TP 21: les pingouins sont-ils des manchots ?

MP2I Descartes, Tours - Vladislav Tempez

5 juillet 2023

1 Description du problème

L'objectif de ce TP est d'utiliser l'algorithme de classification hiérarchique ascendante pour reconstruire des arbres phylogénétiques à partir du génome d'espèces actuelles. Les espèces actuelles ont en commun des espèces ancestrales à partir desquelles elles ont évolué, et un arbre phylogénétique représente ces relations de parenté entre espèces. Dans ce TP, un modèle simplifié à l'extrême est utilisé. Il part du génome de l'hypothétique unique ancêtre commun de tous les êtres vivants et considère que les espèces et leur génome subissent avec le temps deux types d'évènements :

— La *dérive génétique* : le génome d'une espèce subit au cours du temps des mutations qui le rendent différent du génome original

— La *spéciation* : deux espèces distinctes apparaissent à partir d'un unique génome. On peut interpréter ceci comme un changement assez important dans le génome pour rendre celui-ci significativement différent du génome précédent, et donc obtenir une espèce distincte de ce qui existait jusqu'alors, ce qui justifie l'apparition d'une nouvelle espèce.

Un arbre phylogénétique est donc un arbre dans lequel les feuilles sont les espèces actuelles, les nœuds sont des espèces ancestrales plus ou moins connues. Entre un nœud et ses enfants, on peut indiquer une durée correspondant au temps écoulé depuis la spéciation précédente.
Dans ce TP, l'objectif est de reconstruire l'arbre à partir des génomes des espèces. La première étape consiste à calculer la proximité entre deux génomes afin de savoir si les deux espèces correspondantes sont proches ou non dans l'arbre phylogénétique. Pour ceci, dans les hypothèses du modèle simplifié utilisé, la distance entre deux génomes peut s'estimer via le nombre de mutations pour passer d'un génome à l'autre, c'est-à-dire la distance d'édition.
La seconde étape est de reconstruire un arbre via les distances entre génomes via l'algorithme de classification hiérarchique ascendante. L'hypothèse principale étant la suivante : si deux génomes sont issus du génome de leur ancêtre commun, la distance de ces deux génomes entre eux est probablement bien plus faible que leur distance à d'autres génomes d'espèces plus éloignées dans l'arbre, ceux deux espèces sont donc regroupées dans un même sous arbre, et on recommence tant qu'il reste plusieurs arbres.
Dans ce TP on s'intéresse plus précisément à la question des relations de parenté entre pingouins et manchots. Les deux espèces vivent dans des zones géographiques différentes, mais on confond régulièrement les deux pour des raisons linguistiques et à cause de quelques ressemblances morphologiques. Ce TP entend vous faire trancher (de manière très artificielle) la question de leurs liens de parenté réels.
Pour réaliser cela, vous aurez accès à des génomes de plusieurs espèces (ce ne sont bien évidement pas les vrais génomes de ces espèces) répartis dans des fichiers dont l'extension est `.gen`. Vous trouverez les génomes des espèces suivantes : petit pingouin, grand pingouins (espèce éteinte), manchot empereur, manchot royal, manchot pygmée, manchot du cap, manchot antipode, gorfou.
Pour servir de points de comparaison, les génomes des espèces suivantes sont inclus dans les données : pélican gris, pélican frisé, grue couronnée et grue royale. Le fichier `aves_header.info` indique la correspondance entre les noms de fichier contenant les génomes et les espèces.
Ces génomes ont été construits selon le modèle détaillé plus haut : on part d'un unique génome (généré aléatoirement), à chaque spéciation on duplique ce génome, puis on lui applique un nombre de mutation aléatoire en fonction du temps écoulé jusqu'à la spéciation suivante. Les spéciations (et donc la forme de l'arbre) sont décidées à partir du réel arbre phylogénétique des espèces concernées. Les durées entre spéciations sont complètement inventées, tout comme le génome initial. La longueur des génomes est d'à peu près 100 paires de bases, ce qui est très inférieur à la taille des véritables génomes.
En plus des données, des fonctions permettant la lecture des fichiers de données et de calcul de la distance entre génomes (cf la correction du TP 17) sont fournies. Vous n'avez pas directement accès au code de ces fonctions, seulement la signature des fonctions (fichiers `.mli`) et les fichiers objets correspondants (`.cmo`). Ces fichiers objets permettent d'utiliser les fonctions détaillées dans la signature dans d'autres programmes.

— La fonction `read_genome_list_from_header_file : (header_file_name: string) -> (string * char list) list` permet de lire les génomes et d'en renvoyer une représentation en OCaml.

— `edit_distance : 'a list -> 'a list -> int` qui permet de calculer la distance entre deux génomes représentés sous forme de listes.

Pour utiliser le code compagnon contenu dans les fichiers `read_data.ml` et `genome_distance.ml`, il suffit de compiler ceux ci avec les commandes `ocamlc read_data.mli read_data.ml` et `ocamlc genome_distance.mli genome_distance.ml`.

