|  |  |
| --- | --- |
| Scène 1 | |
| Nom | Titre |
| Durée | 10 s |

Nous vous présentons une capsule sur le codage de Huffmann, un codage utilisé pour la compression de donnée.

|  |  |
| --- | --- |
| Scène 2 | |
| Nom | Intro |
| Durée | 3 min 44 4min30s |

Le codage de Huffman, du nom de son inventeur et qui date de 1951, est un algorithme glouton qui permet la compression de donnée sans perte, algorithme basé sur la construction d’un arbre binaire. Il est utilisé pour compresser des données dans de nombreux processus de l’informatique moderne (compression d’images, de vidéos, de son, de fichier, etc.).

La compression de données, ou codage de source, est l’opération informatique qui consiste à transformer une suite de bits A en une suite de bits B plus courte, contenant les mêmes informations, en utilisant un algorithme particulier, ici le codage de Huffman. La décompression est l’opération inverse de la compression. C’est un couple de mécanismes omniprésent dans les systèmes numériques.

Avec un algorithme de compression sans perte comme le codage de Huffman, la suite de bits obtenue après les opérations successives de compression et de décompression est strictement identique à l’originale.

Rappelons qu’il n’existe pas de technique de compression de données sans perte universelle, qui pourrait compresser n’importe quel fichier. Par exemple, un fichier qui est déjà le résultat d’une compression se compresse mal, voire grossit par application d’un compresseur. Mais dans la pratique, les mots, messages ou fichiers que l’on souhaite compresser ne sont pas quelconques. Ils ont une entropie faible, propriété que les compresseurs utilisent.

Un algorithme glouton, comme l’algorithme du codage de Huffman, est un algorithme qui fait à chaque étape le choix le meilleur possible à cet instant en choisissant par exemple ici comme nœuds fils les nœuds de plus petite fréquence possible. Si ce choix est aussi le meilleur possible dans l’absolu, alors l’algorithme glouton est optimal. C’est le cas pour le codage de Huffman, qui produit un code préfixe de longueur moyenne optimale.

(1min45 jusque là)

Prenons l’exemple du texte Hello World. En codage ASCII, à l’aide de mots de 8 bits, 88 bits de mémoire sont nécessaires pour stocker cette chaîne de caractères. Mais on peut remarquer une certaine redondance : la chaîne de caractères contient par exemple 3 L et 2 O. Il y a en tout 8 symboles différents pour une chaîne de 11 caractères.

Si on applique un compresseur comme le codage de Huffman, le code généré sera adapté au nombre de symboles présents et des mots plus courts seront utilisés pour les symboles plus fréquents.

Voilà la suite de bits nécessaire pour stocker en mémoire Hello World à l’aide d’un code de Huffman : 32 bits constitués de mots de taille variable, soit une diminution de taille ou

taux de compression de 64 %.

Les mots de taille variable implique une subtilité. Pour lire un texte codé en ASCII, il suffit de redécouper la suite de bits en mots de 8 bits et d’y associer la lettre correspondante dans le code ASCII. Ce même processus ne fonctionne plus pour un codage à l’aide de mots de taille variable : prendre un mot de 8 bits n’a ici aucun sens.

Alors comment décoder ? Le codage de Huffman nécessite l’usage d’un code préfixe. Un code est dit préfixe si aucun mot de code n’est un préfixe d’un autre mot de code, c’est-à-dire correspond au début d’un autre mot de code. Par exemple, si la lettre A est codée par le mot 00, alors aucun autre mot de code ne pourra commencer par 00.

Voici deux codes : (Tableau à rajouter dans la scène si pas trop long compliqué avec 010 : 2, 0 10 : 14, 01 0 : 31)

|  |  |  |
| --- | --- | --- |
| Chiffre | Code 1 | Code 2 |
| 1 | 0 | 0 |
| 2 | 010 | 10 |
| 3 | 01 | 110 |
| 4 | 10 | 111 |

Avec le code 1, 010 peut-être le code du chiffre 2, du nombre 14 (concaténation de 1 (0) et 4(10)) ou encore du nombre 31 (concaténation de 3(01) et 1 (0).

Avec le code 2, il n’y a aucune ambiguïté possible.

