Séquences génétiques et arbres phylogénétiques

UE IPFL – ENSIIE – S4

Avril 2021

Travail à rendre

Le projet est à réaliser en OCaml individuellement. Il sera accompagné d'un rapport qui doit contenir impérativement la description des choix faits, des types et des fonctions. Le code et le rapport fourniront également des cas de tests accompagnés des résultats retournés. Pour cela, vous pouvez utiliser des assertions. Vous devez rendre votre rapport (en pdf) et le(s) fichier(s) de code rassemblés dans une archive identifiée comme `votre_prenom_votre_nom.tgz`. Une archive qui ne contient pas les fichiers demandés ne sera pas excusable. Votre archive doit être déposée sur `http://exam.ensiie.fr`, dans le dépôt `ipf_projet_2021`, avant 23h59 le 17/05/2021. Vous devrez IMPERATIVEMENT respecter les noms des types/fonctions donnés dans le sujet, mais vous êtes libres de définir des types/fonction auxiliaires qui vous semblent utiles.

Attention :

Un squelette vous est fourni : `http://web4.ensiie.fr/~stefania.dumbrava/ipf_template_projet.ml`.

Séquences ADN/ARN

On représente un brin d'ADN comme une liste de *nucléotides*. Les quatres nucléotides de l'ADN sont l'*adénine*, la *cytosine*, la *guanine* et la *thymine*. On va les noter comme `A`, `C`, `G` et, respectivement, `T`.

L'ADN est constituée de deux brins qui sont liés de manière complémentaires deux à deux, c'est à dire que : l'*adénine* ne peut se lier qu'à la *thymine* et la *cytosine* ne peut se lier qu'à la *guanine*, comme dans la Figure 1.

FIGURE 1 – Structure en double brin de l'ADN

Les types OCaml correspondant à une `nucléotide` et à un `brin` d'ADN sont :

```
type nucleotide = A | C | G | T
type brin = nucleotide list
```

1. Le contenu GC d'un brin d'ADN est la proportion de `C` ou de `G` parmi les nucléotides de la séquence. Écrire une fonction `contenu_gc : brin -> float` qui retourne le contenu GC d'un brin d'ADN passé en argument. Tester votre code.

```
# contenu_gc [A;T;G;T;T;G;A;C]
- : float = 0.375
# contenu_gc [C;T;T;A]
- : float = 0.25
# contenu_gc [A;A;A;T;A]
- : float = 0.
```

2. Écrire une fonction `brin_complementaire : brin -> brin` qui prend en argument un brin et calcule son complémentaire. Tester votre code.

```
# brin_complementaire [T]
- : brin = [A]
# brin_complementaire [C; T; T; C]
- : brin = [G; A; A; G]
# brin_complementaire [C; T; A; A; T; G; T]
- : brin = [G; A; T; T; A; C; A]
```

3. Écrire une fonction `distance : brin -> brin -> int` qui prend comme arguments deux brins de la même longueur (lever une exception sinon) et calcule leur *distance d'édition*, d_e, en comptant combien de nucléotides doivent être changés pour rendre les brins égaux. Tester votre code.

```
# distance [T] [T]
 - : acide list = 0
# distance [T] [C]
 - : acide list = 1
# distance [G; A; G] [A; G; G]
 - : acide list = 2
```

4. Écrire une fonction `similarite : brin -> brin -> float` qui utilise la fonction précédante pour calculer la *similarité procentuelle*, s_p, entre deux brins de la même longueur (lever une exception sinon) avec la formule :

$$s_p(h_1, h_2) = 1 - \frac{d_e(h_1, h_2)}{|h_1|}$$

Tester votre code.

```
# similarite [C;G;A;T] [T;A;G;T]
- : float = 0.25
# similarite [A;G;C;T] [T;A;A;G]
- : float = 0.
# similarite [A;G;C;T] [A;G;C;T]
- : float = 1.
```

5. Des triplets de nucléotides (nommés aussi *codons*) peuvent encoder des acide aminés (valeurs de type `acide`) spécifiques, comme indiqué par la fonction `codon_vers_acide`. Remarquer que, parmi ces acide aminés, on a aussi les acides aminés :