Ces commandes produiront des fichier `.cmo`. Pour utiliser les fonctions présentes dans les fichiers `.cmo` dans un interpéteur (comme celui de VSCode ou d'emacs), il faut entrer la commande `#load "<nom_du_fichier>.cmo"`. Autrement, il est possible de compiler le code en mettant le `.cmo` avant les `.ml` qui en ont besoin dans la commande de compilation avec `ocamlc`. Dans votre cas, il est possible d'ajouter simplement au début de votre code les lignes suivantes :

```
#load "read_data.cmo";;
#load "genome_distance.cmo";;
```

Attention, si vous procédez ainsi, votre code ne sera pas compilable. Pour le rendre compilable il faudra supprimer les deux lignes en question.
Les contenus des fichiers `.cmo` sont accessibles en tant que fonctions de *modules* via la syntaxe `Nom_du_module.nom_de_la_fonctio` sur le même modèle que les fonctions `List.hd, List.Length, ...` du module `List`. Le nom du module sera le nom du fichier dont la première lettre sera en majuscule. Ici vous pourrez donc utiliser les fonctions `Read_data.read_genome_list_from_hea` et `Genome_distance.edit_distance`.

1. Utilisez la fonction de lecture des données pour obtenir une liste des génomes des différentes espèces.
2. Utilisez la fonction de distance pour obtenir la distance entre les deux premiers génomes de la liste.

Remarque: Afin de ne pas bloquer la progression, deux fichiers `cha.mli` et `cha.ml` contenant les fonctions à implémenter sont fournis. Ils vous permettront d'utiliser une fonction sur laquelle vous resteriez bloqués ou bien de comparer leur sortie à la sortie de vos fonctions. Pour les utiliser il faut d'abord le compiler, comme pour les fichiers `genome_distance.ml` et `read_data.ml`. Pour que ce TP soit utile, il est important de ne PAS regarder la correction avant d'avoir essayé par soi-même d'implémenter les fonctions demandées. Et dans ce cas, il est impératif de ne pas copier/coller le code du corrigé, mais simplement de s'en inspirer.
Le fichier `gen_data.ml` contient le code qui permet de générer les génomes artificiels utilisés dans ce TP. Il n'est pas nécessaire de regarder son contenu pour faire ce TP.

2 Algorithme de classification hiérarchique ascendante

On rappelle que le principe de l'algorithme de classification hiérarchique ascendante est le suivant :

— On dispose d'un ensemble de n objets entre lesquels on peut calculer une distance d. Ici les objets sont des génomes et la distance correspondante est la distance d'édition.
— En début d'algorithme, chaque génome est seul dans son groupe.
— On dispose d'une pseudo-distance entre groupes d_g basée sur d.
— Tant qu'il reste plusieurs groupes, on cherche les deux groupes g_i et g_j pour lesquels $d_g(g_i, g_j)$ est minimale.
— On fusionne g_i et g_j.

Dans ce TP, on prendra pour d_g le lien moyen : $d_g(S_1, S_2) = \frac{1}{|S_1||S_2|} \sum_{i \in S_1, j \in S_2} d(i,j)$ Pour implémenter l'algorithme de classification hiérarchique ascendante, il faut procéder à un choix de représentation des différents objets. Les génomes sont initialement représentés par une liste de caractères pour être utilisables par la fonction de calcul de la distance d'édition. Mais pour ne pas manipuler directement les génomes durant l'algorithme, on peut commencer par initialiser une matrice des distances entre ces génomes : la case i, j de cette matrice contiendra directement la distance d'édition entre le i-ième et le j-ième génome. Ceci évitera de la recalculer à chaque fois qu'elle est nécessaire. Il faudra cependant garder un lien entre i et le i-ième génome. On le fera par le biais d'un tableau contenant dans sa case i le couple (nom de la i-ième espèce, génome de la i-ième espèce).

3. Implémentez une fonction `fill_id_to_name_array: (genome_list : ('a * 'b) list) -> ('a * 'b) array` qui remplit ce tableau à partir de la liste fournie par la fonction de lecture des données.
4. Implémentez une fonction `make_distance_array : (genome_array: ('a * 'b list) array) -> int array array` qui remplit la matrice des distances à partir du tableau résultat de la question précédente. On pourra se souvenir de l'usage de `Array.make_matrix`.

On choisit de représenter les groupes manipulés par l'algorithme de classification hiérarchique ascendante par le type d'arbre suivant :

```
type cha_tree = Leaf of string * (char list) | Node of cha_tree * cha_tree
```

Les feuilles contiennent le nom de l'espèce et le génome associé, et chaque nœud possède exactement deux fils. Cette représentation permet de conserver les relations de parenté internes à chaque groupe. Pour plus de simplicité dans la manipulation future des arbres, on associe à chaque arbre un index, et on stocke l'ensemble des arbres non encore fusionnés à l'étape courant dans un dictionnaire implémenté par une table de hash.

