ICC (SV) – Mini-projet

Barbara Jobstmann, Jamila Sam, Samuel Teeporten

Amir Bouchaoui, Miguel Crespo

(Version 1.1)

Ce document est en couleur et contient des liens cliquables. Il est préférable de le consulter en format numérique. Il est très important de bien le lire, **entièrement** et de **comprendre les structures de données fournies** et le rôle des fonctions demandées avant de vous lancer dans le codage.

1 Introduction

Un arbre phylogénétique a pour but de montrer les relations qui ont lié différentes entités au fil du temps. Il expose la proximité évolutive, en termes de caractéristiques héréditaires (telles que des traits morphologiques, des séquences d'ADN/ARN, des séquences d'acides aminés etc.) entre différentes entités. La figure 1 représente un arbre phylogénétique établi sur la base de données ARN. Cet arbre phylogénétique, proposé par Carl Woes, montre la séparation évolutive entre des bactéries, des archées et des eucaryotes. Les entités qui sont proches les unes des autres dans cet arbre ont des séquences d'ARN similaires.

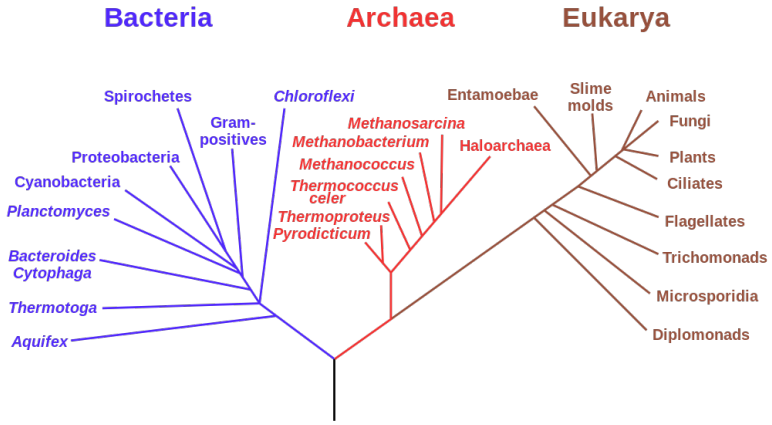

Fig. 1 : Arbre phylogénétique (supposé) des organismes vivants https://en.wikipedia.org/wiki/File:Phylogenetic_tree.svg

Les arbres phylogénétiques sont importants car ils nous aident à comprendre comment différents organismes sont liés par l'évolution ; comment les espèces ont changé et se sont ramifiées au fil du temps. En examinant ces arbres, les scientifiques peuvent déterminer

quelles espèces ont des ancêtres communs, suivre l'évolution des gènes et observer des schémas d'adaptation. Ces arbres sont utilisés dans de nombreux domaines tels que la biologie, l'écologie et la recherche médicale pour étudier la diversité et les connexions dans le monde du vivant.

En épidémiologie, par exemple, ces arbres permettent de suivre la propagation et l'évolution des maladies. En analysant les relations génétiques entre les différentes souches d'un virus ou d'une bactérie, les scientifiques peuvent comprendre comment les infections se propagent dans différentes populations et identifier les sources potentielles d'épidémies.

Dans la recherche sur le cancer, les arbres phylogénétiques aident les chercheurs à comprendre les changements génétiques qui conduisent au développement et à la progression des tumeurs. En étudiant l'histoire évolutive des cellules cancéreuses, les scientifiques peuvent identifier les principales mutations et voies impliquées dans la croissance du cancer, ce qui peut contribuer au développement de thérapies ciblées et d'approches thérapeutiques personnalisées.

2 Objectifs et Exemple

L'objectif de ce projet est de créer des arbres phylogénétiques en utilisant les deux méthodes de regroupement hiérarchique suivantes :

1. WPGMA (Weighted Pair Group Method with Arithmetic Mean)
2. UPGMA (Unweighted Pair Group Method with Arithmetic Mean)

Nous illustrons l'algorithme WPGMA à l'aide d'un petit exemple. L'algorithme UPGMA est très similaire et sera expliqué plus loin. Imaginons que nous ayons quatre organismes ayant des codes génétiques différents, c'est-à-dire des séquences d'ADN composées des quatre bases Adénine (A), Cytosine (C), Guanine (G) et Thymine (T). Comme le code génétique d'une entité peut être très vaste, on se concentre généralement sur une partie intéressante pour comparer les organismes (par exemple, un gène ou moins). Nous appellerons un élément classifiable tel qu'une partie d'une séquence d'ADN un **taxon**. Supposons que nous souhaitions analyser la relation entre les quatre fragments d'ADN suivants (T1, T2, T3, T4).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T1	C	A	T	A	G	A	C	C	T	G	A	C	G	C	C	A	G	C	T	C
T2	C	A	T	A	G	A	C	C	C	G	C	C	A	T	G	A	G	C	T	C
T3	C	G	T	A	G	A	C	T	G	G	G	C	G	C	C	A	G	C	T	C
T4	C	C	T	A	G	A	C	G	T	C	G	C	G	G	C	A	G	T	C	C

Le point de départ de cette méthode de regroupement est une matrice de distance qui indique la distance séparant deux entités. La distance entre deux entités est calculée en les comparant lettre par lettre. Par exemple, la distance entre le taxon T1 et le taxon T2 est de 5 parce qu'ils diffèrent dans les positions 8, 10, 12, 13 et 14. En calculant les distances par paire pour les quatre taxa (pour les quatre taxons), on obtient la matrice de distance donnée par la Table 1.

