Chapitre 21 : Algorithmique du texte

1 Définition

Alphabet et mot

Définition 1

- Un alphabet Σ est un ensemble fini non vide, dont les éléments sont appelés lettres ou symboles ou caractères.
- Un mot ou chaîne de caractères m sur Σ est :
  - soit le mot vide notée ϵ ;
  - soit une suite finie $m_1 \dots m_n$ d'éléments de Σ, avec $n \geqslant 1$.
- L'ensemble des mots sur Σ est noté Σ^*.

Exemple 2

- $\Sigma = \{a, b\}$ qu'on utilisera souvent comme exemple ;
- $\Sigma = \{0, 1\}$ quand on s'intéresse à un flux de bits ;
- l'alphabet « usuel » à 26 lettres ;
- l'ensemble des caractères ASCII standard ;
- $\Sigma = \{A, T, G, C\}$ si l'on traite les séquences d'ADN.

Longueur

Définition 3

- La longueur d'un mot m est notée $|m|$. Elle est définie par :
  - $|\epsilon| = 0$;
  - $|m_1 \dots m_m| = n$ (où $m_1, \dots, m_n \in \Sigma$).
- L'ensemble des mots de longueur n sur un alphabet Σ est noté Σ^n.
- Le nombre d'occurrences d'une lettre x dans m, ou longueur en x de m, est noté $|m|_x$. Il est défini par :

$$|m|_x = \text{Card}\{i \in [1 \dots n], m_i = x\}$$

Concaténation

Définition 4

La concaténation de deux mots u et v sur un même alphabet Σ est noté $u \cdot v$. Elle est définie comme suit :

- $\sigma \cdot v = v$;
- $u \cdot \epsilon = u$;
- $(u_1 \dots u_n) \cdot (v_1 \dots v_p) = u_1 \dots u_n v_1 \dots v_p$.

Pour un mot u et un entier $n \geqslant 0$, on définit u^n par :

- $u^0 = \epsilon$;
- $u^{n+1} = u \cdot u^n$.

Remarques

- L'opération de concaténation est associative : on notera donc $u \cdot v \cdot w$ pour $u \cdot (v \cdot w) = (u \cdot v) \cdot w$.
- On vérifie facilement que les règles de calcul usuelles sur les puissances s'appliquent : $u^m \cdot u^n = u^{m+n}$ et $(u^m)^n = u^{mn}$.
- On écrit parfois uv à la place de $u \cdot v$.

Préfixe, suffixe, facteur

Définition 5

Soit u et v deux mots sur un même alphabet Σ.

- u est un préfixe de v s'il existe un mot w tel que $v = u \cdot w$;
- u est un suffixe de v s'il existe un mot w tel que $v = w \cdot u$;
- u est un facteur de v s'il existe deux mots w et z tels que $v = w \cdot u \cdot z$.

Exercice 6

Soit $u = abab$. Donner les préfixes, les suffixes et les facteurs de u.

Exercice 7

Soient u et u' deux préfixes d'un mot v. Justifier que u est un préfixe de u' ou u' est un préfixe de u.

Sous-mot

Définition 8

Soient $u, v \in \Sigma^*$, $u = u_1 \cdot u_n$ ($n = 0$ si u est vide). On dit que u est un sous-mot de v s'il existe des mots $t_0, \ldots, t_n$ tels que $v = t_0 \cdot u_1 \cdot t_1 \cdot u_2 \ldots t_{n-1} \cdot u_n \cdot t_n$.

Exercice 9

Donner les sous mots de $abcd$.

Exercice 10

Soit u un mot de longueur n dont toutes les lettres sont distinctes. Combien u a-t-il de préfixes, de suffixes, de facteurs, de sous-mots ?

2 Recherche de chaîne

Une des opérations les plus courantes sur le texte est de rechercher une chaîne de caractères dans une autre chaîne de caractères (plus longue). C'est ce qu'on fait avec un simple Ctrl-F, mais c'est également le problème central de l'alignement de séquences en génétiques. Le problème se résume ainsi :

Problème 11

Entrées : — une chaîne de caractères s sur l'alphabet Σ.

— Une autre chaîne de caractères m sur ce même alphabet appelé motif et de longueur plus petite que s.

Sortie : Un résultat partiel correspondant à l'indice de la première occurrence du motif dans la chaîne s'il est présent.

