

INFORMATIQUE

2024 - 2025

Algorithmes sur les Graphes et Combinatoire

Laurent PHILIPPE

SUP^F_C

Algorithmique sur les Graphes et Combinatoire

MASTERS INFORMATIQUE DVL / I2A / ISL
1ÈRE ANNÉE

Chapitre IV - Exercices - Programmation gloutonne



Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

1 Problème du codage de Huffman

Le codage de Huffman est une technique de compression d'informations très utile. Elle permet de gagner entre 20% et 90% selon les cas. Il s'agit d'un algorithme basé sur les fréquences d'apparition de chaque caractère dans un fichier. L'exemple suivant (tableau 1) donne deux codages de Huffman différents, le codage fixe et le codage variable, pour les lettres a, b, c, d, e et f. Pour 100 000 caractères, le codage fixe (3 bits par caractère) du fichier occupe 300 000 bits alors que le codage variable (les lettres sont codées avec un nombre variable de bits) du fichier occupe 224 000 bits, soit une économie de 25%. Il s'agit là d'un codage optimal dont la découverte peut être faite grâce à un algorithme glouton.

| type de codage/lettre | a | b | c | d | e | f |
|-----------------------------|-------|-------|-------|-------|------|------|
| occurrence d'apparition | 45000 | 13000 | 12000 | 16000 | 9000 | 5000 |
| fréquences d'apparition (%) | 45 | 13 | 12 | 16 | 9 | 5 |
| longueur fixe | 000 | 001 | 010 | 011 | 100 | 101 |
| longueur variable | 0 | 101 | 100 | 111 | 1101 | 1100 |

TABLE 1 – Exemple de fréquences d'apparition des lettres dans un fichier et codages associés

Exercice 1 : Codage fixe

Le codage fixe consiste en la construction d'un arbre binaire dont les feuilles sont les lettres. Cet arbre sert au décodage du fichier codé grâce au déplacement dans l'arbre fidèlement à chaque bit rencontré. Si le bit 0 est rencontré, on se déplace vers le fils gauche du nœud courant et si le bit 1 est rencontré, on se déplace vers le fils droit. La figure 1 illustre l'arbre correspondant au codage fixe donné dans le tableau 1.

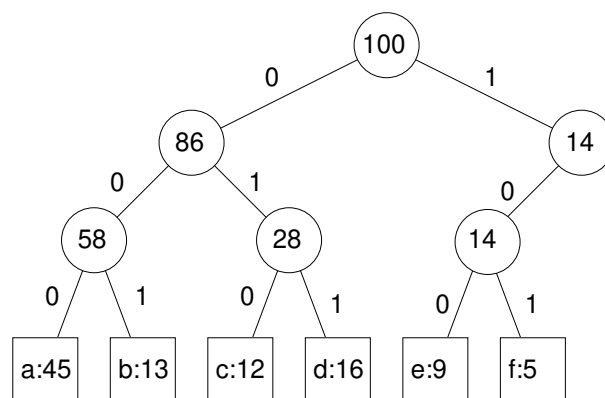


FIGURE 1 – Arbre de décodage des lettres fidèlement à un codage fixe

Question 1.1 : Donner le nombre d'étages (appelé hauteur par la suite) de l'arbre en fonction du nombre de lettres du fichier à coder ou à décoder. En déduire la taille du code

de chaque lettre.

Question 1.2 : Proposer un algorithme de construction d'un arbre de codage fixe conformément à un ensemble de n lettres écrites dans un tableau C et dont la fréquence d'apparition est donnée par une fonction f .

Vous pouvez supposer que le tableau C est trié en fonction de l'apparition des lettres (les plus fréquentes sont au début).

Comme cela est représenté sur la figure 1 l'arbre à créer doit mémoriser sur les nœuds la fréquence d'apparition et sur les feuilles le caractère et sa fréquence.

Question 1.3 : Proposer un algorithme de décodage d'un fichier codé par un arbre A donné en paramètre.