Au départ, chaque entité se trouve dans son propre groupe (« cluster ») . L'algorithme

Tab. 1 : Distance entre T1, T2, T3, et T4

	T1	T2	T3	T4
T1	0	5	4	7
T2	5	0	7	10
T3	4	7	0	7
T4	7	10	7	0

Tab. 2 : Distances entre T1T3, T2, et T4

	T1T3	T2	T4
T1T3	0	6	7
T2	6	0	10
T4	7	10	0

Tab. 3 : Distances entre T1T3T2 et T4

	T1T3T2	T4
T1T3T2	0	8.5
T4	8.5	0

procède par étapes. À chaque étape, deux « clusters » ayant une distance minimale sont fusionnés pour en construire un nouveau. La distance du nouveau « cluster » à tout autre « cluster » C est la moyenne des distances entre chacun des deux « cluster » et le « cluster » C. Dans notre exemple, au départ, chaque taxon se trouve dans son propre cluster, et la distance minimale entre deux clusters est 4, qui est la distance entre T1 et T3. Nous créons un nouveau groupe en fusionnant T1 et T3. La distance entre le nouveau cluster T1T3 et T2 est la moyenne entre (1) la distance entre T1 et T2, qui est de 5, et (2) la distance entre T3 et T2, qui est de 7. La distance entre T1T3 et T2 est donc de $(5 + 7)/2 = 6$. La distance entre T1T3 et T4 est $(7 + 7)/2 = 7$. Le tableau 2 montre la nouvelle matrice de distance après la fusion de T1 et T3. Dans l'étape suivante, nous fusionnons le cluster T1T3 avec T2 et obtenons la matrice de distance présentée dans le tableau 3. Nous continuons à fusionner jusqu'à ce que nous obtenions un seul groupe.

L'arbre phylogénétique est construit en regroupant itérativement les groupes les plus proches en termes de distance et en distribuant les distances sur les différentes branches. La fusion de T1 et T3 nous donne un arbre avec deux branches, une pour T1 et une pour T3. La distance entre T1 et T3 est de 4, qui sera répartie uniformément sur ces deux branches, de sorte que chaque branche se voit attribuer la valeur 2. La figure 2 montre l'arbre correspondant à la première étape de fusion. L'étape suivante, qui fusionne T1T3 avec T2, étend l'arbre d'un niveau avec T1 et T3 d'un côté et T2 de l'autre (voir la figure 3). La distance entre T1T3 et T2 est de 6, donc chacune de ces deux nouvelles branches doit représenter la valeur 3. Comme le sous-arbre de gauche (avec T1 et T3) a déjà la valeur 2, il suffit d'ajouter 1 pour arriver à un total de 3. Dans la dernière étape, le cluster T1T3T2 est fusionné avec T4, ce qui donne une autre couche à l'arbre. La distance entre ces deux groupes est de 8.5, ce qui donne 4.25 pour chaque nouvelle branche. La figure 4 montre l'arbre final généré pour T1, T2, T3 et T4 suivant l'algorithme WPGMA.

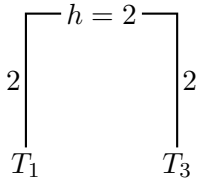

FIG. 2 : Arbre après l'étape 1

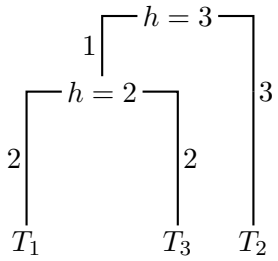

FIG. 3 : Arbre après l'étape 2

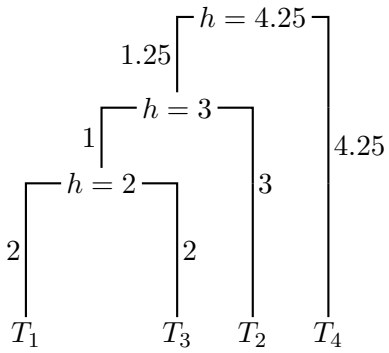

FIG. 4 : Arbre final

3 Structure et code fourni

Ce projet est divisé en 3 étapes :

1. Implémentation de fonctions de déverminage (« debugging »)
2. Initialisation de la structure de données représentant un arbre phylogénétique et calcul des matrices de distances.
3. Construction d'arbres phylogénétiques.

Chaque étape de ce projet nécessite de compléter plusieurs fonctions. Des types et fonctions utilitaires sont fournis dans les fichiers `utils.[hpp][.cpp]`. Les types et structures de données à utiliser sont définis dans le fichier `phylogenetics.hpp`.

Il est important de consulter les types et fonctions fournis dans le fichier `phylogenetics.hpp` et d'en comprendre le rôle.

À noter que si l'on n'utilise pas la directive `using namespace std;`, il faut modifier les identifiants comme `cout`, `vector` etc. en ajoutant `std::` comme préfixe (`std::cout`, `std::vector`, `std::endl`). Ceci est d'un usage courant en C++ et vous trouverez cette façon de faire dans le matériel fourni (nous y reviendrons au second semestre).

Vous devez compléter le code du fichier `phylogenetics.cpp`. Des instructions précises vous sont données dans ce document à ce sujet. Les prototypes des fonctions à implémenter se trouvent dans le fichier `phylogenetics.hpp` et **ne doivent pas être modifiés**.

Notez qu'en dehors des fonctions imposées, vous êtes libre de définir toutes les fonctions ou structures de données supplémentaires que vous jugez pertinentes.

Attention : tout ajout de fonction ou de type de données personnels devra impérativement se faire dans le fichier **`phylogenetics.cpp` car seul ce fichier sera rendu**.

Il relève des bonnes pratiques de créer un code propre et modulaire !

Dans l'annexe (chapitre A), à la fin de ce document, se trouve une description de la différence entre les algorithmes UGPMA et WGPMA.

Notez que dans les traces d'exécution fournies en exemple, vous pourrez voir des flèches telle que : ↪ . Cela ne fait pas partie de la sortie attendue mais désigne tout simplement la continuation d'une ligne coupée artificiellement dans le document pour en faciliter la lecture.

3.1 Utilitaires fournis

Un taxon, par exemple une protéine, peut être une très longue séquence de caractères. Il peut donc vite devenir fastidieux d'avoir à le saisir à chaque exécution du programme.

