads**
  - Copy the bank and unzip it
  - Take one node dumped by 'tool1'
  - Check with the 'grep' command that this node is in at least one read
- **Should it work in every cases ? Are we that lucky ?**

- **Ouch, some nodes given by 'tool1' are not in any read...**
- **Don't panic, it's not a bug because :**
  - When reads are cut into kmers, we don't know which strand has to be considered
  - So, each node (ie. solid kmer) in the graph may be seen in both strands ; in other words, the de Bruijn graph is bi-directed
  - By convention, a node is encoded as the smallest (lexicographically) of the kmer and its reverse complement

- **For a node (given by 'tool1') not present in the reads :**
  - Compute its reverse complement
    (hint : go to http://www.bioinformatics.org/sms/rev_comp.html)
  - Check that the reverse complement is in the reads
- **Everything should be ok now**
- **It's time to improve 'tool1' and learn a little bit about the GATB-CORE API**

- **Have a look at the reference API here :**
  - http://gatb-core.gforge.inria.fr
  - Reachable from http://gatb.inria.fr (see 'API Documentation' on the right)
- **Use the 'search' widget on the right/top**
  - Type 'Graph' and select the first item
- **Select the 'iterator' method (used in main.cpp)**
- **Select the 'Node' class**
  - There is a 'abundance' attribute, giving the number of times the kmer of the node is present in the reads
- **Let's go and modify our code to display the nodes abundances**

- **Exercise 1 : dump nodes abundances**
- **Source code location of 'tool1'**
  - tools/tools1/src/main.cpp

```
// We loop each node. Note the structure of the for loop.
for (it.first(); !it.isDone(); it.next())
{
   // A shortcut: use a local variable for the current node
   Node node = it.item();

   // We dump the required information
   std::cout
      << graph.toString (node)
      << " "
      << node.abundance   //  we display the node's abundance
      << std::endl;
}
```

- **Launch 'tool1' with our first modification**
- **Take a node present in the reads** **(like we did before)**
- **Grep the node against the reads**
- **Count the number of lines found by grep**
  - It should be the same value as the abundance returned by 'tool1'

- **Exercise 2 : dump reverse complement**

- **One minute ago, we used a web site to compute reverse complement**

- **Let's do it with GATB-CORE :**

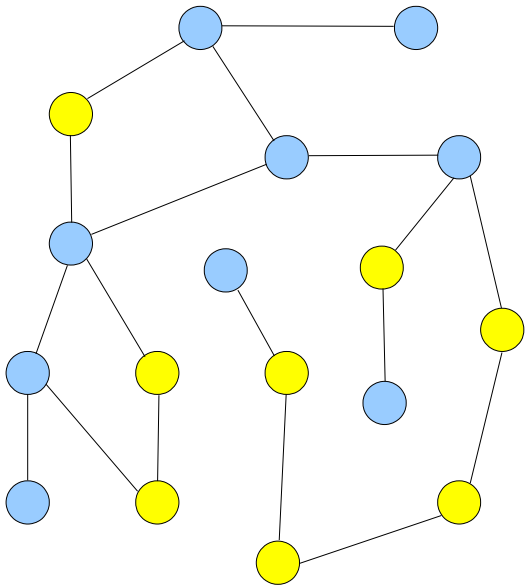
  - In the ref API, search for 'reverse' and select the reverse method for a Node
  - This method returns a node, so we can dump it with the Graph::toString method
  - Modify 'tool1' to dump the reverse complement for each node
  - Now, check that any node or its reverse complement is in the reads

Solution 2

```
// We loop each node. Note the structure of the for loop.
for (it.first(); !it.isDone(); it.next())
{
   // A shortcut: use a local variable for the current node
   Node node = it.item();

   // We dump the required information
   std::cout
       << graph.toString (node)
       << " "
       << graph.toString (graph.reverse(node));
       << std::endl;
}
```

Exercise 3 : dump branching nodes

- Have a look first at Graph::iterator method in API ref
- Modify 'Node' into 'BranchingNode' in the main.cpp file
- Launch 'make' again
- Launch 'tools/tool1 somereads.fasta.h5' again
- Launch 'tools/tool1 somereads.fasta.h5 | wc -l' again



- **Branching nodes in blue, simple nodes in yellow**
- **In general, de Bruijn graphs have much more simple nodes than branching nodes**