Exercice 2 : Codage variable

Le codage de taille variable est basé sur une hypothèse de construction : aucun mot de code (ou plus exactement une lettre codée) ne peut être le préfixe d'un autre. Ceci simplifie considérablement le codage et décodage car il n'y a pas d'ambiguïté. Ainsi le codage des lettres **abc** par le codage de longueur variable donné dans le tableau 1 ci-dessus est $0.101.100 = 0101100$. Comme précédemment, le codage se fait à l'aide d'un arbre binaire dont les feuilles sont les caractères. Pour le décodage, on interprète le mot binaire en se déplaçant dans l'arbre (0 à gauche et 1 à droite). Le codage est optimal si l'arbre est complet. La figure 2 illustre l'arbre de codage de l'exemple donné dans le tableau 1 ci-dessus.

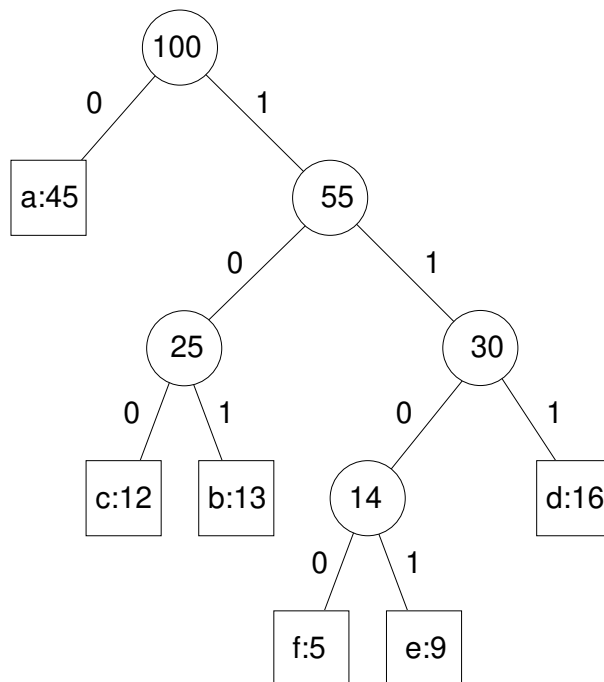


FIGURE 2 – Arbre de décodage des lettres fidèlement à un codage variable

La construction de l'arbre se fait du bas vers le haut. On considère C l'ensemble des

caractères à coder et $n = |C|$ le nombre de feuilles. La fonction $f(c)$ est la fréquence d'apparition du caractère c dans le fichier. On a également recours à une file de priorité F qui donne en sommet de file la lettre la moins fréquente. Les nœuds intermédiaires portent la fréquence d'apparition des caractères dont ils sont les ascendants (il s'agit de la somme de toutes les fréquences d'apparition des caractères situés dans le sous-arbre dont il est la racine).

Le principe de la construction de cet arbre est de fusionner les deux nœuds de moins grandes fréquences d'apparition (il s'agit soit de nœuds de l'arbre en construction, soit de caractères). Le nœud fusionné de plus grande fréquence est placé à droite du nouveau (codage du 1) alors que l'autre est placé à gauche. Le nœud nouvellement créé est porteur de la somme des fréquences des nœuds fusionnés. Il est aussitôt placé dans la file de priorité afin qu'il puisse faire l'objet à son tour d'une prochaine fusion. La racine de l'arbre est un nœud de fréquence 100%.

Question 2.1 : *Écrire l'algorithme de construction de l'arbre du codage de longueur variable de Huffman. Donner la complexité de cet algorithme de construction en fonction de la complexité d'accès à la file de priorité F . Comment cette file de priorité doit être mise en œuvre afin de garantir la meilleure complexité pour la construction de cet arbre ?*

Question 2.2 : *Écrire l'algorithme de décodage d'un fichier binaire codé suivant un arbre A donné en paramètre. Donner la complexité du décodage d'un fichier binaire.*

Exercice 3 : Etpaes de voiture : glouton optimal

Un voyageur de commerce possède une voiture dont le réservoir lui permet de faire N kilomètres. Il possède également une carte qui lui donne les stations d'essence qui sont sur la route de sa tournée. Bien sûr le voyageur de commerce veut s'arrêter le moins souvent possible pour faire le plein.

