TP18: Compression et recherche de motif

1 Recherche de motif

L'archive du TP contient les fichiers texte suivants:

- `gene.txt` qui contient la séquence ADN du gène CTFR [1]. Une mutation de ce gène peut entrainer la mucoviscidose;
- `ADN.txt` qui contient un morceau de l'ADN d'une personne, dont on souhaite déterminer si son gène CTFR a subi une mutation;
- `vingt_mille_lieues.txt`, qui contient le texte de Vingt Mille Lieues sous les mers de Jules Verne.

Nous allons utiliser ces fichiers pour comparer les performances des algorithmes de Rabin-Karp et de Boyer-Moore-Horspool. Les deux objectifs sont:

- Vérifier si le gène CTFR apparaît où non dans le fragment de séquence ADN donné.
- Chercher le nombre d'occurrences de quelques mots dans *Vingt Mille Lieues sous les mers*: "`Nemo`", "`Nautilus`", "`tératologique`", "`quoicoubaka`", "`anticonstitutionnellement`".

Cependant, il faudra aussi écrire vos propres tests, plus courts, pour vous aider à débugger vos programmes au fur et à mesure.

Q1. Lire le fichier `texte.h` de l'archive. Les différentes fonctions qui y sont décrites devront être implémentées dans trois fichiers:

- `texte_naif.c`
- `texte_rabin_karp.c`
- `texte_boyer_moore.c`

Ceci permettra de pouvoir choisir, à la compilation, l'une des trois implémentations.

[1] Les données ont été modifiées pour mieux s'adapter au TP

Recherche naïve

Q2. Implémenter la fonction `int recherche(char* texte, char* motif, int n, int p)` selon une méthode naïve.

Q3. Implémenter la fonction `int recherche_compte(char* texte, char* motif, int n, int p)`, toujours selon une méthode naïve.

Q4. Écrire une fonction `char* lire(char* filename, int len)` lisant un fichier `filename` et renvoyant son contenu sous la forme d'une chaîne de caractères. L'entier `len` indiquera le nombre d'octets que le fichier contient au maximum, ce que vous n'aurez pas besoin de vérifier dans votre fonction. Vous pouvez déterminer le nombre d'octets que contient un fichier en inspectant ses propriétés, ou directement dans le terminal avec la commande `stat`.

Q5. Implémenter dans le main de quoi tester les performances:

- Mesurer le temps pour trouver (ou non) la première occurrence du gène CTFR dans la séquence ADN
- Mesurer le temps pour compter le nombre d'occurrences de chacun des motifs demandés dans *Vingt Mille Lieues sous les mers*

On pourra utiliser la fonction `int clock()` de la librairie `<time.h>`, dont on rappelle l'utilisation:

```
clock_t debut = clock();
fonction_longue_a_executer();
clock_t fin = clock();

float nombre_de_secondes = (float) (fin - debut) / CLOCKS_PER_SEC;
```

Après ce code, `nombre_de_secondes` contient le temps d'exécution de l'appel à la fonction `fonction_longue_a_executer`, en secondes.

Q6. Mesurer les temps d'exécutions de cet algorithme sur les tests proposés.

Rabin-Karp

Pour l'algorithme de Rabin-Karp, on prend q un nombre premier assez grand, et B le nombre de lettres possibles (on prendra $B = 256$ pour les caractères ASCII), et l'on considère un mot $m_0, \dots m_{p-1}$ comme un nombre en base B pris modulo q. Autrement dit, la fonction de hachage est:

$$h : u_0 \dots u_{p-1} \mapsto \sum_{i=0}^{p-1} u_{p-1-i} B^i \mod q$$

On prendra $q = 5003371$, ce qui permettra de ne pas avoir de dépassement d'entier, et des collisions très rares.

Q7. On considère un texte $t = t_0 \dots t_{n-1}$. Rappeler la formule liant $h(t_{i+1} \dots t_{i+p})$ et $h(t_i \dots t_{i+p-1})$.

Q8. Écrire une fonction `int hash_initial(char* s, int p)` calculant le hash de la chaîne formée des p premières lettres de s. Cette fonction servira à initialiser les hash du motif et de la fenêtre du texte.

Q9. Écrire une fonction `int exp(int B, int p, int q)` calculant B^p mod q. On procèdera par exponentiation naïve (en $\mathcal{O}(p)$), et l'on fera **chaque** opération modulo q, afin de ne pas avoir de dépassement d'entier au cours des calculs.

