# Projet LU2SV483 - 2024

## Analyse de séquences codantes
## et construction d'un arbre de distances

Le projet consiste à écrire un programme capable de :

- sélectionner dans une banque de séquences nucléotidiques, un sous-ensemble de séquences d'intérêt ;
- traduire les séquences nucléotidiques choisies en séquences protéiques ;
- aligner (au moyen d'une bibliothèque fournie) les séquences protéiques ;
- construire un arbre des distances entre les séquences protéiques ;
- représenter graphiquement les ressemblances entre séquences.

Ces traitements ne sont pas implémentés dans cet ordre, mais en fonction des techniques nécessaires, acquises au cours du déroulement de l'UE. Chaque tâche doit être réalisée par une (ou plusieurs) fonction(s) distincte(s). Le programme principal se charge d'appeler les fonctions pour les exécuter sur un ensemble spécifique de données (fichiers lus, paramètres).

Le projet s'articule autour de l'analyse d'une famille de gènes : les hbp (« *hemin binding proteins* »). Cependant, il comprend aussi plusieurs fonctions génériques, utiles pour l'analyse de séquences, mais qui ne seront pas utilisées dans le cas des hbp. Ces fonctions doivent tout de même apparaître dans la bibliothèque de fonctions, et un exemple d'utilisation doit apparaître dans le programme principal.

<u>Remarque importante pour l'évaluation du projet :</u>

Le programme rendu sera constitué de fonctions, qui devront être regroupées dans une ou des bibliothèque(s). Un code principal distinct du reste du code doit appeler les différentes fonctions. Sauf exception, une fonction doit effectuer des traitements et retourner un résultat (ou plusieurs), et le code principal doit prendre en charge l'interaction avec l'utilisateur : saisie de données et paramètres, appel des fonctions, affichage des résultats.

Il faut bien réfléchir aux arguments des fonctions et à leur retour en choisissant des structures de données appropriées. Il ne faut pas se contenter d'afficher les résultats obtenus. Pour chaque fonction, on explicitera – sous forme d'un commentaire en début de fonction – ce que fait cette fonction, le(s) type(s) du ou des argument(s) en entrée et le(s) type(s) de la ou des valeur(s) retournée(s).

**LU2SV483 – Projet 2024**

# Étape 1

1. Calculer des données de base concernant une séquence nucléotidique. Chacun des calculs doit être traité par une fonction spécifique.
   La séquence est écrite littéralement dans le code du programme principal (ou importée depuis une bibliothèque) sous forme d'une chaîne de caractères comportant les lettres A, C, G, T. Le programme doit en donner :
   - la longueur ;
   - le taux de G+C global ;
   - la séquence inverse-complémentaire ;
   - la position de tous les motifs ATG et tous les STOP de traduction, sur les deux brins.
2. Ajouter des fonctions pour comparer deux séquences nucléotidiques alignées fournies en paramètres. Les deux séquences sont constituées des lettres A,C,G,T, plus le caractère « - », qui indique une insertion / déletion (indel). Une fonction doit être créée pour chacune des tâches suivantes :
   a) compter les appariements, substitutions et indels.
   b) calculer la distance d'édition.

   **Distance d'édition** C'est le nombre d'opérations (substitutions et indels) nécessaires pour passer d'une séquence à une autre. Plus cette distance est grande et moins les séquences se ressemblent. Par exemple, pour l'alignement suivant :

   ```
   CA-CGTGCTGACCCAACC
   CAGCGCGCTGG--CAGCC
   ```

   on calcule une distance d'édition de 6 (de gauche à droite) :
   1 indel (`-/G`) + 2 substitutions (`T/C`, `A/G`) + 2 indels (`C/-`) + 1 substitution (`A/G`).

   c) calculer le score d'alignement, en tenant compte de poids pour les appariements, substitutions, indels. Ces poids sont donnés comme paramètres à la fonction.

   **Score d'alignement** On attribue un poids à chaque situation possible pour une position de l'alignement : Wa pour un appariement, Ws pour une substitution, Wi pour un indel. Pour que ce score soit d'autant plus grand que les séquences se ressemblent, il faut que le poids d'un appariement soit supérieur au poids d'une substitution ou d'un indel. Par exemple, on pourrait choisir Wa = +2, Ws = -1, Wi = -2. Le score de l'alignement est la somme des poids correspondant à chaque position. Par exemple, pour les poids ci-dessus et l'alignement suivant :

   ```
   CA-CGTGCTGACCCAACC
   CAGCGCGCTGG--CAGCC
   ```

   on calcule un score de 15 (de gauche à droite) :
   2Wa + Wi + 2Wa + Ws + 4Wa + Ws + 2Wi + 2Wa + Ws + 2Wa = 15

   d) calculer le taux d'identité entre les deux séquences.

**LU2SV483 – Projet 2024**

# Étape 2

Le fichier `Bartonella.dat` fourni sur le Moodle est extrait de la banque EMBL, qui contient toutes les séquences génomiques connues. Des explications sur le format des fichiers EMBL sont données en annexe A. `Bartonella.dat` renferme des annotations de séquences génomiques de bactéries du genre Bartonella (α-protéo-bactérie). Les séquences nucléotidiques sont quant à elles dans le fichier `Bartonella.fasta`, également téléchargeable depuis le Moodle.

Pour la suite de l'étude, on s'intéresse à la famille multigénique des « *hemin-binding proteins* », et en particulier à trois gènes : hbpC, hbpD, hbpE, dans les cinq espèces suivantes : *B. henselae*, *B.quintana*, *B. taylorii*, *B. vinsonii*, *B. washoeensis*. Dans la banque EMBL, certaines de ces séquences sont annotées avec leur nom de gène (éventuellement suivi d'un numéro), d'autres comportent une annotation concernant la structure de la protéine produite, qui contient un domaine « *Outer membrane protein beta-barrel domain* » abrégé en « *OMP_b-brl* ». Les autres gènes, y compris les autres hbp (c'est-à-dire hbpA et hbpB), ceux n'appartenant à aucune des cinq espèces choisies, ainsi que les gènes hypothétiques doivent être exclus. L'étude porte donc sur quinze protéines (trois gènes pour chacune des cinq espèces).

1. a) Écrire une fonction qui lit un fichier EMBL, et stocke dans une liste, pour chaque CDS présente dans le fichier, un résumé des informations essentielles : ID de la séquence génomique d'origine, espèce, position de la CDS (attention au sens, et aux CDS incomplètes, *cf* Annexe A), fonction, et identifiant UniProt.
   b) Écrire une autre fonction qui écrit ces données dans un fichier. On devrait obtenir un tableau similaire à celui-ci :

| EMBL ID | Species | dir | begin - end | Uniprot | Function |
|---|---|---|---|---|---|
| AILX01000001 | Bartonella washoeensis | - | 6409 - 6882 | J0ZFN6 | hypothetical protein |
| AILX01000001 | Bartonella washoeensis | - | 6934 - 7647 | J0QUQ6 | hypothetical protein |
| AILX01000001 | Bartonella washoeensis | - | 7658 - 8155 | J1JR39 | hypothetical protein |
| AILX01000001 | Bartonella washoeensis | - | 8119 - 8616 | J0QMI4 | hypothetical protein |
| AILX01000001 | Bartonella washoeensis | - | 8737 - 9135 | J1JPZ3 | hypothetical protein |
| AILX01000001 | Bartonella washoeensis | + | 9670 - 10536 | J0ZFP0 | orotidine 5'-phosphate decarboxylase |
| AILX01000002 | Bartonella washoeensis | + | 198 - 308 | J0QUH6 | hypothetical protein |
| *... En tout, 10177 lignes ...* | | | | | |
| AM260525 | Bartonella tribocorum | + | 2611044 - 2611511 | A9IZX2 | hypothetical protein predicted by Glimmer/Critica |
| AM260525 | Bartonella tribocorum | - | 2611536 - 2612432 | A9IZX4 | chromosome partitioning protein ParB |
| AM260525 | Bartonella tribocorum | - | 2612472 - 2613269 | A9IZX6 | chromosome partitioning protein ParA |
| AM260525 | Bartonella tribocorum | - | 2617239 - 2618504 | A9IZY3 | transcription termination factor rho |

2. Écrire une fonction qui prend en paramètres un fichier EMBL et un fichier contenant des critères de sélection de gènes (qui peuvent être des expressions régulières), et retourne une liste d'ID de séquences, de positions et de références Uniprot correspondant aux gènes qui remplissent les critères.
3. Écrire une fonction qui lit un fichier au format Fasta contenant des séquences nucléotidiques, et les stocke sous une forme convenable.
4. Écrire une fonction qui extrait d'une série de séquences, les portions définies dans une liste composée d'ID de séquences et de bornes.

On utilisera ces fonctions pour sélectionner, à partir des fichiers `Bartonella.dat` et `Bartonella.fasta`, les quinze séquences des CDS choisies selon les critères explicités ci-dessus.

**LU2SV483 – Projet 2024**

# Étape 3

1. Reprendre les fonctions de l'étape précédente, et lorsque c'est approprié, stocker les données dans des dictionnaires.
2. Écrire une fonction capable de trouver toutes les phases ouvertes sur les deux brins d'une séquence nucléotidique. Une phase ouverte est considérée comme une portion de séquence comprise entre un codon `ATG` et le premier codon STOP qui suit, dans la même phase de lecture, et ayant une longueur minimale de 48 nucléotides (soit 15 AA + le codon STOP). Par exemple, on devrait obtenir, pour la séquence :
   `GCATGCGAGCTATATGCATGGCCTACACGCTAAAGATGCAGATGCTAAATACGGAAGCAGACCGTAGCGCTTGATACATCTCGTAAGCATA`
   - trois ORF sur le brin + : 2-73, 13-66, 17-73 ;
   - deux ORF sur le brin - : 1-81 et 12-68.

   *N.B.* Les bornes données ci-dessus correspondent à la numérotation Python : le premier nucléotide est à la position 0.
3. Écrire une fonction qui traduit une phase ouverte en séquence protéique. Le code génétique est lu dans le fichier tabulé `CodeGenetique.tab`, fourni sur le Moodle.

   Utiliser la fonction de traduction pour traduire les quinze séquences de hbp de Bartonella.

# Étape 4

Pour cette étape, il est nécessaire d'importer la bibliothèque `msa` (multiple sequence alignment) fournie sur le Moodle, qui contient une fonction d'alignement multiple.

**Bibliothèque `msa`** Cette bibliothèque de fonctions permet de réaliser un alignement multiple de séquences nucléotidiques ou protéiques. Elle implémente un algorithme d'alignement multiple « en étoile », un grand classique de l'analyse de séquences.
Pour utiliser la bibliothèque, il faut disposer du fichier `msa.pyc`, copié dans le même dossier que celui où se trouve le code à exécuter.
Le code doit commencer par importer la bibliothèque. L'alignement est réalisé par la fonction `star_align`, à laquelle il faut passer deux arguments :

- un dictionnaire dont les clés sont les noms des séquences, et les valeurs sont les séquences protéiques ;
- une matrice de dissimilarité (ou matrice de substitution), sous forme d'un dictionnaire sont les clés sont des tuples de deux AA (écrits avec le code à une lettre des AA), et les valeurs sont les poids de substitution.

La fonction retourne le score d'alignement, ainsi qu'un dictionnaire de séquences similaire à celui qu'elle a reçu en paramètre, où les séquences sont alignées.
*N.B.* Dans l'exemple d'utilisation ci-dessous, les séquences et la matrice sont respectivement lues par les fonctions `read_fasta` et `read_matrix`, incluses dans la bibliothèque `msa`, ceci afin de permettre de réaliser des tests. Cependant, pour le projet, vous devez écrire et utiliser vos propres fonctions de lecture de séquences et de matrice.
Les fichiers `SeqTest.fasta` et `blosum62.mat` sont fournis sur le moodle.

```
############### Test de la bibliothèque msa ###############

import msa

#### Programme principal ###

# SeqTest : Dict{str:str} ; Séquences protéiques
SeqTest = msa.read_fasta('SeqTest.fasta')

# SubMat : Dict{(str,str):float} ; Matrice de substitution
SubMat = msa.read_matrix('blosum62.mat')

# MSA_score : float ; score d'alignement multiple
# MSA_seqs : Dict{str:str} ; Séquences alignées
MSA_score, MSA_seqs = msa.star_align(SeqTest, SubMat)

print(len(SeqTest), 'séquences, score =', MSA_score)
# nom : str ; nom d'une séquence
for nom in MSA_seqs:
    print(nom, MSA_seqs[nom])

############################################################
```

On doit obtenir :

```
4 séquences, score = 278.0
Seq1 MEGKVNEDVAGDANCRLM---LLV
Seq2 MEGKVNEDVAGEANCKLMQP-LLV
Seq3 MEGKVHDDV---SNCKLLQPILLV
Seq4 MEGKVHE----EANCKLMQPILLV
```

# LU2SV483 – Projet 2024

1. Écrire une fonction qui part d'un alignement multiple de protéines et calcule à chaque position de l'alignement le score selon des poids d'appariement, mésappariement et insertion/deletion (ou *indel*).
2. Représenter graphiquement ce taux le long de l'alignement.
3. Écrire une variante de la fonction de calcul de score qui, au lieu des poids, utilise une matrice de dissimilarité (`blosum62.mat`, fournie sur le Moodle), pour calculer un score à chaque position. La matrice assigne un score (positif ou négatif) à chaque paire d'Acides Aminés, ainsi que pour un *indel* (noté « `*` » dans la matrice). Représenter graphiquement ce score le long de l'alignement.
4. Écrire une fonction qui calcule un taux de dissimilarité pour chaque paire de séquences alignées. Pour calculer ce taux, seules comptent les positions pour lesquelles les deux séquences comportent un résidu A.A. (et non un *indel*) ; le taux de dissimilarité se calcule comme la proportion de ces positions pour lesquelles les deux séquences présentent un AA différent. Ce taux est donc une valeur entre 0 (deux séquences identiques) et 1 (aucun AA en commun). La fonction doit retourner une matrice (un tableau 2D, c-à-d une liste de listes) qui contient les taux de dissimilarité de toutes les paires de séquences alignées.
5. À partir de la matrice de dissimilarité, dessiner une *heatmap* de distance.

***Heatmap*** Il s'agit, de façon générale, de représenter graphiquement une propriété par un échelle de couleur, appliquée dans des cases correspondant à un tableau 2D. Ici, le tableau est la matrice des paires de séquences, et la couleur doit représenter la dissimilarité. Prenons un exemple pour un tableau 4×4 :

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 3 | 1 | 2 |
| B | 3 | 0 | 2 | 3 |
| C | 1 | 2 | 0 | 1 |
| D | 2 | 3 | 1 | 0 |

→

| | A | B | C | D |
|---|---|---|---|---|
| A | | | | |
| B | | | | |
| C | | | | |
| D | | | | |

| Échelle |
|---|
| 3 |
| 2 |
| 1 |
| 0 |

*N.B.* Dans un cas plus réaliste, la distance peut prendre beaucoup plus que 4 valeurs, et/ou des valeurs qui ne sont pas forcément des entiers.

6. Écrire une fonction qui, en partant d'une matrice de dissimilarité, calcule un arbre selon l'algorithme UPGMA expliqué en annexe B.

   Utiliser les fonctions ci-dessus pour aligner les quinze séquences protéiques de hbp de Bartonella, dessiner la *heatmap* en prenant les séquences dans un ordre quelconque, puis re-dessiner la *heatmap* en positionnant les séquences selon l'arbre UPGMA (voir dans l'annexe B l'algorithme pour ordonner les séquences d'après un arbre).

# Annexe A
## Les fichiers EMBL

La banque de séquences de l'*European Molecular Biology Laboratory* (*EMBL*) est destinée à stocker toutes les informations connues concernant les séquences biologiques. Une séquence de la banque EMBL peut correspondre à un seul gène (codant une protéine, ou autre chose, comme un ARNt, un ARNr, un micro-ARN, *etc*.) ou à un génome entier dans lequel se trouvent tous les gènes d'un organisme, ou à toute situation intermédiaire entre ces deux extrêmes (chromosome, génome partiellement connu, *etc*.).

Un fichier au format EMBL contient les informations relatives à une ou plusieurs séquences (ou "entrées") de la banque. Il s'agit d'un fichier texte, rédigé en anglais, obéissant à un format strict, afin d'être lu par des programmes informatiques comme celui que vous êtes en train d'écrire. Voici comment est formé un tel fichier (voir plus loin, à titre d'exemple, un extrait de fichier EMBL) :

Chaque ligne comporte 120 caractères au maximum, et contient des données d'une certaine nature ; la nature des données présentes sur une ligne est indiquée par les deux premiers caractères de la ligne, eux-mêmes suivis de trois caractères `[ESPACE]`. Les principales lignes sont :

| `..` | **Nature des données** | **Occurrences pour une entrée** |
|---|---|---|
| `ID` | Identifiant unique de la séquence + quelques informations sur sa nature (linéaire / circulaire, longueur, *etc*.). | Exactement 1 ligne (la première de l'entrée) |
| `AC` | Numéro(s) d'accession. | 1 ou plusieurs lignes |
| `DT` | Date de création, puis date de dernière modification. | Exactement 2 lignes |
| `DE` | Description. | 1 ou plusieurs |
| `KW` | Mots-clés. | 1 ou plusieurs |
| `OS` | Espèce (et, s'il y a lieu, souche). | Exactement 1 |
| `OC` | Classification. | 1 ou plusieurs |
| `R.` | Référence bibliographique : `RN` = numéro, `RX` = référence dans la banque PUBMED, `RA` = auteurs, `RT` = titre, `RL` = revue, numéro, pages. | 1 ou plusieurs |
| `DR` | Références à des banques externes. | 0, 1 ou plus |
| `CC` | Commentaires. | 0, 1 ou plus |
| `FH` | En-tête des lignes de contenu, ou "*Feature Header*" (voir ci-dessous). | Exactement 1 |
| `FT` | Ligne de contenu, ou "*Feature*" (voir ci-dessous). | 0, 1 ou plus |
| `SQ` | Longueur de la séquence nucléotidique, et nombre de A, C, G, T, *etc*. | Exactement 1 |
| `XX` | Ligne vide pour améliorer la lisibilité. | 0, 1 ou plus |
| `//` | Fin de l'entrée. Peut marquer la fin du fichier, sinon, une autre entrée suit, commençant par une ligne `ID`, etc. | Exactement 1 ligne (la dernière de l'entrée) |

Les éléments annotés dans la séquence sont nommés « *Features* » et occupent des lignes préfixées par `FT`, dont la structure obéit à un format spécifique. Elles sont découpées en deux parties, correspondant à deux « colonnes », nommées sur la ligne `FH` (*Feature Header*) : `Key` et `Location/Qualifier`.

