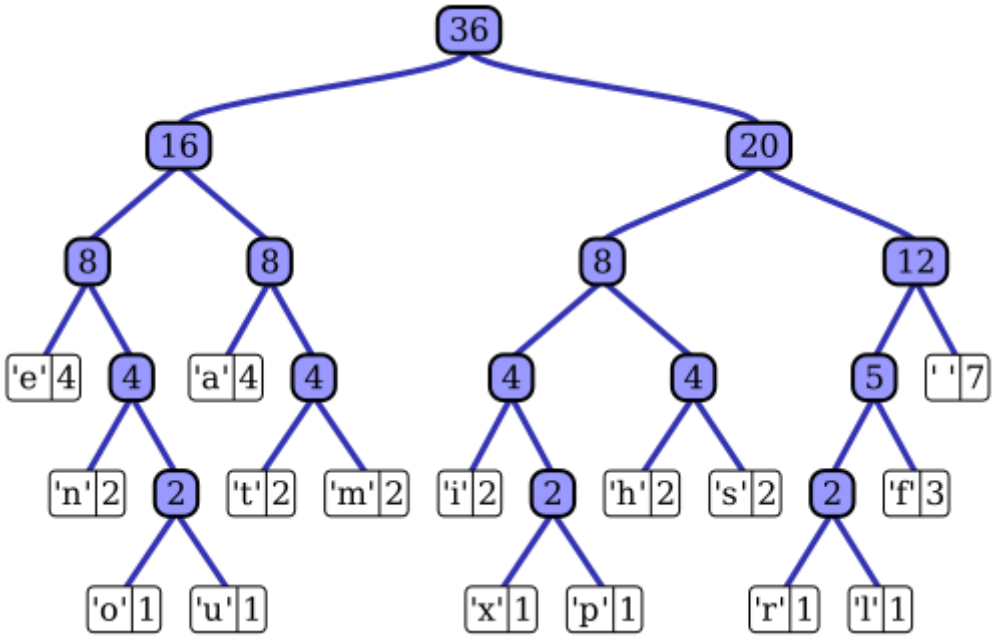


# Étude du codage Huffman

## Cours écrit par jaudi

# Sommaire du cours

# Introduction

Chers membres de Club Alkindi,
Après avoir publié ou créé moi-même de nombreuses énigmes pour le site, j'ai décidé d'écrire mon premier cours sur Club Alkindi pour vous dévoiler les rouages d'un chiffrement qui m'a beaucoup intéressé.
Dans ce cours, nous allons nous plonger dans les coulisses d'un chiffrement plutôt récent que vous ne connaissez pas forcément tous, mais qui néanmoins a eu une importance capitale dans les télécommunications, notamment dans le secteur informatique, durant ce dernier siècle : le codage de Huffman.
Je pense que la majeure partie de ce cours peut être accessible à un niveau collégien et lycéen, car j'ai essayé de bien expliciter les différents procédés utilisés. Néanmoins, je pense que des adultes ou cryptanalystes plus chevronnés y trouveront également leur compte car j'ai essayé d'être exhaustif sur les méthodes, la théorie mathématique, les heuristiques informatiques ainsi que les exemples pour vous fournir un document approfondi sur le sujet.
Ce cours pourra vous être utile pour certaines énigmes du site, déjà parues ou qui arriveront prochainement donc profitez-en !
Bonne lecture,
jaudi

# I – Biographie de David Albert Huffman

Né le 9 août 1925 dans l'Ohio, David Albert Huffman obtint un baccalauréat en génie électrique en 1944, puis fut embauché deux ans dans la Marine américaine en tant qu'officier de maintenance des radars. En 1949, il fut diplômé d'un master en génie électrique à Ohio.

