ADN et arbres PATRICIA

1 Introduction

L'ADN est une macromolécule biologique présente dans toutes les cellules. C'est une double hélice consituée de deux polymères anti-parallèles. Chaque polymère est un assemblage de bases nucléiques. Ces bases nucléiques sont l'adénine (notée A), la cytosine (C), la guanine (G) et la thymine (T).

On définit les quatres bases existantes en Caml par :

```
type base = A | C | G | T;;
```

Un fragment d'ADN pourra donc être représenté par une liste en Caml (`base list`).

On souhaite créer une structure de données permettant de stocker en Caml un ensemble de fragments d'ADN, permettant tout à la fois d'économiser de la mémoire et de permettre une vérification rapide de la présence d'un fragment donné dans l'ensemble.

2 Tries en OCaml

Pour ce faire, on va utiliser une structure similaire au « tries » et aux arbres PATRICIA (de l'anglais « Practical Algorithm To Retrieve Information Coded In Alphanumeric »). Cela consiste à mémoriser l'ensemble des fragments sous la forme d'un arbre où les arêtes correspondent aux différentes bases A, C, G et T. On colorie les nœuds en noir si le chemin allant de la racine au nœud en question correspond à un fragment de l'ensemble, et blanc sinon. L'arbre est infini, mais on ne mémorise que la plus petite partie contenant des nœuds noirs, en élagant les branches ne contenant que des nœuds blancs.

On peut ainsi représenter l'ensemble AA, AC, ATA, ATG, C, G, GA, GCC, GCT, GT par l'arbre suivant :

Ces arbres sont représentés en Caml par le type

```
type trie = Nil | Node of bool * trie * trie * trie * trie;;
```

Dans cette définition, le booléen indique si le nœud est noir (`true` = le nœud est noir), et les quatre (sous-)arbres représentent respectivement, dans cet ordre, les fils pour les bases A, C, G et T.

1. Écrire une fonction `count` de type `trie -> int` qui retourne le nombre de fragments stockés dans l'arbre fourni en paramètre.

2. Proposer une fonction `subtree` de type `base -> trie -> trie` qui prend en argument une base et un arbre, et retourne le sous-arbre correspondant à la base indiquée s'il existe, et `Nil` dans le cas contraire.

3. Écrire une fonction `mem` de type `base list -> trie -> bool` qui prend en argument un fragment et un arbre représentant un ensemble de fragments et retournant un booléen indiquant si le premier est inclus dans le second.

4. Proposer une fonction `subst` de type `base -> trie -> trie -> trie` qui prend en argument une base, un arbre A et un arbre S, et retourne l'arbre dans lequel le sous-arbre de A correspondant à la base fournie a été remplacé par l'arbre S. Si A est un arbre vide, on créera un arbre avec une racine blanche, trois fils `Nil` et l'arbre S comme quatrième fils (celui correspondant à la base indiquée).

5. Proposer une fonction `add` de signature `base list -> trie -> trie` qui prend en argument un fragment sous la forme d'une liste de bases, et un ensemble de fragments représenté par un arbre, et retourne un arbre représentant l'ensemble (puisque l'on ne manipule que des constantes, l'arbre est nécessairement un nouvel arbre).

6. Utiliser la fonction précédente pour créer une fonction `trie_of_list` de signature `base list list -> trie` prenant une liste de fragments et retournant l'arbre correspondant.

7. Écrire une fonction `list_of_trie` de signature `trie -> base list list` qui effectue l'opération inverse.

8. En déduire une fonction `display` de signature `arbre -> unit` qui affiche le contenu d'un arbre (un fragment par ligne). On pourra utiliser les fonctions suivantes :

```
let print_base b =
  print_char ((function A->'A'|C->'C'|G->'G'|T->'T') b);;

let print_fragm = List.iter print_base;;
```

9. Montrer que les fonctions `trie_of_list` et `list_of_trie`, au prix d'une éventuelle modification mineure, permettent d'obtenir un tri des fragments par ordre lexicographique, et proposer une fonction `sort` de signature `base list list -> base list list` qui effectue le tri d'une liste de fragments. Combien d'opérations élémentaires effectue-t-on pour n fragments de longueur inférieure ou égale à p ?

10. Proposer une fonction `remove` de signature `base list -> trie -> trie` qui retire un fragment, passé en argument, d'un ensemble de fragments représenté par un arbre. On se contentera de changer le booléen représentant la « couleur » du nœud concerné, et on déclenchera une erreur si le fragment n'était pas présent.

