TP sur les arbres (2): le codage de Huffman

Stéphane Grandcolas, Valentin Emiya, François Brucker - Aix-Marseille Université

L2 MPCI - Novembre 2024

Le codage de Huffman est une technique de compression de données sans perte. Imaginons par exemple que l'on souhaite coder la chaîne de caractères

```
s = "abbrrrrrrrracccaddabbraddabraaaaaaaaaaaaaaaaaaaaa"
```

Une première solution est d'utiliser une norme telle qu'ASCII ou UTF-8 qui code chacun de ces caractères sur un octet, soit 8 bits, selon une table de codes fixée. Cette chaîne comprenant 49 caractères, elle occuperait alors 49 octets en mémoire, soit 392 bits. C'est la taille du fichier `abrr.txt` fourni. Pour réduire l'espace de stockage ou le débit de transmission, une façon plus économique de coder cette chaîne consiste à adapter les codes de chaque caractère, en utilisant moins de bits pour les caractères les plus fréquents. Par exemple, en français, le `'e'` serait codé avec moins de bits que le `'w'`. Dans le cas ci-dessus, l'algorithme de codage de Huffman calcule le nombre d'occurrences de chaque caractère pour construire le tableau de correspondance suivant :

caractère	`'a'`	`'r'`	`'b'`	`'d'`	`'c'`
nombre d'occurrences	27	10	5	4	3
code binaire	`1`	`00`	`010`	`0111`	`0110`

La chaîne de caractères est alors codée par la séquence de 90 bits (compression de 77%) :

```
101001000000000000000001011001100110101110111101001000101110111101000111111111111111111111
```

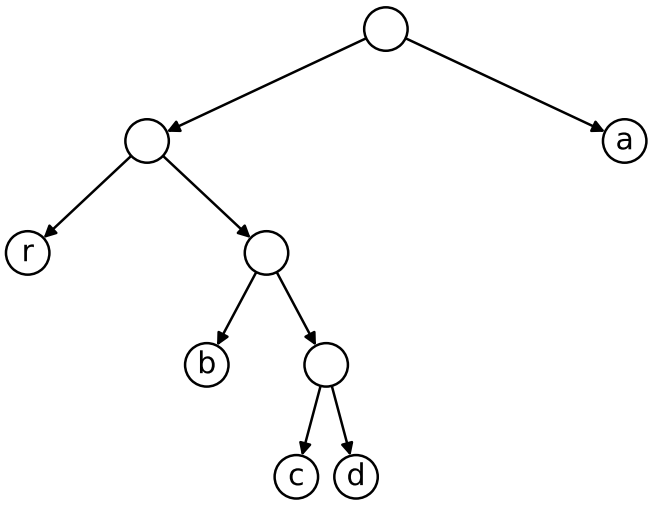

FIGURE 1 – Arbre issu de l'algorithme de Huffmann

La structure de donnée sous-jacente est un arbre binaire tel que représenté sur la figure 1 : les feuilles sont les symboles à coder ; le code d'un caractère correspond à une descente depuis la racine en ajoutant le bit 0 (resp. 1) lorsque l'on sélectionne un fils gauche (resp. droit).

Le codage de Huffman peut s'appliquer à n'importe quelle suite de symboles. On peut par exemple construire un ensemble de codes adapté à la langue française à partir d'un grand corpus de textes. Les principales tâches à traiter sont les suivantes :

— construction des codes : comment construire les codes adaptés à un texte donné, c'est-à-dire permettant de le compresser ?

— encodage : comment encoder un texte lorsque les codes sont connus ?

— décodage : comment décoder une suite de bits lorsque les codes sont connus ?

Instructions générales. Complétez le fichier `huffman.py` fourni en traitant les exercices ci-dessous et en utilisant l'exemple simple précédent pour corriger vos bugs. Ajoutez un test pour chaque fonction ou méthode dans un fichier `test_huffman.py`. Vous utiliserez votre classe `BinaryTree` pour gérer les arbres de codage.