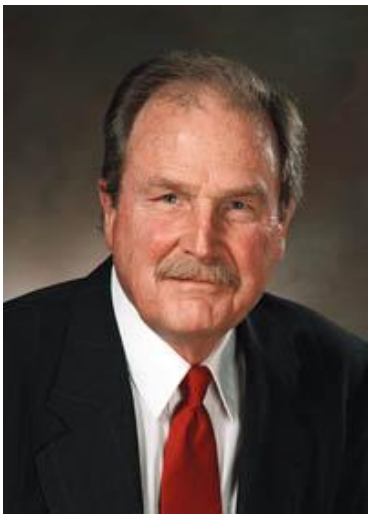

*Figure 1 -Portrait de David Albert Huffman*

Durant sa thèse au MIT, il publia un article intitulé « *A Method for the Construction of Minimum-Redundancy Codes »,* qui apparaît en page 1098 de la revue Proceedings of the Institute of Radio Engineers de septembre 1952. Cet article de 3 pages sur la construction de codes à redondance minimale, apparemment anodin et présentant une méthode optimale de codage pour minimiser le nombre moyen de chiffres du code d'un message, aura des conséquences immenses pour la cryptographie lors de l'essor de l'informatique. En effet, ce codage Huffman permet la compression de données sans perte et a été utilisé durant des dizaines d'années par de nombreux domaines (réseaux informatiques, télévisions, premiers

ordinateurs, télécopieurs…). Petite anecdote, cet article a été écrit alors que les étudiants du cours de Robert Fano (inventeur du codage Shannon-Faro) avaient le choix entre passer un examen final traditionnel ou améliorer un algorithme de pointe pour la compression de données, et Huffman opta alors pour la deuxième solution et proposa une méthode qui surpassa celle de son professeur, créant ainsi le codage de Huffman, que nous allons étudier dans ce cours !

Il soutint ensuite sa thèse, intitulée « *The Synthesis of Sequential Switching Circuits* », sur les circuits de commutation séquentielle sous la direction de Caldwell en 1953. Il rejoint ensuite la faculté du MIT en 1953, puis l'université de Californie Santa Cruz en 1967 pour élaborer les programmes académiques concernant l'informatique. A sa retraite en 1994, il continua d'enseigner en temps que professeur, notamment sur la théorie de l'information et l'analyse des signaux.

Durant toute sa carrière, ce chercheur américain réalisa de grands travaux de recherche en informatique (théorie de l'information, codage, circuits logiques…) et mathématiques (étude des surfaces avec zéro courbure) et reçut de nombreuses distinctions, notamment la médaille Richard-Hamming en 1999, attribuée par l'Institute of Electrical and Electronics Engineers (IEEE), l'année de son décès.

# II – Présentation du codage Huffman

Inventé en 1952 par David Albert Huffman (comme évoqué dans sa biographie), le codage Huffman autorise une économie conséquente de 0 et de 1 en adaptant le chiffrement des lettres à leur fréquence (un peu à l'image du Morse).
Par conséquent, cet algorithme, qui prend en argument un alphabet quelconque, permet une compression des données sans perte, sous format binaire.

On définit un dictionnaire qui, à chaque caractère de l'alphabet, associe un code binaire unique (composé de 0 et de 1). De plus, le codage de Huffman est un code préfixe, ce qui signifie que le dictionnaire est construit de telle sorte qu'aucun des codes binaires d'un caractère de l'alphabet ne serve de préfixe (c'est-à-dire début de code) à un code binaire pour un autre caractère de l'alphabet.

L'intérêt de ce codage est de diminuer la taille moyenne des données transmises, ce qui veut dire qu'en moyenne, le code Huffman fait l'économie de nombreux caractères par rapport aux méthodes de transmission classiques. Par conséquent, la longueur des codes peut varier en fonction des caractères.

A l'image de l'alphabet Morse, l'intérêt de ce codage est d'attribuer des codes courts aux caractères fréquents (comme le E de la langue française) et des caractères plus longs aux caractères peu utilisés (comme le K de la langue française). Alors que le codage ASCII, classiquement utilisé, transmettait tous les caractères sur 8 bits (à savoir une suite de huit chiffres 0 ou 1), ce qui, pour un message de longueur n, implique de transmettre 8n

caractères, le code Huffman, lorsqu'il est bien employé, transmet souvent un nombre bien moindre de caractères.

