Table de hachage : implémentation

INFORMATIQUE COMMUNE - TP n° 3.1 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ utiliser les listes Python
- ☞ écrire des fonctions en Python
- ☞ utiliser une bibliothèque en l'important correctement
- ☞ expliquer le fonctionnement d'une table de hachage (dictionnaire)

L'objectif de ce TP est de construire une table de hachage «à la main», un équivalent des `dict` Python. Dans ce but, il faut dans un premier temps disposer d'une fonction de hachage adaptée. C'est l'objet de la première partie. La seconde partie se focaliser sur l'implémentation de la table de hachage.

On rappelle sur la figure 1 le principe du dictionnaire (ou table de hachage).

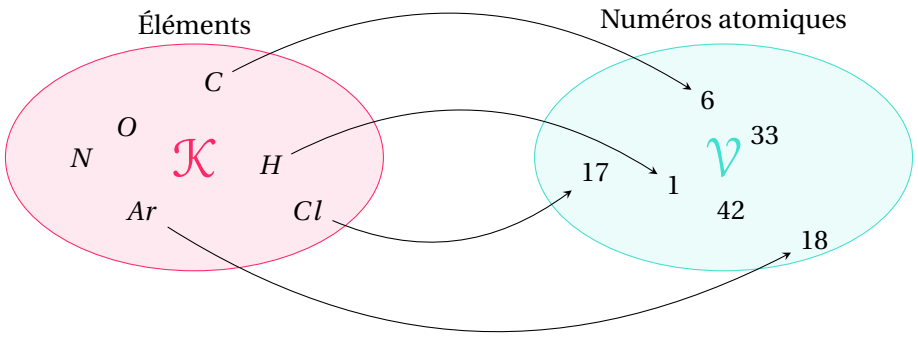

FIGURE 1 – Illustration du concept de dictionnaire : tableau associatif reliant une clef à un numéro atomique.

A Fonctions de hachage et uniformité

R On rappelle que le choix d'une fonction de hachage est délicat et qu'il n'existe pas de méthode pour atteindre l'optimal.

Lorsque les clefs d'une table de hachage sont des chaînes de caractères, il est souvent possible de décomposer une fonction de hachage h en deux étapes :

1. une fonction h_e qui encode la clef d'entrée en un nombre entier (encodage),

2. et une fonction h_c qui compresse ce nombre entier dans l'ensemble des indexes (compression).

Plus formellement, on la fonction de hachage comme une fonction composée :

$$h_e : \mathcal{K} \longrightarrow \mathbb{N} \tag{1}$$
$$h_c : \mathbb{N} \longrightarrow [\![0, n-1]\!] \tag{2}$$

et

$$h = h_c \circ h_e \tag{3}$$

a Encodage

L'idée de l'encodage est de générer un nombre représentant une chaîne de caractères. Deux approches sont considérées :

γ Cette fonction calcule un entier unique pour chaque chaîne de caractères comme suit :

$$\gamma(s) = \sum_{k=0}^{|s|-1} \text{ascii}(s_k) 2^{8k} \tag{4}$$

où $\text{ascii}(s_k)$ est le code ASCII associé au caractère d'indice k de s.

γ_p On dispose des 100 premiers nombres premiers sous la forme d'une variable globale :

```
PRIMES = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
    67, 71, 73, 79, 83, 89, 97, 101, 103,  107, 109, 113, 127, 131, 137, 139,
    149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227,
    229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311,
    313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
    409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
    499, 503, 509, 521, 523, 541]
```

Cette fonction procède comme suit :

$$\gamma_p(s) = \sum_{k=0}^{|s|-1} \text{PRIMES}[(\text{ascii}(s_k) + k) \bmod 100] \tag{5}$$

(P) En Python, la fonction `ord` permet d'obtenir le code ASCII associé à un caractère documentation).

A1. À quoi sert le facteur multiplicatif 2^{8k} dans la formule de γ ? Quelle qualité mathématique confère-t-il à la fonction γ ?

> **Solution :** À décaler de 8 bits vers la gauche les nombres calculés. Cela garantit qu'il n'y aura aucun jumeau : deux chaînes de caractères distinctes auront toujours des codes différents. La fonction γ est donc injective.
>
> Comme les caractères ASCII sont codés sur huit bits au maximum et que la fonction γ décale de `k*8` bits vers la gauche chaque valeur ascii s_k, alors le nombre obtenu dépend des codes des lettres $ascii(s_k)$et de leur position dans le mot (k) : les octets associés à chaque caractères ne se recoupent pas. Les mots proches comportant les mêmes lettres mais pas dans le même ordre ne produisent donc pas le même code et les codes sont mêmes relativement distants les uns des autres :

```
print(gamma("abaa"), gamma("aaba"), gamma("aaab"), gamma("baaa"))
#1633772129 1633837409 1650549089 1633771874
print(gamma("choir"), gamma("music"), gamma("piano"), gamma("song"))
#491395180643 426970936685 478593247600 1735290739
```

A2. Programmer une fonction `gamma(s : str): int` qui calcule $\gamma(s)$.