Le fichier `utils.hpp` fournit une fonction utilitaire `readSequencesFromFile(std::string filename)` permettant de lire des séquences de caractères depuis un fichier nommé `filename`. Vous trouverez dans le fichier fourni `sources/main.cpp` des exemples d'utilisation de cette fonction.

Le fichier `phylogenetics.hpp` fournit aussi le type `Taxon`, synonyme de `string` (chaînes de caractères), ainsi que le type énuméré `AlgoType` qui permettra de préciser quel algorithme on souhaite lancer pour construire les arbres phylogénétiques.

3.2 Fichiers de test fourni

Nous vous fournissons des données dans le dossier `data` afin de tester votre code. Chaque fichier texte (`.txt`) de ce répertoire est censé correspondre à un ensemble de taxa (taxons) à classifier. Vous êtes libre d'ajouter des fichiers de données à des fins de tests, mais ces derniers ne seront pas rendus.

Test et vérification des cas limites

Il est d'usage courant de tester les paramètres d'entrée des fonctions ; par exemple, vérifier qu'un tableau n'est pas vide, et/ou est de la bonne dimension, etc. Ces tests facilitent généralement le débogage, et vous aident à raisonner quant au comportement d'une fonction.

Nous supposerons que les arguments des fonctions sont par défaut corrects et qu'il n'est pas nécessaire de faire dessus des vérifications particulières. Dans

les cas particuliers où cela s'avère nécessaire, il faudra utiliser des instructions C++ de ce type :

```
if (condition) throw std::invalid_argument(" an error message ");
```

Si la condition est vraie, alors le programme affiche le message d'erreur sur le terminal et s'arrête, sinon il continue normalement. Par exemple, si le paramètre de type pointeur nommé `key` d'une fonction donnée `f` ne doit pas valoir `nullptr` pour un fonctionnement correct, vous pouvez écrire :

```
if (key == nullptr) throw std::invalid_argument(" key must not be
    ↪ nullptr in function f ");
```

au début de la fonction. Ce n'est cependant pas une nécessité absolue que ce genre d'instructions soient placée au début. Nous utilisons ici un mécanisme dit de *gestion des exceptions* qui sera étudié en détail au semestre prochain.

> Pour utiliser le mécanisme des exceptions, il suffit d'inclure la bibliothèque `exception`.

Il est de votre responsabilité de vérifier que votre programme se comporte correctement. Il est notamment important de **vérifier soigneusement à chaque étape que vous produisez bien des données correctes avant de passer à l'étape suivante qui utilisera ces données.**

Pour vous aider dans vos vérifications, nous vous fournissons cependant quelques exemples de tests, disponibles dans les fichiers `main.cpp` et `unit_test.cpp` (ceux de ce fichier sont déjà invoqués pour vous dans le fichier `main.cpp`). Vous pouvez utiliser ces tests et les compléter pour vous assurer que tout fonctionne. Notez que le code dans `main.cpp` et `unit_test.cpp` ne seront pas testés en tant que tels pendant la correction. Modifiez-les comme bon vous semble, en fonction de vos besoins.

Pour tester le code que vous avez écrit pour chaque partie, vous pouvez appeler la fonction `run_unit_tests(part)` en lui passant l'index `part` de la partie. Par exemple, si vous voulez tester votre code pour la partie 3 de ce projet, appelez la fonction `run_unit_tests` ↪ `(3)` dans le fichier `main.cpp`.

Attention les tests fournis ne sont pas exhaustifs. Il est recommandé d'essayer de les compléter par soi-même, notamment pour le traitement des cas limites.

Enfin, nous vous recommandons vivement de vous familiariser d'emblée avec l'usage du dévermineur (« debugger », voir à ce sujet le tutoriel de la série 8). Essayez dès le test de votre première fonction de placer des points d'arrêt pour examiner les valeurs des variables impliquées.

4 Implémentation

Ce projet est composés de 3 parties. Dans chaque partie, vous devez implémenter quelques fonctions dans le fichier `phylogenetics.cpp`.

4.1 Partie 1

L'objectif de la première partie du projet est d'implémenter des fonctions utilitaires permettant notamment d'afficher le contenu des structures de données.

Un « Cluster » est un noeud dans un arbre phylogénétique. Pour l'arbre de la figure 3 par exemple, il est prévu de modéliser les « Clusters » impliqués de la façon suivante :

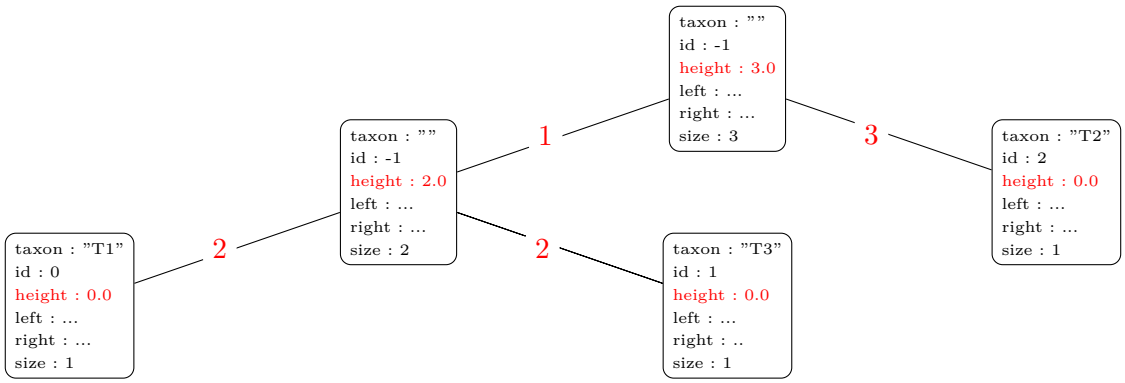

FIG. 5 : Arbre phylogénétique : vue de détail sur les noeuds (les informations en rouge sur les liens entre les noeuds ne font pas partie de la représentation des `Cluster`)