Il existe trois versions différentes de ce codage :

- Codage de Huffman statique
- Codage de Huffman semi-adaptatif
- Codage de Huffman adaptatif

# III – Codage de Huffman à dictionnaire statique

## III.1 – Présentation du codage statique

Pour le codage de Huffman à dictionnaire statique, chaque caractère (ou octet) a déjà un code prédéfini et connu par les personnes qui s'échangent les messages. Par conséquent, il n'y a pas nécessité de transmettre le dictionnaire à son interlocuteur à chaque échange de messages.

## III.2 – Exemple avec le codage statique

Pour cet exemple, on considère l'alphabet gamma = « BACD » (utilisé dans des contrées lointaines). Alice et Bob ont convenu du dictionnaire de correspondances suivant : B=11, A=0, C=1000, D=10010.
Par conséquent, lorsque Bob envoie à Alice le message 1101000, Alice le déchiffre suivant la méthode suivante :

- Elle regarde le premier bit (1), et voit qu'aucun caractère n'a ce codage dans le dictionnaire.
- Elle y ajoute ensuite le deuxième bit (11), et voit que B possède le codage 11. Par conséquent, elle note que la première lettre du clair est un B, et peut ensuite chercher la lettre suivante. En effet, comme le codage Huffman est un code préfixe, elle est sûre qu'aucun autre caractère que B ne commence par le chiffré 11.
- Elle regarde ensuite le troisième bit (0), et voit que A possède le codage 0. Par conséquent, elle note que la deuxième lettre du clair est un A, et peut ensuite chercher la lettre suivante.
- Elle regarde ensuite le quatrième bit (1), et voit qu'aucun caractère n'a ce codage dans le dictionnaire.
- Elle y ajoute ensuite le cinquième bit (10), et voit qu'aucun caractère n'a ce codage dans le dictionnaire.
- Elle y ajoute ensuite le sixième bit (100), et voit qu'aucun caractère n'a ce codage dans le dictionnaire.

- Elle y ajoute ensuite le septième bit (1000), et voit que C possède le codage 1000. Par conséquent, elle note que la troisième lettre du clair est un C.
- Elle est arrivée à la fin du chiffré, et conclut donc que le message transmis est BAC.

# IV – Codage de Huffman à dictionnaire semi-adaptatif

## IV.1 – Présentation du codage semi-adaptatif

Pour le codage de Huffman à dictionnaire semi-adaptatif, l'algorithme prend en argument un alphabet et la fréquence de chacune des lettres de cet alphabet.

Elle construit l'arbre progressivement en fusionnant les deux nœuds les moins fréquents en un nouveau nœud intermédiaire dont la fréquence vaut la somme de ses deux feuilles. Une fois que tous les nœuds sont construits, les nœuds sont classés par longueur (correspondant aux nombres de branches les séparant du tronc) puis par ordre alphabétique.

Contrairement au dictionnaire statique, le codage semi-adaptatif est considéré comme optimal en termes de compression, dans le sens où il permet de bien représenter un message sur un nombre de bits très proche du minimum théorique.

## IV.2 – De nombreux dictionnaires équivalents

Le dictionnaire dépend de l'ordre d'entrée des lettres dans le texte qui a servi à le générer. Par conséquent, il existe de nombreux dictionnaires optimaux possibles, car, à chaque fusion de nœuds, on peut choisir d'inverser les codes des branches droites et des branches gauches. Il existe donc de nombreuses permutations d'un même codage de Huffman.

Ainsi, les utilisateurs du codage Huffman ont dû se mettre d'accord pour convenir d'une convention pour choisir arbitrairement une des permutations parmi ces nombreux dictionnaires équivalents, ce qui a donné lieu à la création du codage canonique.