Solution :

```
def gamma(s):
    g = 0
    for k in range(len(s)):
        g += ord(s[k]) << (k * 8)
    return g
```

A3. La fonction γ_p engendre-t-elle des codes tous différents ? Pourquoi ? Que peut-on en conclure sur la fonction γ_p ?

Solution : Non, car il se peut que $\text{ascii}(s_k) + k$ soit identique pour deux caractères différents placés différemment sur deux chaînes différentes.

A4. Programmer une fonction `gammap(s : str): int` qui calcule $\gamma_p(s)$.

Solution :

```
def gammap(s):
    ind = 0
    sprimes = len(PRIMES)
    for i, c in enumerate(s):
        ind += PRIMES[(i + ord(c)) % sprimes]
    return ind
```

b Compression

Dans un second temps, on cherche à **compresser la valeur encodée dans l'intervalle des index possibles** $[\![0, n-1]\!]$, si n est la taille de la table de hachage.

Une distribution uniforme des clefs dans l'espace d'arrivée peut être obtenue en utilisant des générateurs aléatoires. Les générateurs à congruence linéaires, c'est à dire les fonctions du type $(ax + b) \mod n$ sont de bons candidats pour les fonctions de hachage, pourvu qu'on choisisse bien les constantes a et b du générateur.

On peut choisir :

1. d'utiliser simplement le reste d'une division :

$$h_d : (s, n) \to \gamma(s) \mod n \tag{6}$$

2. d'utiliser une extraction de partie fractionnaire et une multiplication :

$$h_\alpha : (s, n) \to \lfloor n \times \{\alpha\gamma(s)\} \rfloor \tag{7}$$

$\alpha \in]0,1[$ étant une constante réelle et $\{\}$ est l'opérateur qui renvoie la partie fractionnaire d'un nombre flottant. Par exemple, $\{42.567\} = 0.567$. Cette opération peut être réalisée par l'opération `%1` en Python.

A5. Coder les fonctions h_d et h_α en Python. Les paramètres n et α pourront être pris par défaut à 47057 et $\frac{\sqrt{5}-1}{2}$.

Solution :

```
import math
TABLE_SIZE = 47057
ALPHA = (math.sqrt(5) - 1) / 2
PRIMES = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
    61, 67, 71, 73, 79, 83, 89, 97, 101, 103,  107, 109, 113, 127, 131, 137,
     139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
    223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
    293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
    383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
    463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541]

def gamma(s):
    g = 0
    for k in range(len(s)):
        g += ord(s[k]) << (k * 8)
    return g

def gammap(s):
    ind = 0
    sprimes = len(PRIMES)
    for i, c in enumerate(s):
        ind += PRIMES[(i + ord(c)) % sprimes]
    return ind

def hd(key, table_size=TABLE_SIZE):
  return gamma(key) % table_size

def hm(key, table_size=TABLE_SIZE, alpha=ALPHA):
  return math.floor(table_size * (alpha * gamma(key) % 1))
```

A6. Importer tous les mots contenus dans le fichier `"english_words.csv"` dans un liste.

On cherche à tester l'uniformité de la distribution des codes obtenues des fonctions de hachage. On peut facilement vérifier ceci en utilisant le test de Kolmogorov-Smirnov et la bibliothèque scipy et l'instruction :

```
scipy.stats.kstest(codes, "uniform")
#KstestResult(statistic=0.0012179563749926126, pvalue=0.49280753163611735)
```

Si le paramètre `p_value` est plus grand que 0.05, alors la distribution peut être considérée comme uniforme. Le paramètre `statistic` donne une mesure de la distance entre les deux distributions.

A7. Écrire une fonction dont le prototype est `uniform_test(h, table_size)` dont le paramètre `h` est une fonction de hachage et `table_size` la taille de la table de hachage. Cette fonction renvoie le résultat du test de Kolmogorov-Smirnov entre une distribution uniforme et la distribution des codes obtenus avec `h` sur l'ensemble des mots du fichier `"english_words"`. La fonction de `scipy` nécessite un tableau d'entrée Numpy dont les données sont de type `float`.

Solution :

```
def uniformity_test(h, table_size):
  codes = []
  f = open("english_words.txt", "r")
  for line in f:
    words = line.split('\n')
    hash_code = h(words[0], table_size)
    codes.append(hash_code)
  f.close()
  codes = np.array(codes, dtype=float)
  codes = codes / np.max(codes)
  return scipy.stats.kstest(codes, "uniform")
```

A8. Observer les résultats de la fonction précédente pour h_d et h_α en utilisant soit γ soit γ_p et faisant varier la taille de la table de hachage. Que pouvez-vous en conclure?

Solution : La fonction h_d ne fonctionne pas avec γ_p car le nombre entier obtenu est trop petit par rapport à la taille de la table. Toutes les valeurs sont projetées dans le début de la table. Par contre, elle peut éventuellement fonctionner avec γ si la taille de la table de hachage est un nombre premier loin d'une puissance de 2.