Solution 3

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

// We loop each branching node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    std::cout << graph.toString (node) << std::endl;
}
```

A few tips...

- There is a tool 'dbginfo' that dumps information about a graph file ; it dumps the same information as the one displayed during 'dbgh5' execution. You can also do it in your code with the Graph::getInfo method

  (hint : cout << graph.getInfo();)

- You can always learn a library through its reference API but it is often easier to learn it by copying small examples. GATB-CORE provides many snippets to ease the beginners' life

  http://gatb-core.gforge.inria.fr/snippets_page.html

- In this session, we started by a 'HelloWorld' archive that encapsulates all the required material. It eases the way to begin a new project but for advanced usage, it is still possible to customize many things to fit your needs.

- **So far, we iterated all nodes in an order that doesn't represent the graph topology**
  - Nodes order in the HDF5 graph file has no link with its topology
- **Now, we want to start from one node and want to go through a real path of the graph**
- **The Graph API provides several methods to get neighborhoods, for instance :**
  - `Graph::successors`
  - `Graph::predecessors`
- **Have a look to the ref API and to the Graph snippets**

- **Exercise 4 : getting successors of a node**

- **Now, we want to dump all the successors for each node of the graph**

  - Use the Graph::successors method
  - For each successor, dump it with the Graph::toString method

- **Important :** the GATB-CORE structure doesn't allow to retrieve abundance for neighbors nodes, so don't dump the 'abundance' attribute for successors

- **You should check that a lot of nodes have only one successor**

  - Modify the code to dump information only for nodes that have more than one successor for instance

Solution 4

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    std::cout << graph.toString (node) << std::endl;

    // We get the successors of the current node
    Graph::Vector<Node> successors = graph.successors<Node> (node);

    for (size_t i=0; i<successors.size(); i++)
    {
        std::cout << "   " << graph.toString (successors[i]) << std::endl;
    }
}
```

- **Exercise 5 : getting predecessors of a node**
- **Very similar to the previous modification**
  - Use the Graph::predecessors method
  - For each successor, dump it with the Graph::toString method
- **Of course, you should check that a lot of nodes have only one predecessor**
  - Modify the code to dump information only for nodes that have more than one predecessor for instance

Solution 5

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    std::cout << graph.toString (node) << std::endl;

    // We get the predecessors of the current node
    Graph::Vector<Node> predecessors = graph.predecessors<Node> (node);

    for (size_t i=0; i<predecessors.size(); i++)
    {
        std::cout << "   " << graph.toString (predecessors[i]) << std::endl;
    }
}
```

- **Exercise 6 : tell whether a node is branching or not**
- **A node is simple if it has exactly one successor and one predecessor ; it is branching otherwise**
- **Use Graph::successors and Graph::predecessors to check that a node is branching**
  - Hint : use the 'size' method of the Graph::Vector type
- **Use Graph::isBranching to check that you have the same result**
  - You can search 'isBranching' in the ref API
- **Compute the percentage of branching nodes**

Solution 6

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the predecessors and successors of the current node
    Graph::Vector<Node> predecessors = graph.predecessors<Node> (node);
    Graph::Vector<Node> successors   = graph.successors  <Node> (node);

    if (predecessors.size()==1 && successors.size()==1)
    {
        std::cout << “SIMPLE NODE “ << graph.toString (node) << std::endl;
    }
    else
    {
        std::cout << “BRANCHING NODE “ << graph.toString (node) << std::endl;
    }
}
```

- **Exercise 7 : get edges between nodes**
- **So far, we got neighbors (successors and predecessors of a node) as nodes**
- **We can have further information by getting Edge objects**
  - Search for 'Edge' in the ref API
  - Have a look to the available attributes
- **Change the code to get Edge objects**
  - Use Graph::successors with the Edge template
  - Dump the 'from' and 'to' nodes
  - Dump also the transition nucleotide