- La clé occupe les 16 premiers caractères de la ligne et indique la nature des informations qui suivent. Contrairement aux deux caractères en tête de ligne, la clé n'est pas répétée pour une même annotation : si les informations occupent plusieurs lignes, la colonne « `Key` » reste vide à partir de la deuxième ligne.
- Les informations commencent par la localisation, suivies de la description de l'annotation.
    - La localisation est formée de deux nombres (séparés par « `..` ») qui désignent le début et la fin de la zone annotée. Si la zone annotée est incomplète, le début est précédé de « `<` » et/ou la fin est précédée de « `>` ». Une annotation sur le brin inverse-complémentaire est indiquée par « `complement` » suivi des bornes entre parenthèses.
    - Les caractéristiques constituant l'annotation commencent par un caractère « `/` » suivi du type de caractéristique, puis « `=` » et les données elles-mêmes entre guillemets.

Par exemple, dans l'extrait qui suit, on trouve les annotations suivantes :

- La première annotation a pour clé « `source` » et concerne tous les nucléotides de la séquence : de 1 à 9106. On y trouve le nom de l'organisme (`/organism`), la souche (`/strain`), le type de molécule (`/mol_type`) et la référence (`/db_xref`) dans la base taxonomique (`taxon`).

Ensuite, on a conservé les annotations de trois gènes :

- Le premier gène (clé `gene`) se situe sur le brin inverse complémentaire (`complement`), et s'étend sur les 170 premiers nucléotides de la séquence (`1..170`), mais n'est pas complet : la partie manquante se situe en amont de la séquence ici annotée (`<1`). Comme le gène est sur le brin complémentaire, le fragment présent dans la séquence correspond au début du gène. La ligne `FT` suivante est la continuation de l'annotation `gene`, et indique qu'on a attribué une étiquette à ce locus (`/locus_tag`), qui permet de l'identifier de façon unique.
- La clé `CDS` (pour « *CoDing Sequence* ») correspond à la partie du gène qui code pour une protéine, c'est-à-dire sans tenir compte des portions non traduites telles que les 5' et 3' non codant, et les introns. Comme nous avons ici affaire à des prokaryotes, il n'y a naturellement pas d'intron ; si la CDS a exactement les mêmes bornes que le gène, ce n'est pas que ce dernier est dépourvu de 5' et 3' non codant, mais que les bornes de ces régions sont inconnues. Les informations suivantes montrent que la traduction commence (`/codon_start`) au début de la CDS, que le code génétique (`/transl_table`) est le code standard (dénommé « `11` »), et rattachent cette CDS au gène nommé précédemment (`/locus_tag`). Vient ensuite la caractérisation de la protéine traduite à partir de cette CDS : sa fonction (`/product`), sa référence (`/db_xref`) dans une autre banque (ici `UniprotKB/TrEMBL`) et un numéro d'identification propre à EMBL.
- Les autres gènes comportent les mêmes type d'annotation.

# Exemple de fichier EMBL

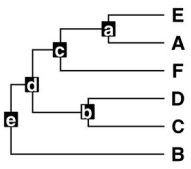
*N.B.* des lignes ont été supprimées pour raccourcir l'exemple, elles apparaissent sous la forme `[...]`

```
ID   AIMD01000001; SV 1; linear; genomic DNA; WGS; PRO; 9106 BP.
XX
AC   AIMD01000001;
XX
DT   16-JUL-2012 (Rel. 113, Created)
DT   26-FEB-2014 (Rel. 119, Last updated, Version 2)
XX
DE   Bartonella taylorii 8TBB cont1.1, whole genome shotgun sequence.
XX
KW   WGS.
XX
OS   Bartonella taylorii 8TBB
OC   Bacteria; Proteobacteria; Alphaproteobacteria; Rhizobiales; Bartonellaceae;
OC   Bartonella.
XX
RN   [1]
RP   1-9106
RG   The Broad Institute Genome Sequencing Platform, The Broad Institute Genome
RG   Sequencing Center for Infectious Disease
RA   Feldgarden M., Kirby J., Kosoy M., Birtles R., Probert W.S.,
[...]
RA   Birren B.;
RT   "The Genome Sequence of Bartonella taylorii 8TBB";
RL   Unpublished.
[...]
DR   ENA; AIMD00000000; SET.
DR   ENA-CON; JH725050.
DR   BioSample; SAMN02596908.
XX
CC   ##Genome-Assembly-Data-START##
CC   Assembly Method       :: allpaths v. R40250
CC   Assembly Name         :: Bart_tayl_8TBB_V1
CC   Genome Coverage       :: 158x
CC   Sequencing Technology :: Illumina
CC   ##Genome-Assembly-Data-END##
XX
FH   Key             Location/Qualifiers
FT   source          1..9106
FT                   /organism="Bartonella taylorii 8TBB"
FT                   /strain="8TBB"
FT                   /mol_type="genomic DNA"
FT                   /db_xref="taxon:1094560"
FT   gene            complement(<1..170)
FT                   /locus_tag="ME9_00001"
FT   CDS             complement(<1..170)
FT                   /codon_start=1
FT                   /transl_table=11
FT                   /locus_tag="ME9_00001"
FT                   /product="hypothetical protein"
FT                   /db_xref="UniProtKB/TrEMBL:J1KJX9"
FT                   /protein_id="EJF97935.1"
FT   gene            410..2401
FT                   /locus_tag="ME9_00002"
FT   CDS             410..2401
FT                   /codon_start=1
FT                   /transl_table=11
FT                   /locus_tag="ME9_00002"
FT                   /product="hypothetical protein"
FT                   /db_xref="GOA:J0RGJ3"
FT                   /db_xref="InterPro:IPR011055"
FT                   /db_xref="InterPro:IPR016047"
FT                   /db_xref="UniProtKB/TrEMBL:J0RGJ3"
FT                   /protein_id="EJF97936.1"
FT   gene            3283..3393
FT                   /locus_tag="ME9_00003"
FT   CDS             3283..3393
FT                   /codon_start=1
FT                   /transl_table=11
FT                   /locus_tag="ME9_00003"
FT                   /product="hypothetical protein"
FT                   /db_xref="UniProtKB/TrEMBL:J1KIW6"
FT                   /protein_id="EJF97937.1"
[...]
SQ   Sequence 9106 BP; 2929 A; 2044 C; 1570 G; 2563 T; 0 other;
//
```