Q10. En utilisant l'algorithme de Rabin-Karp, implémenter les fonctions de `texte.h`. Comparer les performances avec l'algorithme naïf, avec et sans optimisation `-Ofast`.

Q11. Modifier l'algorithme pour qu'il compte le nombre de collisions, c'est à dire le nombre de fois où la fenêtre du texte a le même hash que le motif sans pour autant y être égale.

Q12. Comparer les performances ainsi que le nombre de collisions en prenant $q = 1024$ puis $q = 1023$.

Q13. Comparer en prenant pour B un nombre premier (au plus 10^4), et $q = 2^{31}$.

Boyer-Moore-Horspool

Implémentons maintenant l'algorithme de Boyer-Moore-Horspool. On rappelle que cet algorithme consiste à utiliser la règle du mauvais caractère, qui utilise la définition suivante:

Définition 1. Soit $m \in \Sigma^*$ un motif de taille p et $a \in \Sigma$. La dernière occurence non-finale de a dans m, notée $d_m(a)$, est définie par la formule suivante, avec comme convention $\max \emptyset = -1$:

$$d_m(a) = \max\{i \in [\![0, p-2]\!] \mid m_i = a\}$$

Proposition 1 (Règle du mauvais caractère)**.** Soit t un texte et m un motif. Supposons que m ne se trouve pas à la position i dans t et que j est le premier indice en partant de la droite pour lequel $m_j \neq t_{i+j}$. Alors, la prochaine position où m peut apparaître est $i' = i + j - d_m(t_{i+j})$.

Cette règle donne lieu à l'algorithme de Boyer-Moore-Horspool:

Algorithme 1 : Recherche de motif: Boyer-Moore-Horspool

Entrée(s) : $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
Sortie(s) : i tel que m apparaît à la position i de t

```
1  i ← 0;
2  tant que i < n − p + 1 faire
3      j ← p − 1;
4      tant que j ≥ 0 et t_{i+j} = m_j faire
5          j ← j − 1;
6      si j = −1 alors
7          retourner i
8      sinon
9          i ← i + max(1, j − d_m(t_{i+j}));
10 retourner Pas d'occurence
```

Q14. Écrire une fonction `int* construire_d(char* m)` qui renvoie un tableau de taille 256 donnant les valeurs de $d_m(a)$ pour chaque caractère a. On rappelle qu'en C, un caractère est un entier, si bien qu'écrire `T['a'] = 3;` a pour effet d'écrire 3 dans la case de T d'indice 97 (le code ASCII de 'a').

Q15. Implémenter l'algorithme de Boyer-Moore-Horspool, et le comparer aux algorithmes précédents.

Q16. Recompilez les trois implémentations avec l'option GCC `-Ofast`, qui demande au compilateur d'optimiser le code, et mesurez à nouveau les temps d'exécution.

Q17. (Bonus) Lire la partie du cours sur la règle du bon suffixe, et l'utiliser pour implémenter l'algorithme de Boyer-Moore. Comparer avec les algorithmes précédents.

2 Codage de Huffman

On rappelle le principe du codage de Huffman: étant donné un texte source $t \in \Sigma^*$, on construit un arbre binaire strict dont les feuilles sont étiquetées par les lettres de Σ. Puis, on encode t en remplaçant chaque lettre par le chemin qui y mène dans l'arbre. L'efficacité de l'arbre de Huffman vient du fait que les lettres les plus fréquentes sont proches de la racine, et sont donc encodées par peu de bits. Le procédé peut donc se décomposer en deux parties:

1. Création de l'arbre de Huffman:

Algorithme 2 : Arbre de Huffman

Entrée(s) : $t \in \Sigma^*$ un texte source
Sortie(s) : A arbre de Huffman de t

$f \leftarrow$ dictionnaire des fréquences des lettres de Σ dans t;
$L \leftarrow$ liste d'arbres binaires, initialement vide;
pour $a \in \Sigma$ **faire**
    Ajouter à L une feuille $F(a, f[a])$;
tant que L ***contient au moins 2 arbres*** **faire**
    Sélectionner A_1 et A_2 dans L de fréquences minimales;
    Supprimer A_1 et A_2 de L et les remplacer par un nœud $N(A_1.f + A_2.f, A_1, A_2)$;
retourner ***l'unique élément de*** L

