Licence 2 informatique

Atelier bioinformatique

Hierarchical clustering

Emmanuelle Becker, Olivier Dameron, Wesley Delage, Marine Louarn

May 13, 2022

Version 1.18
The latest version of this document is available at:
`https://gitlab.com/odameron/hemoglobinClustering`

1 Objective

This project's goal is to **compare a gene and its associated proteins across multiple species**. The idea underlying this comparison is that DNA changes over evolution, so that the closest species should have the most similar genes[1] (cf. section 1.1 and Fig. 1). We will see that identifying the groups of similar genes is a computational data analysis problem called clustering (section 1.2). Eventually, we will lay out an approach for performing hierarchical clustering in order to identify groups of similar genes or similar proteins and apply to real data about hemoglobin (cf. section 1.3 and Fig. 2).

1.1 Context: from genes to proteins and the influence of mutations

Genes are transcribed in messenger RNA, which are translated into proteins (chains of amino-acids). During translation, each group of three consecutive nucleotide (a codon) is associated to an amino-acid. The chain of amino-acids forms a protein.

A mutation is a permanent change of DNA. It can be either:

- an insertion of a nucleotide;
- a deletion of a nucleotide;
- a permutation where a nucleotide is replaced by another one.

[1] `https://en.wikipedia.org/wiki/Phylogenetics`

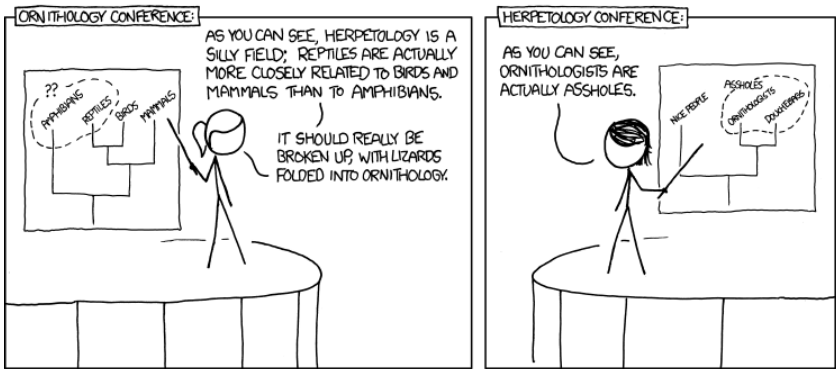

ORNITHOLOGY CONFERENCE:
??
AMPHIBIANS
REPTILES
BIRDS
MAMMALS
AS YOU CAN SEE, HERPETOLOGY IS A SILLY FIELD; REPTILES ARE ACTUALLY MORE CLOSELY RELATED TO BIRDS AND MAMMALS THAN TO AMPHIBIANS.
IT SHOULD REALLY BE BROKEN UP, WITH LIZARDS FOLDED INTO ORNITHOLOGY.
HERPETOLOGY CONFERENCE:
AS YOU CAN SEE, ORNITHOLOGISTS ARE ACTUALLY ASSHOLES.
NICE PEOPLE
ASSHOLES
ORNITHOLOGISTS
DOUCHEBAGS

Figure 1: Species classification using dendrograms and (questionable) analysis according to xkcd 867 (`http://xkcd.com/867/`). The dendrograms are the tree-like representations which are typically computed using hierarchical clustering; they group similar items in clusters which can recursively be included into broader clusters. For example, on the left, Reptiles and Birds form a first cluster (they are the most similar). Mammals are then added, forming a broader cluster.

Insertions and deletions typically induce a frameshift that leads to different codons, hence amino-acids. Several codons code for the same amino-acid, so permutations may or may not result in different proteins. In consequence, **a small difference in the DNA may result in no, little or important differences in proteins**.

For several species, we aim to compare their DNA sequence for the hemoglobin gene. The underlying assumption is that the more similar are the sequences for two species, the closer these species should be phylogenetically. Then, as we have seen that small differences in DNA sequences can lead to no, little or important differences in proteins, we aim to check whether the more similar are the proteins for two species, the closer these species are be phylogenetically.