Voilà le code préfixe utilisé pour coder Hello World à l’aide du codage de Huffman. Pour décompresser la suite de bit, il suffit de passer en revu un à un les bits jusqu’à reconnaître, sans ambiguïté possible, un mot de code. Par exemple, les 4 premiers 1 ne peuvent correspondre qu’à la lettre H. Puis la suite 010 correspond forcément à la lettre e. Avec un code préfixe, il n’y a donc plus besoin de mots de longueur fixe ou de séparateur.

Un code ayant la propriété du préfixe est uniquement déchiffrable/décodable, c’est à dire enon-ambigü

On peut représenter un code préfixe par un arbre binaire dont seules les feuilles de l’arbre sont étiquetées par les lettres de l’alphabet à coder.

Un code est préfixe si et seulement si dans son arbre tous les symboles de S sont portés par des feuilles.

C’est ce que nous allons découvrir maintenant avec un exemple complet.

(2min05 2min35s soit un total de 3 min 50 4min 30 s)

|  |  |
| --- | --- |
| Scène 3 | |
| Nom | TexteArbre |
| Durée | 1 min 46 |

Un exemple pour comprendre.

Considérons un texte de 100 caractères choisis aléatoirement parmi l’ensemble C contenant les caractères de A à F.

Si nous devions stocker ce texte en utilisant le codage ASCII 8 bits, alors 800 bits de mémoire seraient nécessaire.

Appliquons sur ce texte l’algorithme de Huffman afin de trouver un meilleur codage. La première étape est de constituer la file de priorité F.

La lettre A a une fréquence d’apparition de 45 %. La lettre B apparaît avec une fréquence de 13 %. La lettre C avec une fréquence de 12 %. La lettre D avec une fréquence de 16 %. La lettre E avec une fréquence de 9 % et enfin la lettre F avec une fréquence de 5 %.

À partir de ces couples (caractère,fréquence), on génère une file de priorité F dans laquelle sont stockés les différents nœuds correspondant aux caractères avec comme poids associés, leur fréquence d’apparition.

Nous avons donc au final une file de priorité F ordonnée dans l’ordre croissant.

Voyons maintenant comment l’algorithme de Huffman permet d’en tirer un arbre de codage optimal.

|  |  |
| --- | --- |
| Scène 4 | |
| Nom | Algo |
| Durée | 1 min 40 |

L’algorithme de Huffman s’applique sur un ensemble de caractères C.

1. Le nombre de caractères différents est stocké dans n : sa valeur est utilisée pour définir le nombre d’itération de la boucle principale de l’algorithme.

2. Comme on a vu précédemment, une file de priorité F est générée à partir des caractères et de leur fréquence. Cette file de priorité ne contient pour le moment que des feuilles (caractère,fréquence). Au fur et à mesure de l’exécution de l’algorithme, on va y voir apparaître les sous-arbres de l’arbre de codage optimal final.

3. La troisième ligne définit la boucle principale, avec pour variable d’itération i allant de 1 à (n-1). À chaque instant, la file de priorité F contient (n – i + 1) nœuds.

4. À chaque itération de la boucle principale, un nouveau nœud est créé.

5. Ce nœud possède comme fils à gauche, le nœud de poids le plus faible de la file de priorité F.

6. Et comme fils droit, le nœud suivant de poids le plus faible de la file de priorité F.

7. On attribut comme poids à ce nouveau nœud la somme des poids (donc des fréquences d’apparition) des deux nœuds fils.

8. Enfin, on insère le nouveau nœud ainsi créé dans la file de priorité, à la bonne place.

Les itérations se répètent suivant le même processus jusqu’à ce qu’il ne reste plus qu’un unique nœud dans la file de priorité.

9. Ce nœud correspond alors à la racine de l’arbre de codage optimal, que l’algorithme retourne en se terminant.

Voyons maintenant l’application pas à pas de cet algorithme sur notre texte de 100 caractères.

|  |  |
| --- | --- |
| Scène 5 | |
| Nom | Arbre |
| Durée | 1 min 50 s |

Lorsque l’algorithme commence, n prend la valeur 6 car il y a 6 caractères différents dans le texte à coder. Comme nous l’avons vu tout à l’heure, la file de priorité F contient les différents nœuds correspondant aux caractères avec comme poids associés, leur fréquence d’apparition, le tout dans l’ordre croissant.