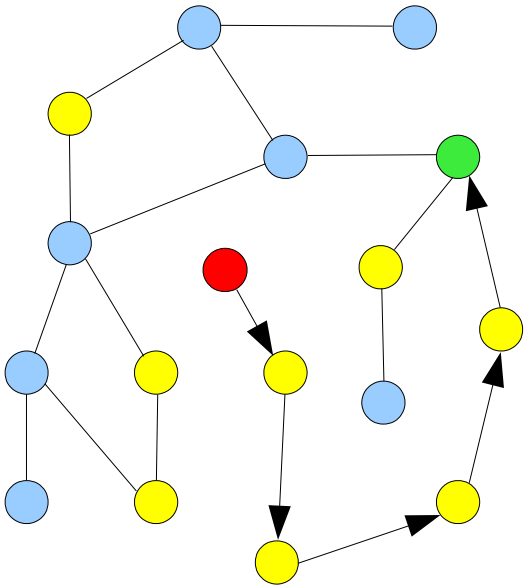
Solution 7

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the successors of the current node
    Graph::Vector<Edge> successors = graph.successors <Edge> (node);

    for (size_t i=0; i<successors.size(); i++)
    {
        std::cout << “from “ << graph.toString (successors[i].from) << “ “
                  << “to “   << graph.toString (successors[i].to)   << “ “
                  << “nucleotide “ << successors[i].nt
                  << std::endl;
    }
}
```

- **Exercise 8 : get neighbors of branching nodes**

- **We can get branching neighbors of a branching node**

  - A path is traversed in the graph from the starting branching node until all branching successors nodes are reached
  - Graph::successors can be used with the template BranchingNode



- **Branching nodes in blue, simple nodes in yellow**
- **Current node in red**
- **Branching successors in green**
  (only one here)

Solution 8

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the branching successors of the current node
    Graph::Vector<BranchingNode> successors =
        graph.successors <BranchingNode> (node);

    for (size_t i=0; i<successors.size(); i++)
    {
        std::cout << “from “ << graph.toString (node) << “ “
                  << “to “   << graph.toString (successors[i].to)
                  << std::endl;
    }
}
```

- **Exercise 9 : get edges between branching nodes**
- **An edge between two branching nodes is called a simple path**
- **An additional attribute gives the number of nodes between the source and the branching successor**
  - Search for 'BranchingEdge' in the ref API
  - Have a look to the available attributes
- **Change the code to get BranchingEdge objects**
  - Iterate all branching nodes
  - Use Graph::successors with the BranchingEdge template
  - Dump the 'from' and 'to' nodes
  - Dump the number of nucleotides between 'from' and 'to'

Solution 9

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the branching successors of the current node
    Graph::Vector<BranchingEdge> successors =
        graph.successors <BranchingEdge> (node);

    for (size_t i=0; i<successors.size(); i++)
    {
        std::cout << “from “ << graph.toString (successors[i].from) << “ “
                  << “to “   << graph.toString (successors[i].to)   << “ “
                  << “path length “ << successors[i].distance
                  << std::endl;
    }
}
```

- **Exercise 10 : get the longest simple path**

- **Simple paths have a special role in assembly process**
  - They represent kind of “atoms” in the assembly process
  - Full assembly tries to link simple path between them (with potential heuristics)

- **Change the code to compute the length of the longest path in the graph**
  - Iterate all branching nodes
  - Use Graph::successors with the BranchingEdge template
  - Remove previous nodes information dump to console
  - At the end of the iteration, dump the size of the longest simple path

Solution 10

```
size_t maxLen = 0;

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

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the branching successors of the current node
    Graph::Vector<BranchingEdge> successors =
        graph.successors <BranchingEdge> (node);

    for (size_t i=0; i<successors.size(); i++)
    {
        if (maxLen < successors[i].distance)
        {
            maxLen = successors[i].distance;
        }
    }
}

std::cout << “maximum simple path length : “ << maxLen << std::endl;
```

- **Exercise 11 : progress information**

- **During processing of huge graphs, it would be nice for user to have progress feedback**

- **GATB-CORE provides this feature through specific iterators**

- **We previously used a Graph::Iterator ; we can use a ProgressGraphIterator**
  - Go to http://gatb-core.gforge.inria.fr/snippets_graph.html
  - Look for ProgressGraphIterator in one snippet
  - Replace our current iterator with this ProgressGraphIterator class
  - Launch the longest simple path search and see the console

Solution 11

```
size_t maxLen = 0;

// We get an iterator for all branching nodes of the graph.
ProgressGraphIterator<BranchingNode,ProgressTimer> it
     (graph.iterator<BranchingNode>(), “running”);

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the branching successors of the current node
    Graph::Vector<BranchingEdge> successors =
        graph.successors <BranchingEdge> (node);

    for (size_t i=0; i<successors.size(); i++)
    {
        if (maxLen < successors[i].distance)
        {
            maxLen = successors[i].distance;
        }
    }
}

std::cout << “maximum simple path length : “ << maxLen << std::endl;
```

- **Exercise 12 : dot file generation**

- **Example of a .dot file**