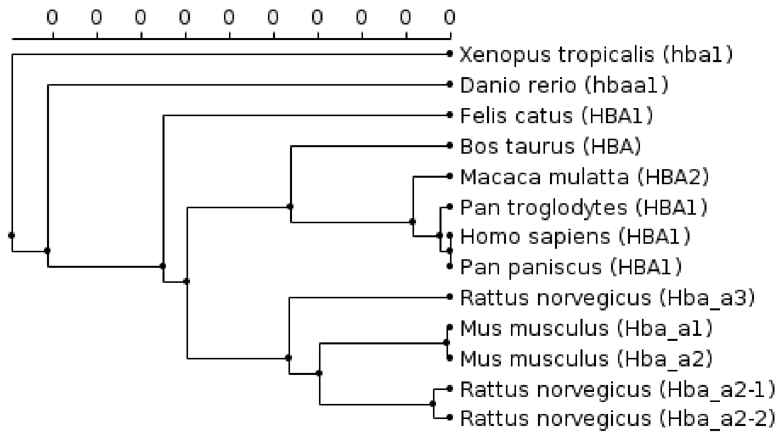
For more information on how to identify the coding regions of a genome, read "*À la recherche des régions codantes*"[2] by François Rechenmann.

1.2 Why is it a clustering problem (and what is clustering, by the way)?

Clustering consists in grouping similar elements. It relies on some **similarity measure** to assess how similar are two elements. These measures are usually in $[0;1]$ (0 for completely dissimilar elements, and 1 for identical elements). Note that similarity is different from distance (similar elements have a distance close to 0 whereas dissimilar elements have a greater distance).

There are many ways of clustering elements. Small clusters are usually highly homogeneous (i.e. composed of very similar elements). Notice that there is an implicit **maximality criterion**, as we would like the cluster to be composed of as many similar elements as possible (otherwise, we would end up with

[2] `https://interstices.info/a-la-recherche-de-regions-codantes/`



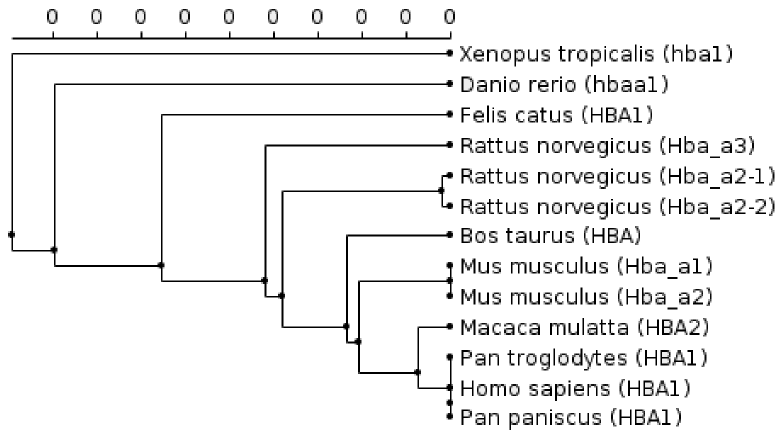


Figure 2: Clustering of hemoglobin according to the gene sequences (top) and the protein sequences (bottom). Sequence distances were calculated after global alignment (cf. section 5.1)

similar elements scattered in several clusters which would give the misleading impression that these elements are not similar).

However, the more elements we add to the cluster, the less homogeneous the cluster becomes, because of its elements' variability. Therefore, we also need to find a balance between many highly homogeneous small clusters, and few heterogeneous large clusters. The first situation fails to convey that elements from different clusters may be similar, and add little value to the original set of elements. The second situation lacks cohesion and also adds little value to the original set of elements.

One way to find the balance is to introduce a **cohesiveness criterion** so that each element of a cluster is more similar to every other element of the cluster than to any element outside the cluster.

At this point, we should obtain a set of clusters that form a partition of the original set of elements.