Question 3.1 : *Proposer une stratégie gloutonne permettant au voyageur de commerce de choisir ses points d'arrêt.*

Question 3.2 : *Cette stratégie est-elle optimale ? Prouvez votre réponse.*

Correction Exercice 1 : Codage fixe

Solution question 1.1 : Donner le nombre d'étages (appelé hauteur par la suite) de l'arbre en fonction du nombre de lettres du fichier à coder ou à décoder. En déduire la taille du code de chaque lettre.

Si C est l'ensemble des caractères à coder et n le nombre de ces caractères ($n = |C|$). L'arbre binaire à n feuilles a une hauteur h . Dans l'exemple de la figure 1 la hauteur de l'arbre est de 4.

$$h = \lceil \log_2(n) \rceil + 1$$

$$\text{taille du code} = h - 1$$

Solution question 1.2 : Proposer un algorithme de construction d'un arbre de codage fixe conformément à un ensemble de n lettres écrites dans un tableau C et dont la fréquence d'apparition est donnée par une fonction f .

À partir de la taille du tableau C , il est possible de trouver la hauteur h de l'arbre à construire. À partir du tableau C , nous initialisons une pile de caractères p où le caractère le plus fréquent se trouve au dessus de la pile. Nous utilisons les fonctions :

- `creerFeuille()`, qui rend une structure de feuille avec deux attributs : c est le caractère associé à la feuille et f la fréquence de ce caractère,
- `creerNoeud()`, qui rend une structure de nœud où `filsg` et `filsd` sont des nœuds fils ou des feuilles de l'arbre et f est la fréquence associée au nœud.

On peut utiliser deux méthodes pour créer l'arbre : une méthode de construction de haut en bas ou de bas en haut.

La méthode de création du haut en bas est une méthode récursive. L'implémentation de cette approche est donnée dans l'algorithme 1. Si la hauteur de l'arbre est atteinte, elle crée une feuille avec les valeurs associées au caractère, sinon elle crée un nœud et s'appelle récursivement pour créer les fils droit et gauche du nœud. L'algorithme se termine lorsque la pile est vide. À noter ici que la pile pourrait aussi bien être une file, ou que nous pourrions parcourir le tableau C en mémorisant l'indice courant.

La méthode de création de bas en haut est une méthode itérative pour laquelle il n'est pas nécessaire de connaître la hauteur de l'arbre avant de la créer, contrairement à l'algorithme précédent. L'implémentation de cette approche est donnée dans l'algorithme 2. L'idée est donc de d'abord créer les feuilles (lignes 4 à 9). Ensuite les nœuds de la base sont créés en associant deux par deux, au sein d'un nœud, les feuilles des caractères de la file $q1$ au premier passage. Chacun des nœuds créés est ensuite mis dans la file $q2$. À l'issue du premier passage nous avons donc tous les nœuds créés dans la file $q2$. Il faut faire la même chose avec les éléments de cette file pour construire l'étage suivant de l'arbre : les grouper deux par deux. L'algorithme repasse donc dans $q1$ le contenu de la file $q2$. Il continue ensuite ainsi jusqu'à ce qu'il n'y ait plus qu'un seul élément dans $q2$, la racine.

À noter que l'algorithme utilise deux files sans quoi la gestion des arbres binaires incomplets ne serait pas correcte, étant donné que les feuilles sont toutes à la même hauteur dans l'arbre de codage fixe.

Algorithme 1 : Codage de Huffman fixe, version de haut en bas

Données :

p : une pile des lettres de C

l : lettre formée du caractère $car(l)$ et d'une fréquence d'apparition $f(l)$

Résultat : le nœud courant de l'arbre

```
1 fonction HuffmanFixe1( $h, p$ )
2 début
3   si  $\neg p.empty()$  alors
4     si  $h = 1$  alors
5        $x \leftarrow creerFeuille()$ 
6        $x.c \leftarrow p.pop()$ 
7        $x.f \leftarrow f(x.c)$ 
8     sinon
9        $x \leftarrow creerNoeud()$ 
10       $x.filsg \leftarrow (HuffmanFixe1(h - 1, p))$ 
11       $x.filsd \leftarrow HuffmanFixe1(h - 1, p)$ 
12       $x.f \leftarrow x.filsg.f + x.filsd.f$ 
13   sinon  $x \leftarrow \text{NULL}$ 
14   retourner  $x$ 
```