2. Encodage du texte source:

Algorithme 3 : Codage de Huffman

Entrée(s) : $t \in \Sigma^*$ un texte source
Sortie(s) : Suite de bits encodant t, A arbre de Huffman ayant servi à l'encodage

$A \leftarrow$ arbre de Huffman de t;
$c \leftarrow$ dictionnaire associant à chaque $a \in \Sigma$ son chemin $c[a]$ dans l'arbre A;
$res \leftarrow []$;
pour a ***lettre de*** t **faire**
    Ajouter $c[a]$ à res;
retourner res

Plutôt que d'utiliser les fréquences, on utilisera le nombre d'occurrences des lettres, ce qui est équivalent mais permet de ne manipuler que des entiers. On utilisera tout de même le terme "fréquence", abusivement. On propose d'utiliser le type suivant:

```
type huffman =
  | Leaf of int * char
  | Node of int * huffman * huffman
```

L'attribut `int` correspondra donc au nombre total d'occurrences dans le texte de tous les caractères de l'arbre.

Q1. Écrire une fonction utilitaire `freq: huffman -> int` qui renvoie la fréquence d'un arbre de Huffman.

Q2. Écrire une fonction `fusion : huffman -> huffman -> huffman` qui permet de fusionner deux arbres de huffman en un seul nœud, en sommant les fréquences.

Initialisation des feuilles Pour créer la liste initiale des feuilles, on compte le nombre d'occurrences de chaque lettre dans le texte source. Afin de stocker ces informations, nous allons utiliser un dictionnaire, implémenté par table de hachage. En OCaml, le module `Hashtbl` implémente cette structure de donnée. Le type `('a, 'b)Hashtbl.t` représente ainsi les tables de hachages où les clés sont de type `'a` et où les valeurs sont de type `'b`. On souhaite donc construire à partir d'un texte un objet de type `(char, int)Hashtbl.t`, qui à chaque lettre associe son nombre d'occurrences dans le texte.

Voici quelques fonctions utiles du module `Hashtbl`:

- `Hashtbl.create (n: int): ('a, 'b)Hashtbl.t` crée une table de hachage à n alvéoles. La taille est dynamique et change automatiquement au fur et à mesure que la table grandit, vous pouvez donc choisir une taille arbitraire à la création.
- `Hashtbl.mem (tb: ('a, 'b)Hashtbl.t)(k: 'a): bool` renvoie un booléen indiquant si la clé k apparaît dans la table tb.
- `Hashtbl.replace (tb: ('a, 'b)Hashtbl.t)(k: 'a)(v: 'b): unit` met à jour la table de hachage tb avec l'association (k, v). **Malgré son nom, cette fonction marche même si k n'existe pas déjà dans la table tb.**
- `Hashtbl.find (tb: ('a, 'b)Hashtbl.t))(k: 'a): 'b` renvoie la valeur associée à k dans la table tb, et lève une exception si la clé n'apparaît pas.

Vous pouvez trouver l'ensemble des fonctions du module Hashtbl dans la documentation d'OCaml. Sur les machines virtuelles, celle-ci se trouve directement dans le logiciel Zeal, sinon vous la trouverez sur internet: v2.ocaml.org/api/Hashtbl.html.

Q3. Écrire une fonction `count_freqs : string -> (char, int)Hashtbl.t` qui construit le dictionnaire des occurrences à partir d'une chaîne de caractères (*Cette fonction s'écrit plus simplement en impératif qu'en fonctionnel*).

Q4. La fonction `Hashtbl.fold` permet, comme la fonction `List.fold_left`, d'appliquer une fonction successivement sur tous les éléments d'une table de hachage. Par exemple, la fonction suivante renvoie la liste des couples (clé, valeur) d'une table:

```
1 (* liste des couples (clé, valeur) de tbl *)
2 let entrees (tbl: ('a, 'b) Hashtbl.t) : ('a * 'b) list =
3   Hashtbl.fold (fun cle valeur res -> (cle, valeur) :: res) tbl []
```

Regarder la documentation de la fonction `Hashtbl.fold` et s'inspirer de la fonction `entrees` ci-dessus pour écrire une fonction `huffman_leaves : string -> huffman list` qui prend en entrée un texte t et renvoie la liste initiale des feuilles de l'algorithme de Huffman.

Réduction de la liste Nous allons garantir comme invariant que la liste des arbres est toujours triée par fréquence croissante. Cela permettra de trouver facilement les deux arbres de fréquences minimales, et l'insertion ne sera pas dure. A noter que la complexité sera moins bonne qu'en utilisant une file de priorité (tas binaire, arbre binaire de recherche).
La première étape est donc de trier la liste initiale des feuilles.

