# Hierarchical clustering

BioInformatic: Report Module 3
Groupe 2

**Thomas FEUILLETIN**
thomas.feuilletin@etudiant.univ-rennes1.fr
**Florian EPAIN**
florian.epain@etudiant.univ-rennes1.fr

May 20, 2022

**Abstract**

We are close to get something nice but we put our hearts into it.

# Contents

# 1 Introduction

This project's goal is to compare a gene and its associated proteins across multiple species. We will process some basic algorithm with basic data sample. We're using a method called clusterize by agglomerative.

First, we have to compare each sequence. Using the Levenshtein distance, we will know how far -or close- is a sequence from another. With this well-known distance, we group the closest sequences and repeat the process with the remaining and this new cluster.

Tests are performed on a hemoglobin dataset. Algorthims coded in rust. Resuls observed in terminal and/or in *.md* by foam vc-code extension. The section 5 is a brief tutorial on how to install rust and run our program and how to install foam and visualize the result graph.

Our project outline, gave by subject:

> 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

This record is writen with the v0.9.0 of our program.

Some final results revealed:

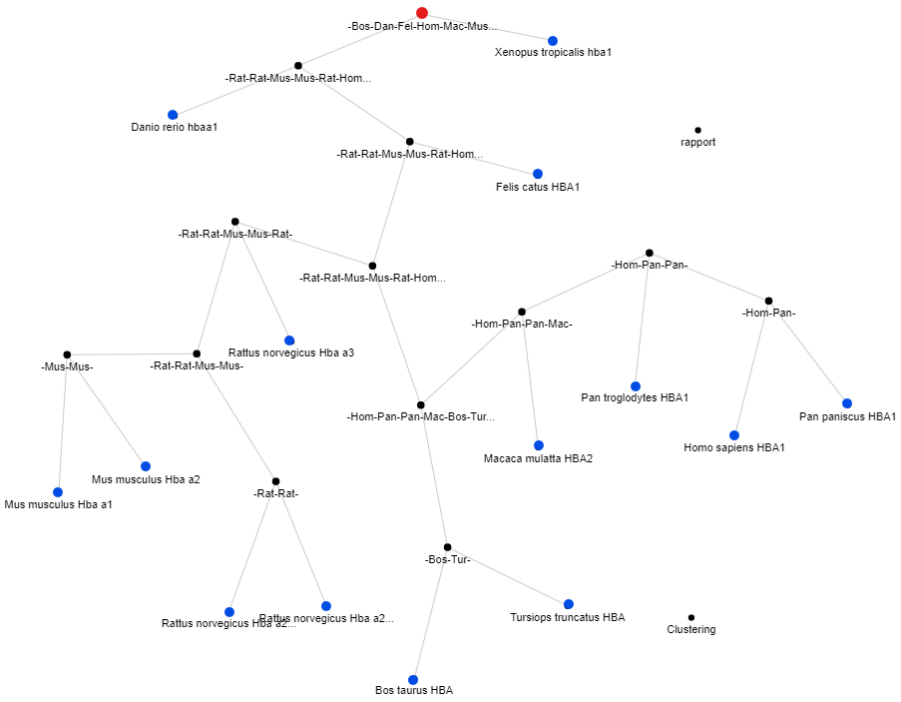


Figure 1: Hierarchically clustered hemoglobin dataset

# 2 Compare sequences

This section aims at computing the distance between two sequences.

## 2.1 Step 1

To do so, let's create a structure to represent Sequence.

```
#[derive(Clone)]
pub struct Sequence {
    name: String,
    seq: String,
}
```

Let's implement this struture with constructor

```
impl Sequence {

    pub fn new() -> Sequence{
        Sequence {
            name: String::new(),
            seq: String::new(),
        }
    }

    pub fn new_with_string(name: String, seq: String) -> Sequence {
        Sequence {
            name,
            seq,
        }
    }

    pub fn new_with_sequence(name: String, s: Sequence) -> Sequence {
        Sequence {
            name,
            seq: s.seq,
        }
    }

    pub fn get_name(&self) -> String {
        self.clone().name
    }


    ...
}
```