La structure de données `Cluster` est fournie pour modéliser la notion de « Cluster ». **Elle ne doit pas être modifiée** et contient les informations suivantes :

- `id` : un identifiant concis du noeuds sous la forme d'un entier ; cet identifiant n'est pertinent que pour les noeuds terminaux (feuilles de l'arbre) et vaudra -1 autrement ;
- `taxon` : une chaîne de caractères de type `Taxon` identifiant les noeuds terminaux. Il s'agira concrètement :
  - de l'intitulé du taxon (par exemple, une séquence d'ARN) pour les noeuds terminaux (feuilles) ;
  - d'une chaîne de caractère vide, pour les noeuds internes.
- `left` & `right` : des pointeurs sur les deux noeuds agrégés par le noeud courant : un pointeur pour celui de droite et un pour celui de gauche ;
- `height` : la hauteur du noeud qui donne la distance le séparant des noeuds terminaux, tel qu'expliqué dans l'exemple d'introduction. Par exemple, la distance séparant T1 et T3 étant de quatre (selon la matrice de distance du tableau 1), l'agrégat T1+T3 sera de hauteur quatre divisé par deux (répartition équitable de la distance sur chacune des branches).

- `size` : la « taille » du noeud définie comme le nombre de noeuds agrégés par le noeud courant. Par exemple (dans la figure 3) le noeud qui agrège les deux noeuds d'identifiants T1 (id=0) et T3 (id=1) est de taille 2 ; celui qui agrège ce noeud (de taille 2) avec le noeud d'identifiant T2 (de taille 1) est de taille 3. Notez que cette taille ne peut être négative et est un nombre entier.

Avant de poursuivre, examinez les alias de types fournis (voir `phylogenetics.hpp`). Vous constaterez notamment qu'un arbre (`Tree`) pour ce projet n'est autre qu'un tableau de `Cluster`. Cette structure de donnée `Tree` s'utilise conjointement avec une structure de donnée `DistanceMatrix` qui indique les distances séparant deux de ses `Cluster`. **L'ordre dans ces structures de données est important** : l'entrée `[i][j]` dans une `DistanceMatrix` donne la distance entre l'entrée `i` et l'entrée `j` du `Tree` correspondant.

Il vous est ensuite demandé de coder quelques fonctions utiles pour afficher le contenu des structures de données. Leur usage vous aidera notamment au déverminage de votre programme. Chacune des fonctions suivantes renvoie une *string* formatée selon des spécifications précises. Ces chaînes de caractères peuvent ensuite être affichées pour voir le contenu des structures de données correspondantes.

Veillez à ce que vos résultats correspondent à ceux fournis dans les exemples et les tests unitaires fournis.

Les fonctions qui ne sont pas encore codées contiennent la ligne `NotImplemented();` qui provoquera une erreur si elle est exécutée. Supprimez cette ligne une fois le corps de la fonction codé.

Voici le liste des fonctions à implémenter :

- `std::string toString(const DistanceMatrix& matrix, bool verbose = false ↪ );` Si `verbose` vaut `false`, cette fonction produit une chaîne de caractères représentant les valeurs d'une matrice de distances. Les valeurs sont séparées par un espace, et un saut et fait entre chaque ligne (le caractère `'\n'` représente un saut de ligne). Si `verbose` vaut `true`, la chaîne de caractères représente la matrice sous la forme d'un ensemble de $i - j = dist_{i,j}$ où i est un numéro de ligne, j un numéro de colonne et $dist_{i,j}$ la valeur de l'entrée $[i][j]$ de la matrice. Cette valeur de l'entrée doit être convertie en chaîne de caractères au moyen de la fonction fournie `double_to_string` (qui limite le nombre de décimales).

  La partie de la fonction `test_part1` dédiée à cette fonction et appelée par le `main` devrait produire l'affichage suivant pour la matrice testée `dist` (la méthode `toString` y est appelée deux fois, une fois avec le paramètre `verbose` valant `false` et une autre fois où il vaut `true`) :

```
 ======= Testing toString for DistanceMatrix =======
 ======= printing in non verbose mode by default ====
0 17 21 31 23
17 0 30 34 21
21 30 0 28 39
31 34 28 0 43
23 21 39 43 0
 ======= printing in verbose mode ====
0 - 0 = 0
0 - 1 = 17
0 - 2 = 21
0 - 3 = 31
0 - 4 = 23
1 - 0 = 17
1 - 1 = 0
1 - 2 = 30
1 - 3 = 34
1 - 4 = 21
2 - 0 = 21
2 - 1 = 30
2 - 2 = 0
2 - 3 = 28
2 - 4 = 39
3 - 0 = 31
3 - 1 = 34
3 - 2 = 28
3 - 3 = 0
3 - 4 = 43
4 - 0 = 23
4 - 1 = 21
4 - 2 = 39
4 - 3 = 43
4 - 4 = 0
 ======= End testing toString for DistanceMatrix =======
```

- `std::string toString(const ClusterIdPair& pair);` retourne la représentation sous forme de `string` d'un `ClusterIdPair` en utilisant le format `id1-id2\n`, où `id1` est le premier élément de `pair`, et `id2` le second. Le caractère `'\n'` est un saut de ligne en C++. Par exemple, si `pair` est initialisé avec 2 et 3, l'affichage de la `string` correspondante devrait être :

  ```
  2-3
  ```

  suivie d'un saut de ligne.