A cluster can be more similar to another cluster than to a third. For example, once we have created a cluster of humans, a cluster of apes and a cluster of fishes, we may want to convey that the set of humans and the set of apes can be grouped in a superset of mammals, which itself is part of the set of animals, along with the fishes. Note that we switched from comparing elements to comparing sets of elements, but from there on, the process remains the same.

We rely on the **transitive relation of set inclusion** between clusters, that form a partial ordering and leads to **hierarchical clustering**.

Dendrograms are a graphical representation of this hierarchical clustering that combine:

- recursively grouping clusters into more general super-clusters in a tree-like structure;
- displaying how similar the sub-clusters of a super-cluster are by varying the lengths of the branches (short branches denote similar clusters).

For more information (in French) on phylogenetic trees, there is an introductory article on the (excellent) blog `https://bioinfo-fr.net`: "*Les arbres phylogénétiques : construction et interprétation*"[3]

1.3 Project outline

We want to verify whether similar species do have similar genes, and whether these genes produce similar proteins.
The first part of the project consists in implementing a generic analysis workflow on any kind of sequence. The second part is an application of the workflow on hemoglobin.
Section 2 focuses on determining the distance between a pair of genes (resp. of proteins). Section 3 groups similar sequences by performing hierarchical clustering according to their distances. Eventually, section 4 focuses on hemoglobin genes and proteins.

Note that we will limit ourselves to the primary structure of proteins, i.e. the chain of amino-acids they are composed of. Proteins being long chains of amino-acids, their folding results in complex 3D structures which have a major influence on protein functions. Therefore, substituting one amino-acid by another (a small change in the primary structure) may have different outcomes which can be computed (this is usually CPU-intensive) but that remain out of the scope of this project.

2 Compare sequences

This section aims at computing the distance between two sequences. It proposes a crude but easy way for computing this distance so that you can procede quickly to the hierarchical clustering section.

[3]`https://bioinfo-fr.net/les-arbres-phylogeniques-construction-et-interpretation`

As an optional improvement, a more precise approach consists in determining the best match between two sequences according to their global alignment (section 5.1.1), and to use the alignment score for computing an improved distance (section 5.1.2).

The distance between two sequences can be approximated by their proportion of pairwise different elements.

$$distance(\mathtt{s1}, \mathtt{s2}) = \frac{(\sum_{i \in [0;min(|\mathtt{s1}|,|\mathtt{s2}|)[} \delta(\mathtt{s1}, \mathtt{s2}, i)) + abs(|\mathtt{s1}| - |\mathtt{s2}|)}{max(|\mathtt{s1}|, |\mathtt{s2}|)}$$

where

$$\delta(\mathtt{s1}, \mathtt{s2}, i)) = \begin{cases} 0 & \text{if } \mathtt{s1}[i] == \mathtt{s2}[i] \\ 1 & \text{if } \mathtt{s1}[i] \neq \mathtt{s2}[i] \end{cases}$$