2.1 Solution naïve

Une solution naïve consiste à parcourir chaque position de `t` afin de tester si le motif est présent à partir de cette position.

Exercice 12

- Écrire une fonction `est_prefixe` telle que l'appel `est_prefixe t m` renvoie `true` si et seulement si `m` est un prefixe de `t`.
- En déduire `est_present` telle que l'appel `est_present t m` renvoie `some i` si la première occurrence de `m` dans `t` est à l'indice `i` et renvoie `none` si le mot n'est pas présent dans `t`.
- Écrire une fonction `occurrences` telle que l'appel `occurrences t m` renvoie la liste des occurrences de `m` dans `t`.

La complexité temporelle dans le pire cas de cet algorithme correspond au maximum de comparaisons. Le pire cas est constitué du mot `mot = aa...aab` dans le texte `s = aa...aa`. La complexité est donc en $\mathcal{O}(np)$ où n est la taille du texte et p la taille du mot.
Remarque Dans une application usuelle de cet algorithme, au bout d'une ou deux comparaisons, on pourra invalider la position et passer à la suivante. On va alors une complexité en $\mathcal{O}(n+p)$ en considérant en plus la validation du motif dans le cas où il est présent.

2.2 Algorithme de Boyer-Moore

En 1977 Boyer et Moore proposent une amélioration de l'algorithme naïf en permettant de sauter certains des tests au prix d'un pré-traitement du motif m, et en comparant $s[i : i+i]$ et m de droite à gauche.
Dans ce cours, nous présenterons que la version simplifié de cet algorithme connu sous le nom de Boyer-Moore-Horspool (1980) (c'est la seule qui est explicite au programme).

Principe

On cherche le motif m dans le texte t. On teste s'il y a une occurrence en ℓ (c'est-à-dire si $m = t[\ell : p+\ell]$), et pour faire ce test on commence à comparer les caractères à partir du dernier caractère du motif.

- Si l'on ne trouve aucun caractère différent entre motif et facteur du texte, on a trouvé notre mot, on renvoie ℓ.
- Sinon, soit i le plus grand indice tel que $m_i \neq t_{\ell+i}$, soit $x = t_{\ell+i}$ et $droite(x)$ le plus grand indice j tel que $m_j = x$ (-1 si aucun indice ne convient). On peut être sûr qu'il n'y a pas d'occurrence du motif commencant avant $\ell + i - droite(x)$.

  En effet, en commencant à $s < i + \ell - droite(x)$, il faudrait que $t_{\ell+i} = m_{\ell+i-s}$, or $\ell + i - s > droite(x)$, ce qui contredirait la maximalité de $droite(x)$.

  Notons que $\ell + i - droite(x)$ peut être négatif si jamais $\ell = 0$ et qu'il y a une occurrence de x à droite de i dans m : il faut donc se décaler de $max(1, i - droite(x))$.

Le principe est beaucoup plus facile à comprendre sur une figure.

Exemple 13

Dérouler le principe sur le texte *abcaababbaabaaaab* avec le mot *abaaa*

Complexité

Par rapport à l'algorithme naïf, on voit bien qu'on économise parfois des comparaisons. Ce n'est toutefois pas systématiques : essentiellement, l'algorithme sera d'autant plus rapide que l'on effectuera souvent des sauts importants.

Implémentation

Pour que l'algorithme soit efficace, il est indispensable de pouvoir déterminer $droite(x)$ (avec les notations ci-dessus) sans parcourir tout le motif. Pour cela, on précalcule les $droite(x)$ pour tous les caractères x de l'alphabet, ce qui peut se faire sans problème en temps $\mathcal{O}(|m|)$.

Exercice 14

Écrire une fonction OCaml `calcule_droite` prenant en entrée une chaîne `m` et renvoyant un tableau de longueur 256 contenant tel que, pour tout code ASCII du caractère x, on ait sa dernière apparition dans `m` ou -1

Exercice 15

Écrire une fonction `boyer_moore_horspool` prenant en entrée un texte et un motif à chercher, et renvoyant un type option indiquant la présence du motif.