Algorithme 2 : Codage de Huffman fixe, version de bas en haut

Données :

$q1, q2$: les files des nœuds de l'arbre en construction, **initialisé à** lettres de C
pour $q1$ et $fileVide$ pour $q2$

$nbElt$: taille de la file $q2$, **initialisé à** 0

Résultat : la racine de l'arbre

```
1 fonction HuffmanFixe2( $h, p$ )
2 début
3    $fin \leftarrow \text{FAUX}$ 
4    $q1 \leftarrow C$ 
5   tant que  $\neg q1.empty()$  faire
6      $x \leftarrow creerFeuille()$ 
7      $x.c \leftarrow q1.poll()$ 
8      $x.f \leftarrow f(x.c)$ 
9      $q2.add(x)$ 
10   $q1 \leftarrow q2$ 
11  tant que  $\neg fin$  faire
12    tant que  $\neg q1.empty()$  faire
13       $x \leftarrow creerNoeud()$ 
14       $x.filsg \leftarrow q1.poll()$ 
15       $x.f \leftarrow f(x.filsg.f)$ 
16      si  $\neg q1.empty()$  alors
17         $x.filsd \leftarrow q1.poll()$ 
18         $x.f \leftarrow x.f + f(x.filsd.f)$ 
19       $q2.add(x)$ 
20       $nbElt++$ 
21    si  $nbElt = 1$  alors  $fin \leftarrow \text{VRAI}$ 
22     $q1 \leftarrow q2$ 
23     $q2 \leftarrow fileVide()$ 
24  retourner  $x$ 
```

Une fois les arbres obtenus, il suffit de les utiliser de base en haut pour le codage et de haut en bas pour le décodage.

Solution question 1.3 : *Proposer un algorithme de décodage d'un fichier codé par un arbre A donné en paramètre.*

Pour le décodage nous suivons la file des valeurs binaires, que nous interprétons à l'aide de l'arbre A . L'algorithme 3 réalise le décodage du contenu du fichier codé, mémorisé dans la file f . La première boucle permet de parcourir l'ensemble de la file. La boucle for (lignes 5 à 8) permet elle le parcours de l'arbre de haut en bas : à chaque itération nous prenons la valeur binaire suivante dans la file et, en fonction de sa valeur, nous continuons le parcours de l'arbre à partir du fils gauche ou du fils droit, jusqu'à arriver à une feuille qui nous donne le caractère attendu. Il suffit ensuite de recommencer l'opération à partir de la racine pour obtenir le caractère suivant.

Algorithme 3 : Algorithme de décodage de HuffmanFixe

Données :

f : la file des valeurs binaires du fichier codé

A : l'arbre de codage fixe

```

1 fonction DecodeHuffmanFixe( $A, h$ )
2 début
3    $a \leftarrow \text{racine}(A)$ 
4   tant que  $\neg f.\text{empty}()$  faire
5     pour  $i$  de 1 à  $h - 1$  faire
6        $b \leftarrow f.\text{poll}()$ 
7       si  $b = 0$  alors  $a \leftarrow a.\text{fils}_g$ 
8       sinon  $a \leftarrow a.\text{fils}_d$ 
9      $\text{afficher}(a.c)$ 
10     $a \leftarrow \text{racine}(A)$ 

```

Correction Exercice 2 : Codage variable

Solution question 2.1 : *Écrire l'algorithme de construction de l'arbre du codage de longueur variable de Huffman. Donner la complexité de cet algorithme de construction en fonction de la complexité d'accès à la file de priorité F . Comment cette file de priorité doit être mise en œuvre afin de garantir la meilleure complexité pour la construction de cet arbre ?*

Il est possible de construire l'arbre de Huffman avec un algorithme glouton donnant un codage optimal, c'est à dire avec le plus faible nombre de lettres pour le codage. Cet algorithme est glouton car nous faisons le choix localement des nœuds à ajouter dans l'arbre pour que globalement, ce choix soit le choix optimal. L'algorithme suivant est l'algorithme classique de Huffman, utilisant une file de priorité (c'est à dire une file permettant l'accès à un élément que l'on peut considérer comme MIN ou MAX suivant une métrique donnée). Par construction, étant donné le nombre n de lettre à coder, il

faut $n - 1$ étapes pour construire l'arbre binaire.

Algorithme 4 : Codage de Huffman variable

Données : $n = |C|$

f : une file de priorité permettant l'accès à $\min()$, la lettre de plus faible fréquence d'apparition ou le nœud représentant les lettres globalement de plus faible fréquence d'apparition. Cette fonction extrait physiquement l'objet ainsi désigné.

Résultat : racine de l'arbre

```

1 fonction HuffmanVariable( $A, h$ )
2 début
3    $F \leftarrow C$ 
4   pour  $i$  de 1 à  $n - 1$  faire
5      $x \leftarrow \text{creerNoeud}()$ 
6      $x.\text{filsg} \leftarrow f.\text{min}()$ 
7      $x.\text{filsd} \leftarrow f.\text{min}()$ 
8      $x.f \leftarrow x.\text{filsg}.f + x.\text{filsd}.f$ 
9      $f.\text{add}(x)$ 
10  retourner  $\min(f)$ 
```

La complexité de l'algorithme 4 est $O(n \times g(n))$ où $g(n)$ est la complexité maximale entre l'extraction à l'élément $\min()$ et l'insertion d'un nouvel élément dans f . En implémentant la file de priorité f dans une structure de Tas, chaque accès ou extraction a une complexité $O(\log_2(n))$ si n est la taille du Tas. En effet extraire un élément se fait en prenant la racine (complexité en $O(1)$). Le vide est comblé par le dernier élément du Tas dont la taille est réduite de 1. Afin de respecter la propriété de Tas, cette nouvelle racine prend sa place en *tombant* en direction des branches le long desquelles des nœuds prennent sa place étape par étape. Cette chute a une plus grande hauteur égale à la hauteur de l'arbre, d'où une complexité dans le pire des cas en $O(\log_2(n))$. De plus, chaque insertion d'un nouvel élément dans le Tas prend au pire $O(\log_2(n))$ opérations. En effet, cet ajout se fait en fin de Tas, avec sa taille qui augmente cette fois de 1. L'élément ajouté remonte comme une bulle dans les branches du Tas jusqu'à trouver sa place en respect des propriétés du Tas. Dans le pire des cas, l'élément remonte jusqu'à la racine, d'où la complexité annoncé.

La complexité de l'algorithme de Huffman est donc $O(n \log_2(n))$.

Solution question 2.2 : *Écrire l'algorithme de décodage d'un fichier binaire codé suivant un arbre A donné en paramètre. Donner la complexité du décodage d'un fichier binaire.*

L'idée du décodage d'un fichier codé suivant un arbre A est de suivre les branches de l'arbre ('0' en direction du fils gauche, '1' en direction du fils droit) jusqu'à une feuille qui donne la valeur de la lettre à décodée. Le codage étant non-ambigu, le décodage est sûr. L'algorithme 5 illustre ce parcours de décodage.

Algorithme 5 : Decodage de Huffman variable

Données : f : les valeurs binaire du code dans une file

```
1 fonction DecodeHuffmanVariable(A)
2 début
3    $a \leftarrow \text{racine}(A)$ 
4   tant que  $\neg f.\text{empty}()$  faire
5       tant que  $a$  n'est pas une feuille faire
6            $b \leftarrow f.\text{poll}$ 
7           si  $b = 0$  alors  $a \leftarrow a.\text{fils}_g$ 
8           sinon  $a \leftarrow a.\text{fils}_d$ 
9       afficher( $a.c$ )
10   $a \leftarrow \text{racine}(A)$ 
```

La complexité de cet algorithme dépend de la taille de l'arbre de Huffman. Dans le meilleur des cas il est complètement équilibré, dans le pire des cas il est linéaire.