— `START` (encodé par le codon `(T,A,C)`) et le

— `STOP` (encodé par les codons `(A, T, T)`, `(A, C, T)` et `(A, T, C)`).

```
type acide = Ala | Arg | Asn | Asp | Cys | Glu | Gln | Gly | His | Ile |
             Leu | Lys | Phe | Pro | Ser | Thr | Trp | Tyr | Val | START | STOP

let codon_vers_acide n1 n2 n3 = match n1, n2, n3 with
| (A, A, A) -> Phe | (A, A, G) -> Phe | (A, A, T) -> Leu  | (A, A, C) -> Leu
| (G, A, A) -> Leu | (G, A, G) -> Leu | (G, A, T) -> Leu  | (G, A, C) -> Leu
| (T, A, A) -> Ile | (T, A, G) -> Ile | (T, A, T) -> Ile  | (T, A, C) -> START
| (C, A, A) -> Val | (C, A, G) -> Val | (C, A, T) -> Val  | (C, A, C) -> Val
| (A, G, A) -> Ser | (A, G, G) -> Ser | (A, G, T) -> Ser  | (A, G, C) -> Ser
| (G, G, A) -> Pro | (G, G, G) -> Pro | (G, G, T) -> Pro  | (G, G, C) -> Pro
| (T, G, A) -> Thr | (T, G, G) -> Thr | (T, G, T) -> Thr  | (T, G, C) -> Thr
| (C, G, A) -> Ala | (C, G, G) -> Ala | (C, G, T) -> Ala  | (C, G, C) -> Ala
| (A, T, A) -> Tyr | (A, T, G) -> Tyr | (A, T, T) -> STOP | (A, T, C) -> STOP
| (G, T, A) -> His | (G, T, G) -> His | (G, T, T) -> Gln  | (G, T, C) -> Gln
| (T, T, A) -> Asn | (T, T, G) -> Asn | (T, T, T) -> Lys  | (T, T, C) -> Lys
| (C, T, A) -> Asp | (C, T, G) -> Asp | (C, T, T) -> Glu  | (C, T, C) -> Glu
| (A, C, A) -> Cys | (A, C, G) -> Cys | (A, C, T) -> STOP | (A, C, C) -> Trp
| (G, C, A) -> Arg | (G, C, G) -> Arg | (G, C, T) -> Arg  | (G, C, C) -> Arg
| (T, C, A) -> Ser | (T, C, G) -> Ser | (T, C, T) -> Arg  | (T, C, C) -> Arg
| (C, C, A) -> Gly | (C, C, G) -> Gly | (C, C, T) -> Gly  | (C, C, C) -> Gly
```

Un brin encode des *chaînes* consécutivess d'acides aminés , chaque étant délimité par un acide aminé START (son début) et par un acide aminé STOP (sa fin).

On dit que le brin est *bien formé* s'il correspond bien a des chaînes consécutives, dont chaque commence par `START` et se finit par `STOP`.

Attention : ces chaînes sont potentiellement séparés par des nucléotides ; en particulier, la premiere chaîne peut être précédé par des nucléotides.

Écrire une fonction `brin_vers_chaine : brin -> acide list` qui prend en argument un brin et qui, si le brin est bien formé, décode les codons à l'intérieur (entre les délimiteurs) de sa première chaîne et les renvoie dans une liste d'acides aminés. Si le brin est mal formé, la fonction va lever une exception avec le message `"brin invalide"`.

Tester votre code.

```
# brin_vers_chaine [T; A; C; G; G; C; T; A; G; A; T; T;
                    T; A; C; G; C; T; A; A; T; A; T; C]
  - : acide list = [Pro; Ile].
# brin_vers_chaine [T; A; C; T; A; C]
Exception: Failure "brin invalide".
# brin_vers_chaine [T; A; C; G; G; A; T; C]
Exception: Failure "brin invalide".
```

6. Question optionnelle : Écrire une fonction `brin_vers_chaines : brin -> acide list list` qui prend en argument un brin et décode toutes ses chaînes. Tester votre code.

```
# brin_vers_chaines [T; A; C; G; G; C; T; A; G; A; T; T;
                     T; A; C; G; C; T; A; A; T; A; T; C]
  - : acide list list = [[Pro; Ile]; [Arg; Leu]].
```