5. Implémentez une fonction `fill_tree_table : (genome_array: (string * char list) array) -> (int, cha_tree) Hashtbl.t` qui initialise ce dictionnaire avec les arbres qui initiaux pour l'algorithme : une feuille pour chaque espèce. On prendra pour index de chacun de ces arbres l'index déjà associé à l'espèce, c'est-à-dire l'index de la case du tableau dans laquelle est l'espèce.

Pour calculer plus aisément la distance entre deux groupes, on va tenir à jour une table associant à chaque arbre la liste des points contenus dans l'arbre. On stocke ces listes de points dans un dictionnaire, et les clés sont les index associés aux arbres.

6. Implémentez une fonction `fill_content_table : (genome_array : 'a array) -> (int, int list) Hashtbl.t` qui initialise une table associant à chaque arbre initial (qui sont tous des feuilles) son contenu, c'est-à-dire le numéro du génome présent dans la feuille.
7. Implémentez une fonction `compute_set_distance_average_link : (i:'a) -> (j:'a) -> (distance_matrix: int array array) -> (content_table: ('a, int list) Hashtbl.t) -> float` qui calcule la distance entre les arbres d'indice `i` et `j` à l'aide de la matrice des distances entre génomes et du dictionnaire de contenu des arbres initialisé à la question précédente. Pour parcourir les listes on pourra écrire une ou plusieurs fonctions récursives ou bien utiliser la fonction `List.fold_left` (cf. TP 19).
8. Implémentez une fonction `find_best_merge : (distance_matrix: int array array) -> (current_trees_table: ('a, 'b) Hashtbl.t) -> (content_table: ('a, int list) Hashtbl.t) -> (i,j,d :'a * 'a * float)` qui détermine les index des deux arbres `i,j` les plus proches selon le lien moyen et la longueur de ce lien `d`. Il est pour cela nécessaire de parcourir les arbres courant stockés dans la table `current_trees_table`, ce qu'on pourra faire via la fonction `Hashtbl.iter` (cf TP 17). Vous pourrez commencer par implémenter une fonction `get_closest_tree_index : (distance_matrix: int array array) -> (content_table: ('a, int list) Hashtbl.t -> (current_tree_table: ('a, 'b) Hashtbl.t) -> (i: 'a) -> ((j,d): ('a * float)` qui calcule l'index `j` de l'arbre le plus proche de l'arbre d'index `i` et la distance entre ces deux arbres `d`. Là encore, il sera nécessaire d'utiliser `Hashtbl.iter`.
9. Implémentez une fonction `merge_trees : (i:'a) -> (j:'a) -> (content_table: ('a, 'b list) Hashtbl.t) -> (current_trees_table : ('a, cha_tree) Hashtbl.t) -> (k : 'a) -> unit` qui fusionne les arbres d'index `i` et `j` en un nouvel arbre à qui on associe un indice `k` et qui met à jour le dictionnaire des contenus ainsi que le dictionnaire des arbres courants. Attention, les arbres d'index `i` et `j` sont fusionnés et ne doivent donc plus apparaître dans les arbres courants.
10. Implémentez une fonction récursive `build_cha_tree_recursively : (distance_matrix : int array array) -> (content_table : (int, int list) Hashtbl.t) -> (current_trees_table : (int, cha_tree) Hashtbl.t) -> (n: int) -> (k : int) -> cha_tree` qui construit l'arbre de classification hiérarchique ascendante à partir d'un ensemble courant d'arbres, du contenu de ces arbres, de la matrice des distances, du nombre `n` d'arbre actuel et du premier indice libre pour un nouvel arbre `k` puis une fonction `build_cha_tree : (list_genome : (string * char list) list) -> cha_tree` qui initialise les structures nécessaires et appelle récursivement la fonction précédente. Alternativement, vous pourrez préférer n'implémenter qu'une fonction et utiliser une boucle `for`.
11. Implémentez une fonction qui affiche un parcours infixe de l'arbre pour vérifier le résultat obtenu.
12. Dessinez l'arbre obtenu, qu'en déduisez-vous ?
13. (Bonus) Implémentez une fonction pour découper cet arbre en `k` groupes différents.
14. (Bonus) modifiez votre algorithme et la structure d'arbre pour inscrire dans k l'arbre la distance de chaque nœud à son ancêtre.
15. (Bonus) Quelle est la complexité en espace de la fonction calculant la distance entre deux génomes ? Est-ce utilisable pour de véritables génomes dont la longueur se compte en dizaine de milliers de paires de base ? À l'aide de la technique précisée dans le DS 6, implémentez une fonction distance d'édition dont la complexité en espace permet de travailler avec des génomes de cette taille.
16. (Bonus) Améliorez la complexité temporelle de l'algorithme de classification hiérarchique ascendante via l'approche proposée dans le DS 7.