Exercice 1 (Construction des codes (1/2) : compter le nombre d'occurrences des caractères)**.** Une étape préliminaire à la construction des codes consiste à compter le nombre d'occurrences de chaque caractère. complétez la fonction `count_symbols(s)` qui prend en argument une chaîne de caractères `s`, compte le nombre d'occurrences de chaque caractère, et renvoie un dictionnaire dont les clés sont les caractères et les valeurs leur nombre d'occurrences. Dans le cas de l'exemple ci-dessus, on obtient le dictionnaire

```
{'a': 27, 'b': 5, 'r': 10, 'c': 3, 'd': 4}
```

Exercice 2 (Construction des codes (2/2) : construire l'arbre)**.** L'algorithme de construction de l'arbre et des codes est le suivant :

Algorithm 1 Algorithme de construction des codes de Huffman

Require: ensemble de symboles à coder avec leur nombre d'occurrences
Ensure: arbre de codage

1: créer une feuille pour chaque caractère
2: créer une file de priorité contenant initialement les feuilles avec leur nombre d'occurrences comme poids
3: **while** la file de priorité contient au moins deux éléments **do**
4: extraire le nœud u de poids minimal n_u
5: extraire le nœud v de poids minimal n_v
6: créer un nœud dont le fils gauche est u et le fils droit v
7: ajouter ce nœud à la file de priorité avec le poids $n_u + n_v$
8: **end while**
9: renvoyer l'arbre dont la racine est le nœud restant dans la file de priorité

Il s'agit d'un algorithme glouton qui donne un résultat optimal pour compresser un texte. À la main, déroulez l'algorithme sur l'exemple précédent.

Complétez la fonction `build_tree(symbol_count)` qui prend en argument un dictionnaire tel que renvoyé par la fonction `count_symbols` et construit l'arbre en appliquant l'algorithme ci-dessus. Vous utiliserez votre classe `BinaryTree` pour construire l'arbre et la bibliothèque `heapq`[1] pour gérer la file de priorité via un tas min. Une file de priorité est construite à partir

1. `https://docs.python.org/3/library/heapq.html`

d'une liste ordinaire `heap`, initialement vide (`heap=[]`) : on ajoute un élément `e` via la fonction `heapq.heappush(heap, e)` et on extrait le minimum avec la fonction `heapq.heappop(heap)`. Les éléments dans la file peuvent être des tuples de la forme `(w, x)` où `w` est le poids à considérer et `x` est l'objet associé à ce poids (dans notre cas, `x` est donc un `BinaryTree`).

Dans la classe `HuffmanCoding`, codez la méthode statique `from_string(s)` qui prend en argument une chaîne de caractère, construit l'arbre de codage puis renvoie un objet de type `HuffmanCoding` construit avec cet arbre.

Exercice 3 (Cas particulier)**.** Cette question est facultative dans un premier temps, n'hésitez pas à la faire à la fin. Testez la fonction `build_tree(symbol_count)` avec lorsque deux symboles ont le même nombre d'occurrences. Vous devriez rencontrer une erreur. Il faut gérer le cas particulier où des éléments `(w, x)` et `(w, y)` ont le même poids `w` : le minimum est alors déterminé en essayant de comparer les objets suivants dans le tuple, c'est-à-dire `x` et `y`. Hors si ces objets ne sont pas comparables, une erreur est alors générée. Ce problème se pose dans notre cas puisque les objets sont des arbres. Deux solutions existent, au choix :

— la première solution consiste à rendre les arbres comparables, par exemple en ajoutant à la classe `BinaryTree` la méthode spéciale `__lt__(self, other)` qui renvoie le résultat de la comparaison de l'arbre courant `self` et d'un autre arbre `other` ; une fois cette méthode définie, on peut utiliser l'opérateur < pour comparer deux instances de `BinaryTree` ; dans le cas présent où le résultat de la comparaison importe peu, vous pouvez par exemple comparer les adresses en mémoire des deux arbres via la fonction `id(obj)` qui renvoie l'adresse d'un objet `obj`.

— la seconde solution consiste à utiliser des tuples de taille trois `(w, n, x)` dans le tas en ajoutant un compteur `n` différent pour chaque nouvel élément : si deux éléments ont le même poids, leurs compteurs permettent de les comparer sans comparer les objets en dernière position dans le tuple [2].

Exercice 4 (Encodage)**.** Pour encoder un texte, c'est-à-dire pour passer d'une suite de caractères à une suite de bits, on parcourt chaque caractère et on le remplace par son code. Cette association caractère → code peut se faire facilement via un dictionnaire dont les clés sont les caractères et les valeurs sont les codes. Un code est ici stocké comme une chaîne de caractères contenant des 0 et des 1. Dans le cas de l'exemple ci-dessus, on obtient le dictionnaire :

```
{'r': '00', 'b': '010', 'c': '0110', 'd': '0111', 'a': '1'}
```

Codez la fonction `build_huffman_dict(huffman_tree)` qui prend en argument un arbre tel que fourni par `build_tree`, construit et renvoie le dictionnaire d'encodage associé. Cette fonction est appelée dans le constructeur de la classe `HuffmanCoding` qui stocke le dictionaire dans l'attribut `huffman_dict`

Dans la classe `HuffmanCoding`, codez la méthode `encode(self, txt)` qui prend en argument un texte et renvoie le code correspondant, sous la forme d'une chaîne de caractères contenant des 0 et des 1.