Step 1 *Create a class* ***`Sequence`****. Add the constructors and a* ***`public double distance(Sequence otherSeq)`*** *method that computes the distance with another sequence. Overriding* ***`toString()`*** *could have been replaced by a* ***`String getSeq()`*** *accessor.*

```java
public class Sequence {

  protected String seq; // we will need it later in a subclass

  public Sequence() {
    // complete
  }

  public Sequence(String s) {
    // complete
  }

  public Sequence (Sequence s) {
    // complete
  }

  public String toString() {
    return this.seq;
  }

  public double distance(Sequence otherSeq) {
    // returns the distance between the two sequences
  }


  public static void main(String[] args) {
    Sequence seq1 = new Sequence("ATTACG");
    Sequence seq2 = new Sequence("ATATCG");
    Sequence seq3 = new Sequence("ACCCCG");
    Sequence seq4 = new Sequence("GGGGAA");
    Sequence seq5 = new Sequence("TTTACG");
    Sequence seq6 = new Sequence("ATTAC"); // beginning of seq1
    Sequence seq7 = new Sequence("ATATC"); // beginning of seq2

    System.out.println("Comparing same-length sequences");
    System.out.println("dist(seq1,seq1): " + seq1.distance(seq1));
    System.out.println("dist(seq1,seq2): " + seq1.distance(seq2));
    System.out.println("dist(seq2,seq1): " + seq2.distance(seq1));
    System.out.println("dist(seq6,seq7): " + seq6.distance(seq7));
    System.out.println("dist(seq7,seq6): " + seq7.distance(seq6));
    System.out.println("dist(seq1,seq3): " + seq1.distance(seq3));
    System.out.println("dist(seq2,seq3): " + seq2.distance(seq3));
    System.out.println("dist(seq1,seq4): " + seq1.distance(seq4));

    System.out.println("");
    System.out.println("Comparing different-length sequences");
    System.out.println("dist(seq1,seq6): " + seq1.distance(seq6));
    System.out.println("dist(seq6,seq1): " + seq6.distance(seq1));
    System.out.println("dist(seq1,seq7): " + seq1.distance(seq7));
    System.out.println("dist(seq2,seq7): " + seq2.distance(seq7));
  }
}
```

Step 2 *Now that we have a distance between sequences, we can proceed directly to the fun part on hierarchical clustering (section 3). When the clustering works, you can resume to section 5.1 in order to take sequence alignment into account when computing the distance between sequences.*

3 Hierarchical clustering

This section first lays the general principles of hierarchical clustering (section 3.1), and then guides you for implementing a clustering of sequences (section 3.2).

3.1 Principle

3.1.1 Agglomerative vs. divisive approaches

Classifying consists in organizing a set of elements in groups based on the elements' similarities or differences.

Hierarchical clustering is a kind of recursive classification that consists in organizing the set of elements into subsets included in to each others in a tree-like structure. There are two main approaches for determining this organization:

- the **agglomerative approach** (also called ascending) starts by creating one (atomic) cluster for each element, and then iteratively generates new clusters composed of the most similar two, until there only remains one cluster;

- the **divisive approach** (also called descending) starts by gathering all the elements into a single cluster, and then iteratively decompose the clusters into subclusters until each of them is only composed of a single element.

The divisive approach requires more operations than the agglomerative one and is therefore usually longer... except when we only need the most general clusters (e.g. to separate a sample into two groups).

3.1.2 Distance measures between elements and between clusters

For both the agglomerative and ther divisive approaches, clustering depends on two main parameters:

- a **distance measure between elements** (also simply called *distance*). There are several classical ones: euclidian distance, Manhattan distance... In our case, we will consider that the distance between two sequences is the one determined in sections 2 or 5.1.2;

- a **distance measure between clusters** (also called *linkage*) that relies on the *distance* between elements of the two clusters. There are several classical linkage measures: the average of the distances between all the combinations of elements, their maximum, their minimum... In our case, we will consider that the distance between two clusters of sequences is the average of the distances between all the elements of the first cluster and all the elements of the second cluster.

3.2 Class `ClusterOfSequences`

This section aims at implementing the `ClusterOfSequences` class for representing a cluster of `Sequence` instances. A simple cluster is composed of a single instance of `Sequence`. A complex cluster is composed of several sub-clusters which can themselves be either simple or complex clusters. A complex cluster has a tree-like structure where all the leaves are simple clusters. As we will see, there is no need to distinguish the simple and the complex clusters as subclasses of `ClusterOfSequences`.

Initially, a complex cluster is only composed of simple clusters (Fig. 3). After clustering, a complex cluster is composed of sub-clusters that are intermediate complex clusters (Fig. 4)

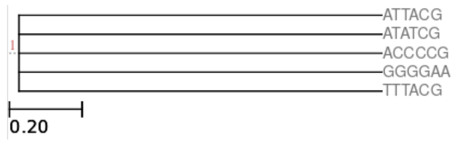


Figure 3: Complex cluster in its initial state. It is composed of five simple sub-clusters, each composed of a sequence.