Dans certaines situations (comme lorsque l'on effectue des suppressions), l'arbre obtenu peut ne pas être le plus petit possible : un sous-arbre peut ne contenir que des nœuds blancs, et peut donc être supprimé sans que cela ne change l'ensemble représenté par l'arbre.

11. Écrire une fonction `is_empty` de signature `trie -> bool` qui indique si un ensemble représenté par un arbre est vide.

12. Écrire une fonction `is_compact` de signature `trie -> bool` qui indique s'il n'existe pas d'arbre plus petit que celui passé en argument permettant de décrire le même ensemble de fragments. On s'efforcera d'obtenir une complexité linéaire.

13. Proposer une fonction `cut` de signature `trie -> trie` qui retire d'un arbre toutes les branches inutiles (ne contenant que des nœuds blancs). Là encore, on cherchera une complexité linéaire en la taille de l'arbre.

3 Tries compressés

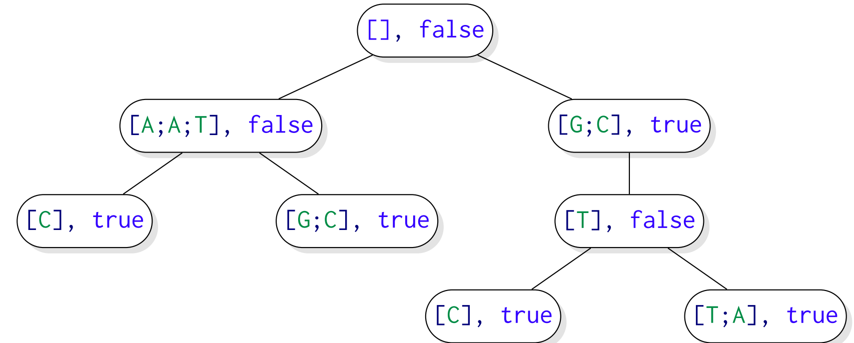
En fait, il y a un peu de gâchis de place dans l'arbre, lorsque l'on a des successions de nœuds qui n'ont qu'un seul fils. Plutôt que de ne conserver qu'une base par « étage » de l'arbre, on peut compresser ces branches en indiquant à chaque nœud la *succession* de bases qui y conduisent.

Un arbre compressé aura pour signature

```
type compr_trie = CNode of base list * bool * compr_trie list;;
```

Un arbre représentant l'ensemble vide sera représenté par le nœud `CNode ([], false, [])`. Dans un arbre représentant un ensemble non-vide, on impose qu'un nœud « `false` » ait toujours *au moins deux enfants*, et que pour tout nœud, la liste des sous-arbres ne contient que des nœuds étiquetés avec des listes de bases non-vides et commençant par des bases différentes. On ne se préoccupera pas ici de garder un ordre lexicographique pour les différents enfants.

Ainsi, un arbre contenant AATC, AATGC, GC, GCTC et GCTTA aura cette structure :



L'ensemble réduit au seul fragment GATCAG serait, de même, décrit simplement par `CNode ([G;A;T;C;A;G], true, [])`.

14. Proposer une fonction `is_prefix` de signature `base list -> base list -> bool` prenant deux listes et retournant un booléen indiquant si la première est un préfixe de la seconde, c'est-à-dire, en notant $a_0, a_1, \ldots a_{n-1}$ les éléments de la première liste et $b_0, b_1, \ldots a_{p-1}$ ceux de la seconde on a $n < p$ et $\forall k < n, a_k = b_k$.

15. Proposer une fonction `compr_mem` de signature `base list -> compr_trie -> bool` qui teste si un fragment est présent dans un arbre compressé.

16. Écrire une fonction `list_of_compr_trie` de signature `compr_trie -> base list` qui prend en argument un arbre compressé et retourne une liste de fragments.

17. Proposer une fonction `compr_add` dont la signature Caml sera `compr_trie -> base list -> compr_trie` qui ajoute un fragment à un ensemble de fragments décrit par un arbre compressé.

18. Écrire une fonction `compr_remove` retirant un fragment de l'ensemble (l'arbre devant garder les propriétés d'un arbre compressé, c'est-à-dire qu'un nœud « `false` » doit toujours avoir au moins deux enfants ; on déclenchera une erreur si l'on tente de retirer le dernier fragment de l'arbre).