We now code a pretty close Levenshtein distance, as described in wikipedia.

```rust
impl Sequence {
    ...

    /**
     * ---------------
     * ^ ----------- ^
     * ^ ^ATCCCTGCA^ ^
     * ^ ^ATCCCTGCT^ ^
     * ^ ----------- ^
     * ^  AGCCCTGCT  ^
     * ---------------
     *
     * Levenshtein distance
     * return the distance between the two sequences
     */
    pub fn levenshtein_distance(&self, other_seq: &Sequence) -> usize {
        // convert our seq to Vec<Char>
        let s1: String = self.seq.to_string();
        let s2: String = other_seq.seq.to_string();
        let seq_1: Vec<char>= s1.chars().collect();
        let seq_2: Vec<char> = s2.chars().collect();

        // initialization of result matrix
        let mut d: Vec<Vec<usize>> = vec![vec![0;s2.len()+1]; s1.len()+1];
        // println!("d.len(): {}", d.len());

        // write a meter
        for i in 1..d.len() {
            d[i][0] = i
        }
        for j in 1..d[0].len() {
            // println!("j: {}, d[0].len(): {}", j, d[0].len());
            d[0][j] = j
        }

        // idk what it determine
        let mut subtitution_cost: usize;
        for i in 1..d.len(){
            for j in 1..d[i].len() {
                // let subtitution_cost: i8;
                // maybe to expensive to recreate every case
                if seq_1[i-1] == seq_2[j-1] {
                    subtitution_cost = 0;
                }else {
                    subtitution_cost = 1;
                }

                d[i][j] = lib::minimum_3(d[i-1][j] + 1,                 // deletion
                                         d[i][j-1] + 1,                 // insertion
                                         d[i-1][j-1] + subtitution_cost)// substitution
            }
        }

        d[seq_1.len()][seq_2.len()]
    }
}
```

### 2.2 Step 2

A simple example of these methods

```
let seq_1 = Sequence::new_with_string("seq_1".to_string(), "ATTACG".to_string());
println!("{}", seq_1.levenshtein_distance(&seq_1));
```

Here the distance calculated by our levenshtein_distance() method:

| levenshtein_distance() | | | | | | | |
|---|---|---|---|---|---|---|---|
| | ATTACG | ATATCG | ACCCCG | GGGGAA | TTTACG | ATTAC | ATATC |
| ATTACG | 0 | 2 | 3 | 6 | 1 | 1 | 3 |
| ATATCG | 2 | 0 | 3 | 6 | 3 | 3 | 1 |
| ACCCCG | 3 | 3 | 0 | 6 | 4 | 4 | 4 |
| GGGGAA | 6 | 6 | 6 | 0 | 6 | 5 | 6 |
| TTTACG | 1 | 3 | 4 | 6 | 0 | 2 | 4 |
| ATTAC | 1 | 3 | 4 | 5 | 2 | 0 | 2 |
| ATATC | 3 | 1 | 4 | 6 | 4 | 2 | 0 |

## 3 Hierarchical clustering

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:

- Agglomerative approach.
- Divisive approach.

We will focus on the Agglometive approach.

### 3.1 Initialization

For the creation of this sturture, we choose to use vector.

*Note:* I added a name attribute which slow down the whole clustering process but helps with the foam representation

```
/**
 * we're able to create a pure copy of a cluster with a new ref by the Clone derive
 *
 * sub_clusters correspond to all the clusters contained in this clusters
 * elements contains all element in this cluster, eventually from the sub_clusters
 */
#[derive(Clone)]
pub struct ClusterOfSequence {
    name: String,
    sub_clusters: Vec<ClusterOfSequence>,
    elements: Vec<Sequence>,
}
```