```
digraph mygraph {
    ACGGCT -> CGGCTG ;
    CGGCTG -> GGCTGA ;
}
```

- **Use GATB API to generate a .dot file**

  - Iterate all nodes
  - Get neighbors of the kmer, not the node (see Graph::neighbors)
  - Dump information in a output .dot file
  - Test (bank of 1 or 2 short sequences), with dbgh5 options as follow :

    -kmer-size 6 -abundance 1

- **Command to build a .pdf from a .dot file**

  - `dot -Tpdf output.dot -o output.pdf -Kcirco`

BANK

```
>seq1
CTACAGCAGCTAGTTC
```

- **Remember !**
  Node value :=
  min (forward, revcomp)

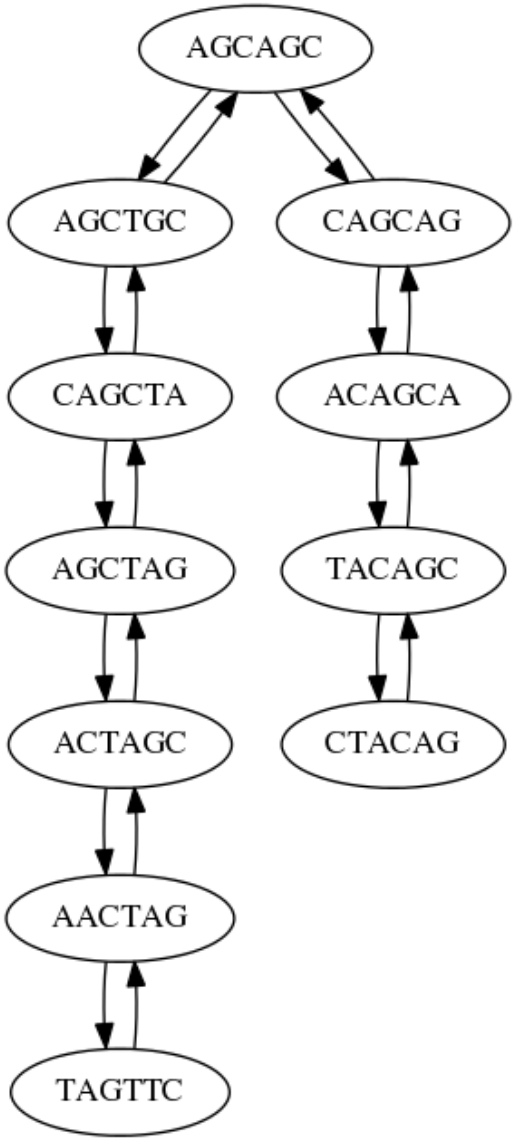
We use

A=0
C=1
T=2
G=3

6-mers of seq 1

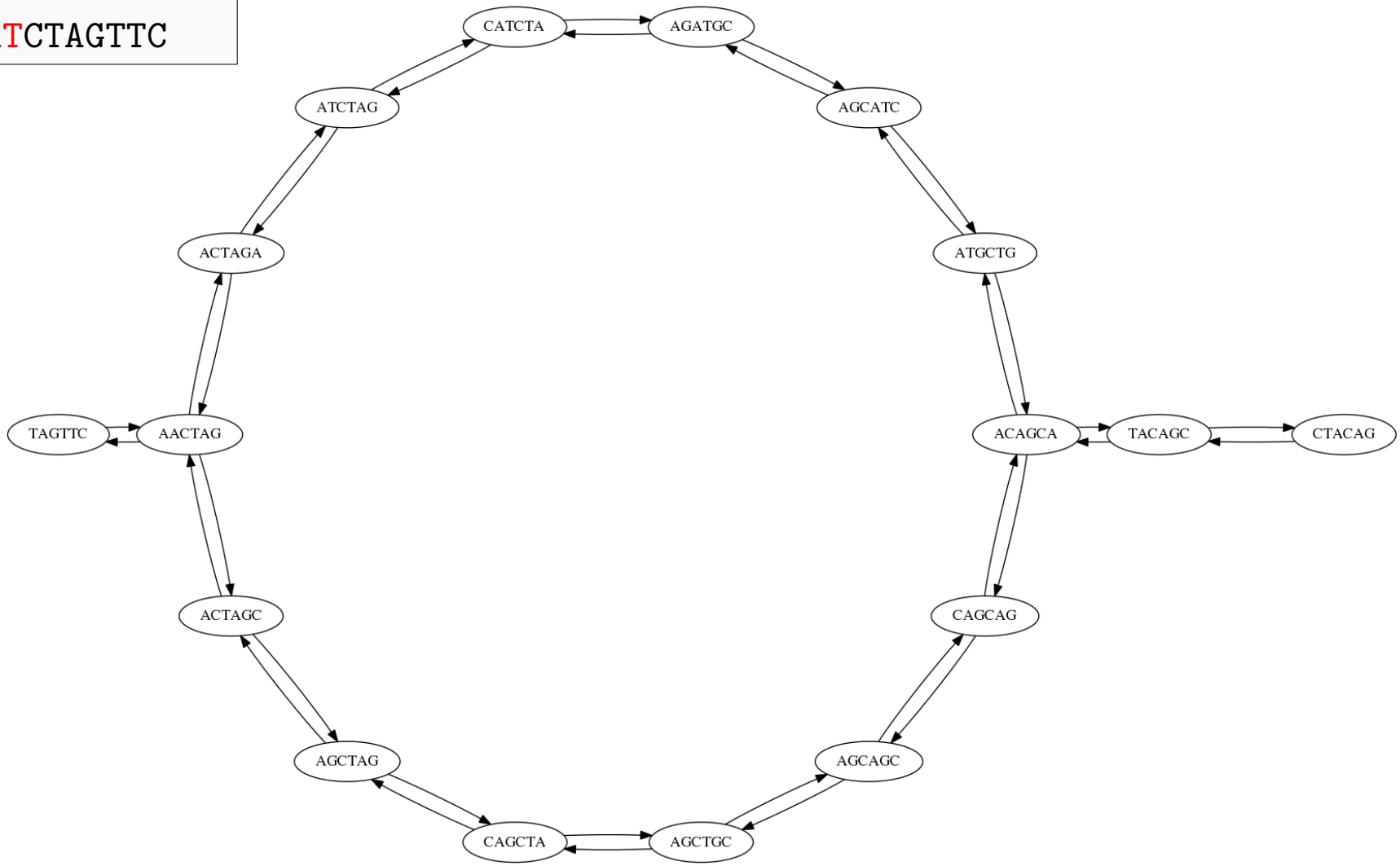
FORWARD	REVCOMP
CTACAG	CTGTAG
TACAGC	GCTGTA
ACAGCA	TGCTGT
CAGCAG	CTGCTG
AGCAGC	GCTGCT
GCAGCT	AGCTGC
CAGCTA	TAGCTG
AGCTAG	CTAGCT
GCTAGT	ACTAGC
CTAGTT	AACTAG
TAGTTC	GAACTA

Created from our generated .dot file



BANK