- Dans le meilleur des cas, l'accès à une feuille se fait en $\Omega(\log_2(n))$ où n est le nombre de lettres différentes dans le fichier à décoder.
- Dans le pire des cas l'accès se fait au plus profond de l'arbre en $O(n)$.

Par conséquent, la complexité est en $\Omega(N \times \log_2(n))$ et $O(N \times n)$ avec n le nombre de lettres différentes dans le fichier contenant N caractères.

Correction Exercice 3 : Etpaes de voiture : glouton optimal

Solution question 3.1 : *Proposer une stratégie gloutonne permettant au voyageur de commerce de choisir ses points d'arrêt.*

Une solution gloutonne consiste à toujours prendre, dans la configuration en cours, la solution qui optimise localement le résultat. Ici, la solution consiste, à partir d'une station dans laquelle nous venons de faire le plein, à choisir la station la plus éloignée de notre position, parmi celles qui sont situées à moins de N kilomètres de notre position.

Solution question 3.2 : *Cette stratégie est-elle optimale ? Prouvez votre réponse.*

Pour prouver l'optimalité de l'algorithme glouton nous réalisons un raisonnement par l'absurde en supposant que l'algorithme ne donne pas une solution optimale que nous comparons à la solution optimale. Nous considérons donc la solution $F = (x_1, x_2, \dots, x_p)$ fournie par notre algorithme, où (x_1, x_2, \dots, x_p) est une suite de stations services (en fait leur distance par rapport au point de départ) triées par ordre d'arrêt, et une solution optimale $G = (y_1, y_2, \dots, y_q)$.

Puisque G est la solution optimale nous avons $p \geq q$, notre solution contient au moins autant d'arrêts à des stations services que la solution optimale. Pour montrer que notre solution est optimale nous devons alors prouver que $p = q$.

Soit k le plus petit entier tel que : $\forall i < k, x_i = y_i$, et $x_k \neq y_k$. x_k est la première station service dans notre solution qui est différente de celle qui est utilisée par la solution

optimale. Alors, par définition de notre algorithme, nous avons $x_k > y_k$, puisque nous prenons toujours la station la plus éloignée.

Nous construisons, à partir de la solution optimale $G = (y_1, y_2, \dots, y_{k-1}, y_k, y_{k+1}, \dots, y_q)$, la suite de stations services $G' = (y_1, y_2, \dots, y_{k-1}, x_k, y_{k+1}, \dots, y_q)$ telle que G' est une liste de même taille que G . Montrons que c'est aussi une solution, et donc une solution optimale puisqu'elle n'utilise pas plus de stations que G . Pour cela il nous faut vérifier qu'il n'y a pas de risque de panne sèche c'est-à-dire que : $x_k - y_{k-1} \leq N$ et $y_{k+1} - x_k \leq N$.

- F est une solution, donc $x_k - x_{k-1} \leq N$. Par construction, nous avons $y_{k-1} = x_{k-1}$ donc $x_k - y_{k-1} \leq N$ et il n'y a pas de risque de panne pour aller de y_{k-1} à x_k .
- G est une solution, donc $y_{k+1} - y_k \leq N$. Or $y_{k+1} - x_k < y_{k+1} - y_k \leq N$ en effet nous avons choisi la station la plus éloignée dans notre algorithme et la distante restante est la plus petite possible. Donc il n'y a pas non plus de risque de panne pour aller de x_k à y_{k+1} . De plus, si $y_{k+1} < x_k$ alors nous pourrions supprimer y_{k+1} ce qui contredirait l'optimalité de G .

Nous pouvons alors supprimer y_k de G' et obtenir une solution strictement meilleure ce qui contredirait l'optimalité de G . G' est donc bien une solution optimale de notre problème. Nous avons donc construit à partir de G une solution G' qui contient un élément en commun de plus avec F . Nous itérons jusqu'à avoir une solution optimale contenant F . Alors $p \leq q$, ce qui prouve le résultat attendu.