L’itération commence : i prend la valeur 1. Un nouveau nœud est créé. Il prend comme nœud fils à gauche le minimum de la file F et comme nœud fils à droite le nouveau minimum de F. Le poids associé à ce nouveau nœud correspond à la somme des fréquences des deux nœuds fils, ici 9 + 5 = 14. On insère alors ce nouveau nœud dans la file de priorité : de part son poids, il va être inséré entre le nœud contenant B et le nœud contenant D.

Une nouvelle itération commence : i prend la valeur 2. Un nouveau nœud est créé. Il prend comme nœud fils à gauche le minimum de la file C et comme nœud fils à droite le nouveau minimum de la file B. Son poids est assigné à 12 + 13 soit 25. On insère alors ce nouveau nœud dans la file de priorité : de part son poids, il va être inséré entre le nœud contenant D et le nœud contenant A.

Nouvelle itération : i = 3. Un nouveau nœud est créé. Le minimum de la file F correspond à un sous-arbre de poids 14. Il devient le nœud fils gauche du nouveau nœud. Le nœud fils droit correspond au minimum de la file, maintenant le nœud contenant F. Le poids du nœud créé est assigné à 14 + 16 = 30. Il va alors être inséré dans la file entre le sous-arbre de poids 25 et la nœud contenant A.

Nouvelle itération : i vaut 4. Un nouveau nœud est créé. Pour le nœud fils gauche, le minimum de la file F correspond au sous-arbre de poids 25 puis pour le nœud fils droite au sous-arbre de poids 30. Notre nouveau nœud se voit donc assigner un poids de 25 + 30 = 55. Il est inséré dans la file F à la suite du nœud contenant A dont le poids (ou fréquence) vaut 45.

On arrive à la dernière itération : i = 5 donc n-1. Un nouveau nœud est créé. Le minimum de la file, donc le nœud contenant A devient le nœud fils gauche et le dernier nœud de la file (le sous-arbre de poids 55) le nœud fils droit. Le nouveau nœud a donc un poids de 45 + 55 = 100, ce qui est normal puisque tous les poids initiaux, donc toutes les fréquences d’apparition des caractères, ont été sommées. Notre nouveau nœud, qui correspond à la racine de l’arbre de codage optimal, est inséré dans la file de priorité F dont il est l’unique nœud.

On sort de la boucle. L’algorithme retourne alors le minimum de la file F qui correspond comme on l’a vu à l’arbre de codage optimal.

Maintenant que l’arbre de codage est obtenu, voyons comment en tirer le codage de Huffman.

|  |  |
| --- | --- |
| Scène : | |
| Nom | Caractéristiques |
| Durée | 1min30 s |

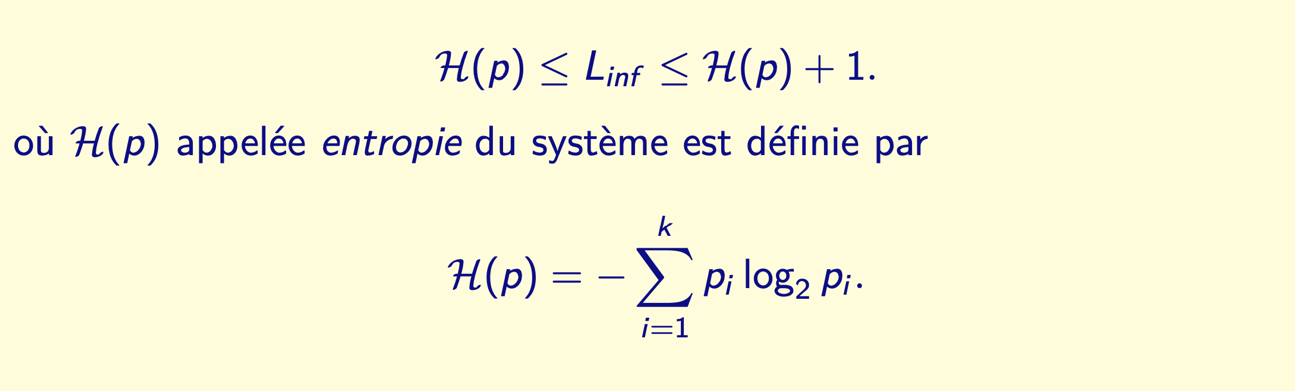
Premier slide avec

Inégalité de Kraft et longueur moyenne d’un code : formules à faire apparaître Inégalité + longueur moyenne