The cluster's name depends on what it contains

- One sequence -> Sequence's name
- Many sequences -> The three first letter of all sequence name

```rust
impl ClusterOfSequence
{
    pub fn new(element: Sequence) -> ClusterOfSequence
    {
        ClusterOfSequence {
            name: element.clone().name,
            sub_clusters: Vec::new(),
            elements: vec![element],
        }
    }

    pub fn new_with_sequences(elements:Vec<Sequence>) -> ClusterOfSequence
    {
        ClusterOfSequence {
            name:
            {
                let mut res = String::new();
                res.push('-');
                for elem in &elements {
                    // to avoid Err :
                    if elem.name.len() <= 3 {
                        res.push_str(&elem.name[0..elem.name.len()]);
                    }else {
                        res.push_str(&elem.name[0..3]);
                    }
                    res.push('-');
                }
                res
            },
            sub_clusters:
            {
                let mut res = Vec::new();
                for e in &elements {
                    res.push(ClusterOfSequence::new(e.clone()))
                }
                res
            },
            elements,
        }
    }

    ...
}
```

Iterating through all the sequence names of a cluster is time consuming, one way to counter this is to give the cluster a name when needed (Foam Graph Representation)

```
impl ClusterOfSequence
{
    ...

    /**
     * ClusterOfSequence without reference &?
     */
    pub fn new_with_clusters(clusters_1: ClusterOfSequence,
                         clusters_2: ClusterOfSequence) -> ClusterOfSequence
    {
        ClusterOfSequence {
            name:
            {
                // we have to make sure there is no doublon
                // to avoid conflict and wrong relation in foam
                let mut foo= String::new();
                foo.push('-');
                for elem in &clusters_1.elements {
                    if elem.name.len() <= 3 {
                        foo.push_str(&elem.name[0..elem.name.len()]);
                    }else {
                        foo.push_str(&elem.name[0..3]);
                    }
                    foo.push('-');
                }
                for elem in &clusters_2.elements {
                    if elem.name.len() <= 3 {
                        foo.push_str(&elem.name[0..elem.name.len()]);
                    }else {
                        foo.push_str(&elem.name[0..3]);
                    }
                    foo.push('-');
                }
                foo
            },
            sub_clusters: vec![clusters_1.clone(), clusters_2.clone()],

            // get all seq of clust_1 and clust_2 to elements
            elements:
            {
                let mut n = clusters_1.clone().elements;
                n.extend(clusters_2.clone().elements);
                n
            }
        }
    }

    ...
}
```

### 3.2 Visualization

Step 5:

```
let seq_0_test = Sequence::new_with_string("ATTACG".to_string(), "ATTACG".to_string());
let seq_1_test = Sequence::new_with_string("ATATCG".to_string(), "ATATCG".to_string());
let seq_2_test = Sequence::new_with_string("GCCGAG".to_string(), "GCCGAG".to_string());
let seq_3_test = Sequence::new_with_string("ACCCCG".to_string(), "ACCCCG".to_string());
let seq_4_test = Sequence::new_with_string("TCCCCG".to_string(), "TCCCCG".to_string());
let sequences_test = vec![seq_0_test, seq_1_test, seq_2_test, seq_3_test, seq_4_test];
let mut bio_cluster = ClusterOfSequence::new_with_sequences(sequences_test);
```

Here the simple newick method

```
pub fn get_newick_old(&self) -> String
{
    let mut res = String::from("(");
    // we could set "!= 0" to "!= 1"
    // to avoid displaying clusters containing a single sequence
    if self.sub_clusters.len() != 0 {
        for i in 0..self.sub_clusters.len() {
            res.push_str(&self.sub_clusters[i].get_newick_old().as_str());
            // e != self.elements.last().unwrap() -> != not implement
            if i != self.sub_clusters.len()-1 {
                res.push(',');
            }

        }
    }else {
    // this else occurs when a cluster contains only sequences
        for e in 0..self.elements.len() {
            res.push_str(&self.elements[e].name.as_str());
            if e != self.elements.len()-1 {
                res.push(',');
            }

        }
    }
    res.push(')');
    res
}
```

Step 7:

```
println!("bio_cluster: \n{}", bio_cluster.get_newick_old());
```

results: bio_cluster: ((ATTACG),(ATATCG),(GCCGAG),(ACCCCG),(TCCCCG))

