Chapitre 18 - Algorithmique du texte

Vocabulaire :

On travaille avec un alphabet $\sum$, ensemble fini non vide de symboles appelées des lettres.

Alphabets classiques :

- $\sum = \{\text{caractères ASCII}\}$
- $\sum = \{0, 1\}$
- $\sum = \{\text{unicode}\}$

Un mot est une suite, éventuellement vide, de lettres sur un alphabet. L'ensemble des mots sur un alphabet $\sum$ est noté $\sum^*$. Le mot vide est noté ϵ.

La taille d'un mot $m \in \sum^*$ est noté $|m|$, et est le nombre de lettres de la suite qui le compose.

La concaténation de deux mots u et v, notée $u \cdot v$ est défini par :

- $\epsilon \cdot v = v$
- $u \cdot \epsilon = u$
- $(u_0 u_1 \dots u_{|u|-1}) \cdot (v_0 v_1 \dots v_{|v|-1}) = u_0 u_1 \dots u_{|u|-1} v_0 v_1 \dots v_{|v|-1}$

$u \in \sum^*$ est un préfixe d'un mot $v \in \sum^*$ si il existe $\omega \in \sum^*$ tel que $v = u \cdot \omega$.

$u \in \sum^*$ est un suffixe d'un mot $v \in \sum^*$ si il existe $\omega \in \sum^*$ tel que $v = \omega \cdot u$.

$u \in \sum^*$ est un facteur d'un mot $v \in \sum^*$ s'il existe $\omega_1, omega_2 \in \sum^*$ tel que $v = \omega_1 \cdot u \cdot \omega_2$.

Proposition :

Un mot $u \in \sum^*$ possède :

- |u| + 1 préfixes
- |u| + 1 suffixes
- $\frac{|u|(|u|+1)}{2} + 1$ facteurs

L'occurrence d'un mot $u \in \sum^*$ dans un mot $v = v_0 v_1 \dots v_{|v|-1} \in \sum^*$ est un indice i tel que $u = v_i \dots v_{i+|u|-1}$.

On parle d'algorithmique du texte quand le nombre de lettres de l'alphabet est négligeable devant la taille des mots.

I. Recherche d'un motif dans un texte

Entrées :

- Un mot sur un alphabet $\sum$, appelé le texte.
- Un mot sur le même alphabet $\sum$, appelé le motif m, avec $|m| \leq |t|$(et généralement $|m| << |t|$).

Sortie :

- Première variante : booléen indiquant si m est un facteur de t.
- Deuxième variante : première occurrence de m dans t.
- Troisième variante : trouver toutes les occurrences de m dans t.

On peut utiliser un algorithme de recherche par force brute où pour chaque indice possible $0 \leq i < |t|$, on détermine si i est une occurrence de m dont la complexité est $\mathcal{O}(|m|)$. Ainsi la complexité totale est en $\mathcal{O}(|t| \times |m|)$.

1. Algorithme de Rabin-Karp

Il permet de diminuer la complexité du test d'une occurrence.

$\rightarrow$ On veut éviter de faire $|m|$ complexité pour chaque indice.

$\rightarrow$ On compare l'empreinte du motif avec l'empreinte du facteur à la position actuelle.

$\rightarrow$ L'empreinte est le résultat entier d'une fonction de hachage.

$\rightarrow$ Si les empreintes ne correspondent pas, on passe à l'indice suivant. Sinon, on compare (car risque de collision).

Exemple :

$\sum$ = {alphabet à 26 lettres}

Fonction de hachage $h(u_0 \dots u_{|u|-1}) = \sum_{i=0}^{|u|-1} P_{u_i}$ avec P_{u_i} la position dans l'alphabet;

$t = abacbbabca$ et $m = abc$.

<u>Etape 0 :</u> L'empreinte du motif est calculé. $h(abc) = 1 + 2 + 3 = 6$.

<u>Etape 1 :</u> $h(aba) = 4 \neq 6$ donc aucune comparaison.