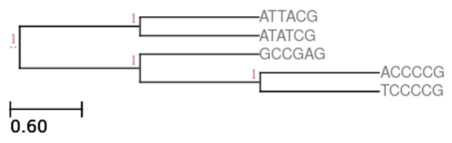


Figure 4: Complex cluster after clustering. It is composed of two intermediate complex sub-clusters. The first is itself composed of two simple clusters (ATTACG and ATATCG). The second is composed of a simple sub-cluster (GCCGAG) and a complex sub-cluster composed of two simple sub-clusters (ACCCCG and TCCCCG).

3.2.1 Initialization

Step 3 *Create a `ClusterOfSequences` class with an attribute `subclusters` that represents the list of its sub-clusters. This list is obvisouly empty for simple clusters, and non-empty for complex clusters. For simplifying the clustering step (when marshalling the sequences composing the cluster), add an attribute `elements` that represents the set of sequences constituting the leaves of the cluster. This list is never empty; it contains a single element for simple clusters, and two elements or more for complex clusters.*

Step 4 *Add the following constructors:*

- *a constructor for simple clusters `ClusterOfSequences(Sequence element)`;*
- *a constructor for complex clusters before clustering `ClusterOfSequences(ArrayList<Sequence> eltList)`;*
- *a constructor for creating a complex cluster by merging two existing clusters (which become its subclusters) `ClusterOfSequences(ClusterOfSequences cluster1, ClusterOfSequences cluster2)`.*

```java
import java.util.ArrayList;

public class ClusterOfSequences {

  private ArrayList<ClusterOfSequences> subClusters;
  private ArrayList<Sequence> elements;

  public ClusterOfSequences(Sequence element) {
    // complete
  }

  public ClusterOfSequences(ArrayList<Sequence> eltList) {
    // complete
  }

  public ClusterOfSequences(ClusterOfSequences cluster1,
                              ClusterOfSequences cluster2) {
    // complete
  }


  private String getNewickIntermediate() {
    // complete
  }

  public String getNewick() {
    return this.getNewickIntermediate() + ";";
  }


  public double linkage(ClusterOfSequences aCluster) {
    // complete
  }

  public void clusterize() {
    //complete
  }

  public static void main(String[] args) {
    Sequence seq1 = new Sequence("ATTACG");
    Sequence seq2 = new Sequence("ATATCG");
    Sequence seq3 = new Sequence("ACCCCG");
    Sequence seq4 = new Sequence("GCCGAG");
    Sequence seq5 = new Sequence("TCCCCG");

    ArrayList<Sequence> listSeq = new ArrayList<Sequence>(5);
    listSeq.add(seq1);
    listSeq.add(seq2);
    listSeq.add(seq3);
    listSeq.add(seq4);
    listSeq.add(seq5);

    ClusterOfSequences bioCluster = new ClusterOfSequences(listSeq);
    System.out.println(bioCluster.getNewick());
    bioCluster.clusterize();
    System.out.println(bioCluster.getNewick());
  }

}
```

Step 5 *In `ClusterOfSequences`' `main(...)` main method, create five instances of simple clusters `cl1` to `cl5` initialized respectively with `seq1` to `seq5`, and an instance of complex cluster `bioCluster` initialized with `a list of 5 sequences`.*

3.2.2 Visualization

The Newick format[4] provides a straightforward representation of trees and dendrograms, and is supported by most visualization tools. You can use the Tree Viewer web server[5] or T-REX[6] or the dedicated softwares FigTree[7], dendroscope[8] (free use in an academic context; getting a licence is not required for the basic functions). FigTree seems to give the best results.

The dendrogram from Fig. 3 can be represented by
`((ATTACG,ATATCG),(GCCGAG,(ACCCCG,TCCCCG)));`.

NB : for visualizing dendrograms, we could as well have used the R functions via the Java–R binding, but it is more complicated, and writing Newick files makes for an interesting exercice anyway.