We tried to have a cool looking newick rep in the terminal but it not goes like intended...

```rust
/**
 * TODO recheck step 6 and 7
 * if you want to get a cleaner look change :
 *      .elements[e].seq to .elements[e].name
 * if you want to get the precise sequence in the representation :
 *      .elements[e].name to .elements[e].seq
 */
pub fn get_newick(&self) -> String
{
    self.get_newick_with_space(0)
}

fn get_newick_with_space(&self, space: i32) -> String
{
    let mut res = String::new();
    if self.sub_clusters.len() != 0 {
    for i in 0..self.sub_clusters.len() {
            // res.push_str("|");
            res.push_str(
                &self.sub_clusters[i].get_newick_with_space(space+1)
                                    .as_str());

            // e != self.elements.last().unwrap()
            // TODO : check this if else usefulness
            if i != self.sub_clusters.len()-1 {
                res.push_str("\n");
            }else {
                res.push_str("\n");
            }

            }
    }else {
        for e in 0..self.elements.len() {
            if e ==0 || e == &self.elements.len() -1{
                for _count in 0..space {
                    res.push_str("-");
                }
            }
            // .elements[e].seq or .elements[e].name depending
            res.push_str(&self.elements[e].name.as_str());

            if e != self.elements.len()-1 {
                res.push_str("\n");
            }

        }
    }
    // res.push_str("\n");
    res
}
```

```
println!("bio_cluster: \n{}", bio_cluster.get_newick());
bio_cluster.clusterize_agglomerative();
println!("bio_cluster: \n{}", bio_cluster.get_newick());
```

imagining that clusterize method exists, it goes:

```
bio_cluster:
-ATTACG
-ATATCG
-GCCGAG
-ACCCCG
-TCCCCG

clusterize...
ended in 1.1177ms

bio_cluster:
---ACCCCG
---TCCCCG

--GCCGAG

--ATTACG
--ATATCG
```

And now for the visualization by graph (not professional graph but np). We opted for the FOAM vs-code extension which reads links between markdown and create a cool constellation. Go to Section 5, for more information on how to install foam in vs-code.

What does the create_foam_rep() method ?
Create a serie of markdown file and each file contains:

- [[*wikilinks*]] to subclusters
- The list of elements contained

Make sure you install foam vscode, and have theses folder :

- _layout
- .vscode
- .foam
- assets

At the end, you will prompt the foam graph in vscode by : ctrl+p ">Foam: Show Graph"

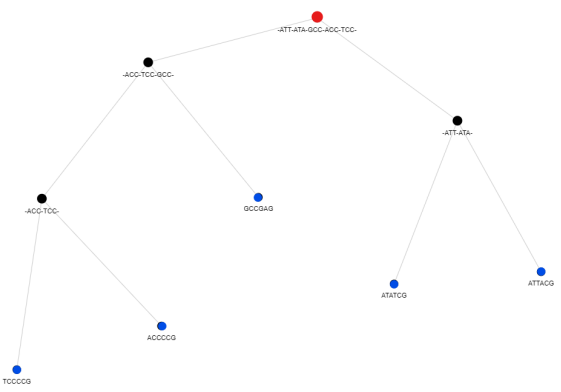


Figure 2: Hierarchically clustered random small dataset

```rust
pub fn create_foam_rep(&self, folder_name: &str) // -> Result<()>
{

    let mut foo =
    "---
    \ntitle: name
    \ntype: bio_inf_results
    \n---
    \n".to_string();

    // Add up all sub_cluster links
    for counter in 0..self.sub_clusters.len() {
        let mut bar = "\n [[wikilink]]"
                    .replace("wikilink", &self.sub_clusters[counter].clone().name);

        if counter != self.sub_clusters.len()-1 {
            bar.push_str(", ");
        }
        foo.push_str(&bar);

        // now create the md file associated with this sub cluster
        self.sub_clusters[counter].create_foam_rep(folder_name);
    }

    // separator
    foo.push_str("\n");

    // if you prefer have the sequence prompted
    // change elment.name to element.seq
    for element in &self.elements {
        let mut bar = "element"
                    .replace("element", &element.name);
        if element.name != self.elements.last().unwrap().name {
            bar.push_str(", ");
        }
        foo.push_str(&bar);
    }

    // write this foobar to a .md file

    let name = self.clone().name;
    // change title
    foo = foo.replace("name", &name);

    // create docs folder if it did not exist
    let folder_path = "./foam_rep/docs/folder_name/"
                            .replace("folder_name", folder_name);
    fs::create_dir_all(&folder_path);
    let md_path = "./foam_rep/docs/folder_name/name.md"
                            .replace("folder_name", folder_name)
                            .replace("name", &name);

    fs::write(md_path, foo)
            .expect("Unable to write file");

    // Ok(())
}
```