<u>Etape 2 :</u> $h(bac) = 6 = 6$ donc $|m|$ comparaisons, collisions.

<u>Etape 3 :</u> $h(acb) = 6 = 6$ donc $|m|$ comparaisons, collision.

<u>Etape 4 :</u> $h(cbb) \neq 6$, pas de comparaisons.

<u>Etape 5 :</u> idem.

$\dots$

<u>Etape 7 :</u> $h(abc) = 6$, $|m|$ comparaisons, occurrence trouvée.

<u>Critères sur la fonction de hachage :</u>

→ Eviter les collisions.

→ Le calcul initial des empreintes du motif et du facteur du texte à l'indice 0 se fait en $\mathcal{O}(|m|)$.

→ Le calcul de l'empreinte à l'indice $i+1$ doit pouvoir être fait en $\mathcal{O}(1)$ à partir de l'empreinte de l'indice i.

<u>Fonction respectant ces critères est utilisée par Rabin-Karp :</u>

$h(l_0 l_1 \dots l_{|l|-1}) = (\sum_{i=0}^{|l|-1} l_i b^{|l|-1-i}) mod(p)$ avec

- $*l_i$ un code entier assimilé à chaque lettre de l'alphabet (exemple : code ASCII).
- b la base, plus grande que le nombre de lettres de $\sum$.
- p choisi entier premier et le plus grand possible (souvent $2^{31} - 1$).
- avec l'algorithme de Horner, on a bien un calcul initial en $\mathcal{O}(|m|)$.

- $h(l_{i+1}\dots l_{i+|l|}) = b(h(l_i \dots l_{i+|l|-1}) - l_i b^{|l|-1}) + l_{i+|l|}$ avec l le facteur du texte de taille $|m|$. Le calcul de h entre parenthèses est en $\mathcal{O}(1)$.

Complexité :

Au plus $|t|$ itérations. Le calcul initial est en $\mathcal{O}(|m|)$ pour les 2 empreintes.

La nouvelle empreinte est en $\mathcal{O}(1)$. On a seulement 1 comparaisons sur les empreintes ne correspondent pas. La complexité est en $\mathcal{O}(|m|)$ sinon. En notant C le nombre de collisions :

Complexité : $\mathcal{O}(|m|) + \mathcal{O}(|t|) \times \mathcal{O}(1) + C \times O(|m|) = \mathcal{O}(|t| + C \times |m|)$

Admis, sauf cas pathologique, C est de l'ordre de $\frac{1}{p}$.

2. Algorithme de Boyer-Moore

Il permet de diminuer le nombre d'itérations.

Première version : Boyer-Moore-Horspool

Première étape : Pré-traitement du motif

```
Pour chaque préfixe du motif i :
        Pour chaque lettre de l'alphabet :
                        On stocke la dernière occurence de la lettre
dans le préfixe dans une table à l'indice [i,j] (et -1 si n'apparaît
pas)
```

Exemple :

$\sum = \{a, b, c\}$; $m = abaaa$

l	a	b	c
ϵ	-1	-1	-1
a	0	-1	-1
ab	0	1	-1
aba	2	1	-1
$abaa$	3	1	-1
$abaaa$	4	1	-1

Deuxième étape :

→ On compare le motif avec le facteur actuel de la droite vers la gauche.

→ Quand la comparaison échoue entre une lettre l du texte et une lettre à l'indice j du motif, on décale le motif pour la prochaine comparaison de $j - décalage$, avec $décalage =$ entier donné par la table du prétraitement, ligne j, colonne l.

Exemple :

$t = abcaababbaabaaaab$.

l	**a**	**b**	**c**	**a**	**a**	**b**	**a**	**b**	**b**	**a**	**a**	**b**	**a**
$i = 0$	a	b	a	a	a								
$i = 3$				a	b	a	a	a					
$i = 6$							a	b	a	a	a		
$i = 7$								a	b	a	a	a	
$i = 10$											a	b	a

Complexité : Même que celle de l'algorithme par force brute : $\mathcal{O}(|m| \times |t|)$

Deuxième version : Boyer-Moore, version complète

- Il construit plusieurs tables de décalage lors du pré-traitement du motif (étape 1).
- Lors du parcours du texte et des comparaisons de droite à gauche (étape 2), on utilise le meilleur décalage fourni par les tables construites.

La table de décalage construite dans la version de Horspool, aussi utilisée pour la Boyer-Moore complet, utilise la "règle du mauvais caractère" (seule à connaître et savoir implémenter).

Les autres règles, par exemple "règle du bon suffixe", seront explicitées dans les sujets.

Avec toutes les règles combinées, on a une complexité de $\mathcal{O}(|m| + |t|)$.

II. Compression d'un texte

Compression sans perte :

On possède un mot $\in \sum^*$, appelé le texte t. On veut écrire deux fonctions $comp$, $decomp$ $\sum^* \rightarrow \sum^*$ telles que $decomp(comp(t)) = t$. Pour les mots qui nous intéressent, $|comp(t)| < |t|$.

Le résultat de la compression d'un texte est appelé codage du texte. Les règles utilisées pour les construites sont des codes.

<u>Exemple :</u>

$texte = abaac$; $code$: $a = 00, b = 01, c = 11$

Alors le codage est : 0001000011.

Un code à longueur fixe est tel que pour tout c et c' deux codes, $|c| = |c'|$.

Un code préfixe est tel que pour tout c et c' deux codes, alors c n'est pas un préfixe de c'.

Un code à longueur fixe ou un code préfixe permet une compression sans perte.

Il y a ainsi deux algorithmes :

- Algorithme de Huffman qui construit un code préfixe
- Algorithme de Lempel-Ziv-Welch pour un code à longueur fixe.

1. Algorithme de Huffman

<u>Compression :</u>

<u>Première étape :</u>

Compter le nombre d'occurrences de chaque caractères du texte à compresser.

<u>Exemple :</u>

mot : "satisfaisant", $s = 3, a = 3, t = 2, i = 2, f = 1, n = 1$

<u>Deuxième étape :</u>

On construit l'arbre de Huffman optimal.

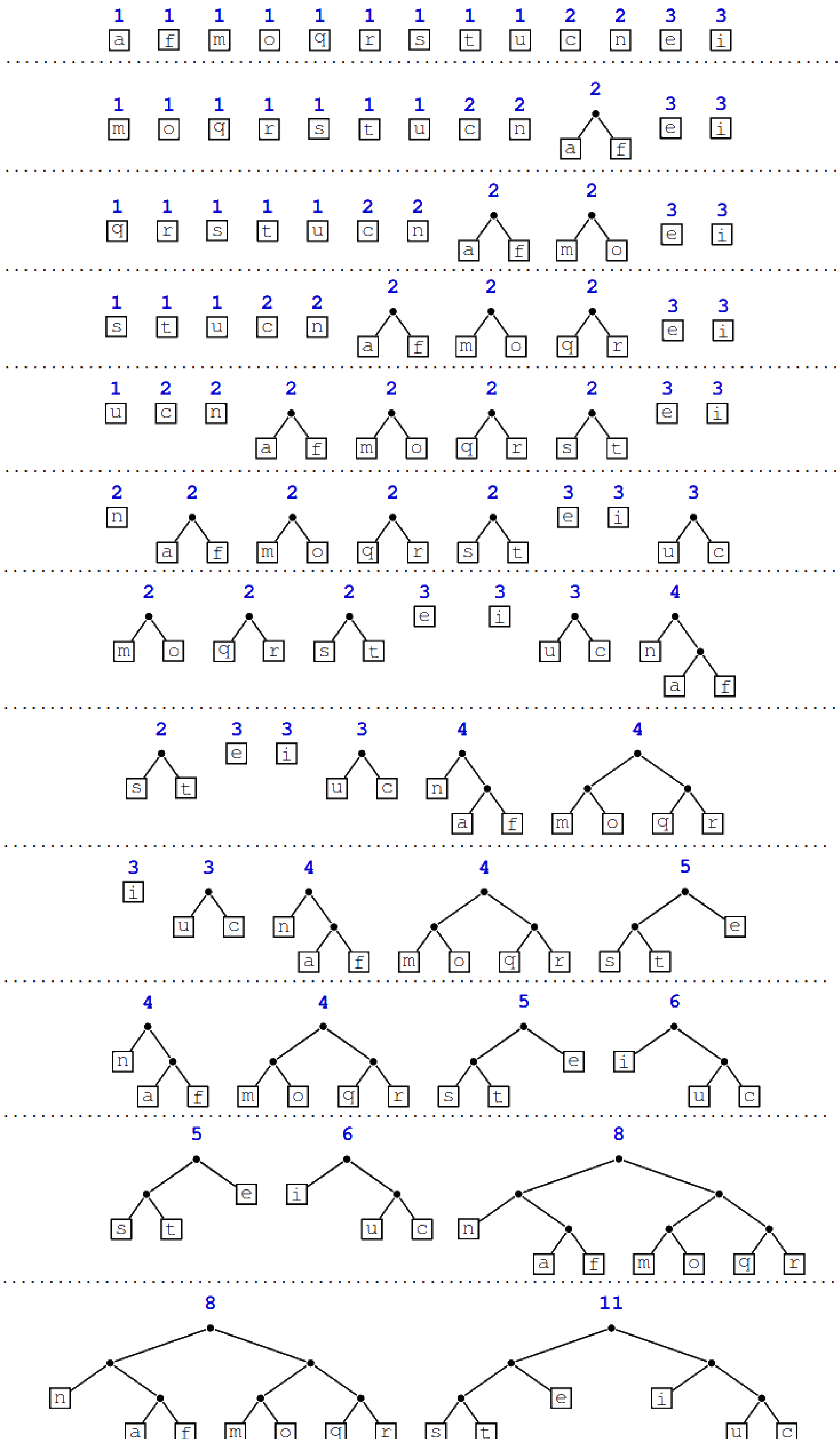
On construit une forêt d’arbres réduits à une feuille, étiquetées par chaque lettre apparaissant dans le texte, de poids leur nombre d’occurrences.

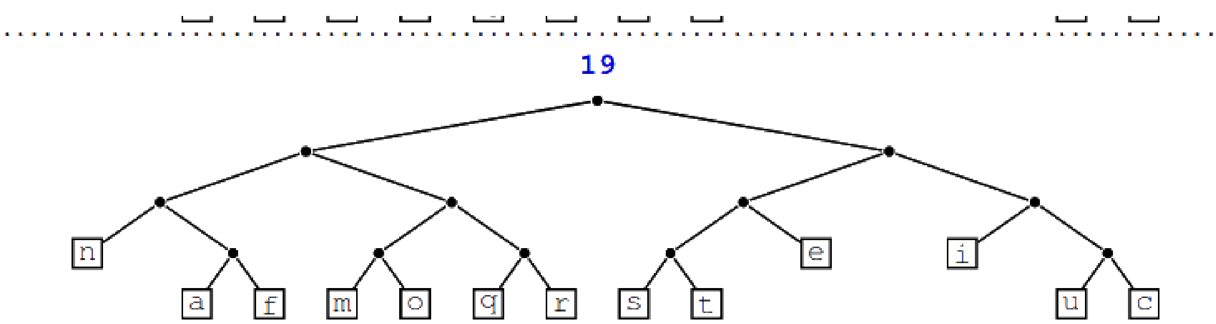
Tant que la forêt contient plus d’un arbre, on sélectionne deux arbres de la forêt g et d, on construit un nouvel arbre N(g,d) de poids le poids de g + le poids de d et on le remet dans la forêt.

L’algorithme de Huffman est un algorithme glouton, les deux arbres choisis sont de poids minimaux. Le seul restant est l’arbre de Huffman optimal.

<u>Exemple :</u>

Pour le mot “scienceinformatique” :





Troisième étape :

On construit le code :

Pour chaque lettre a un code différent, déterminé par sa position dans l'arbre. On suit le chemin de la racine à la feuille, en ajoutant 0 si gauche et 1 si droite.

<u>Exemple :</u>

$a : 00, n : 010, f : 011, s : 10, i : 110, t : 111.$

<u>Propriété :</u>

Par construction de l'arbre, le code est préfixe.

<u>Quatrième étape :</u>

On utilise le code pour construire le codage du texte.

$satisfaisant \to 10001111101001100\ldots1101000010111$ (compression).

<u>Dernière étape :</u>

Le résultat de la compression (ce qu'il faut stocker/envoyer) est le texte compressé et l'arbre de Huffman.

<u>Propriété :</u>

Le code de Huffman est optimal pour un code préfixe, c'est-à-dire qu'il minimise

$\sum_{\text{lettre } l_i} |\text{code de } l_i| \times \frac{\text{nb d'occ. de } l_i}{|\text{text}|}$ (Admis, se montre par récurrence sur le nombre de lettres de $\sum$).

Décompression :

On reçoit l'arbre et $comp(t)$, comment retrouver t.

On parcourt $comp(t)$ en descendant à gauche, à droite dans l'arbre, quand on arrive à une feuille, on l'ajoute à la décompression et on repart de la racine.

Le mot “satisfaisant” devient ainsi “saint”.

2. Algorithme de Lempel-Ziv-Welch

C'est un code à longueur fixe, la longueur est choisi au préalable. On stocke les codes au fur et à mesure de la compression dans une table T. Chaque lettre de l'alphabet est codé par lui-même (typiquement code ASCII).

Donc tous les codes entre 0 et $|\sum| - 1$ sont pris.

On lit le texte de gauche à droite, lettre par lettre, en gardant dans un “$buffer$” les lettres lues et non compressées.

- Quand $buffer.caractèrelu \in T$, on ajoute $caractèrelu$ au $buffer$.
- Sinon, on l'ajoute à T avec un nouveau code pas encore pris, et on compresse le $buffer$.

A la fin, on compresse ce qui est resté dans le $buffer$.

$\sum = \{e, n, t, d\}$, Texte = entendent

T	$buffer$	**caractère lu**	**compression**
$E \leftrightarrow 0, N \leftrightarrow 1, T \leftrightarrow 2, D \leftrightarrow 3$			
		e	
$en \leftrightarrow 4$	e	n	0
$nt \leftrightarrow 5$	n	t	1
$te \leftrightarrow 6$	t	e	2
	e	n	
$end \leftrightarrow 7$	en	d	4
	d	e	3
	e	n	
$ent \leftrightarrow 9$	en	t	4
	t	rien	2

On a alors : $comp(texte) = 0124342$.

La table T n'est pas conservée, on garde la taille choisie.

<u>Décompression :</u>

$comp(t) = 0124342$, $\sum = \{e, n, t, d\}$.

T est reconstruite au fur et à mesure de la décompression. On ajoute un nouveau code à la table pour la décompression précédente concaténée à la décompression actuelle (première lettre).

T	**Code lu**	**décompression**
$e \leftrightarrow 0, n \leftrightarrow 1, t \leftrightarrow 2, d \leftrightarrow 3$		
	0	e
$en \leftrightarrow 4$	1	n
$nt \leftrightarrow 5$	2	t
$te \leftrightarrow 6$	4	en
$end \leftrightarrow 7$	3	d
$de \leftrightarrow 8$	4	en
$ent \leftrightarrow 9$	2	t

$\sum = a, comp(t) = 01$.

T	**lu**	**décompression**
$a \leftrightarrow 0$		
	0	a
$aa \leftrightarrow 1$	1	aa

Quand ce cas se produit, on ajoute à T un code pour (décompresser avant).(décompresser avant première lettre), ici (a).(première lettre de a) $= aa$.