2.3 Algorithme de Rabin-Karp

Principe de Rabin-Karp

La complexité des algorithmes précédents repose sur les n comparaisons sur des chaînes de longueur p. Le principe de l'algorithme de Rabin-Karp est de limiter le nombre de comparaison coûteuses à effectuer, en commencant à chaque fois par comparer un hash du facteur `t[i:i+p]` avec la chaîne cherchée. Un hash étant un entier, on peut faire une telle comparaison en $\mathcal{O}(1)$ puis :

- si les empreintes diffèrent, on est sûr qu'il n'y a pas d'occurrence en i ;
- si les empreintes coïncident, on peut être en présence du mot ou d'une collision il faut comparer les chaînes pour trancher.

Présenté comme cela, l'algorithme de Rabin-Karp n'a pas l'air très efficace. Il faut a priori un temps proportionnel à p pour calculer `t[i:i+p]`. On obtient une complexité pas meilleure que la recherche naïve.
L'idée essentiel de Rabin-Karp est de calculer le hash de `t[i:i+p]` en un temps constant à partir du hash de `t[i+1:i+1+p]`.

Choix d'une fonction de hachage

Ceci est possible en choisissant une base b, on définit la fonction h

$$h(\text{"}s_0 s_1 \cdots s_{p-1}\text{"}) = \sum_{i=0}^{n} b^{p-i-1}\text{code}(s_i)$$

où $\text{code}(s_i)$ est le code ASCII du caractère s_i.
En effet, on a

$$h(t_{i+1}t_{i+2}\dots t_{i+p}) = b\left(h(t_i t_{i+1}\dots t_{i+p-1}) - b^{p-1}t_i\right) + t_{i+p}$$

Pour la base, on prendra typiquement $b = 256$ si l'on travaille par octet (b doit être supérieur à la plus grande valeur possible pour un t_i).
Mathématiquement, avec cette version, il ne peut y avoir de collision.
Cependant, la taille de $h(t_i t_{i+1}\dots t_{i+p-1})$ n'est pas bornée. En pratique, avec $b = 2^8$, on dépasse la capacité d'un entier 64 bits dès que la longueur du motif recherché dépasse 7 … On travaille donc modulo q, où q est un entier que l'on choisira premier et assez grand mais pas trop . C'est-à-dire tel que q^2 tiennent sur un entier de 64 bits. On choisira par exemple $q = 2^{31} - 1$.
Ce choix de nombre premier nous permet de faire un calcul rapide modulo à partir de l'écriture du nombre en base 2^{31}. En effet si $a = \sum_{k=0}^{n} a_k 2^{31k}$ comme $2^{31} - 1 | 2^{31k} - 1$ pour $k \geqslant 1$, on a $2^{31k} \equiv 1[q]$ et ainsi $a \equiv \sum_{k=0}^{n} a_k [q]$.
Le calcul rapide du modulo dépendra du calcul rapide de $a_k \bmod q$.

Exercice 16

- Écrire les a_k en fonction de $2^{31}-1$ et de $31k$ avec des `lsr` et `land`.
- En déduire la fonction `calcul_mod a` qui calcule rapidement $a \bmod (2^{31}-1)$.

Implémentation de l'algorithme

Il ne reste plus qu'à écrire l'algorithme. Ce travail est à faire en exercice.

Exercice 17

- Écrire la fonction `hash s` qui prend en argument une chaîne de caractère et renvoie son hash avec la fonction définie ci-dessus.
- Écrire la fonction `delta rp a b` qui prend en arguments la valeur du hash de $rp = h(t_i t_{i+1} \dots t_{i+p-1})$, les caractères $a = t_i$ et le caractère $b = t_{i+p}$ et renvoie $h(t_{i+1} t_{i+2} \dots t_{i+p})$.
- En déduire la fonction `rapin_karp t m` qui renvoie l'indice de la première apparition de `m` dans `t`.

Complexité

Complexité 18

En notant c la proportion de collisions, la complexité temporelle de l'algorithme de Rabin-Karp est en $\mathcal{O}(n + cnp)$. Sauf cas pathologique, c est de l'orde de $1/q$, et comme q peut être choisi de l'ordre de 2^{32}, on a $cn < 1$.

Cette complexité est théorique, en pratique le nombre de comparaisons est beaucoup plus faibles.

3 Compression

On s'intéresse ici à la compression parfaite d'un texte, c'est-à-dire, étant donné un alphabet fixé Σ, on cherche à réaliser un couple de fonctions $comp, decomp : \Sigma^* \to \Sigma^*$ telles que :

- $decomp \circ comp \circ m = m$.
- Pour la plupart des mots m qui nous intéressent (qui ont une structure), $|comp(m)| < |m|$.