```
>seq1
CTACAGCAGCTAGTTC
>seq2
CTACAGCATCTAGTTC
```



- **Generate a .dot file for branching nodes only**
  - Iterate all branching nodes
  - Get branching successors
  - Dump information in a output .dot file
  - Test (bank of 1 or 2 short sequences), with dbgh5 options as follow :

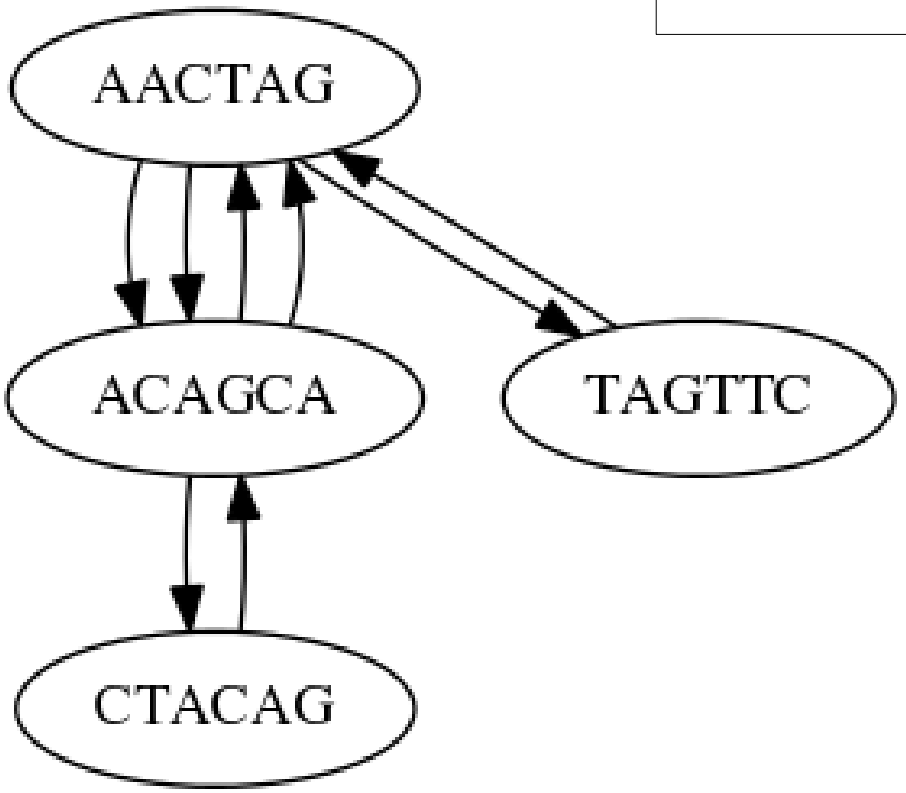
    -kmer-size 6 -abundance 1

- **Command to build a .pdf from a .dot file**
  - `dot -Tpdf output.dot -o output.pdf -Kcirco`

BANK