### 3.3 Clustering

Linkage method creation

```rust
pub fn linkage(&self, a_cluster: ClusterOfSequence) -> Option<f32>
{
    // covert the division by 0
    if self.elements.len() == 0 || a_cluster.elements.len() == 0 {
        None
        // f32::MAX;
    }else {
        let mut result = 0.00;
        let mut counter = 0.00;
        /*
         * compare every e1: Sequence (in the self called Cluster)
         *      with every e2: Sequence (in a_cluster)
         *      by the levenshtein_distance
         * return the average distance between all sequence
         */
        for e1 in &self.elements {
            for e2 in &a_cluster.elements {
                result += e1.levenshtein_distance(&e2) as f32;
                counter += 1 as f32;
            }
        }

        result = result/counter;
        Some(result)
    }
}
```

With this cluster : ((ATTACG,ATATCG),(GCCGAG,(ACCCCG,TCCCCG)))
We have : (cl1, cl2) and cl2 = (GCCGAG, cl3)
So, we should have cl3.linkage(cl2) < cl1.linkage(cl2). The average distance between cl1 and cl2 should be pretty high > 2. Same for cl1 and cl3. But cl2 and cl3 should be quite close (being in the same cluster must bring us closer)

| linkage() | | | |
|---|---|---|---|
| | cl1 | cl2 | cl3 |
| cl1 | 1 | 4 | 3.5 |
| cl2 | 4 | 1.55 | 1.33 |
| cl3 | 3.5 | 1.33 | 0.5 |

To create the clusterize_agglomerative() method we imagined some approach :
First approach :

- Compare each sequence.
- Group two close sequences in new cluster
- Group other sequences in another cluster
- Clusterize the second cluster

---

Second approach :

- Calculate the distance for each sequence two by two
- Compare this distance with ALL of the other sequence
- If the first distance is the smallest
  - Create the sub-cluster containning the two sequence
- Verify if the members are coherent
  - Distance between all of them is smaller than any distance of a another sequence (in the universe)
- If this cluster is good to go, create it for good.
- Compare two cluster

---

Third approach :

state 1. Create clusters for all sequence indiv (ok when initilization)

state 2. Compare all sub-clusters one by one by distance

state 2.1 Get the smallest distance

state 2.2 Check cluster condition if it's correct confirm this cluster else switch to the next