Remarques

- En réalité, *decomp* ne sera généralement pas défini sur Σ^* en entier, mais seulement sur $comp(\Sigma^*)$. Le comportement de *decomp* sur un mot qui n'est pas le résultat d'une opération de compression pourra être quelconque.
- Ce qui nous intéresse dans un algorithme de compression, ce n'est pas d'arriver à compression une entrée quelconque, mais d'ariver à compresser efficacement une entrée avec une structure. Par exemple un texte, du code, ...

Si vous compressez avec zip un fichier texte vous pouvez aurez aurez un énorme ratio de compression.
Un fichier quelconque composé de bits aléatoires risque d'avoir un fichier compressé plus gros que le fichier d'origine

3.1 Algorithme de Huffman

L'algorithme de Huffman a été vu en TP : on rappele ici le principe et on retournera voir le TP pour sa mise en œuvre.
L'algorithme de Huffman repose sur l'idée suivante : si certains caractères du texte à compresser apparaissent souvent, il est préférable de les représenter par un code court. Par exemple, dans le texte « satisfaisant », les caractères `'a'` et `'s'` apparaissent souvent, à savoir trois fois chacun.

On peut ainsi choisir de représenter le caractère `'a'` par la séquence `'01'` et le caractère `'s'` par la séquence 10 et les caractères `'f'`, `'n'`, `'i'` et `'t'` par des séquences plus longues encore, par exemple respectivement `'000'`, `'001'`, `'110'`, `'111'`. Le texte compressé sera alors :

```
1001111110100000111010010011111
```

3.2 Algorithme de Lempel-Ziv-Welch

L'algorithme d'Huffman est efficace, mais il présente un désavantage majeur : il nécessite de lire le contenu d'un fichier dans son intégralité pour pouvoir déterminer un code préfixe optimal. Il est toutefois possible de modifier l'algorithme pour lever cette limitation. Dans ce paragraphe, nous allons plutôt étudier une autre technique de compression qui, bien que moins efficace en pratique que Huffman, se programme assez facilement et permet de compresser des flux plutôt que des fichiers. C'est-à-dire qu'on peut compresser et décompresser des données au fur et à mesure qu'elles sont transmises.
Il s'agit de l'algorithme de Lempel-Ziv-Welch, appelé communément compression LZW, et qui est une modification faite en 1984 par Welch de l'algorithme de LZ78 de Lempel et Ziv.

Principe de la compression

L'idée de l'algorithme LZW est de faire avancer une fenetre sur le texte en maintenant une table des motifs déjà rencontrés. Quand on rencontre un motif déjà vu, on le code avec une référence vers la table et quand on rencontre un nouveau motif, on le code tel quel en rajoutant une entrée dans la table.

Algorithme

L'algorithme procède ainsi pour compresser :

Algorithme 19

- Initialiser la table avec une entrée pour chaque caractère ASCII.
- on maintient une variable m contenant le plus long suffixe du texte lu qui soit présent dans la table, il est initialisé avec la première lettre du texte. On lit alors chaque caractère x :
  - Soit mx est dans la table, et alors on remplace le motif courant par $m \leftarrow mx$.
  - Soit mx n'est pas dans la table, par construction m y est nécessairement on produit alors le code correspondant à m, on rajoute une entrée dans la table pour mx et on repart de $m \leftarrow x$.
- quand tous les caractères ont été lus, on produit le code correspondant à m.

Exemple

Exemple 20

Considérons la chaîne "AABABAAAB" et remarquons juste que le code ASCII de "A" est 65

Décompression

Pour décompresser, on ne dispose pas de la table : il faut donc la reconstituer au fur et à mesure du décodage, en déterminant les opérations de compression qui ont nécessairement conduit à l'émission des codes lus.

Algorithme de décompression 21

- On initialise la table t avec les valeurs des 256 octets, puis l'on traite le premier code lu en emmetant cet octet. Pour tous les codes suivants :
  - On lit un code n où $n < |t|$ et $t[n] = xm'$;
  - On écrit xm' dans le fichier de sortie.
  - On ajoute à la table mx, où c est le précédent code lu et $t[c] = m$.

Exemple 22

Déroulons l'algorithme sur 65 65 66 257 256 256.

Il