```
>seq1
CTACAGCAGCTAGTTC
>seq2
CTACAGCATCTAGTTC
```

Note !
Double path between
AACTAG and ACAGCA



Solution 12 (nodes)

```
std::cout << “digraph mygraph {” << std::endl ;

// We get an iterator for all nodes of the graph.
ProgressGraphIterator<Node,ProgressTimer> it (graph.iterator<Node>(), “running”);

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the neighbors of the kmer of the current node
    Graph::Vector<Node> successors = graph.neighbors <Node> (node.kmer);

    for (size_t i=0; i<successors.size(); i++)
    {
        std::cout << graph.toString (node.kmer)
                  << “ -> “
                  << graph.toString (successors[i].kmer)
                  << “;” << std::endl;
    }
}

std::cout << “}“ << std::endl;
```

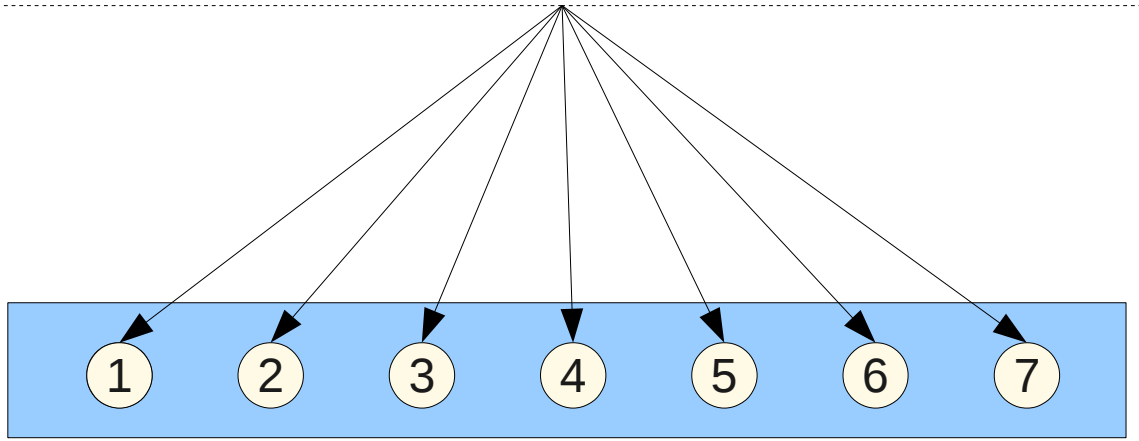
Solution 12 (branching nodes)

```
std::cout << “digraph mygraph {” << std::endl ;

// We get an iterator for all nodes of the graph.
ProgressGraphIterator<BranchingNode,ProgressTimer> it
    (graph.iterator<BranchingNode>(), “running”);