La fonction h_α ne fonctionne pas avec γ car les nombres produits sont trop grands. La raison est qu'en multipliant par α de grands nombres entiers, la partie fractionnaire obtenue par l'opération `%1` peut être nulle, à cause de la différence de plage d'exposant des deux nombres flottants α et $\gamma(s)$. Avec les flottants, l'erreur absolue $\epsilon_a = v - \overline{v}$ dépend de la plage des exposants, la précision limitée. Un grumeau se forme dans la table à l'indice 0.

Par contre, la fonction h_α fonctionne correctement avec γ_p. Ce qui montre que l'encodage n'a pas à être injectif pour obtenir une fonction de hachage efficace.

```
def uniformity_test(h, table_size):
  codes = []
  f = open("english_words.txt", "r")
  for line in f:
    words = line.split('\n')
    hash_code = h(words[0], table_size)
    codes.append(hash_code)
    f.close()
  codes = np.array(codes, dtype=float)
  codes = codes / np.max(codes)
  return scipy.stats.kstest(codes, "uniform")
```

A9. Afficher les histogrammes associés aux différentes distribution de codes obtenues à l'aide de la bibliothèque `matplotlib` et à la fonction `hist`.

Solution :

```
def plot_hist(h):
  codes = []
  f = open("english_words.txt", "r")
  for line in f:
    words = line.split('\n')
    hash_code = h(words[0], TABLE_SIZE)
    codes.append(hash_code)
  f.close()
  codes = np.array(codes, dtype=float)
  codes = codes / np.max(codes)
  plt.hist(codes, 50)# range=(np.min(codes), np.max(codes)))
  plt.title("Codes Distribution "+h.__name__)
  plt.show()
```

B Implémentation d'une table de hachage

On souhaite créer une table de hachage d'après un fichier qui recense les capitales des pays du monde entier. Cette table possède donc des clefs de type `str` (le pays) et des valeurs de type `str` (la capitale).

B1. Écrire une fonction `import_csv()` qui importe les données du fichier `"capitals.csv"`. Cette fonction renvoie une liste de tuples `(pays, capitale)`.

Solution :

```
def import_csv(filename):
  # conventions :
  # -- data are strings
  with open(filename, "r") as f:
    data = []
    headers = f.readline().split(",")
    for line in f:
      words = line.split(',')
      data.append((words[0], words[1]))  # (country,capital)
  return data
```

B2. Écrire une fonction de prototype `init_hash_table(elements,table_size)` qui renvoie une table de hachage initialisée d'après le paramètre `elements`. Ce paramètre est la liste de tuples créée à la question précédente.

Solution :

```
def init_hash_table(elements, table_size):
  hashtable = [[] for i in range(table_size)]
  for key, value in elements:
    index = hd(key)
    hashtable[index].append((key, value))
```

```
    return hashtable
```

B3. Écrire une fonction de prototype `get_value(table, table_size, input_key)` qui permet d'accéder à l'élément de clef `input_key` de la table de hachage `table`. Par exemple, `get_value(ht, "Italy"), n` renvoie `"Roma"`.

Solution :

```
def get_value(table, table_size, input_key):
  index = hd(input_key)
  element = table[index]
  for key, value in element:
    if key == input_key:
      return value
  return None
```

B4. Créer l'ensemble de toutes les clefs de la table pour lesquelles il existe une valeur, puis parcourir la table à partir de cet ensemble. Les capitales apparaissent-elles dans un ordre quelconque ? Faire apparaître les sous-listes de la table s'il y en a. Combien y-a-t-il de clefs si on utilise h_d ? Même question si on utilise la fonction interne de Python :

```
def hashp(s, table_size=TABLE_SIZE):
  return hash(s) % table_size
```

Solution : L'ordre est quelconque puisque la table de hachage est répartie uniformément à partir de codes pseudo-aléatoires. Avec h_d, comme les capitales possèdent souvent plus de cinq lettres, on obtient 246 clefs, ce qui est presque injectif (248 capitales). On ne doit parcourir que deux sous listes. Avec la fonction interne de Python, on trouve 247 clefs ! En fait, comme la Bolivie a deux capitales, il est normal d'avoir au moins une sous-liste.

```
data = import_csv("capitals.csv")
print(len(data), " capitales", data)
ht = init_hash_table(data, gammap, TABLE_SIZE)
print(ht)

print("Access to capital of Italy --> ", get_value(ht, gammap, TABLE_SIZE, "
    Italy"))
print("Access to capital of France --> ", get_value(ht, gammap, TABLE_SIZE,
    "France"))

keys = []
jumeaux = 0
for i, e in enumerate(ht):
    if len(e) > 1:
        print("Jumeaux --> ", e)
        jumeaux += 1
    if e:
        keys.append(i)
```

```
    # print(len(keys), keys)

    print("#capilates --> ", len(data))
    print("#keys --> ", len(keys))
    print("#jumeaux --> ", jumeaux)
    print("Facteur de remplissage --> ", len(keys) / TABLE_SIZE)
```

R Naturellement, si par la suite vous avez besoin d'une table de hachage, il faut utiliser le type `dict` de Python et ne pas réinventer la poudre !