Mise en œuvre de la recherche de facteur par l'algorithme de Rabin-Karp

Julien Reichert

Algorithme de Rabin-Karp

On note $\mathtt{s}$ le texte, n sa taille, $\mathtt{m}$ le motif et k sa taille.

On considère une fonction de hachage f sur les chaînes de caractères.

- On calcule une fois pour toutes $f(\mathtt{m})$.
- Pour tous les facteurs de $\mathtt{s}$ de taille k, on calcule l'image par f de ce facteur. Si elle est identique à celle de $\mathtt{m}$, on compare le facteur à $\mathtt{m}$ à la manière de l'algorithme naïf. Sinon on passe directement au facteur suivant.
- On pourra calculer les hachés modulo un certain entier p (si f est compatible avec les divisions euclidiennes, c'est mieux).

Algorithme de Rabin-Karp

- Fonctions de hachage classiques : évaluation polynomiale en un entier arbitraire avec les codes des caractères pour coefficients.
- Problème : dans quelle mesure le calcul de l'image par f serait-il plus rapide que la comparaison des caractères ?
- Solution : calcul en temps constant et indépendant de k du haché du facteur suivant une fois le premier haché disponible ($\rightarrow$ Horner adapté... d'ailleurs, qui se souvient de Horner ?).
- La complexité en temps sera encore $\mathcal{O}(nk)$ si les hachés des facteurs sont trop souvent égaux à celui de `m`. Mais en moyenne cela va plus vite.

Explication sur un exemple

Soient `s = "CUAGCAGUCAUGCAG"` et `m = "AUG"`.

Fonction de hachage choisie pour une chaîne `x` de taille trois, dont les codes des caractères sont notés x_0, x_1 et x_2 :

$$x_0 \cdot 97^2 + x_1 \cdot 97 + x_2.$$

La fonction qui réalise ce calcul est intuitive.

Explication sur un exemple

La fonction précédente est compatible avec les restes dans la division euclidienne par un certain p.

Avantage des divisions euclidiennes : on évite les problèmes de dépassement en pratique, quitte à avoir plus souvent des égalités entre hachés.

Explication sur un exemple

Passage du haché du facteur $s_i s_{i+1} s_{i+2}$ à celui du facteur $s_{i+1} s_{i+2} s_{i+3}$ (en assimilant un caractère à son code), soit de $s_i \cdot 97^2 + s_{i+1} \cdot 97 + s_{i+2}$ à $s_{i+1} \cdot 97^2 + s_{i+2} \cdot 97 + s_{i+3}$:

- Retirer $s_i \cdot 97^2$ (valeur de 97^2 stockée une fois pour toutes)
- Multiplier par 97
- Ajouter s_{i+3}

On peut considérer que 97 est remplacé par `b`, globale ou argument, au choix.

On propose le pseudo-code ci-après pour la recherche de la première occurrence de `m` en tant que facteur.

L'algorithme

```
fonction premiere_correspondance(s, m) {
  n <- taille(s); k <- taille(m);
  gros <- b ** k modulo p;
  hm <- hache(m); hs <- hache(s[:k]); // slice à la Python
  pour i de 0 à n - k {
    si hs = hm {
      j <- 0;
      tant que j < k et s[i+j] = m[j] {
        incrémenter j;
      }
      si j = k alors renvoyer i;
    }
    hs <- hache_suivant(...); // Trouvez les arguments !
  }
  erreur("Pas de correspondance")
}
```

Mise en œuvre

Préparation pour l'instance actuelle :

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`n = 15` et `k = 3`, d'où 13 tours de boucle.

`p = 19`

`hm = 7` (ASCII et évaluation en 97 puis reste modulo 19)

97^2 modulo 19 vaut 4, mémorisé une fois pour toutes

Pour information, les codes rencontrés : 65 pour `A`, 67 pour `C`, 71 pour `G` et 85 pour `U`.

Mise en œuvre

s = "CUAGCAGUCAUGCAG"

m = "AUG"

hm = 7

hs = 9 (calculé initialement)

i = 0

Hachés différents : on passe au facteur suivant.

hs devient ((9 - 67 * 4) * 97 + 71) mod 19, soit 9.

Mise en œuvre

s = "CUAGCAGUCAUGCAG"

m = "AUG"

hm = 7

hs = 9

i = 1

Hachés différents : on passe au facteur suivant.

hs devient ((9 - 85 * 4) * 97 + 67) mod 19, soit 13.

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 13`

`i = 2`

Hachés différents : on passe au facteur suivant.

`hs` devient `((13 - 65 * 4) * 97 + 65) mod 19`, soit 8.

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 8`

`i = 3`

Hachés différents : on passe au facteur suivant.

`hs` devient `((8 - 71 * 4) * 97 + 71) mod 19`, soit 13.

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 13`

`i = 4`

Hachés différents : on passe au facteur suivant.

`hs` devient `((13 - 67 * 4) * 97 + 85) mod 19`, soit 12.

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 12`

`i = 5`

Hachés différents : on passe au facteur suivant.

`hs` devient `((12 - 65 * 4) * 97 + 67) mod 19`, soit 8.

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 8`

`i = 6`

Hachés différents : on passe au facteur suivant.

`hs` devient `((8 - 71 * 4) * 97 + 65) mod 19`, soit 7. **Ah !**

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 7`

`i = 7, j = 0`

Caractères différents : fin de la boucle while, facteur suivant.

`hs` devient `((7 - 85 * 4) * 97 + 85) mod 19`, soit 8.

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 8`

`i = 8`

Hachés différents : on passe au facteur suivant.

`hs` devient `((8 - 67 * 4) * 97 + 71) mod 19`, soit 7. **Ouf !**

Mise en œuvre

`s = "CUAGCAGUCAUGCAG"`

`m = "AUG"`

`hm = 7`

`hs = 7`

`i = 9, j = 0`

Caractères identiques, la boucle while continue.

Spoiler alert : `m` sera reconnu et la fonction s'arrêtera et renverra 9.

Bilan

Nombre de calculs effectués :

- 2 hachés calculés intégralement (coût dans le cas général linéaire en k)
- 1 exponentiation (et un modulo)
- 9 hachés calculés en une fois par quatre opérations
- Le nombre de tours de boucle et tout ce qui tourne autour des structures de contrôle.

Nombre de comparaisons effectuées :

- 10 entre des entiers modulo 19
- 4 entre des caractères.

Bilan

L'algorithme naïf aurait fait 14 comparaisons de caractères, ce n'est pas tellement plus.

Gros avantage non exploité ici : on pouvait chercher plusieurs facteurs de même taille à la fois (utiliser un dictionnaire pour les hachés des facteurs cherchés).

C'est essentiellement à ce titre que l'algorithme de Rabin-Karp sera utilisé en pratique.