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the neighbors of the kmer of the current node
    Graph::Vector<BranchingNode> successors =
        graph.neighbors <BranchingNode> (node.kmer);

    for (size_t i=0; i<successors.size(); i++)
    {
        std::cout << graph.toString (node.kmer)
                  << “ -> “
                  << graph.toString (successors[i].kmer)
                  << “;” << std::endl;
    }
}

std::cout << “}“ << std::endl;
```

- **Exercise 13 : speedup with multithreading**
- **In the longest simple path example, we start from each branching node and look for branching successors**
- **This leads to independant searches in the graph; if our computer has N cores, we could run N searches in parallel**
- **GATB-CORE provides a way to easilly parallelize iterations (including nodes iteration)**
  - Concept of Dispatcher that takes an iterator as input, creates N thread ; each thread receives and processes iterated items from the iterator through a functor object.
  - We need to modify our previous code a little bit.

BEFORE

```
// MAIN THREAD
{

    // we iterate nodes
    for (it.first(); !it.isDone(); it.next())
    {
        // it.item() is the currenly iterated node
        // do some process on it
    }
};
```



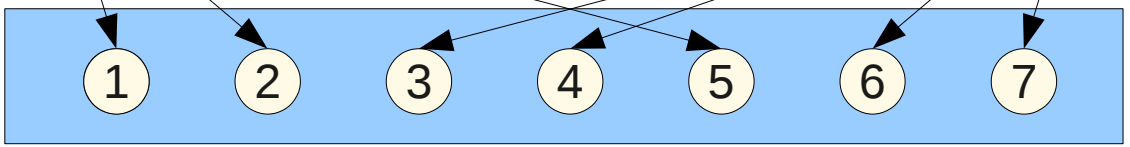
Iterated nodes

AFTER

```
// MAIN THREAD
{
    int N = 2;
    // we use a dispatcher that
    // will create N threads

    Dispatcher(N).iterate (iter, Functor());

    // here, the dispatcher waits for
    // the end of the N threads it just created
};
```

```
// THREAD 1
void Functor::operator() (Node node)
{
    // do something with the node
};
```

```
// THREAD 2
void Functor::operator() (Node node)
{
    // do something with the node
};
```



Iterated nodes

Functor definition

```
// We define a functor that computes BranchingEdge objects from a given
// BranchingNode object and keeps the local maximum distance among them
// The global maximum distance is then set during destruction of the functor.

struct LongestPathFunctor
{
    const Graph& graph;    int& globalMaxValue;     int localMaxValue;

     LongestPathFunctor (const Graph& graph, int& value)
        : graph(graph), globalMaxValue(value) {}

    ~LongestPathFunctor ()
    {
        if (localMaxValue > globalMaxValue) { globalMaxValue = localMaxValue; }
    }

    void operator() (const BranchingNode& bnode)
    {
        Graph::Vector<BranchingEdge> ngb = graph.successors<BranchingEdge> (bnode);

        for (size_t i=0; i<ngb.size(); i++)
        {
            if (ngb[i].distance>localMaxValue) { localMaxValue = ngb[i].distance; }
        }
    }
};
```

- **Dispatcher call**

```
int nbCores     = 4;
int longestPath = 0;

// We use a dispatcher with 4 cores. It will create 4 instances of
// LongestPathFunctor that will receive and process branching edges
// through the operator() method
Dispatcher(nbCores).iterate (
    it,
    LongestPathFunctor (graph, longestPath)
);

// Now, the LongestPathFunctor 4 instances have been created,
// runned and deleted. Each one computed a local maximum distance
// and each one updated the global longest path during destruction

// Now, 'longestPath' should be the global longest simple path
cout << “longest simple path : “ << longestPath << endl;
```

- **What have we learnt so far ?**
  - Load a de Bruijn graph from a HDF5 file
  - Iterate nodes (simple or branching ones) and use some of their attributes
  - Get neighbors of a given node (successors, predecessors)
  - Get edges from a given node to its neighbors
  - Get progress information during nodes iteration
  - Speed up nodes iteration by using multithreading

- **Now, we have minimal knowledge to build small programs that can provide useful services**

Exercise 14 : detect patterns in the graph

Why is the de Bruijn graph structure interesting ?

- Encodes most (but not all) of the reads information
- Smart implementations of this structure can have low memory footprint
- Allows to find information not directly reachable from the reads

Applications of the de Bruijn graphs

- De novo assembly
- Search of biological patterns (like Single Nucleotide Polymorphism)

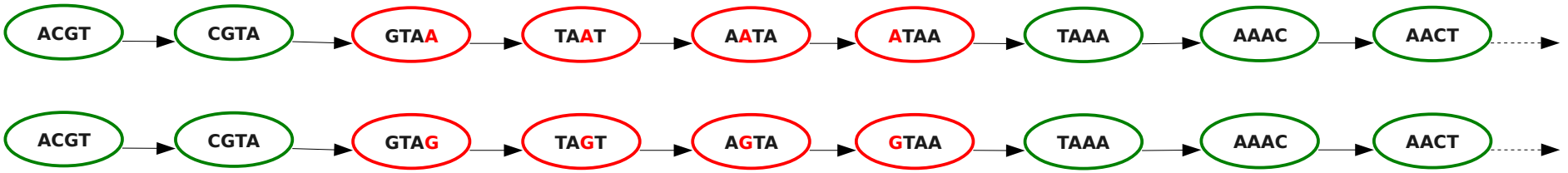
Our next code improvement : discovering SNPs

- **Example of SNP between two reads**

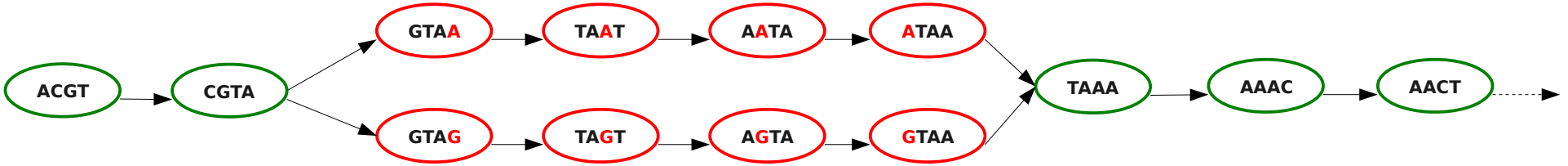
```
>read 1
ACGTAATAAACTACGATCGACT
```

```
>read 2
ACGTAGTAAACTACGATCGACT
```

- **kmers of size 4 will be**

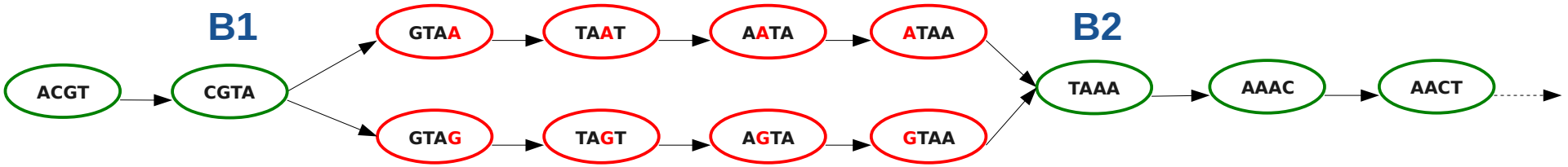


- **The de Bruijn graph is therefore**



A (simple) SNP is a « bubble » in the de Bruijn graph

- The bubble has two simple paths of size K (K=4 here)
- Its bounds are two branching nodes, B1 and B2



One way to detect SNP :

- Iterate each branching node B
- Get branching successors of B
- Check that there are 2 successors and that it is the same branching node
- If found, dump the SNP to the console (ie. 2 sequences showing the SNP)
- For testing, create a FASTA bank with two sequences with a SNP

Solution 14

```
 // We get an iterator for all nodes of the graph.
ProgressGraphIterator<BranchingNode,ProgressTimer> it
    (graph.iterator<BranchingNode>(), “SNP”);

// We loop each node.
for (it.first(); !it.isDone(); it.next())
{
    Node node = it.item();

    // We get the successors of the current node
    Graph::Vector<BranchingEdge> successors = graph.successors <BranchingEdge> (node);

    // We look whether this is a simple SNP or not
    if (successors.size()==2 && successors[0].to==successors[1].to)
    {
        std::cout << std::endl << "SNP" << std::endl;

        std::cout << graph.toString(node) << ascii(successors[0].nt)
                  << graph.toString (successors[0].to)  << std::endl;

        std::cout << graph.toString(node) << ascii(successors[1].nt)
                  << graph.toString (successors[1].to)  << std::endl;
    }
}
```

Questions ?