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

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.

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

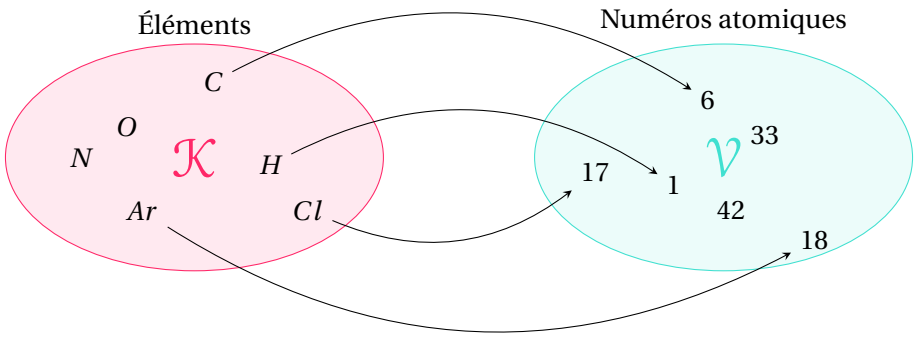

FIGURE 1 – Illustration du concept de dictionnaire : tableau associatif reliant une clef à un numéro atomique.

A Fonctions de hachage et uniformité

Pour être adaptée à l'usage par une table de hachage, une fonction de hachage devrait être :

1. rapide,
2. cohérente : pour une même clef, on obtient un même code,
3. injective : pour des clefs différentes, on obtient des codes différents. Pour des codes identiques, les clefs sont nécessairement identiques. Dans le cas contraire, on obtient une **collision** qu'on cherche

à minimiser. Comme la table de hachage sera de dimension finie, les collisions sont inévitables. Donc l'injectivité sera sacrifiée.

4. uniformément répartie : pour des clefs qui se ressemblent, les codes obtenus doivent être très différents, ceci pour limiter les collisions.

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.

Dans un premier temps, on opère l'**encodage de la chaîne de caractère en nombre entier** $\gamma(s)$ pour chaque chaîne de caractère s de la manière suivante :

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

où $\text{ascii}(s_k)$ est le code ASCII associé au caractère d'indice k de s. Cette étape est importante, car elle permet déjà de générer des codes souvent différents pour des chaînes très similaires.

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

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, on peut choisir :

1. d'utiliser simplement une division :

$$h_d : (s, n) \to \gamma(s) \mod n \tag{2}$$

2. d'utiliser une multiplication et une division :

$$h_\alpha : (s, n) \to \lfloor n \times (\alpha\gamma(s) \mod 1) \rfloor \tag{3}$$

$\alpha \in]0,1[$ étant une constante réelle.

Le choix d'une fonction de hachage est délicat et il n'existe pas de méthode pour atteindre l'optimal.

A1. La fonction γ est-elle injective ? Expliquer pourquoi la fonction γ renvoie un nombre unique associé à une chaîne de caractères.

Solution : 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 le codes sont mêmes 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. Coder les fonctions γ, h_d et h_α en Python. Les paramètres n et α pourront être pris par défaut à 47057 et $\frac{\sqrt{5}-1}{2}$. Ne pas oublier également que les puissances de deux peuvent être facilement calculées grâce aux opérateurs de décalage binaire.

Solution :

```python
import math
TABLE_SIZE = 47057
ALPHA = (math.sqrt(5) - 1) / 2


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


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))
```

A3. 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 :

```python
    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.

A4. 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 :

```python
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")
```

A5. Observer les résultats de la fonction précédente pour h_d et h_α en faisant varier la taille de la table de hachage. Que pouvez-vous en conclure ?

Solution : La fonction h_d fonctionne :

- si la clef est d'une certaine taille (typiquement > 5 caractères), sinon, pour de courtes chaînes de caractères, le modulo TABLE_SIZE ne garantit pas l'uniformité puisqu'il est inopérant, le code $\gamma(s)$ étant plus petit que TABLE_SIZE. Des grumeaux se forment dans la table.
- si la taille de la table de hachage est un nombre premier loin d'une puissance de 2.

La fonction h_α n'est sensible pas sensible au choix de la taille de la table de hachage mais ne fonctionne que pour des clefs de tailles <7. La raison est qu'en multipliant par α de grands nombre flottants (les mots de 7 lettres en produisent), la partie décimale 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.

```
def uniformity_test(h, table_size):
  codes = []
  f = open("english_words.txt", "r")
  for line in f:
    words = line.split('\n')
    if len(words[0]) > 5:  # On joue avec la taille des clefs essayer
        aussi < 5 ou > 7
      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")
```

A6. 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(data)
ht = init_hash_table(data, TABLE_SIZE)
print(ht)

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

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

print("#keys --> ", len(keys))

for k in keys:
print(k, ht[k])
```

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 !