Arbres phylogénétiques

Un *arbre phylogénétique* est un *arbre binaire complet* (c'est-à-dire dans lequel chaque nœud sauf les feuilles a deux enfants). Dans cette arbre, chaque nœud qui est une *feuille* contient un brin et chaque nœud qui *n'est pas une feuille* contient un couple formé par un brin, ainsi qu'une valeur (qu'on va appeler *malus*) qui correspond à la somme entre les distances d'édition entre le nœud et chaqu'un de ses deux enfants et celle de leur *malus* respectifs. Attention : tous les brins dans un arbre donné ont la même longueur ! Un exemple est donné dans la figure suivante :

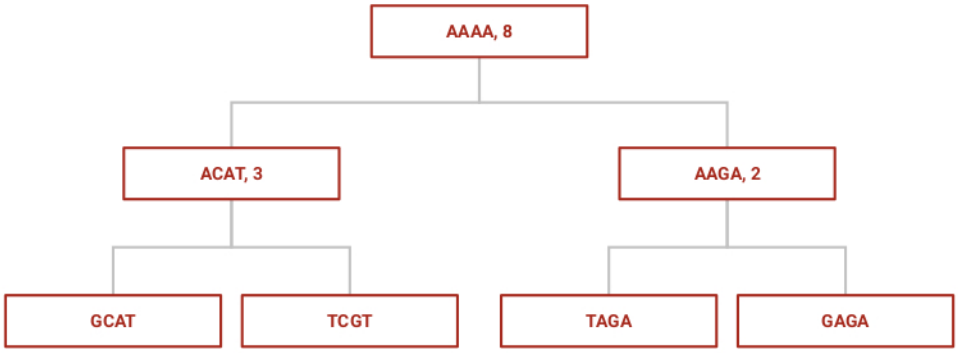

FIGURE 2 – Exemple d'arbre phylogénétique ($\mathcal{A}_1$)

Par exemple, dans la Figure 2, on a que le malus correspondant au nœud contenant `ACAT` est $d_e(\texttt{ACAT}, \texttt{GCAT}) + d_e(\texttt{ACAT}, \texttt{TCGT}) = 1 + 2 = 3$, que celui correspondant au nœud contenant `AAGA` est $d_e(\texttt{AAGA}, \texttt{TAGA}) + d_e(\texttt{AAGA}, \texttt{GAGA}) = 1 + 1 = 2$ et, finalement, que celui que celui correspondant au nœud contenant `AAAA` est $d_e(\texttt{AAAA}, \texttt{ACAT}) + d_e(\texttt{AAAA}, \texttt{AAGA}) + 3 + 2 = 8$.

On peut réprésenter ces *arbres phylogénétique* en OCaml avec le type `arbre_phylo` :

```
type arbre_phylo = Lf of brin | Br of arbre_phylo * brin * int * arbre_phylo
```

1. Écrire une fonction `arbre_phylo_vers_string : arbre_phylo -> string` qui prend en argument un arbre et retourne une representation unique de cet arbre sous forme de chaîne de caractères. Tester votre code.

2. Écrire une fonction `similaire: arbre_phylo -> arbre_phylo list -> arbre_phylo` qui calcule, pour un arbre phylogénétique donné et une liste d'arbres qui on tous la même forme (donc même hauteur, car ils sont tous d'arbres binaires complètes), l'arbre le plus similaire avec lui.

   Pour calculer la similarité S entre deux arbres on va comparer leurs éléments deux par deux, en suivant le même parcours, et calculer la somme de leur similarités procentuelles. Par exemple, en utilisant un parcours préfixe (racine - sous-arbre gauche - sous-arbre droit), on peut calculer la similarité $S(\mathcal{A}_1, \mathcal{A}_2)$ entre les arbres $\mathcal{A}_1$ (Figure 2) et $\mathcal{A}_2$ (Figure 3) avec la formule suivante :

   $$s_p(\texttt{AAAA}, \texttt{AATA}) + s_p(\texttt{ACAT}, \texttt{GCTT}) + s_p(\texttt{GCAT}, \texttt{GAAT}) + s_p(\texttt{TCGT}, \texttt{CAGT}) + s_p(\texttt{AAGA}, \texttt{AAGA}) + s_p(\texttt{TAGA}, \texttt{TAGA}) + s_p(\texttt{GAGA}, \texttt{GTGA})$$

   Tester votre code.