Step 6 *Add a `getNewick()` method to the class `ClusterOfSequences` that returns a string representing the dendrogram in the Newick format. Because of the final semicolon, you may need to introduce an intermediate function (aptly named `getNewickIntermediate()`). For marshalling the tree, you will make your life easier by considering a recursive approach (but this is not mandatory). Should these methods' visibility be public, protected or private?*

Step 7 *Generate a Newick representation of `bioCluster` and check (for example with T-REX or dendroscope) whether you get something similar to Fig. 3.*

3.2.3 Clustering

Step 8 *Add a method `linkage(ClusterOfSequences anotherCluster)` that returns the distance between the current cluster and `anotherCluster`. Choosing the average of the distance for each combination of sequence from each cluster is probably the easiest solution.*

Step 9 *In `ClusterOfSequences`'s `main(...)` method, check whether the distances between `cl1`, `cl2` and `cl3` are what you expect them to be (check the six combinations).*

Step 10 *In `ClusterOfSequences`'s `main(...)` method, create the complex cluster `cl6` by merging `cl3` and `cl5` and check whether its distance with `cl4` is what you expect.*

Step 11 *Add a method `clusterizeAgglomerative()`. Perform clustering on `bioCluster` and display the result as a Newick string. It should be similar to Fig. 4.*

4 Application to hemoglobin

The `data` directory constains the nucleotide sequences of hemoglobin for several species in fasta format. They were downloaded from Ensembl[9].

[4] `http://evolution.genetics.washington.edu/phylip/newicktree.html`
[5] `http://etetoolkit.org/treeview/`
[6] `http://www.trex.uqam.ca/`
[7] `http://tree.bio.ed.ac.uk/software/figtree/`
[8] `http://ab.inf.uni-tuebingen.de/software/dendroscope/`
[9] `https://www.ensembl.org/`

4.1 Read the gene sequence from a fasta file

As any protein, hemoglobin is composed of several hundreds of nucleotides. Using directly the `Sequence` class from section 3 would result in correct but messy display.

Step 12 *Create a class `SequenceLabeled` that extends `Sequence` and has an additional `private String label` attribute that will be used for display. Adapt the constructors and override the `toString()` method so that it returns the label.*

Step 13 *Add a new constructor `SequenceLabeled(File f, String l) throws FileNotFoundException` that creates an instance of `SequenceLabeled` by reading the sequence from a fasta file (you can use the `readFasta(File f)` static method from the `Utils` class.*

4.2 Cluster of gene sequences

Step 14 *The class `Utils`'s main method performs a hierarchical clustering of the hemoglobin nucleotide sequences.*

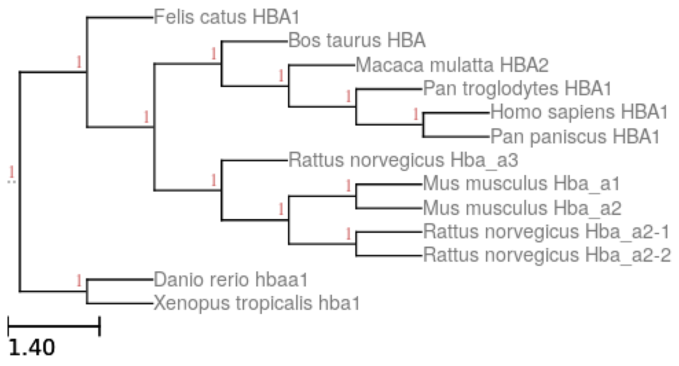


Figure 5: Cluster of hemoglobin genes according to their nucleotide sequences. This diagram was generated using the crude sequence distance, but the clusters are sensible (hominids together, rats and mice together, etc.)

