UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

TD3 - Calcul de la distance phylogénétique

Adil BELMOKADDEM
April 30, 2017

1 CALCUL DE LA DISTANCE PHYLOGÉNÉTIQUE

On a *15* fichiers au format .fna, contenant des chaînes d'ADN, sous forme de lettres. On a comme objectif, la compression de ses fichiers volumineux. On utilise la fonction *gzip.NewWriterLevel* du package *"compress/gzip"*, pour créer les archives. On aura *135* archives au total, *15* fichiers archivés, *15* duplications archivées, et *105* archives de touts les concaténations possibles entre deux fichiers. Les archives sont au format *.gz*

1.1 CHOIX D'UNE MÉTRIQUE

L'implémentation du programme est faite en langage Go. Au début, le programme crée trois répertoires via la fonction *os.Mkdir*, le répertoire *fnaFiles* pour mettre les *15* fichiers *fna* dedans, le répertoire *append* pour contenir les concaténations des fichiers *fna*, et le répertoire *archives* pour contenir les *135* archives. Avant l'exécution du programme, il faut mettre les fichiers *fna* dans le même dossier avec le code source.

1.1.1 CONCATÉNATION

On met les fichiers dans le dossier *fnaFiles* avec la fonction *os.Rename* La lecture du contenu d'un répertoire est fait par la fonction *ioutil.ReadDir*, pour sauvegarder les noms des fichiers dans un tableau. La fonction *append* implémente la concaténation des fichiers avec eux même et avec le reste des autres fichiers. Cette fonction prend deux fichiers en paramètres pour les concaténer, la lecture des deux fichier est faite par la fonction *ioutil.ReadFile*, puis on crée une chaîne de caractères qui est la somme des des noms des deux fichiers, la fonction *strings.Trim* renomme le fichier, et *os.Create* crée le fichier **vide** concaténé dans le dossier *append*. Enfin, *ioutil.WriteFile* écris ce fichier concaténé.

Il y a deux boucles *for* pour faire la concaténation, la première concatène le chaque fichier avec lui même, la deuxième concatène chaque fichier avec les autres fichiers.

1.1.2 COMPRESSION

Maintenant que le dossier *append* contient les fichiers concaténés, on scanne ce dernier pour sauvegarder les noms des fichiers concaténés dans un tableau.

La fonction *compress* implémente la compression, elle retourne la taille de l'archive (*float64*), et prend comme paramètres, le chemin du fichier à compresser, le chemin de l'archive à créer, et le nom du fichier à compresser. la fonction *gzip.NewWriterLevel* est utilisée pour compression, et *Write([]byte(file2))* pour écrire le contenu de l'archive. Pour récupérer la taille, on utilise la fonction *os.Stat* pour récupérer les informations à propos de l'archive, et puis la fonction *stat.Size* pour récupérer la taille, on fait un *cast* de type *float64* sur le retour de la fonction de compression, pour avoir la taille en double précision. Maintenant, on peut scanner le dossier *archives* pour récuperer les noms des *135* archives dans un tableau, et calculer les distances.

1.1.3 CALCUL DE LA DISTANCE PHYLOGÉNÉTIQUE

La formule utilisée pour calculer la distance phylogénétique est comme suit:

$$Z(A,B) = \frac{Z(A\&B)}{Z(A)+Z(B)} - \frac{Z(A\&A)}{4*Z(A)} - \frac{Z(B\&B)}{4*Z(B)} \tag{1.1}$$

La boucle *for* de la ligne *304*, remplie la matrice des distances. On ne parcoure que la partie triangulaire supérieure de la matrice, et on initialise l'indice transposé par la même valeur de la distance de l'indice courant, j'ai calculé la distances entre un fichier et lui même pour remplir la diagonale, les distances de la diagonale sont bien nulles. L'ordre dans lequel les 135 archives sont stockées, est de tel façon que, l'archive d'un seul fichier (non concatńé), est suivie par les archives de la concaténation de ce fichier avec lui même, ainsi que, les archives de sa concaténation avec les autres fichiers. De ce fait, il faut passer les bons indices à la fonction *distance*, pour avoir des résultats cohérants pour les distances. La fonction *distance* calcule la distance entre deux archives selon la formule définie dans (1.1), pour que la valeur de retour soit positive, j'ai utilisé la fonction *math.Abs* pour avoir la valeur absolue. La matrice r_'esultante doit être symmétrique définie positive.