Exercice 5 (Décodage)**.** Le décodage consiste à parcourir une suite de bits codant un texte et à la transformer en suite de caractères pour retrouver le texte original. La procédure repose sur l'utilisation de l'arbre construit à l'exercice 2. En partant de la racine et en parcourant la suite de bits, on effectue des descentes successives dans l'arbre selon les cas suivants :

— si l'on est dans une feuille, on émet le caractère stocké dans la feuille et on se déplace à la racine de l'arbre et on passe au bit suivant ;

2. Voir `https://docs.python.org/3/library/heapq.html#priority-queue-implementation-notes`

— sinon, on se déplace à gauche si le symbole courant est un 0, à droite si c'est un 1, puis on passe au bit suivant.

Codez la méthode `decode(self, bits)` qui prend en argument une suite de bits sous forme de chaîne de caractères, et renvoie la chaîne de caractères correspondant au texte décodé, en utilisant l'arbre stocké en attribut dans l'instance courante.

Exercice 6 (Jeux de données et expériences)**.** Plusieurs fichiers `.txt` contenant des suites de caractères sont fournis pour tester votre code. Vous pouvez les utiliser pour construire les codes et/ou pour encoder des chaînes de caractères. Vous avez en particulier :

— `abrr.txt` (49 caractères) : l'exemple court donné au début de ce sujet, principalement pour vous aider dans le développement de votre code ;

— `Deux beaux présents - Olivier Salon et Michèle Audin.txt` (2551 caractères) : un texte en français, vous pourrez chercher ce qu'est un beau présent du côté de l'Oulipo.

— `asyoulik.txt` (125179 caractères) : un extrait de Shakespeare !

— `pi.txt` (10^6 caractères) : le premier million de décimales du nombre π.

Pour chaque exemple, vous pouvez réaliser cette expérience :

— lisez le fichier pour obtenir la chaîne de caractères à coder via la fonction `read_txtfile` fournie ;

— construisez l'arbre à partir de cette chaîne ; affichez ses caractéristiques telles que nombre de feuilles et hauteur ;

— encodez la chaîne et estimez le taux de compression, en fonction de la longueur des chaînes et du nombre de bits pour chaque caractère selon les cas ; interprétez les résultats obtenus ;

— décodez la chaîne et vérifiez que vous retrouvez la chaîne originale.

Remarque : pour certains fichiers, il y a des symboles qui ont le même nombre d'occurrences, ce qui peut générer une erreur

```
TypeError: '<' not supported between instances of 'BinaryTree' and 'BinaryTree'
```

si vous n'avez pas traité l'exercice 3.

Exercice 7 (Pour aller plus loin)**.** Vous pouvez ensuite, au choix :

— chercher à convertir la chaîne de caractères 0 et 1 en suite de bits et les stocker dans un fichier binaire puis comparer les tailles des fichiers pour voir si le taux de compression estimé est le même en pratique.

— essayer d'utiliser les codes appris sur un fichier pour coder le texte d'un autre fichier pour répondre à une question du type « dans quelle mesure des codes adaptés au français sont-il efficaces sur de l'anglais ? » ; cela peut nécessiter d'enrichir le premier texte avec les caractères qui n'apparaissent que dans le second (par exemple en les ajoutant une fois à la fin) ; comparez ensuite les gains de compression lorsque l'on utilise les codes adaptés au fichier ou les codes issus d'un autre fichier.

— enrichir votre jeu de données avec d'autres textes.

— réfléchir à l'unicité de la solution (et à la preuve associée) : deux séquences de symboles différentes peuvent-elles être encodées par la même séquence binaire ?

— réfléchir à l'existence d'une solution (et à la preuve associée) : toute séquence binaire correspond-elle à une séquence de symboles ou bien y a-t-il des séquences binaires qui ne peuvent pas être décodées ?

— imaginer des principes de compression qui peuvent donner des résultats encore meilleurs.