state 3. Back to state 2 until the total of sub-cluster =< 2

---

The problem is this method used a while loop that contains two nested for loop. The first for loop determine the smaller distance between all clusters. The second verify the coherence of the potential new cluster (with i from the while 'main_loop and keep from the first for loop). Once the cluster is verified (if not we're incrementing i and get back to the 'main_loop), we create the new cluster (i and keep), remove the sub-cluster i and keep from the general list of sub-clusters. The new cluster is inserted in place 0. And we reiter the while loop ('main_loop) from 0, to compare the fresh cluster to others.

```rust
pub fn clusterize_agglomerative(&mut self)
{   // start timer
    let st = SystemTime::now();

    let mut i = 0;
    'main_loop:
    while i < self.sub_clusters.len() && self.sub_clusters.len() > 2 {

        let mut min: f32 = f32::MAX;
        let mut keep = 0;
        // state 2
        for j in 0.. self.sub_clusters.len() {
            if i != j {
                // state 2.1
                let dist = self.sub_clusters[i]
                               .linkage(self.sub_clusters[j].clone())
                               .unwrap();
                if dist < min {
                    min = dist;
                    keep = j;
                }
            }
            // we tried here to verify at the same time the cluster's coherence
            // but we need to keep track of a smallest distance BEFORE checking
            // the coherence (in the case if the closest is after j)

        }
        // state 2.2
        for j in 0..self.sub_clusters.len() {
            if j != keep && j != i {
                if self.sub_clusters[keep]
                       .linkage(self.sub_clusters[j].clone())
                       .unwrap() < min {
                    i+=1;
                    continue 'main_loop;
                    // the while exit is ONLY here
                }
            }
        }

        // create a new subCluster which contains two coherent element
        let sub_1 = self.sub_clusters[i].clone();
        let sub_2 = self.sub_clusters[keep].clone();

        // commit: from here I remove all the ref & to self call

        // remove the two clusters from the first cluster
        self.sub_clusters.remove(i);
        if keep > i {
            self.sub_clusters.remove(keep-1);
        }else {
            self.sub_clusters.remove(keep);
        }

        // add the new cluster which contains the two last clusters
        self.sub_clusters.insert(
            0, ClusterOfSequence::new_with_clusters(sub_1, sub_2));

        i = 0;
    }
    // check timer
    let ed = SystemTime::now();
    println!("{:#?}", ed.duration_since(st).unwrap());
}
```

# 4 Application to hemoglobin

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

## 4.1 Read the gene sequence from a fasta file

After reformed all ( ) fasta to be :

```
>some text that is skipped
ATGGTGCTGTCTGCCGCCGACAAGGGCAATGTCAAGGCCGCCTGGGGCAAGGTTGG
```

utils.rs

```rust
use std::{
    fs,
    time::SystemTime
};
use crate::sequence::*;

fn read_fasta(path: &str) -> String {
    let start = SystemTime::now();
    let content = fs::read_to_string(path).expect("Can't touch it");
    let mut res:String = String::new();
    //print!("{:?}", content);
    for line in content.split("\n") {
        if !line.starts_with(">") {
            res.push_str(&line.replace("\r", ""));
        }
    }
    let end = SystemTime::now();
    println!("read content: {:?}", end.duration_since(start).unwrap());
    res
}
```

### 4.2 Cluster of gene sequences

We created a method that read automatically all .fa file and return a Cluster containing all raw sequence.

```rust
/**
 * it's meant to simplify the main method by just one method
 * calling as many time the read_fasta method for all the
 * fasta files in the path (in param).
 * @param a specific folder name in the res folder
 */
pub fn analyze_from_ressource_folder() -> ClusterOfSequence{
    let paths = fs::read_dir("./data_reformed").unwrap();

    let mut sequences: Vec<Sequence> = Vec::new();

    for path in paths {

        let mut name = path.as_ref()
                           .unwrap()
                           .path()
                           .display()
                           .to_string();

        name = name.replace("./data_reformed\\", "")
                   .replace("_sequence.fa", "")
                   .replace("_", " ");

        sequences.push(Sequence::new_with_string(
                        name.to_string(),
                        read_fasta(&path.unwrap().path().display().to_string()))
                      );
        println!("name of the sequence: {}", name);

    }

    let bio_cluster = ClusterOfSequence::new_with_sequences(sequences);

    bio_cluster


}
```

# 5 Tutorial

## 5.1 Install Rust

For your own good: Install Rust
Open a terminal and check the cmd : $ rustup -V and $ cargo -V
If it doesn't work GL (internet is your friend).
In terminal, go to the root of the Clustering project, and type $ cargo run
Here we go the program runs !

## 5.2 Install Foam

Foam is a developing tools for thought. But we can use it for bioinfo ! Launch Visual Studio Code, install the Foam extension.
Open the Clustering project by >File >Open Folder...
Now, type $ctrl + p$ and $> Foam : ShowGraph$ The Graph will normally display.