- `std::string clusterToString(const Cluster* cluster, bool verbose = false ↪ );` Cette fonction permet de représenter la structure d'un `Cluster` sous la forme d'une `string` par une approche récursive : s'il s'agit d'un noeud terminal, on produira la string représentant le taxon correspondant, sinon on produira celle

représentant son fils gauche puis son fils droit séparés par une virgule et entourés de parenthèses. Si `verbose` vaut `true`, le taxon sera affiché tel quel, suivi des valeurs des champs `id`, `height`, et `size`, séparés par des virgules et entourés par des crochets (`[]`) (voir la trace d'exécution plus bas pour la description exacte du format attendu). La valeur du champ `height` sera convertie en `string` au moyen de la fonction fournie `double_to_string`. Si `verbose` vaut `false`, ce sera une forme abrégée qui concatène le mot `Taxon` avec l'identificateur entier du taxon. Si `cluster` vaut `nullptr`, une chaîne vide sera retournée. La fonction `test_part1` appelée par le `main` fournit le résultat attendu pour un réseau similaire à celui de de la figure 3 :

```
======= Testing clusterToString =======
======= printing in non verbose mode by default ====
((Taxon_0,Taxon_2),Taxon_1)
======= printing in verbose mode ====
((ACGTAACCTTGGG[i:0,h:0,s:1],GTAGTAGTAGTAG[i:2,h:0,s:1])[i:-1,h:2,s
    ↪ :2],AGGGTCTATATGT[i:1,h:0,s:1])[i:-1,h:3,s:3]
======= End testing clusterToString =======
```

- `std::string toString(const Tree& tree, bool verbose = false);` qui produit la représentation sous forme de `string` de tous les « clusters » d'un arbre au moyen de leur fonction `toString` dédiée. La fonction `test_part1` appelée par le `main` devrait produire l'affichage suivant pour les arbres testés `tree1` et `tree2` :

```
======= Testing toString for Tree =======
printing tree1:
---------------
 ======= printing in verbose mode ====
ACGTAACCTTGGG[i:0,h:0,s:1];
AGGGTCTATATGT[i:1,h:0,s:1];
 ======= printing in non verbose mode ====
Taxon_0;
Taxon_1;
printing tree2:
---------------
 ======= printing in verbose mode ====
((ACGTAACCTTGGG[i:0,h:0,s:1],GTAGTAGTAGTAG[i:2,h:0,s:1])[i:-1,h:2,s
    ↪ :2],AGGGTCTATATGT[i:1,h:0,s:1])[i:-1,h:3,s:3];
 ======= printing in non verbose mode ====
((Taxon_0,Taxon_2),Taxon_1);
 ======= End testing toString for Tree =======
```

Tests

Pour tester cette partie, vous pouvez utiliser le programme principal fourni. Ouvrez le fichier `main.cpp`, vous y verrez que de nombreux tests sont fournis et **vous êtes libre de décommenter les instructions pertinentes pour tester votre code**. La fonction `test_part1` vous donne un exemple de tests simples que vous pouvez compléter à votre guise pour la première partie du projet. Par ailleurs, les dernières instructions exécutent la fonction `run_unit_tests`. Elle est définie dans `unit_test.cpp`. Ouvrez le fichier `unit_test.cpp` et examinez la fonction `run_unit_tests`. Vous verrez qu'elle appelle différents tests préprogrammés pour chaque partie du projet.

Pour cette partie du projet, vous pouvez appeler cette fonction `run_unit_tests(1)` qui exécutera le test préprogrammé pour la partie 1 du projet. L'exécution devrait afficher le message `[Passed]` pour tous les tests. En cas d'erreur, vous devriez obtenir un message d'erreur avec `[Failed]` vous indiquant la différence entre la valeur attendue et la valeur calculée.

Par exemple :

```
---------------
test_distanceMatrixToString
-----------------------
[Passed]
[Passed]
[Passed]
[Passed]
```

ce qui indique que la fonction `toString` qui prend en paramètre une matrice de distance a passé avec succès les tests.

Sinon, elle affichera par exemple :

```
--------------
test_distanceMatrixToString
-----------------------

[Failed]
 expected:
 0 17 21
 17 0 30
 21 30 0
 found:
 0 17 21 17 0 30 21 30 0
```

ce qui indique que `toString` a produit l'affichage donné sous `found`, au lieu de la valeur correcte affichée sous `expected`.

4.2 Partie 2

Note : Si `ptr_c` est un pointeur sur un cluster et `m` le nom d'un champ de ce cluster, alors les notations `(*ptr_c).m` et `ptr_c->m` sont équivalentes en C++.

L'objectif de la deuxième partie du projet est d'implémenter les fonctions nécessaires à l'initialisation de la structure de données représentant un arbre phylogénétique et au calcul des matrices de distances. Il vous est demandé pour cela d'implémenter les fonctions suivantes :

- `int calculateDistance(const Taxon& seq1, const Taxon& seq2);` calculant la distance entre le taxon `seq1` et le taxon `seq2`. Cette distance sera le nombre de fois où `seq1[i]` est différent de `seq2[i]` pour tout `i`. Les deux taxa peuvent être de tailles différentes. Les caractères en plus dans un des taxa seront comptés comme des différences.

- `Tree initTree(std::vector<Taxon> taxa);` qui retourne un arbre initial construit à partir des taxa passés en paramètre. Pour chaque taxon de l'ensemble `taxa`, un cluster dynamiquement alloué sera ajouté à l'arbre. Les « clusters » seront numérotés (identificateur entier du « cluster ») dans l'ordre de construction en commençant depuis le numéro zéro ;

- `void deleteTree(Tree& tree);` qui vide un arbre donné en libérant la mémoire associée à l'ensemble de ses clusters ; cette fonction doit s'exécuter sans crash y-compris dans les cas où certains « clusters » de `tree` valent `nullptr`. Attention, une libération incomplète de la mémoire peut entraîner des erreurs plus tard, dans la dernière partie du projet.

- `DistanceMatrix initDistanceMatrix(const Tree& tree);` qui construit et retourne la matrice de distances correspondant à l'arbre passé en paramètre. Pour chaque paire `(i,j)` de clusters de l'arbre, l'entrée `[i][j]` de la matrice sera calculée comme la distance séparant leur champs de type Taxon. La distance sera calculée au moyen de la fonction `calculateDistance`. Cette fonction n'a de sens que si l'arbre passé en paramètre ne contient que des feuilles et si aucune d'entre elle ne vaut `nullptr`, ce qu'il faut vérifier par le mécanisme des exceptions. Vous écrirez une fonction `isLeafNodeTree` qui teste la première condition. Elle retournera `true` si tous les « clusters » de l'arbre sont de taille une.