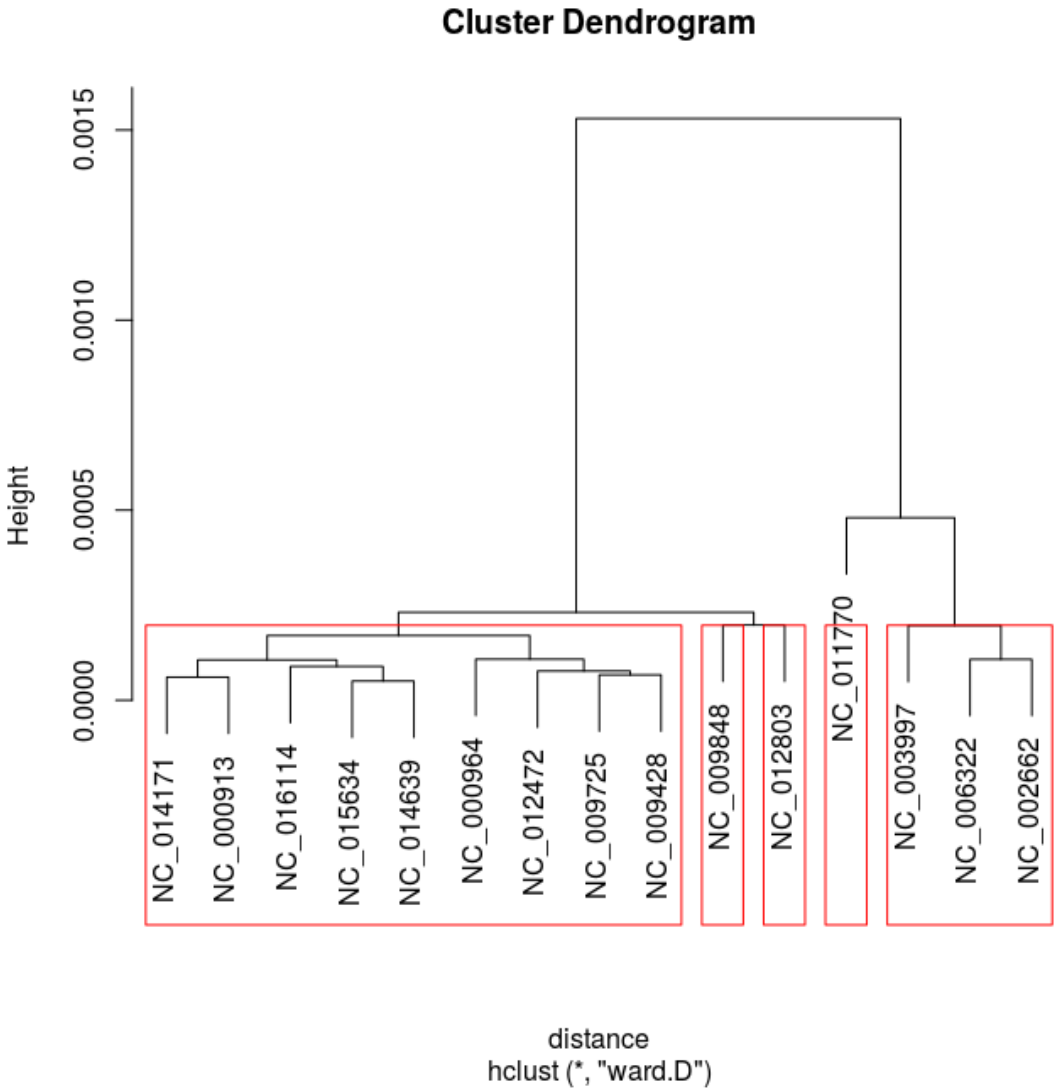
La fonction *distanceFileFillIn* génère deux fichiers, le premier est *Distances.txt* qui sera passé à Njplot pour afficher l'arbre. Le deuxième fichier est *Distances_R.txt* qui sera utilisé dans le script *R*, pour faire la clusterisation hiérarchique.

1.2 CLUSTERISATION HIÉRARCHIQUE AVEC *R*

Le script en langage *R* pour faire la clusterisation hiérarchique, est le suivant:

```r
dist_mat = read.table ('Distances_R.txt',TRUE, sep=';')
dist_mat =dist_mat [, 1:16]
file_names = names (dist_mat)
file_names=file_names [2:16]
library (data.table)
setattr(dist_mat , "row.names", file_names)
dist_mat =dist_mat [,2:16]
distance <- dist(dist_mat , method = "euclidean")
fit <- hclust(distance, method="ward.D")
plot(fit)
groups <- cutree(fit, k=5)
rect.hclust(fit, k=5, border="red")
```

Clusterisation hiérarchique générée par le script *R*



2 Implémentation en Golang de la méthode dite single linkeage (saut minimal)

La matrice des distances est prête, alors on peut calculer le *Dendrogramme*. L'idée est de faire les itérations jusqu'à ce que l'on obtient un seul ensemble, en éliminant les colonnes et les lignes contenant la distance la plus petite, puis on regroupe les lignes et les colonnes d'indices i et j (qui sont les indices du plus petit élément trouvé), et comparant les cellules des colonnes *i* et *j*, et les cellules des lignes *i* et *j*, on met le minimum entre deux cellules (resp. la ligne) de l'indice le plus petit. On remplace les valeurs dans les lignes et colonnes du plus petit élément par une grande valeur (*90*).

Les fonctions suivantes ont été implémentées:

- func singleLinkage(matrix [][]float64)
  - implémente l'algorithme du *saut minimal*, et génère le fichier *Tree.txt*, qui contient la chaîne de la forme standard *Netwick*.
- func min_matrix(matrix [][]float64) (int, int, float64)
  - Cette fonction, en quête de performance, ne parcourt que la partie triangulaire supérieure de la matrice symmétrique pour trouver le plus petit élément.
- func minimum_maximum(x, y int) (int, int)
  - Compare deux éléments entiers, *x* et *y*, et retourne le plus petit et le plus grand.
- func fmini(x, y float64) float64
  - Compare deux éléments de type double précision flottante, *x* et *y*, et retourne le plus petit des deux.

La chaîne *netwick*, est construite comme suit:

- Variable contenant la chaîne
  - chaine := ""
- Mise à jour
  - chaine = "(" + fnaNames[i_index] + "," + fnaNames[j_index] + ")"
- Sauvegarde du nouvel ensemble :
  - fnaNames[ind_ensemble] = chaine
- Finir la chaîne par un point-virgule
  - chaine = chaine + ";"

Cette chaîne se trouve dans le fichier *Tree.txt*.

3 CONSTRUCTION DE L'ARBRE

On utilise *Njplot* pour construire l'arbre de la standard *Netwick*, on tape cette commande dans le terminal:

- njplot -lengths -psonly Tree.txt

Cette commande génère l'arbre dans le fichier *Tree.ps* (*post-script*).

Voici l'arbre généré par *Njplot*: