Chapitre 22 : Algorithmique du Texte

Notations et Vocabulaire

Notations

On appelle alphabet, souvent noté Σ, un ensemble fini de symboles.

Exemple : $\Sigma = \{ASCII\}$

$|\Sigma| = 256$

Exemple : En bio-informatique

$\Sigma = \{A, T, C, G\}$

$|\Sigma| = 4$

On fixe un alphabet sur Σ.

Un mot est une suite finie de symboles de Σ. $u = a_0, ...a_{n-1}$. On note $|u| = n$ la longueur de u. Les a_i sont les lettres de u. On notera $u[i] = a_i$ le i^e caractère de u.

On note ϵ le mot vide (l'unique mot de longueur 0).

On note Σ^* l'ensemble des mots.

On note $\circ$ la concaténation des mots. $u \circ v = a_0...a_{n-1}b_0...b_{m-1}$ avec $u = a_0...a_{n-1}$ et $v = b_0...b_{m-1}$. On écrit souvent uw au lieu de $u \circ w$.

Vocabulaire

u est un préfixe de v si

- $u = a_0....a_{n-1}$
- $v = a_0...a_{n-1}a_n...a_{m-1}$

Respectivement "suffixe"

u est un facteur de v s'il existe un mot w tel que wu est un préfixe de v.

$m[i:j]$ est le facteur de m entre position i (inclus) et j (exclus), $m = a_0...a_{n-1} \Rightarrow m[i:j] = a_i...a_{j-1}$

I - Recherche de Mot dans un Texte

1. Algorithme de Robin Karp

a. Algorithme naïf

Première approche

$m, T \in \Sigma^*$

On veut savoir si m est un facteur de T

Pour $i_0 \in [0, |T| - |m|[$

Tester si $m = T[i_0 : i_0 + |m|]$

```
algo_naif(texte, motif) # tiré de wikipédia
1.  n ← longueur(texte)
2.  m ← longueur(motif)
3. pour i de 1 à n-m+1 faire
4.    si texte[i..i+m-1] = motif[1..m]
5.       motif trouvé dans le texte à la position i
6. motif non trouvé
```

Complexité en $O((|T| - |m|)|m|)$

Exemple pire cas :

m = “dauphin”

T = “dauphi dauphi dauphi…”

Idée de l’algorithme

On prend une fonction de hachage h et on remplace le test $m == T[i_0, i_0 + m]$ par le test $h(m) = h(T[i_0, i_0 + m])$

S’il y a égalité des haches, alors on fait le premier test.

Empiriquement, si h est “bien faite” (peu de collisions) alors on s’attend à réaliser le premier test beaucoup moins souvent.

Problème : Le calcul de h(m) est en $O(|m|)$

Pour l’algorithme on choisit $h(m) = \sum_{i=0}^{|m|-1} c(m[i])|\Sigma|^i mod(N)$ avec $c : \Sigma \to [0, |\Sigma|[$ une énumération de Σ.

$\Rightarrow h(m)$ se calcule en $O(|m|)$

L’algorithme

On obtient $h(T|i_0 : i_0 + |m|] = h(T[i_0 - 1 : i_0 - 1 + |m|])/|\Sigma\ | + \sum^{|m|-1} c(T[i_0 + |m| - 1])mod(N)$

La somme est calculée qu’une seule fois.

On prend h_m le haché de m

On calcule $\sum^{|m|-1}$.

h_T = haché $T[0 : |m| - 1]$*$|\Sigma|$

Pour $i_0 \in [0, |T| - |m|[$

$h_T = h_T/|\Sigma| + \sum^{|m|-1} *c(T[i_0 + |m| - 1])$

Si $h_T == h_m$:

Si $m == T[i_0 : i_0 + |m|]$:

Retourner vrai

Retourner faux

```
rabin_karp(texte, motif) # tiré de wikipédia
 1.  n ← longueur(texte)
 2.  m ← longueur(motif)
 3.  hn ← hach(texte[1..m])
 4.  hm ← hach(motif[1..m])
 5.  pour i de 0 à n-m+1 faire
 6.    si hn = hm
 7.      si texte[i..i+m-1] = motif[1..m]
 8.        motif trouvé dans le texte à la position i
 9.    hn ← hach(texte[i+1..i+m])
10. motif non trouvé
```

Complexité : Dans le pire cas on effectue le test `m == T[...]` à chaque fois et on a donc rien gagné. L'analyse de complexité pire cas n'est pas pertinente ici. L'efficacité empirique de cet algorithme repose sur le fait que lorsque $m \space !=T[i_0:i_0+|m|]$ alors $h_m ! = h_T$ dans la plupart des cas.

Autrement dit, les cas où $h_T = h_m$ et $m \space!= T[...]$ sont rares.

2. Algorithme de Boyer-Moore

a. Algorithme (version 1)

```
algo
1. i0 = 0
2. Tant que i0 <  |T| - |m|
3.     tester m == T[i0 : i0 + |m|] en partant de la droite
4.     Si cela échoue, en prenant compte de T[i0 + |m| - 1]
5.         On se décale intelligament : i0 = i0 + decalage(T[i0+|m -1])
```

Comment construire `décalage` ?

Pour représenter cette fonction on pourrait utiliser un tableau "offset" et une énumération de Σ, notée c de sorte que $offset[c(a)] = decalage(a) \forall a \in \Sigma$

Inconvénient : Beaucoup d'espace utilisé pour rien puisque $\forall a \in \Sigma$ qui n'es pas dans m on a $decalage(a) = |m|$.

On utilise donc un dictionnaire dont les clés sont les caractères présents dans m et la valeur associée à $a \in \Sigma$ sera $decalage(a)$.

Complexité :

- Pire des cas : Si $i_0 = i_0 + 1$ à chaque boucle (irréaliste) alors on est en $O((|T| - |m|)|m|)$. Cette analyse n'est pas adaptée, l'amélioration est empirique.
- Meilleur cas : $i_0 = i_0 + |m|$.

  On est en $O\left(\frac{|T|-|m|}{|m|}\right)$.

L'algorithme sera d'autant plus efficace que |m| est grand.

b. Algorithme (version 2 - hors-programme)

Elle prend compte des suffixes dans la table.

II - Compression de Texte

1. Codage de Huffman

Exemple : "AATACGCATAAATA"

On peut s'intéresser à cette séquence en RAM en prenant

A – 01

T – 01

C – 10

G – 11

Espace RAM occupé : 2*14 = $28bits$

Autre codage possible (fonction c)

A – 0

T – 10

C – 110

G – 111

Calcul de l'espace mémoire utilisé : $\sum_{a\in\Sigma}|c(a)|*freq_T(a) = 23bits$

On donne un poids différent entre chaque caractère selon sa fréquence d'apparition.

Décompression : Algorithme glouton. Il fonctionne si on impose la contrainte suivante

$\forall a \in\Sigma, \forall b\in\Sigma, a \space !=b \Longrightarrow c(a)\in\{0,1\}^*$ n'est pas un préfixe de $c(b) \in \{0,1\}^*$

Définition : Soit Σ un alphabet, on appelle codage une fonction injective $c : \Sigma \to \{0,1\}^*$. On dit qu'un codage est admissible si $\forall a \space != b, c(a)$ n'est pas un préfixe de c(b).

Objectif : Etant donné $T \in \Sigma^*$, trouver le **meilleur** codage admissible, c'est-à-dire celui qui minimise la consommation mémoire de T.

On note $c_{mT}(c) = \sum_{i=0}^{|T|-1}| \ c(T[i]) \ | = \sum_{a\in\Sigma}|c(a)|*freq_T(a)$

Où $freq_T(a)$ est le nombre d'occurrences de a dans T.

a. Représentation des Codages

On propose de voir un mot de $\{0,1\}^*$ comme un chemin dans un arbre binaire

- 0 : aller à gauche
- 1 : aller à droite

Pour $a \in \Sigma$, on écrit a comme étiquette du nœud d'arbre c(a).

Ainsi, on visualise les codages comme des arbres.

$c : A \to 00, T \to 01, C \to 10, G \to 11$

Si le codage est admissible dans les symboles $a \in \Sigma$ étiquettent des feuilles sur l'arbre.

Pour minimiser la consommation mémoire, on peut ne considérer que les codages pour lesquels tout nœud interne de l'arbre associé a exactement 2 enfants.

> Remarque : Pour $a \in \Sigma$, on a |c(a)| = profondeur du nœud étiqueté par a dans l'arbre.

Donc faire “remonter” un symbole dans l’arbre a pour effet de réduire |c(a)| sans changer |c(b)| pour $b \in \Sigma\backslash\{a\}$. Donc cela réduit $\sum_{a\in\Sigma}|c(a)|freq_T(a)$

Conclusion : Un codage optimal correspond forcément à une arbre binaire strict. On prendra dans la suite

```
type codage = Leaf of Sigma | Node of codage * codage
```

b. Recherche du codage Optimal

On fixe un texte T.

Propriété 1 : Soit σ une bijection de Σ dans Σ et c un codage admissible. Alors $c \circ \sigma$ est un codage admissible et si c correspond à un arbre binaire strict alors $c \circ \sigma$ aussi.

Propriété 2 : Soit c un codage optimal pour T. Soit $a, b \in \Sigma$ tels que

$freq_T(a) < freq_T(b)$

Alors $|c(a)| \geq |c(b)|$.

Preuve : Si ce n’est pas le cas, on applique au codage la transposition τ_{ab}. Par la première propriété, $c \circ \tau_{ab}$ est un codage admissible donc par optimalité

$c_{mT}(c) \leq c_{mT}(c \circ \tau_{ab})$

et pourtant

$c_{mT}(c \circ \tau_{ab}) = \sum_{d\in\Sigma}|c \circ \tau_{ab}(d)|freq_T(d)$

demander à quelqu’un pour les étapes intermédiaires

$c_{mT}(c \circ \tau_{ab}) = c_{mT}(c)$

$-|c(a)|freq_T(a)$

$-|c(b)|freq_T(b)$

$+|c(h)|freq_T(a)$

$+|c(a)|freq_T(b)$

$=< 0$

Donc absurde

Preuve : Soit c un codage optimal pour T.

Soient a_1, a_2 deux frères dans l'arbre c de profondeur maximale.

Par la propriété 2, $frqu_T(a_1)$ et $ freq_T(a_2)$ sont minimales parmi

$\{freq_T(a) \mid a \in \Sigma\}$

On définit $\Sigma' = \Sigma \setminus \{a_1, a_2\} union \{a'\}$ où a' est un nouveau symbole qui n'appartient pas à Σ.

Et $T' = T$ dans lequel on remplace tous les a_1, a_2 par a'.

$freq_{T'}(a') = freq_T(a_1) + freq_T(a_2)$ et T' est un texte sur Σ' et $freq_{T'}(a) = freq_T(a)$ pour $a \in \Sigma union \Sigma'$

Comme a_1 et a_2 sont frères dans l'arbre, ils s'écrivent $a_1 = u_0$ et $a_2 = u_1$ (ou l'inverse).

On définit $c': \Sigma \longrightarrow \{0,1\}^* \\c'(a) = c(a) \\c'(a') = u$.

Lemme : c est optimal pour T' et Σ'.

Preuve : Supposons par l'absurde que c' n'est pas optimal. Il existe donc c'_{opt} tel qu $c_{mT}(c'_{opt}) < c_{mT'}(c')$.

On va définir c_{opt} un codage pour T et Σ qui sera "mieux" que c.

On définit

$c_{opt} : \Sigma \longrightarrow \{0,1\}^* \\c_{opt}(a) = c_{opt}'(a) \space si \space a \in \Sigma union \Sigma' \\ c_{opt}(a_1) = c_{opt}'(a').0 \\ c_{opt}(a_2) = c_{opt}'(a').1$

$c_{mT}(c_{opt}) = \sum_{d \in \Sigma} |c_{opt}(d)| * freq_T(d)$

$ = c_{mT'}(c_{opt}') - |c_{opt}'(a')| *freq_{T'}(a') + |c_{opt}(a_1)|*freq_T(a_1) + |c_{opt}(a_2)| *freq(a_2)$

demander à quelqu'un pour les étapes intermédiaires

$= c_{mT'}(c_{opt}') + (freq_T(a_1) + freq_T(a_2)) * 1$

De même, $c_{mT}(c) = \sum_{d \in \Sigma} |c(a)| * freq_T(d) = c_{mT}(c') + (freq_T(a_1) + freq_T(a_2))$

Conclusion : $c_{mT}(c_{opt}) = c_{mT'}(c_{opt}{}') + ...$

c. Algorithme Glouton Provisoire

```
codage_optimal(S, T):
1.    calculer freqT(a) pour a dans S
2.    a1, a2 = minimums de freqT(a)
3.    S' = S sans a1 et a2, et avec a'
4.    T' = T dans lequel on remplace tous les a1 et a2 par a'
5.    c' = codage_optimal(S', T')
6.    c = le codage tel que Pour tout a commun à S et S'
7.        c(a) = c'(a)
8.        c(a1) = c(a').0
9.        c(a2) = c(a').1
```

Cas de base :

Si $|\Sigma| = 1$, renvoyer $c(a) = 0$ pour a l'unique symbole de Σ.

Si $|\Sigma| = 2$, renvoyer $c(a) = 0, c(b) = 1$ pour $\Sigma = \{a, b\}$.

Terminaison : variant $|\Sigma|$

Correction : C'est la preuve qui précède, par récurrence sur $|\Sigma|$

Initialisation : $|\Sigma| \in \{1, 2\}$ c'est évident.

Hérédité : Par HR, le $c' = codage\_optimal(\Sigma', T')$ est optimal.

Le c est alors optimal : s'il ne l'est pas, $\exist c_{opt}$ qui est meilleur pour Σ, T et on en déduit $c_{opt}{}'$ meilleur pour Σ', T' que c'.

Complexité

- Calcul des fréquences : $O(|T|)$
- Calcul de a1 et a2 : $O(|\Sigma|)$
- Construction de Σ' : $O(1)$
- Construction dans T' : $O(|T|)$
- Construction de c : $O(|\Sigma|)$

$u_{|\Sigma|,|T|} = O(|T| + |\Sigma|) + u_{|\Sigma|-1,|T|}$

$u_{n,p} = C(n + p) + u_{n-1,p}$

$\Rightarrow u_{n,p} = O(n^2 + np)$

Pour améliorer cette complexité on peut calculer une seule fois les fréquences au début. On peut également utiliser une file de priorité pour les calculs de minimaux. On peut également se passer de la construction de T'. Enfin on peut construire c d'une meilleure façon.

d. Algorithme

```
codage_optimal(S, T)
1.    On calcule les freqT(a) pour tout a de S
2.    In initialise une file de priorité pq qui contient tous les
3.        Leaf(a) pour a de S avec priorité freqT(a)
4.    Faire |S|-1 fois:
5.        a1,p1 = extract_min pq
6.        a2,p2 = extract_min pq
```

```
7.          add pq Node(a1,a2) (p1+p2)
8.      (c1, _) = extract_min pq
9.      return c
```

```
(* Rappel *)
type codage = Leaf of S | Node of codage * codage
```

e. Complexité

On calcule la complexité totale

- Calcul des fréquences : $O(|T|)$
- Initialisation de la file de priorité : $O(|\Sigma|)$
- Boucle : $|\Sigma|$ fois $O(log(|\Sigma|)$

Ce qui donne $O(|T| + |\Sigma| log |\Sigma|)$.

2. Algorithme de Lempel Ziv

a. Introduction

Avantages :

- Algorithme Online (streaming)
- la décompression ne nécessite pas de connaître le codage

Exemple : “ATCATGTATCATGTAA”

On maintient une table $facteur \rightarrow nouveau\_symbole$.

Ici pour simplifier, les nouveaux symboles sont des entiers.

On initialise la table : $A\rightarrow0\\T\rightarrow1\\C\rightarrow2\\G\rightarrow3$

Compression : On ajoute à la table le motif si on ne le connais pas et on incrémente le motif précédent connu.

Décompression : On remonte l’algorithme.

b. Algorithme de Compression

```
def compression(T):
    i = 0
    d = dict([(S[i],i) for i in range(len(S))])
    symbole = n
    while i < len(T):
        Trouver le plus petit j tel que T[i:j] not in d
        d[T[i:j]] = symbole
        symbole += 1
        print(d[T[i:j-1]])
        i = j-1
    return d
```

c. Algorithme de Décompression

```
def decompression(c):
    d = {}
    for i in range(len(S)): d[i] = S[i]
    print(d[c[0]])

    precedent = d[c[0]]
    for i in range(1, len(c))
```

```
        d[output_precedant+d[c[i]][0]]
        print(d[c[i]])
```

Remarque : Il y a plusieurs cas à distinguer, on ne peut pas toujours accéder à d[c[i]].

Invariant à retenir :

Si un facteur u appartient au dictionnaire alors tous ses préfixes aussi.