## IV.3 – Présentation du codage canonique

Pour le codage canonique, le premier symbole dans le classement reçoit un code de longueur n (avec n le nombre de branches le séparant du tronc) et avec que des 0, puis le suivant aura (n-1) fois 0 puis un 1 (puis éventuellement autre chose s'il est plus long que le premier) et ainsi de suite, en veillant à ce qu'aucun des codes n'aient leur début correspondant au code d'une lettre plus courte.

Cette convention, usuellement employée dans les algorithmes, n'est pas toujours celle utilisée sur des sites de cryptographie (notamment Dcode, dont la convention sera détaillée ultérieurement).

Avec cette convention canonique, on peut même envisager ne fournir que la longueur du codage utilisé en guise de dictionnaire.

## IV.4 – Exemple de codage canonique

Pour cet exemple, on considère l'alphabet gamma = « BACD » (utilisée dans des contrées lointaines). Dans cette langue, la fréquence des lettres est B : 51 %, A = 15 %, D = 15 % et C = 19 %.

On construit le dictionnaire par la procédure suivante :

- On fusionne A et D pour former nœud_1 (30 %).
- On fusionne C et nœud_1 pour former nœud_2 (49 %).
- On fusionne B et nœud_2 pour former nœud_3 (100 %), qui correspond donc au tronc.

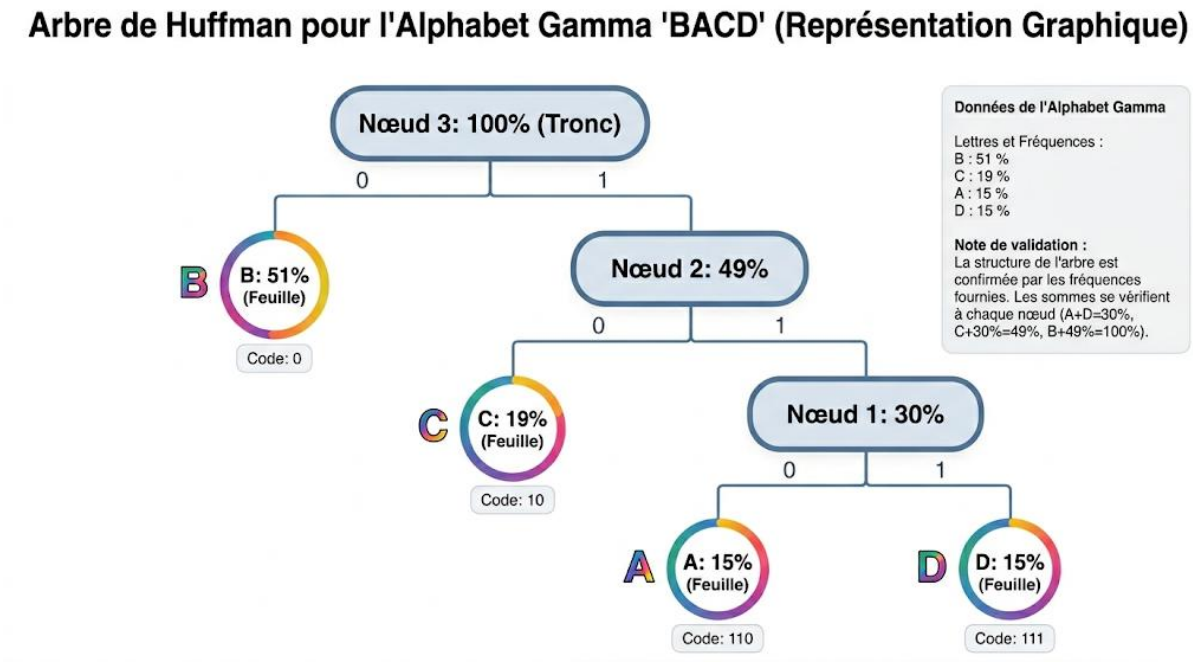


*Figure 2 - Construction du dictionnaire canonique (source : jaudi)*

Ainsi, les lettres présentes dans le dictionnaire sont donc B (codage de longueur 1 bit), C (2 bits), A (3 bits) et D (3 bits).