- `void eraseColumn(DistanceMatrix& distances, size_t c)` permettant de supprimer d'une matrice de distances la colonne d'indice `c` ; cet indice doit être compatible avec la taille de la matrice sinon cela devra être signalé par une exception.

- `void eraseRow(DistanceMatrix& distances, size_t r)` permettant de supprimer d'une matrice de distances la ligne d'indice `r` ; cet indice doit être compatible avec la taille de la matrice sinon cela devra être signalé par une exception.

Note : La suppression d'un élément à la position p dans un `vector`, `v`, peut se faire au moyen de l'instruction :

```
v.erase(v.begin() + p);
```

Attention, si l'instruction `v.erase(v.begin() + p);` est utilisée dans une itération, il faut que ce soit une itération classique (`for (size_t i(..);..;..)`) et non pas une itération sur ensemble de valeurs (`for (auto val :...)`). En effet, changer la taille d'une collection alors que l'on est en train d'itérer dessus avec une boucle `for (auto val :...)` cause un crash (les explications vous seront fournies au prochain semestre !).

Tests

Pour tester cette partie, vous pouvez utiliser la fonction `test_part2`. Les affichages attendus pour cette partie sont donné dans l'annexe A.2.1. Vous pouvez aussi appeler la fonction de test unitaire comme suit : `run_unit_tests(2)`
Si votre code est correct, vous ne devriez obtenir que les messages `[Passed]`.

4.3 Partie 3

Dans la troisième partie, vous devez implémenter les algorithmes de construction des arbres phylogénétiques. Un type énuméré `AlgoType` sera utilisé pour indiquer quelle variante l'on veut voir implémentée (`UGPMA` ou `WGPMA`). Voici les fonctions qu'il vous est demandé de coder (pour chacune d'elle, vous trouverez un exemple d'exécution dans la fonction `test_part3`) :

- `ClusterIdPair minimumDistance(const DistanceMatrix& distances);` qui retourne la paire de `Cluster` distincts séparés par la plus petite distance dans la matrice `distances`. Le parcours de la matrice devra se faire ligne par ligne et pour chaque ligne, colonne par colonne. Si la distance minimale est présente à plusieurs reprises, seule la paire avec les plus petits indices sera retournée (la première recncontrée). Par exemple, pour la matrice de distance :

  ```
  {
    {0, 1, 2},
    {1, 0 , 1},
    {2, 1 , 0}
  }
  ```

  La paire retournée serait `{0,1}`. Ceci signifie que les deux `Cluster` les plus proches (en termes de distance) sont ceux occupant la position 0 et la position 1 dans l'arbre phylogénétique correspondant à cette matrice de distance. On considérera que cette fonction ne peut fonctionner correctement que si le paramètre est une matrice non vide et carrée. Le mécanisme des exceptions devra le garantir.

Indication : la plus grande valeur possible pour un `double` est donnée par `std::numeric_limits<double>::max()`.

- `void mergeClusters(const ClusterIdPair& pair, Tree& tree, DistanceMatrix & distances, AlgoType algorithm = WPGMA);` qui permet de fusionner la paire de `Clusters` identifiés par `pair` dans l'arbre `tree` et d'adapter la matrice des distances, `distances`, en conséquence. Le noeud de l'arbre obtenu par fusion contiendra les données telles que décrites dans l'exemple de la figure 5.
  Vous adopterez notamment la même façon de coder les identifiants de type `Taxon` et entier que dans cet exemple. La fusion se fait de sorte à ce que dans l'arbre `tree`, le noeud à la position `pair[0]` soit remplacé par le nouveau noeud obtenu par fusion et que le noeud à la position `pair[1]` disparaisse de l'arbre. La matrice de distances sera adaptée en conséquence : pour chaque indice possible `p`, les entrées `[pair[0]][p]` et `[p][pair[0]]` seront réévaluées selon les modalités spécifiques à l'algorithme utilisé (moyenne des distances ou moyenne des distances pondérées par la taille des noeuds selon que l'on utilise `WPGMA` ou `UPGMA`, voir la différence entre les deux dans l'annexe). Par exemple, le premier appel à la fonction `test_merge_clusters` par `test_part3` devrait produire l'affichage suivant :

```
======= Testing mergeCluster (WPGMA) =======

The pair to merge:
0-1

Before merging:
-----------------
Tree:
------
ACGTAACCTTGGG[i:0,h:0,s:1];
ACGGTCTATTGGA[i:1,h:0,s:1];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 6 11
6 0 11
11 11 0

After merging clusters:
------------------------------
Tree:
------
(ACGTAACCTTGGG[i:0,h:0,s:1],ACGGTCTATTGGA[i:1,h:0,s:1])[i:-1,h:3,s
    ↪ :2];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 11
```

```
11 0

======= End testing mergeCluster (WPGMA) =======
```

Cette fonction ne peut fonctionner correctement que si les paramètres fournis sont correctement constitués : éléments distincts et compatibles avec la taille de la matrice dans `pair`, arbre ne contenant pas de `nullptr`, taille de l'arbre identique à celui de la matrice de distances et matrice de distance carrée. Ces conditions doivent être garanties au moyen du mécanisme des exceptions.

- `void buildPhylogeneticTree(Tree& tree, DistanceMatrix& distances, AlgoType algorithm = WPGMA);` qui applique l'algorithme consistant à fusionner les noeuds les plus proches en termes de distance, jusqu'à ce qu'il ne reste plus qu'un seul noeud dans l'arbre. L'arbre représentera alors l'arbre phylogénétique recherché. Pour que cette fonction s'exécute correctement, `tree` ne doit pas contenir de `nullptr`, la taille de l'arbre doit être la même que celle de la matrice de distance, et cette dernière doit être carrée. Si l'une de ces conditions n'est pas vérifiée, une exception doit être levée.