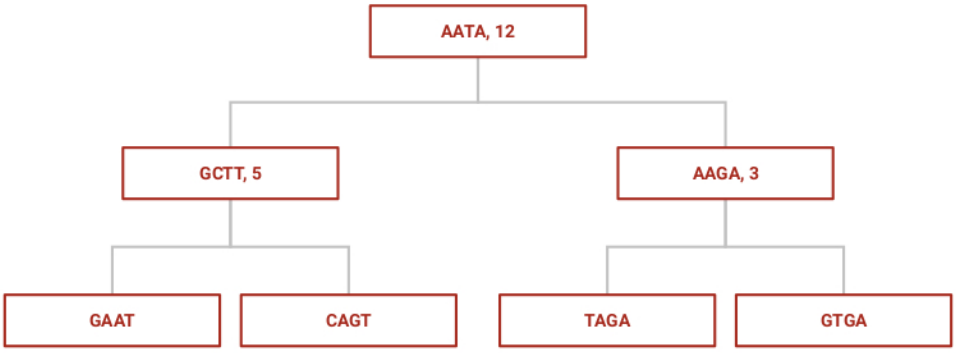

FIGURE 3 – Autre exemple d'arbre phylogénétique ($\mathcal{A}_2$)

3. Écrire les fonctions auxiliaires suivantes :
   — `get_root : arbre_phylo -> brin` qui extrait le *brin* de la racine d'un `arbre_phylo`.
   — `get_malus : arbre_phylo -> int` qui extrait le *malus* de la racine d'un `arbre_phylo`.
   — `br : arbre_phylo * brin * arbre_phylo -> arbre_phylo` qui permet de construire un `arbre_phylo` à partir d'un sous-arbre gauche, d'un brin et d'un sous-arbre droit.

   Tester votre code.
4. Écrire une fonction `gen_phylo : brin -> brin -> brin -> arbre_phylo list` qui prend en paramètres trois brins de la même taille et renvoie la liste de tous les arbres phylogénétiques que l'on peut générer avec ces brins. On va exclure les cas triviaux des arbres formés juste d'une feuille (aucun fils). Tester votre code.
5. Écrire une fonction `min_malus : arbre_phylo list -> arbre_phylo` qui prend en paramètre une liste d'arbres phylogénétiques et renvoie celui qui a la valeur *minimale* de *malus global* (le malus de la racine). Par exemple, si on passe à cette fonction la liste contenant les arbres $\mathcal{A}_1$ et $\mathcal{A}_2$, elle va retourner l'arbre $\mathcal{A}_1$, car la valeur 8 de son malus global est plus petite que la valeur 12, correspondant au malus global de l'arbre $\mathcal{A}_2$. Tester votre code.
6. Écrire une fonction `gen_min_malus_phylo : brin list -> arbre_phylo` qui prend en paramètre une liste de brins de la même taille et renvoie l'arbre phylogénétique qui a comme feuilles ces brins et qui a la plus petite valeur du malus global parmi tous les arbres qu'on puisse générer. Expliquer l'algorithme conçu. Tester votre code.

   Astuce 1 : Utiliser les fonctions auxiliaires définis précédemment.

   Astuce 2 : Utiliser une approche *diviser pour régner*[1] (comme vu aussi en cours) en décomposant le problème en des sous-problèmes plus petits (par exemple, avec l'aide d'une fonction qui permet de découper de toutes les façons possibles une liste de brins en deux sous-listes de tailles qui conviennent).

   Vérifier que votre algorithme est correct, en comparant la `similarite` entre l'arbre réponse obtenu à partir d'une liste de trois brins et celui obtenu en appelant la fonction `min_malus` sur la liste d'arbres générés avec `gen_phylo`.

1. https://fr.wikipedia.org/wiki/Diviser_pour_r%C3%A9gner_(informatique)