# Annexe B
## L'algorithme UPGMA

### 1. Introduction

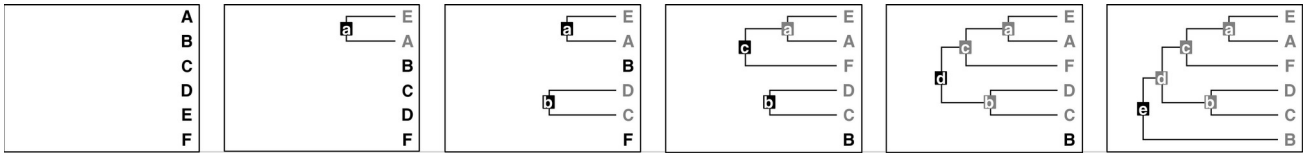
La méthode « *Unweighted Pair Group Method with Arithmetic mean* » (*UPGMA*) construit un arbre qui représente le degré de ressemblance entre des objets dont on connaît au départ la distance deux à deux. Comme son nom le suggère, l'algorithme consiste à grouper les objets par paires, chaque paire devenant un nouvel objet qui remplace les deux objets qui la constituent. L'arbre obtenu est dit « binaire » parce que chaque nœud interne possède deux descendants directs.




Un arbre binaire. Les objets classés sont ici désignés par les lettres majuscules : **A**, … **F**. Ils constituent les « feuilles », ou « nœuds terminaux » de l'arbre, c'est-à-dire qu'ils n'ont pas de descendants. Les « nœuds internes » sont ici nommés par les lettres minuscules **a**, … **e**. Chaque nœud interne a un parent et deux descendants, qui peuvent être des nœuds internes ou terminaux. Par exemple, le nœud **c** a pour parent le nœud **d**, et pour descendants le nœud interne **a** et la feuille **F**.

### 2. Des feuilles à l'arbre

Pour obtenir un tel arbre, UPGMA part de la liste des feuilles, et crée un à un les nœuds internes :




| Départ | Étape 1 | Étape 2 | Étape 3 | Étape 4 | Étape 5 |
|---|---|---|---|---|---|
| Il y a six objets à classer, ce seront les feuilles de l'arbre. | **A** et **E** sont groupés. Leur parent commun est **a**. Il reste cinq objets à classer : **a**, **B**, **C**, **D**, **F**. | Le nouveau nœud interne, **b**, a pour descendants **C** et **D**. Il reste à classer **a**, **B**, **b**, **F**. | Les nœuds **F** et **a** sont groupés, avec **c** comme parent. Il reste à classer **b**,**c**,**B**. | Deux nœuds internes sont groupés. Leur parent est nommé **d**. Il ne reste à classer que **d** et **B**. | Le dernier nœud regroupe **b** et **B**. Ce nœud, **e**, est la racine de l'arbre. |

L'ordre de ces étapes de regroupement est dicté par les distances entre objets. Au commencement de l'algorithme, les distances entre toutes les paires d'objets forment une matrice. À chaque étape :

- On groupe les deux objets les plus proches dans la matrice : ceux dont la distance est la plus petite. Nommons cette distance $d_{min}$. On crée un nouveau nœud interne : leur parent.
- On supprime de la matrice les deux objets qu'on vient de grouper, et on y ajoute leur parent. La matrice compte donc maintenant une ligne et une colonne de moins qu'au départ.
- Du nouveau nœud partent deux branches dont la longueur jusqu'aux feuilles est $d_{min}/2$. Attention : cela signifie que la longueur de la branche du nœud parent à l'un de ses descendants est $d_{min}/2$ seulement si ce descendant est une feuille. Si au contraire le descendant est lui-même un nœud interne, alors la distance est $d_{min}/2 - d_f$, où $d_f$ est la longueur de branche entre le descendant et une feuille de sa propre descendance.
- La distance entre le parent nouvellement ajouté et chaque autre objet de la matrice, est la moyenne des distances entre cet objet et les deux nœuds nouvellement groupés. Cette moyenne est une moyenne pondérée : le poids de chacun des deux nœuds est le nombre de feuilles qu'il a dans sa descendance, en attribuant à chaque feuille un poids de 1. Par exemple, sur l'arbre ci-dessus, le poids du nœud **b** est $w_b = 2$, et le poids du nœud **d** est $w_d = 5$. Le poids de chaque nœud interne est donc la somme des poids de ses deux nœuds descendants directs.

  Tous ces calculs sont explicités sur l'exemple de la page suivante.

**Construction d'un arbre pas à pas** : la plus courte distance dans la matrice est surlignée en jaune. Le nombre qui suit chaque nœud interne est son poids (nombre de feuilles dans sa descendance). Les branches sont dessinées à l'échelle.

## ÉTAPE 1

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 42 | 36 | 40 | 16 | 32 |
| B | 42 | 0 | 45 | 49 | 31 | 53 |
| C | 36 | 45 | 0 | 22 | 40 | 34 |
| D | 40 | 49 | 22 | 0 | 36 | 42 |
| E | 16 | 31 | 40 | 36 | 0 | 28 |
| F | 32 | 53 | 34 | 42 | 28 | 0 |


$d_{aE} = d_{aA} = d_{EA}/2 = 16 / 2 = 8$

$w_a = w_E + w_A = 1 + 1 = 2$

Exemple de calcul pour la nouvelle matrice :

$d_{aD} = (w_E \times d_{ED} + w_A \times d_{AD}) / (w_E+w_A)$

$= (1 \times 36 + 1 \times 40) / (1 + 1) = 38$

## ÉTAPE 2

| | B | C | D | F | a |
|---|---|---|---|---|---|
| B | 0 | 45 | 49 | 53 | 36.5 |
| C | 45 | 0 | 22 | 34 | 38 |
| D | 49 | 22 | 0 | 42 | 38 |
| F | 53 | 34 | 42 | 0 | 30 |
| a | 36.5 | 38 | 38 | 30 | 0 |


$d_{bD} = d_{bC} = d_{DC}/2 = 22 / 2 = 11$

$w_b = w_D + w_C = 1 + 1 = 2$

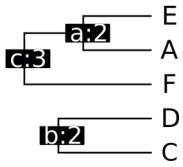
Exemple de calcul pour la nouvelle matrice :

$d_{bB} = (w_D \times d_{DB} + w_C \times d_{CB}) / (w_D+w_C)$

$= (1 \times 49 + 1 \times 45) / (1 + 1) = 47$

## ÉTAPE 3

| | B | F | a | b |
|---|---|---|---|---|
| B | 0 | 53 | 36.5 | 47 |
| F | 53 | 0 | 30 | 38 |
| a | 36.5 | 30 | 0 | 38 |
| b | 47 | 38 | 38 | 0 |




$d_{cE} = d_{cA} = d_{cF} = d_{aF}/2 = 30/2 = 15$

$d_{ca} = d_{cE} - d_{aE} = d_{cA} - d_{aA} = 15 - 8 = 7$

$w_c = w_a + w_F = 2 + 1 = 3$

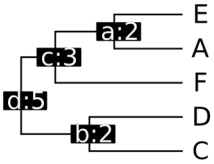
Exemple de calcul pour la nouvelle matrice :

$d_{cB} = (w_a \times d_{aB} + w_F \times d_{FB}) / (w_a+w_F)$

$= (2 \times 36.5 + 1 \times 53) / (2 + 1) = 42$

## ÉTAPE 4

| | B | b | c |
|---|---|---|---|
| B | 0 | 47 | 42 |
| b | 47 | 0 | 38 |
| c | 42 | 38 | 0 |




$d_{dE} = d_{dA} = d_{dF} = d_{dD} = d_{dC} = d_{cb}/2 = 38/2 = 19$

$d_{dc} = d_{dE} - d_{cE} = d_{dA} - d_{cA} = d_{dF} - d_{cF} = 19 - 15 = 4$

$d_{db} = d_{dD} - d_{bD} = d_{dC} - d_{bC} = 19 - 11 = 8$

$w_d = w_c + w_b = 3 + 2 = 5$

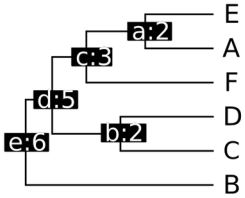
Exemple de calcul pour la nouvelle matrice :

$d_{dB} = (w_c \times d_{cB} + w_b \times d_{bB}) / (w_c+w_b)$

$= (3 \times 42 + 2 \times 47) / (3 + 2) = 44$

## ÉTAPE 5

| | B | d |
|---|---|---|
| B | 0 | 44 |
| d | 44 | 0 |




$d_{eE} = d_{eA} = d_{eF} = d_{eD} = d_{eC} = d_{eB} = d_{dB}/2 = 44/2 = 22$

$d_{ed} = d_{eE} - d_{dE} = d_{eA} - d_{dA} = d_{eF} - d_{dF}$

$= d_{eD} - d_{dD} = d_{eC} - d_{dC} = 22 - 19 = 3$

$w_e = w_d + w_B = 5 + 1 = 6$

## FIN

| | e |
|---|---|
| e | 0 |

La matrice comporte maintenant un seul taxon.
La construction de l'arbre est terminée, sa racine est e.

## 3. L'algorithme UPGMA en détail

L'algorithme présenté ici constitue une implémentation possible d'UPGMA, mais il est aussi permis d'introduire des variations, en particulier en ce qui concerne les structures de données.

```
fonction upgma(matrice) :
    nœuds, matdist = matrice
    arbre = {}
    racine = None
    tant_que taille(matdist) > 1 :
        Ni, Nj = nœuds_les_plus_proches(matdist)
        [Li, Lj] = longueurs_des_branches(Ni, Nj, nœuds, matdist, arbre)
        parent, Wi, Wj = ajouter_nœud(arbre, Ni, Li, Nj, Lj, nœuds)
        remplacer_nœuds(Ni, Nj, parent, noeuds)
        matdist = mise_a_jour(matdist, Ni, Nj, arbre, Wi, Wj, nœuds)
    retourner arbre, parent
```

**matrice** désigne la matrice des distances initiale. Il s'agit d'un tuple composé de la liste des noms des nœuds (nommée **nœuds** ; au lancement de l'algorithme, c'est la liste des objets à classer, donc les feuilles) et de la véritable matrice de distances (nommée **matdist**), qui est une liste de listes de nombres où, bien entendu, les lignes et les colonnes suivent le même ordre que la liste **nœuds**. Comme cette matrice carrée est symétrique par rapport à la diagonale (car la distance de A à B est toujours égale à la distance de B à A), on peut en remplir une seule moitié. Par exemple :

La matrice
```
 0 30 12 46
30  0 23 31
12 23  0 27
46 31 27  0
```
s'écrit comme la liste de listes : `[[0],[30,0],[12,23,0],[46,31,27,0]]`

L'arbre est ici stocké sous forme d'un dictionnaire, dont les clés seront les nœuds internes. C'est pourquoi ce dictionnaire est vide au début. Ceci permet d'identifier facilement les feuilles : si un nœud n'est pas une clé du dictionnaire **arbre**, alors c'est une feuille. La valeur associée à chaque clé est un tuple, qui doit contenir le poids du nœud, et pour chaque descendant de ce nœud, son nom et la longueur de la branche qui y mène.

La fonction **nœuds_les_plus_proches** prend en entrée la matrice de distances, et retourne les numéros des deux nœuds pour lesquels la distance est la plus petite : $N_i$ et $N_j$.

La fonction **longueurs_des_branches** calcule la longueur entre chaque branche et son parent. Comme nous l'avons vu plus haut :

- on sait que la distance entre le nœud parent et une feuille est **d** = **dmat**[$N_i$][$N_j$] / 2.
- Pour la branche menant du parent à son descendant **i** :
  - si ce descendant est une feuille, on garde **d** comme longueur de branche entre le nœud parent et le descendant **i**.
  - sinon :
    - on diminue **d** de la distance entre **i** et l'un de ses descendants (peu importe lequel, le calcul donne le même résultat pour tous les chemins jusqu'à une feuille quelconque).
    - on recommence jusqu'à atteindre une feuille.
- La même procédure est appliquée pour le descendant **j**.

La fonction retourne les deux longueurs de branches, sous forme d'une liste.

La fonction **ajouter_nœud** crée le nouveau nœud parent, à qui il donne un nom. Chaque nom de nœud doit être unique. Il faut calculer le poids de ce nœud comme la somme des poids de ses deux descendants (Rappel : si un descendant est une feuille, son poids est 1, sinon son poids est stocké dans l'arbre). Toutes les informations relatives au nouveau nœud sont alors disponibles, et on peut créer un nouvel élément du dictionnaire **arbre** (*cf* ci-dessus). La fonction retourne le nom du nouveau nœud parent, ainsi que les poids de ses deux descendants (nous en aurons bientôt besoin).

La fonction **remplacer_nœuds** modifie la liste **nœuds** : elle enlève les nœuds **i** et **j** de la liste, et y ajoute leur parent à la fin. Elle ne retourne rien.

La fonction **mise a jour** crée une nouvelle matrice, où le nouveau nœud parent **p** remplace ses deux descendants, **i** et **j**. Les distances autres que celles avec ces nœuds ne changent pas. Il suffit donc de recopier toutes les valeurs de la matrice pour les couples de nœuds qui ne sont ni **i** ni **j**. Ainsi, la nouvelle matrice contient toutes les distances inchangées, et préserve l'ordre des nœuds, conforme à celui de la liste **nœuds**. Une nouvelle ligne peut alors être ajoutée : les distances entre le nouveau nœud **p** et tous les autres. Pour chaque nœud **k** (autre que **i** ou **j**), la nouvelle distance est :

$$d_{p,k} = ( matdist[i][k] * Wi + matdist[j][k] * Wj ) / ( Wi + Wj )$$

La fonction retourne la nouvelle matrice.

La fonction **upgma** réitère ces étapes tant que la matrice contient au moins deux nœuds. Le dernier nœud restant est le dernier parent qui vient d'être créé. C'est la racine de l'arbre. Il est possible de retrouver cette racine dans le dictionnaire **arbre** (il s'agit du seul nœud qui n'a pas de parent) mais le plus simple est que la fonction retourne cette racine.

**4. Comment afficher l'arbre, et vérifier qu'il est correct ?**

L'affichage du dictionnaire permet de vérifier que l'arbre construit est correct. Avec la matrice donnée en exemple, on doit obtenir (sans oublier que dans un dictionnaire, l'ordre des clés n'a pas d'importance) :

```
{'a': (2, ('E', 8.0), ('A', 8.0)), 'b': (2, ('D', 11.0), ('C', 11.0)), 'c': (3, ('a', 7.0), ('F', 15.0)), 'd': (5, ('c', 4.0), ('b', 8.0)), 'e': (6, ('d', 3.0), ('B', 22.0))}
```

En partant de la racine : **e**, il est possible de reconstituer l'arbre. Cependant, on ne peut pas dire que la structure saute aux yeux. La structure d'un arbre se représente plus classiquement sous forme de parenthèses emboîtées : **((((E,A),F),(D,C)),B)**.

Noter que l'ordre des descendants d'un nœud interne est sans importance pour la structure de l'arbre. Ainsi, les arbres suivants sont en fait identiques (et identiques à l'arbre ci-dessus) :

`(B,((C,D),(F,(A,E))))` `((((E,A),F),(C,D)),B)` `(B,((F,(A,E)),(D,C)))`

En revanche, les arbres suivants ne sont pas identiques (ni entre eux, ni avec le précédent) :

`(B,(((C,D),F),(A,E)))` `((((E,F),A),(C,D)),B)` `((B,F),((A,E),(D,C)))`

**4.a. Ordonner grâce à une pile**

Afin de passer du dictionnaire à la forme parenthésée, nous allons utiliser une structure de données classique en algorithmique : une pile. Elle ressemble à une pile d'assiettes, pour laquelle l'on dispose de deux opérations :

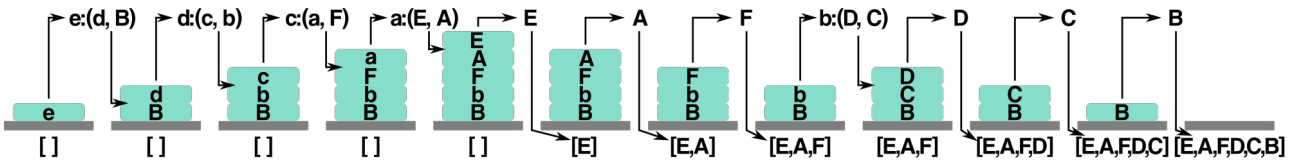
- empiler = déposer un objet sur le dessus de la pile ;
- dépiler = retirer l'objet qui se trouve au sommet de la pile.

Dans un premier temps, nous voulons seulement obtenir les feuilles, dans l'ordre où elles apparaissent dans l'arbre sous forme parenthésée. Le résultat sera une liste, que nous initialisons comme une liste vide : **Res=[]**. Commençons par déposer sur la pile, la racine de l'arbre. La pile contient donc un objet : **e**.

Répétons ensuite les opérations suivantes, jusqu'à ce que la liste soit vide :

- Dépiler.
  - Si l'objet est un nœud interne de l'arbre, empiler ses deux descendants (peu importe l'ordre) ;
  - Sinon (il s'agit d'une feuille), ajouter l'objet à la liste.