- `std::string phylogeneticTreeToString(const Cluster* root, bool verbose = false);` qui permet d'afficher de façon plus visuelle un arbre phylogénétique en partant de sa racine et en affichant la hauteur de chaque cluster (voir ci-dessous pour un exemple d'affichage souhaité). Si `root` vaut `nullptr`, une chaîne vide sera retournée. **Indication :** utilisez une fonction annexe récursive prenant un paramètre supplémentaire représentant la « profondeur » d'un appel récursif donné (cette profondeur donnera le nombre de symboles `'|'` dans une ligne donnée).

La fonction `test` appelée par `test_part3` , devrait produire l'affichage suivant pour l'exemple tiré de Wikipédia :

```
----- Wikipedia Example with UPGMA ----
(((Taxon_0,Taxon_1),Taxon_4),(Taxon_2,Taxon_3));
 (16.5)
|   +--- (11)
|   |   +--- (8.5)
|   |   |   +---a
|   |   |   +---b
|   |   +---e
|   +--- (14)
|   |   +---c
|   |   +---d
----- Wikipedia Example with WPGMA ----
(((a[i:0,h:0,s:1],b[i:1,h:0,s:1])[i:-1,h:8.5,s:2],e[i:4,h:0,s:1])[i:-1,h:11,s:3],(c[i:2
 (17.5)
|   +--- (11)
|   |   +--- (8.5)
|   |   |   +---a
|   |   |   +---b
```

```
|   |   +---e
|   +--- (14)
|   |   +---c
|   |   +---d
```

Tests

Pour tester cette partie, vous pouvez appeler la fonction `test_part3`. Les affichages attendus pour cette partie sont donnés dans l'annexe A.2.2. Vous pouvez aussi appeler la fonction de test unitaire comme suit : `run_unit_tests(3)`
Si votre code est correct, vous ne devriez obtenir que les messages `[Passed]`.

A Annexe

A.1 UPGMA vs WGPMA

Les deux algorithmes WPGMA et UPGMA ne diffèrent l'un de l'autre que dans la façon de calculer les distances entre « clusters ». Souvenez vous que dans la variante WPGMA, lorsque deux clusters C_1 et C_2 sont agrégés, la distance entre C_1C_2 et un tierce cluster C_3 se calcule comme une distance moyenne ; c'est-à-dire :

$$d(C_1C_2, C_3) = \frac{d(C_1, C_3) + d(C_2, C_3)}{2}.$$

Pour UPGMA, la distance dépend aussi de la taille des clusters : si un cluster agrège plus de composants, il aura plus d'influence sur les distances.

Plus précisément, la distance entre le cluster C_1C_2 et un tierce cluster C_3 est une moyenne pondérée par la taille ; c'est-à-dire :

$$d(C_1C_2, C_3) = \frac{|C_1| \cdot d(C_1, C_3) + |C_2| \cdot d(C_2, C_3)}{|C_1| + |C_2|},$$

où $|C_1|$ et $|C_2|$ désignent les tailles de C_1 et C_2, respectivement.

En guise d'illustration, nous allons dans ce qui suit, appliquer l'algorithme UPGMA à l'exemple de la Section 2. Nous commençons avec la matrice de distances de la Table 4. La distance minimale est entre T_1 et T_3, ce qui conduit à créer le cluster T_1T_3. Initialement, la taille de tous les clusters est 1 car chacun d'eux ne comporte qu'un seul taxon. Par conséquent, les distance entre T_1T_3 et les autres clusters est alors identiques pour les deux algorithmes (WPGMA and UPGMA). Ces distances sont données dans la Table 5. À l'étape suivante, les clusters T_1T_3 et T_2 sont fusionnés. Notez que T_1T_3 est de taille 2 et T_2 de taille 1. En utilisant l'information sur la taille la distance entre le cluster $T_1T_3T_2$ et T_4 est calculée comme suit :

$$d(T_1T_3T_2, T_4) = \frac{2 \cdot d(T_1T_3, T_4) + 1 \cdot d(T_2, T_4)}{2+1} = \frac{2 \cdot 7 + 1 \cdot 10}{2+1} = 8$$

Ceci produit la matrice de distance finale de la Table 6 et à l'arbre correspondant de la figure 6.