Entropie et Théorème de Shannon : faire apparaître une photo de Shannon ? + Théorème de Shannon

Une image contenant texte, homme, personne, habits

Description générée automatiquement



Intro à faire …

Pour tout code ayant la propriété du préfixe on a : Somme de i=1 à k de ½^l\_i inférieure ou égale à 1

et réciproquement si on a une telle inégalité alors il existe un code avec la propriété du préfixe dont les mots-codes ont pour longueur l\_i. Il s’agit de l’inégalité de Kraft.

Les codes préfixes assurent qu’ils sont décodables sans erreur, ce qui indispensable, mais on peut se demander si certains certains sont meilleurs que d’autres ?

Soit un alphabet avec k symboles à coder dont les probabilités d’apparition sont {p1, · · · , pk }.

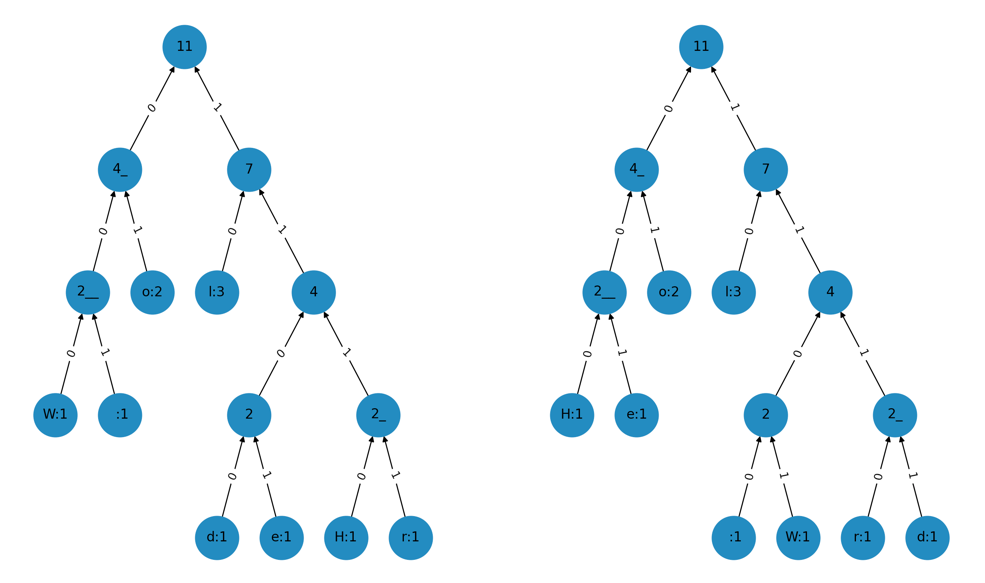
La longueur moyenne du code L(C) est donnée par la formule suivante : = somme de i=1 à k de p\_i l\_i

On note L\_inf = inf \_C préfixe L(C), la longueur moyenne du meilleur code préfixe

En 1948, Shannon à une séquence de lettres ou mots son entropie, avec une définition assez analogue à celle en physique. L’entropie représente la quantité d’informations contenue dans la séquence de lettres/mots. Ainsi une séquence redondante a une entropie faible alors qu’une séquence aléatoire a une entropie plus importante. Ce résultat lie la longueur moyenne du meilleur code aux probabilités d’apparition (proportions), et ce indépendamment du code choisi.

* Il y a une borne inférieure absolue : on ne pourra pas compresser l’information au delà d’un certain facteur H(*p*) qui correspond à l’entropie du système.
* D’après Kraft, il est possible de construire le meilleur code possible en affectant à chaque symbole *si* un code de longueur ⌈− log2 *pi* ⌉

Deuxième slide : celui déjà fait avec Ajout de cette figure dans la scène



Avec en dessous des arbres

H : 1110 H : 000

e : 1101 e : 001

l : 10 l : 10

o : 01 o : 01

: 001 : 1100

W : 000 W : 1101

r : 1111 r : 1110

d : 1100 d : 1111

Dans le texte déjà écrit remplacer « éléments » par symboles

Voici maintenant quelques caractéristiques du codage de Huffman.