On en déduit alors le dictionnaire utilisé : B=0, C=10, A=110 et D=111.
Par exemple, BAC sera chiffré 011010 (6 bits, ce qui est un gain énorme par rapport aux 3 × 8 = 24 bits habituellement nécessaire pour représenter ce code par ASCII).

## IV.5 – Propriétés mathématiques du codage

De la définition du codage de Huffman donnée précédemment, on peut en déduire un

certain nombre de propriétés intéressantes sur cette méthode de compression, qui aident à comprendre plus en profondeur son fonctionnement logique et sa puissance.

Propriété 1 :
Soient deux caractères A et B, et leurs codes Huffman respectifs huff(A) et huff(B). Alors, le fait que le codage Huffman soit un code préfixe implique que, quels que soient A et B, huff(A) ne correspond pas au début de huff(B) et huff(B) ne correspond pas au début de huff(A).

Propriété 2 :
Soient deux textes T(1) et T(2), correspondant au chiffrement par le codage canonique d'un même texte clair T(0) mais par deux dictionnaires de Huffman distincts D(1) et D(2). Si les deux dictionnaires D(1) et D(2) sont équivalents, c'est-à-dire que, pour chaque caractère A, ses codages dans D(1) et D(2) ont la même longueur, alors le texte T(1) correspond à une substitution monoalphabétique du texte T(2).

Propriété 3 :

La fréquence $f_0$ des caractères dans les branches les plus éloignées de la racine est inférieure à toutes les autres fréquences $f_i$ des branches plus proches de la racine. On en déduit que : plus les feuilles (ou les nœuds) sont proches de la racine, plus leur fréquence est élevée et, de manière analogue, plus les feuilles (ou les nœuds) sont éloignées de la racine, plus leur fréquence est basse.

Propriété 4 :

Si le dictionnaire est optimal pour la chaîne de caractères donnée en entrée, alors tous les paquets d'un même niveau sont compris entre une fréquence $f_0$ et son double $2 \times f_0$.

Propriété 5 :

Pour le codage de Huffman semi-adaptatif optimal, la longueur idéale du codage d'un caractère correspond au logarithme en base deux de sa fréquence $\log_2(f_0)$, qui correspond à son entropie de Shannon. La longueur réelle du codage essaie de se rapprocher au maximum de cette longueur idéale, notamment lorsque tous les caractères sont représentés.

Propriété 6 :

Pour le codage de Huffman semi-adaptatif optimal, le cas le moins optimal arrive lorsque toutes les fréquences suivent la suite de Fibonacci (vérifiant $u_0 = 0$, $u_1 = 1$ et $\forall n \geq 2$, $u_n = u_{n-1} + u_{n-2}$). Pour un caractère A de fréquence $f_0$, cela implique donc l'encadrement suivant pour la longueur L(A) de son chiffré : $1 \leq L(A) \leq \log_\phi(f_0)$ (avec $\phi$ le nombre d'or).

Propriété 7 :
Soient deux caractères A et B. Si la fréquence du caractère A est $2^N$ fois plus grande que celle du caractère B, avec N = sup(n) tels que $freq(A) \geq 2^n \times freq(B)$ et que chaque fusion successive impliquant B reste inférieure à la fréquence de A, alors on en déduit que le caractère B possède un chiffré au moins N bits plus long que le caractère A.

Propriété 8 :

Dans la plupart des cas (ceux pour lesquels l'arbre est complet) pour le codage de Huffman semi-adaptatif (sauf en cas d'une distribution asymétrique comme celle de Fibonacci), si la fréquence $f_0$ d'un caractère A vérifie l'encadrement $2^n \leq 1/f_0 \leq 2^{n+1}$, alors le caractère A aura un chiffré de longueur n ou n+1 bits.

## IV.6 – Exemple : construction d'un dictionnaire canonique complexe

Pour cet exemple, on considère l'alphabet beta = « ABCEDFGHIJKLMNOPQRSTUVWXYZ_ », comprenant les 26 lettres et l'espace.

On utilise les fréquences françaises de cette alphabet, présentes dans le dictionnaire ci-dessous.

freq_fr = { "_": 17.4, "E": 14.33, "A": 6.77, "S": 6.55, "I": 6.22, "N": 5.92, "T": 5.77, "R": 5.49, "L": 4.89, "U": 4.73, "O": 4.57, "D": 3.31, "C": 2.75, "M": 2.45, "P": 2.41, "V": 1.15, "G": 0.90, "F": 0.89, "Q": 0.86, "H": 0.77, "B": 0.76, "X": 0.39, "J": 0.28, "Y": 0.26, "Z": 0.08, "K": 0.05, "W": 0.02 }

On réalise ensuite la construction de l'arbre de Huffman, en fusionnant successivement les deux éléments avec la plus faible fréquence :

- Étape 1 : Fusion de W (0.02) et K (0.05) → Nœud (0.07)
- Étape 2 : Fusion de Z (0.08) et du Nœud (0.07) → Nœud (0.15)
- Et ainsi de suite jusqu'à constituer l'arbre entier (représenté ci-dessous en deux parties).

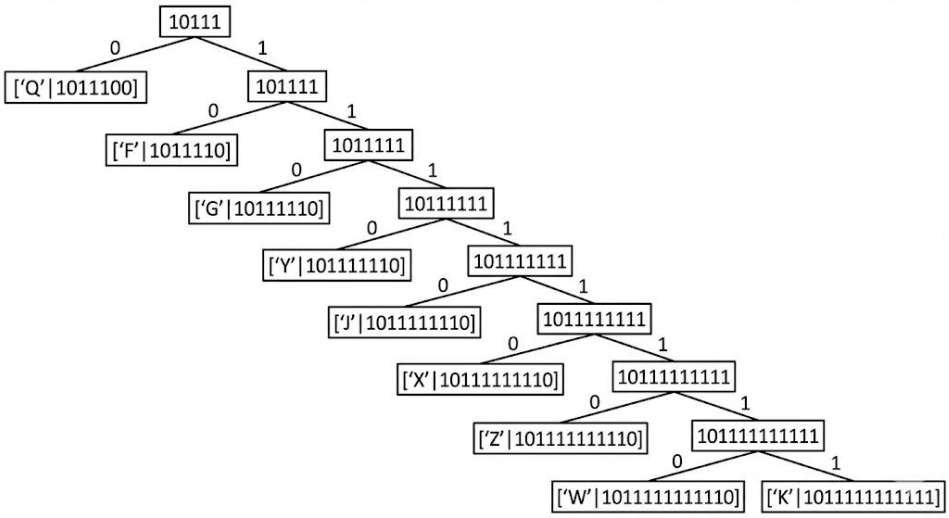


*Figure 3 - Construction de l'arbre pour l'alphabet beta à 27 caractères - partie 2 (source : jaudi)*