On peut visualiser ce processus ainsi (en supposant connue une version simplifiée du dictionnaire arbre, qui associe à chaque nœud interne un tuple formé de ses deux descendants) :



La liste obtenue est dite « ordonnée selon l'arbre ». Pour implémenter cet algorithme en Python, le plus simple est de représenter la pile au moyen d'une liste. Empiler un élément revient à l'ajouter à la fin de la liste avec `append()`. Dépiler un élément revient à enlever le dernier élément de la liste grâce à `pop(-1)`.

### 4.b. Ajouter les parenthèses

Pour obtenir l'arbre parenthésé sous forme d'une chaîne de caractères, il suffit de modifier légèrement l'algorithme précédent :

- au lieu d'une liste vide, on initialise le résultat comme une chaîne vide.
- au lieu d'empiler simplement les descendants d'un nœud interne, on empile cinq éléments dans l'ordre :
  1. une parenthèse fermante : le caractère `')'`
  2. un nœud descendant
  3. une virgule : le caractère `','`
  4. l'autre nœud descendant
  5. une parenthèse ouvrante : le caractère `'('`

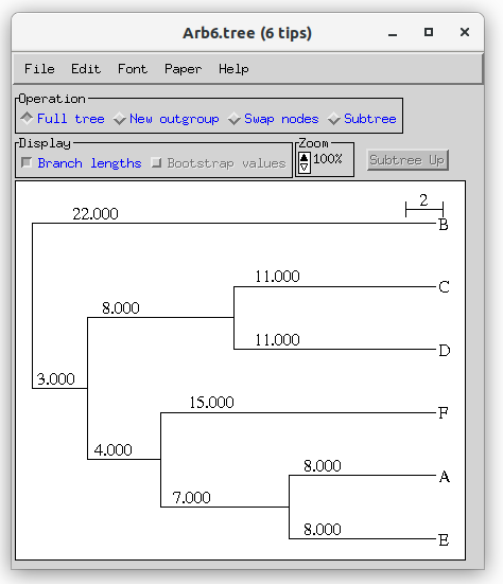
Comme les parenthèses et virgules ne sont pas des nœuds internes de l'arbre (ce ne sont pas des clés du dictionnaire), elles sont toujours ajoutées à la chaîne parenthésée lorsqu'elles sont dépilées. Voyons le détail du fonctionnement au moyen d'un tableau : l'opération « Empiler » consiste à empiler les cinq éléments définis ci-dessus ; l'opération « Ajouter » consiste à allonger la chaîne parenthésée en ajoutant le dernier élément dépilé.

| Pile | Dépilé | Opération | Chaîne parenthésée |
|---|---|---|---|
| `['e']` | `'e'` | Empiler | `''` |
| `[')','B',',','d','(']` | `'('` | Ajouter | `'('` |
| `[')','B',',','d']` | `'d'` | Empiler | `'('` |
| `[')','B',',',')','b',',','c','(']` | `'('` | Ajouter | `'(('` |
| `[')','B',',',')','b',',','c']` | `'c'` | Empiler | `'(('` |
| `[')','B',',',')','b',',',')','F',',','a','(']` | `'('` | Ajouter | `'((('` |
| `[')','B',',',')','b',',',')','F',',','a']` | `'a'` | Empiler | `'((('` |
| `[')','B',',',')','b',',',')','F',',',')','A',',','E','(']` | `'('` | Ajouter | `'(((('` |
| `[')','B',',',')','b',',',')','F',',',')','A',',’E']` | `'E'` | Ajouter | `'((((E'` |
| `[')','B',',',')','b',',',')','F',',',')','A',',']` | `','` | Ajouter | `'((((A,'` |
| `[')','B',',',')','b',',',')','F',',',')','A']` | `'A'` | Ajouter | `'((((E,A'` |
| `[')','B',',',')','b',',',')','F',',',')']` | `')'` | Ajouter | `'((((E,A)'` |
| `[')','B',',',')','b',',',')','F',',']` | `','` | Ajouter | `'((((E,A),'` |
| `[')','B',',',')','b',',',')','F']` | `'F'` | Ajouter | `'((((E,A),F'` |
| `[')','B',',',')','b',',',')']` | `')'` | Ajouter | `'((((E,A),F)'` |
| `[')','B',',',')','b',',']` | `','` | Ajouter | `'((((E,A),F),'` |
| `[')','B',',',')','b']` | `'b'` | Empiler | `'((((E,A),F),'` |
| `[')','B',',',')',')','C',',','D','(']` | `'('` | Ajouter | `'((((E,A),F),('` |
| `[')','B',',',')',')','C',',','D']` | `'D'` | Ajouter | `'((((E,A),F),(D'` |
| `[')','B',',',')',')','C',',']` | `','` | Ajouter | `'((((E,A),F),(D,'` |
| `[')','B',',',')',')','C']` | `'C'` | Ajouter | `'((((E,A),F),(D,C'` |
| `[')','B',',',')',')']` | `')'` | Ajouter | `'((((E,A),F),(D,C)'` |
| `[')','B',',',')']` | `')'` | Ajouter | `'((((E,A),F),(D,C))'` |
| `[')','B',',']` | `','` | Ajouter | `'((((E,A),F),(D,C)),'` |
| `[')','B']` | `'B'` | Ajouter | `'((((E,A),F),(D,C)),B'` |
| `[')']` | `')'` | Ajouter | `'((((E,A),F),(D,C)),B)'` |
| `[]` | Fin de l'algorithme | | `'((((E,A),F),(D,C)),B)'` |

Enfin, si l'on veut obtenir un arbre comprenant les longueurs des branches, on peut également empiler, avant chaque nœud, une chaîne formée du caractère « : » suivi de la longueur de sa branche. Avec notre exemple, on doit alors obtenir la chaîne :

```
((((E:8.0,A:8.0):7.0,F:15.0):4.0,(D:11.0,C:11.0):8.0):3.0,B:22.0)
```

Ce format de représentation est nommé « newick ». Il peut être lu et interprété par de nombreux programmes. Par exemple, le logiciel NJPlot[1] permet d'afficher et sauvegarder une représentation graphique de l'arbre :




1 https://doua.prabi.fr/software/njplot