Q5. Écrire une fonction `merge_sort: huffman list -> huffman list` triant une liste d'arbres par ordre croissant de fréquences, suivant le principe du tri fusion.

Q6. Écrire une fonction `insert_huffman : huffman -> huffman list -> huffman list` qui prend en entrée un arbre de Huffman h et une liste d'arbres l triés par fréquence croissante, et qui insère h dans l au bon endroit.

Q7. En déduire une fonction `fusion_huffman : huffman list -> huffman` qui prend en entrée une liste d'arbres de Huffman supposée triée, et qui renvoie l'arbre de Huffman obtenu en ayant itéré le processus de fusion jusqu'à ce que la liste soit réduite à un élément.

Q8. Déduire des fonctions précédentes une fonction `huffman_tree : string -> huffman` qui construit l'arbre de Huffman d'un texte.

Q9. Quelle est la complexité de cette fonction ? Quelle serait-elle si l'on utilisait des tas binaires pour stocker les arbres de Huffman au lieu d'une simple liste ?

Construction du dictionnaire A partir de l'arbre de Huffman, il faut construire le dictionnaire associant à chaque lettre le chemin y menant dans l'arbre. On cherche donc à construire une table de type `(char, bool list)Hashtbl.t`.

Q10. Écrire une fonction

```
1 construire_table: huffman -> (char, bool list) Hashtbl.t
```

construisant le dictionnaire en question. On pourra commencer par créer une table de hachage `chemins` vide, puis définir une fonction auxiliaire

```
1 construire_table_chemin: huffman -> bool list -> unit
```

qui prend en entrée un arbre de Huffman et une liste de booléens correspondant au chemin déjà parcouru **stocké à l'envers**, et qui rajoute à la table `chemins` les associations (lettre, chemin) dans l'arbre. On rappelle l'existence de la fonction `List.rev` servant à renverser l'ordre d'une liste.

Codage de Huffman

Q11. Écrire une fonction `compression_huffman : string -> bool list * huffman` qui prend en entrée un texte et renvoie un couple (C, h) où h est l'arbre de Huffman calculé à partir du texte et C le texte codé en utilisant h.

Décodage de Huffman La dernière étape de cette section est la mise en place de l'algorithme de décompression associé. On en rappelle le principe: étant donné une suite de bits C et un arbre de codage h, on lit C en suivant dans h le chemin correspondant. Lorsque l'on arrive à une feuille dans h, on note le caractère lu et on revient à la racine.

Q12. Écrire une fonction `read_path : huffman -> bool list -> char * bool list` prenant en entrée un arbre de Huffman h et une suite de booléens C, qui renvoie le caractère lu sur la feuille atteinte en suivant le chemin indiqué par C dans h, ainsi que la liste des booléens non lus restants. Par exemple, si la liste C est $[1, 0, 0, 1, 0]$ et que l'on atteint une feuille en allant à droite (1) puis à gauche (0), la liste restante sera $[0, 1, 0]$.

Q13. Écrire une fonction `decompression_huffman : huffman -> bool list -> string` qui décode une liste de booléens en utilisant un arbre de huffman, et qui renvoie le texte ainsi reconstruit. On pourra utiliser la fonction `String.make: int -> char -> string` pour transformer un caractère en un string de longueur 1, et utiliser l'opérateur de concaténation `^`.

Performances

Q14. Écrire une fonction `compare: string -> unit` qui calcule le nombre de bits utilisés pour encoder une chaîne via l'algorithme de Huffman, et sans encodage, et qui affiche les deux nombres ainsi que leur ratio. Pour calculer le nombre de bits utilisés sans encodage, on compte le nombre de lettres p distinctes utilisées, et on considère que chaque lettre du texte prend $\lceil \log_2 p \rceil$ bits.

Q15. (Bonus) Renseignez-vous sur les fonctions suivantes permettant de lire des fichiers:

- `open_in: string -> in_channel`;
- `input_line: in_channel -> string`;
- `close_in: in_channel -> unit`.

Utilisez les pour tester l'efficacité de l'algorithme de Huffman sur le fichier "vingt_mille_lieues.txt" de l'archive. **Attention:** pour que votre code fonctionne, il faudra que la plupart de vos fonctions soient récursives terminales !