TAB. 4 : Distance entre T_1, T_2, T_3, et T_4

	T_1	T_2	T_3	T_4
T_1	0	5	4	7
T_2	5	0	7	10
T_3	4	7	0	7
T_4	7	10	7	0

TAB. 5 : Distances entre T_1T_3, T_2, et T_4

	T_1T_3	T_2	T_4
T_1T_3	0	6	7
T_2	6	0	10
T_4	7	10	0

TAB. 6 : Distances entre $T_1T_3T_2$ et T_4

	$T_1T_3T_2$	T_4
$T_1T_3T_2$	0	8
T_4	8	0

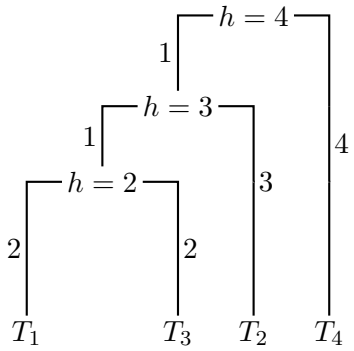

FIG. 6 : Arbre final avec UPGMA

A.2 Sorties des tests fournis

A.2.1 Part 2

```
 ========= TESTING PART 2 ================
 ======= Testing calculateDistance =======
Distance between CGTAACCTTGGG and CGTGAGCTTA: 5
 ======= end Testing calculateDistance =======
 ======= Testing initTree =======
The initial vector of taxa is: {"a", "b", "c", "d", "e"}
Tree constructed by initTree, printed in non verbose mode:
Taxon_0;
Taxon_1;
Taxon_2;
Taxon_3;
Taxon_4;
Same tree printed in verbose mode:
a[i:0,h:0,s:1];
b[i:1,h:0,s:1];
c[i:2,h:0,s:1];
d[i:3,h:0,s:1];
e[i:4,h:0,s:1];
 ======= end testing initTree =======
 ======= Testing initDistanceMatrix =======
Input data: a tree constructed using the taxa ACGTAACCTTGGG,
    ↪ ACGGTCTATTGGA and GTAGTAGTAGTAG
Distance matrix constructed by initDistanceMatrix:
0 6 11
6 0 11
11 11 0
 ======= end testing initDistanceMatrix =======
 ======= Testing eraseColumn and eraseRow =======
Initial distance matrix:
0 17 21 31 23
17 0 30 34 21
21 30 0 28 39
31 34 28 0 43
23 21 39 43 0
The previous matrix after erasing column 3:
0 17 21 23
17 0 30 21
21 30 0 39
31 34 28 43
23 21 39 0
The previous matrix after erasing row 3:
0 17 21 23
17 0 30 21
21 30 0 39
```

```
23 21 39 0
 ======= end testing eraseColumn and eraseRow =======
```

A.2.2 Part 3

```
 ========= TESTING PART 3 ===============
 ======= Testing minimumDistance =======
Input data: the distance matrix
Input data: the distance matrix {0, 1, 2}, {1, 0, 1}, {2, 1, 0}
The pair with the minimum distance is:
0-1
 ======= End testing minimumDistance =======
 ======= Testing mergeCluster (WPGMA) =======

The pair to merge:
0-1

Before merging:
-----------------
Tree:
------
ACGTAACCTTGGG[i:0,h:0,s:1];
ACGGTCTATTGGA[i:1,h:0,s:1];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 6 11
6 0 11
11 11 0

After merging clusters:
------------------------------
Tree:
------
(ACGTAACCTTGGG[i:0,h:0,s:1],ACGGTCTATTGGA[i:1,h:0,s:1])[i:-1,h:3,s:2];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 11
11 0

 ======= End testing mergeCluster (WPGMA) =======
 ======= Testing mergeCluster (UPGMA) =======
Input data: a tree with three leaf nodes "ACGTAACCTTGGG", "ACGGTCTATTGGA
    ↪ " and "GTAGTAGTAGTAG"
```

```
The pair to merge:
0-1

Before merging:
-----------------
Tree:
------
ACGTAACCTTGGG[i:0,h:0,s:1];
ACGGTCTATTGGA[i:1,h:0,s:1];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 6 11
6 0 11
11 11 0

After merging clusters:
------------------------------
Tree:
------
(ACGTAACCTTGGG[i:0,h:0,s:1],ACGGTCTATTGGA[i:1,h:0,s:1])[i:-1,h:3,s:2];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 11
11 0

 ======= End testing mergeCluster (UPGMA) =======
 ======= Testing build and print phylogenetic trees =======
Input data:
Tree:
------
Verbose printing:
CATAGACCTGACGCCAGCTC[i:0,h:0,s:1];
CATAGACCCGCCATGAGCTC[i:1,h:0,s:1];
CGTAGACTGGGCGCCAGCTC[i:2,h:0,s:1];
CCTAGACGTCGCGGCAGTCC[i:3,h:0,s:1];
Non verbose printing:
Taxon_0;
Taxon_1;
Taxon_2;
Taxon_3;

Distance matrix:
```

```
-----------------
0 5 4 7
5 0 7 10
4 7 0 7
7 10 7 0
----- Build Phylogenetic Tree with WPGMA ----

After calling buildPhylogeneticTree:
Tree:
------
(((Taxon_0,Taxon_2),Taxon_1),Taxon_3);

Distance matrix:
-----------------
0

Calling phylogeneticTreeToString:

Verbose printing:
 (4.25)
| +--- (3)
| | +--- (2)
| | | +---CATAGACCTGACGCCAGCTC
| | | +---CGTAGACTGGGCGCCAGCTC
| | +---CATAGACCCGCCATGAGCTC
| +---CCTAGACGTCGCGGCAGTCC
Non verbose printing:
 (4.25)
| +--- (3)
| | +--- (2)
| | | +---Taxon_0
| | | +---Taxon_2
| | +---Taxon_1
| +---Taxon_3

----- Build Phylogenetic Tree with UPGMA ----

After calling buildPhylogeneticTree:

Tree:
------
(((Taxon_0,Taxon_2),Taxon_1),Taxon_3);

Calling phylogeneticTreeToString:

Verbose printing:
 (4)
```

```
| +--- (3)
| | +--- (2)
| | | +---CATAGACCTGACGCCAGCTC
| | | +---CGTAGACTGGGCGCCAGCTC
| | +---CATAGACCCGCCATGAGCTC
| +---CCTAGACGTCGCGGCAGTCC
Non verbose printing:
 (4)
| +--- (3)
| | +--- (2)
| | | +---Taxon_0
| | | +---Taxon_2
| | +---Taxon_1
| +---Taxon_3
----- Wikipedia Example with UPGMA ----
(((Taxon_0,Taxon_1),Taxon_4),(Taxon_2,Taxon_3));
 (16.5)
| +--- (11)
| | +--- (8.5)
| | | +---a
| | | +---b
| | +---e
| +--- (14)
| | +---c
| | +---d
----- Wikipedia Example with WPGMA ----
(((a[i:0,h:0,s:1],b[i:1,h:0,s:1])[i:-1,h:8.5,s:2],e[i:4,h:0,s:1])[i:-1,h
    ↪ :11,s:3],(c[i:2,h:0,s:1],d[i:3,h:0,s:1])[i:-1,h:14,s:2])[i:-1,h
    ↪ :17.5,s:5];
 (17.5)
| +--- (11)
| | +--- (8.5)
| | | +---a
| | | +---b
| | +---e
| +--- (14)
| | +---c
| | +---d
======= end testing build and print phylogenetic trees =======
```