4.3 Cluster of protein sequences

Step 15 *The class `Utils` contains a `codon2aa(...)` method that returns the amino-acids associated to a codon. Use it to write a `public static String nucleotidesToAminoAcids(String nuclSeq)` method that converts a sequence of DNA nucleotides into the corresponding sequence of amino-acids*

Step 16 *Complete the class `Utils`'s main method to perform a hierarchical clustering of the hemoglobin protein sequences.*

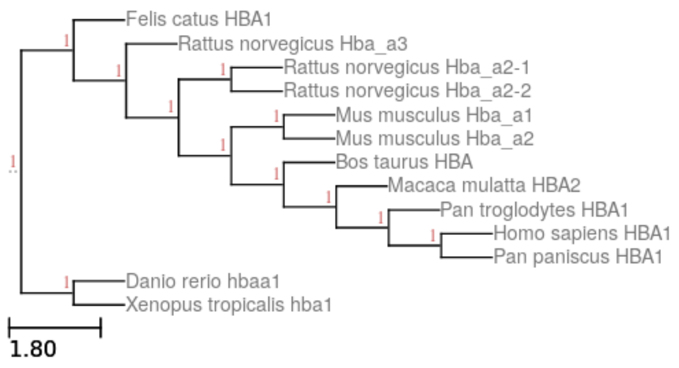


Figure 6: Cluster of hemoglobin proteins according to their amino-acids sequences. This diagram was generated using the crude sequence distance. Notice that mice (*Mus musculus*) hemoglobin gene is more similar to rats, whereas the corresponding proteins is more similar to the cluster of homonids and cows.

5 Optional extensions

5.1 Align sequences for computing their distance

5.1.1 Align sequences

Throughout the project, the sequences are going to be rather similar, so we will use a **global alignment strategy** based on the Needleman and Wunsch algorithm.

As a first approximation, we will assume that:

- a **match** has a score of +5;
- a **mismatch** has a score of -4;
- an **indel** has a score of -3.

When comparing sequences of amino-acids, we may refine our approach by considering that certain pairs of amino-acids are more likely to be substituted than others (section 5.1.3).

Step 17 *Create a class `AlignmentNW` that has two `String` attributes (the sequences to compare), as well as some match, mismatch and indel scores and an alignment matrix initialized according to the Needleman and Wunsch algorithm.*

Step 18 *In the class `AlignmentNW`, add a method `void printMatrix()` that prints the alignment matrix to `System.out`*

```java
public class AlignmentNW {
  private String s1;
  private String s2;
  private int scoreMatch;
  private int scoreMismatch;
  private int scoreIndel;
  private int[][] alignmentMatrix;

  public AlignmentNW(...) {
    // initialize the attributes
  }

  public void printMatrix() {
    // pretty prints alignmentMatrix
  }
}
```

Step 19 *In the* `main(...)`*, create the following sequences, and print their three respective alignments (*`seq1,seq2`*), (*`seq1,seq3`*), (*`seq2,seq3`*):*

```
seq1 = "ATTACG";
seq2 = "ATATCG";
seq3 = "ACCCCG";
seq4 = "GGGGAA";
seq5 = "TTTACG";
```

5.1.2 Compute the distance between sequences

Step 20 *In the class* `AlignmentNW`*, add a method* `int getScore()` *that returns the score of the alignment.*

This score can be used to compute a distance ($\in [0;1]$) between the two sequences. The distance is `0.` if the sequences are identical, and `1.` if they are completely dissimilar.

For normalizing the distance, we consider the maximal and minimal possible scores for two strings of the same length as `s1` and `s2`. You can either let your `getScore()` method compute them for you, or compute them directly. The first approach is the laziest; the second one is the most efficient. You can also do both and check that they return the same result.

For the lazy way, create two sequences `s1max` and `s2max` of `'A'` of the same length as `s1` and `s2`, and `s2min` of `'T'` of the same length as `s2`. The maximal score is the score of the (`s1max`,`s2max`) alignment, and the minimal score is the score of the (`s1max`,`s2min`) alignment.

For the efficient way, the maximal score is

$$scoreMax(\texttt{s1},\texttt{s2}) = \texttt{scoreMatch} \times min(|\texttt{s1}|, |\texttt{s2}|)$$

$$scoreMin(\texttt{s1},\texttt{s2}) = (\texttt{scoreIndel} \times (|\texttt{s1}| + |\texttt{s2}|))$$

$$distance(\texttt{s1},\texttt{s2}) = \frac{scoreMax(\texttt{s1},\texttt{s2}) - score(\texttt{s1},\texttt{s2})}{ScoreMax(\texttt{s1},\texttt{s2}) - scoreMin(\texttt{s1},\texttt{s2})}$$

Step 21 *In the class `AlignmentNW`, add the methods `int getScoreMax()` and `int getScoreMin()` that respectively return the maximal and minimal scores of the alignment.*

Step 22 *In the class `AlignmentNW`, add the methods `double getDistance()` that returns the distance between the two strings of the alignment. Compute the respective distances of (`seq1`,`seq2`), (`seq1`,`seq3`), (`seq2`,`seq3`)*

5.1.3 Optional: use a substitution matrix

To acknowledge that some mismatches have a greater influence than others, we can have a mismatch penalty that depends on the two elements being compared. This is usually important when comparing two proteins' sequences of amino-acids.

A generic approach can rely on a substitution matrix. There are several such matrices (e.g. from`ftp://ftp.ncbi.nih.gov/blast/matrices/`). Typical examples include:

- NUC.4.4 for comparing nucleotides. You can see that is follows an uniform scoring scheme with an exact match being worth +5, and a mismatch being worth -4;
- BLOSUM65 for comparing amino-acids. Notice that the mismatch penalty is not uniform.

Step 23 *Write a function that reads a substitution matrix from a text file (follow the format from the NCBI). Lines starting with a hash are comments.*

Step 24 *Modify your function that computes the alignment matrix so that is uses the value from a substitution matrix when considering matches and mismatches.*

5.2 Improve the dendrograms

The Newick format allows to specify the branches' length. For visualizing the result, not all the tools mentionned previously support this feature. Rather use FigTree[10] or the Tree Viewer[11] website.

Step 25 *Improve the `getNewick()` method so that all the leaves are at the same level (i.e. aligned on the right, contrary to Fig. 4). Run javadoc, commit your changes and push them to your remote repository.*

Step 26 *Improve the `getNewick()` method so that all the leaves are at the same level and the branches' length are proportional to the (absolute value of) the difference between the grades. This is usually called the* cophenetic distance[12]*. Run javadoc, commit your changes and push them to your remote repository.*

[10]`http://tree.bio.ed.ac.uk/software/figtree/`

[11]`http://www.proweb.org/treeviewer/`

[12]`https://en.wikipedia.org/wiki/Cophenetic`

5.3 Divisive approach

Step 27 *Add a method `clusterizeDivisive()` to the class `ClusterOfSequencess`. Perform clustering on `bioCluster` and compare with the agglomerative approach.*

5.4 Modeling considerations for `ClusterOfSequences`

In the class `ClusterOfSequences`, each instance of `Sequence` appears twice:

- in the attribute `subClusters` because the dendrogram has as many subcluster leaves as sequences;
- in the attribute `elements` that maked marshalling the elements easier by avoiding to traverse the cluster recursively.

One could have the impression that this results in doubling the memory usage (even if in our case the overhead would be perfectly acceptable, as each instance takes up a small space in memory and there are few sequences). However, Java obviously does not duplicates the `Sequence` instances in both attributes. Each attributes only contains references to the `Sequence` instances (i.e. their address).

Overall, using two attributes seemingly redundant because they contain (references to) the same objects:

- has the main advantage of improving processing performances by avoiding to traverse the dendrogram when retrieving the list of sequences (which happens often during clustering). This was actually the motivation for introducing the `elements` attribute.
- has the secundary advantage of dispensing you from writing the dendrognam traversal function that would have been necessary for retrieving the list of elements.
- has the drawback of increasing the memory footprint.

Step 28 *Create a class `ClusterOfSequencesBis` that only constains the attribute `subClusters`. Compare the respective clustering time of `ClusterOfSequences` and `ClusterOfSequencesBis`.*