Tout d’abord la non-unicité : pour un même alphabet, on peut produire des arbres, et donc des codes, différents.

En effet, pour les symboles qui ont la même occurrence, il y a un/des choix à faire. Dans quel ordre prend-on ces symboles ? Dans notre exemple, « Hello World », à part le « l » qui apparaît 3 fois et le « o » qui apparaît deux fois, toutes les autres lettres et l’espace n’apparaissent qu’une seule fois. On voit donc sur l’image ci-dessous qu’en permutant les symboles n’apparaissant qu’une seule fois, on obtient deux arbres différents et donc des codes différents pour les symboles de notre alphabet.

De plus, pour chaque nœud, il y a un choix entre gauche et droite à effectuer.

Dans notre exemple, nous avons choisi de toujours mettre à gauche le fils dont la valeur associée est plus petite et à droite, celui, dont la valeur est plus grande.

Effacer ce qu’il y a sur la scène et ne garder que le titre « Caractéristiques du codage de Huffman ». Rajouter 2 points :

Conséquence

Codage canonique

Cette non-unicité à des conséquences non négligeables. En effet, cela implique de communiquer le code pour procéder au décodage ou au minimum d’indiquer les règles/choix effectués.

Il existe une forme canonique de l’arbre de Huffman. Elle peut être obtenue par construction (beaucoup plus simple avec des files de priorités/tas binaire) ou par canonisation des codes choisis pour chaque symbole (i.e., directement sur la table de compression).

…

|  |  |
| --- | --- |
| Scène | |
| Nom | Limitations |
| Durée | 30s |

Si possible : premier « slide » : Titre et les deux limitations :

-nombre entier de bits

- codage statique

Puis 2ème slide : titre avec la 1ère limitation + texte qui va avec

Et enfin 3ème slide : titre avec la 2ème limitation + texte qui va avec

Il y a deux limitations principales au codage de Huffman. Tout d’abord, le codage de Huffman impose d’utiliser des nombres entiers de bits pour un symbole source

…

Par ailleurs, ainsi présenté le codage Huffman est « statique » il se base sur la fréquence évaluée au début du codage des occurrences des symboles. Si la source évolue et que les propriétés statistiques diffèrent le codage de Huffman n’est alors plus aussi optimal. Pour prendre en compte ceci, il y a le codage de Huffman évolutif

…

|  |  |
| --- | --- |
| Scène | |
| Nom | Preuve |
| Durée | Environ 2 min |

1re slide : preuve avec les deux parties

* Proprité du choix glouton
* Sous structure optimale

2 slide : Titre « Propriété du choix glouton » + « Lemme des fréquences faibles » avec le texte du lemme avec image ci-dessous et les deux formules de B(T) et celle de B(T)-B(T’) (dans le livre)

(image du livre)

Une image contenant texte, tableau blanc

Description générée automatiquement

3e slide : Titre « Sous-strucure optimale » + « Lemme de propagation de l’optimalitée » avec le texte du lemme.

Démontrer que l’algorithme glouton de Huffman est correct, revient montrer que le problème consistant à déterminer un codage préfixe optimal nécessite

* La propriété du choix glouton (lemme des fréquences faibles)
* La sous-structure optimale (lemme de propagation de l’optimalité)

Lemme des fréquences faibles

Lemme à recopier ici.

Le principe est de prendre… ( j’ai mon texte)

Lemme de propagation de l’optimalité

Phrase intro

Lemme à recopier ici.

On commence par montrer que …. (j’ai mon texte)

|  |  |
| --- | --- |
| Scène | |
| Nom | Complexité |
| Durée | 10 s |

Chaque itération construit un nœud interne d’un arbre à k feuilles

L’algorithme nécessite donc k-1 itérations

La file de priorité est implanté dans un tas, les coûts d’extraction et d’insertion sont donc en O(log k)

D’où une complexité en klog(k) de l’ algorithme pour un alphabet de k symboles.

Remarque : Le nombre de nœuds dans la file à priorité est un invariant de boucle.

…

|  |  |
| --- | --- |
| Scène | |
| Nom | Implémentation |
| Durée |  |

Inclure sur un slide :

* la vidéo montrant l’exécution du code avec un arbre pour le texte de 100 caractères
* les points-clé suivants :
* table de fréquence
* construction d’un arbre
* code de Huffman pour chaque symbole
* fonction pour coder avec le code de Huffman
* code inverse pour chaque symbole et fonction pour décoder
* le codage de Huffman pour les 6 symboles A : 0 B : 101 C : 100 D : 111 E : 1101 F : 1100
* l’algo de l’arbre :

Huffman (C)

N=ICI

Q = C

**pour i** = 1 à n-1

Allouer un nouveau noeud z

z.gauche = x = EXTRAIRE-MIN(Q)

z.droit = y = EXTRAIRE-MIN(Q)

z.freq = x.freq + y.freq

INSERER(Q,z)

**retourner** EXTRAIRE-MIN(Q)

Les points clés du programme qui s’exécute ci-dessous sont les suivants :

1. Création d’une table de fréquence qui pour chaque symbole ( 6 dans notre exemple) associe un nombre d’occurrences du symbole
2. Construction d’un arbre, avec Networkx, avec les règles expliquées précédemment :

On rejoint toujours 2 lettres dont les occurrences sont les plus petites

On somme les 2 occurrences(probabilités) pour en faire un nœud

Voici un rapide descriptif de l’algorithme :

* 1. On détermine le nombre de caractères n
  2. On initialise la file de priorité minimale avec les caractères de C
  3. La boucle pour extrait de manière répétée les 2 nœuds x et y ayant les férquences les plus basses de la file, puis on les remplace dans la file par un nouveau nœud z, représentant leur fusion. La fréquence de z est la somme des fréquences de x et y.
  4. On a mis x comme fils gauche et y comme fils droit, mais c’est un choix arbitraire)
  5. Après n-1 fusions, on retourne le seul nœud qui reste dans la file, à savoir la racine de l’arbre de codage.

1. De l’arbre binaire, on déduit le code de Huffman pour chaque symbole : en partant de la racine et en descendant jusqu’à la feuille du symbole correspondant, on juxtapose les 0et 1 figurant sur les arêtes entre les nœuds. Ainsi dans notre exemple A est codé par 0, B est codé par 101, C est codé par 100, D par 111, E par 1101 et F par 1100
2. Création d’un dictionnaire associant chaque symbole à son code binaire
3. Fonction utilisant le dictionnaire codant le texte ( le dictionnaire ou table de code est donc suffisant)
4. Création d’un dictionnaire inverse et fonction utilisant ce dictionnaire inverser pour décoder : étape de vérification. Pour le décodage, grâce à la propriété du préfixe, dès qu’un mot code est reconnu on le décode ( le dictionnaire inverse suffit donc). Mais pour être efficace, on utilise l’arbre : on suit le chemin désigné par la suite de bits initiale du mot binaire jusqu’à arriver à une feuille, on obtient alors le symbole. Puis on repart de la racine.

Proposition : rajouter la « définition » du codage de Hufffman comme support visuel du résumé ci-dessous

Le codage de Huffman :

algorithme glouton

qui construit un codage préfixe optimal

en construisant l’arbre binaire de façon ascendante

en utilisant une file de priorité pour trouver l’élément le moins fréquent

En résumé, le principe du codage de Huffman est de chercher un code binaire C ayant la propriété du préfixe, pour qu’il soit facile à décoder, et minimisant la longueur moyenne du code, pour qu’il soit efficace. Pour ce faire, on construit un arbre C correspondant de façon ascendante en combinant deux à deux les symboles de fréquences les plus faibles.

|  |  |
| --- | --- |
| Scène | |
| Nom |  |
| Durée |  |

|  |  |
| --- | --- |
| Scène | |
| Nom |  |
| Durée |  |

|  |  |
| --- | --- |
| Scène | |
| Nom |  |
| Durée |  |

|  |  |
| --- | --- |
| Scène | |
| Nom | Références |
| Durée | 10 s |

|  |  |
| --- | --- |
| Scène | |
| Nom | Crédits |
| Durée | 11 s |

|  |  |
| --- | --- |
| Scène | |
| Nom |  |
| Durée |  |