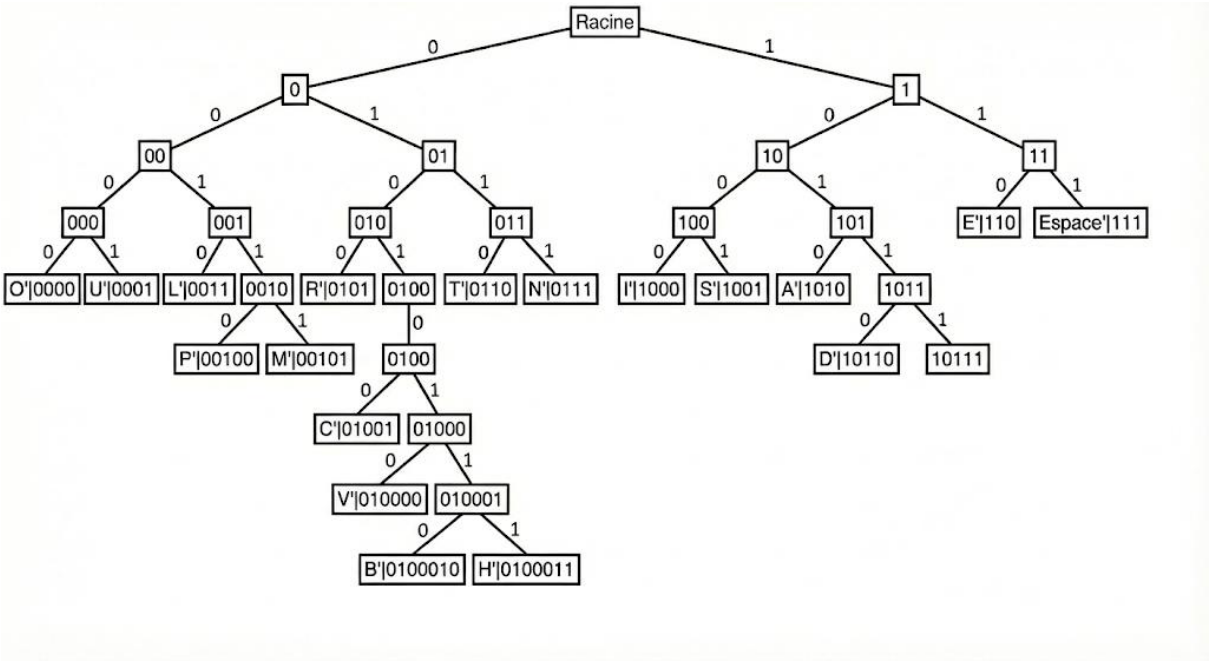


*Figure 4 - Construction de l'arbre pour l'alphabet beta à 27 caractères (source : jaudi)*

--- DICTIONNAIRE DE HUFFMAN OPTIMAL (FRANÇAIS) ---
'E': 110 (3 bits)
'Espace': 111 (3 bits)
'O': 0000 (4 bits)
'U': 0001 (4 bits)
'L': 0011 (4 bits)
'R': 0101 (4 bits)
'T': 0110 (4 bits)
'N': 0111 (4 bits)
'I': 1000 (4 bits)
'S': 1001 (4 bits)
'A': 1010 (4 bits)
'P': 00100 (5 bits)
'M': 00101 (5 bits)
'C': 01001 (5 bits)
'D': 10110 (5 bits)
'V': 010000 (6 bits)
'B': 0100010 (7 bits)
'H': 0100011 (7 bits)
'Q': 1011100 (7 bits)
'F': 1011101 (7 bits)
'G': 1011110 (7 bits)
'Y': 101111101 (9 bits)
'J': 101111110 (9 bits)

'X': 101111111 (9 bits)
'Z': 1011111001 (10 bits)
'W': 10111110000 (11 bits)
'K': 10111110001 (11 bits)

Comme il est possible d'inverser tous les 0 et les 1 (ce qui donne un facteur 2) et qu'il y a 2 caractères chiffrés par 3 bits, 9 par 4 bits, 4 par 5 bits, 1 par 6 bits, 5 par 7 bits, 0 par 8 bits, 3 par 9 bits, 1 par 10 bits ainsi que 2 par 11 bits, le nombre total de dictionnaires possibles vaut donc :
$2! \times 9! \times 4! \times 1! \times 5! \times 0! \times 3! \times 1! \times 2! \times 2 = 50\,164\,531\,200$

Ainsi, cette correspondance trouvée ne s'agit que d'une possibilité de dictionnaire optimal pour l'alphabet français avec espace parmi les 50 milliards de variantes possibles, ce qui rend le brute-force d'un tel dictionnaire quasiment impossible sans une convention claire.

# V – Codage de Huffman à dictionnaire adaptatif

## V.1 – Présentation du codage adaptatif

Pour le codage de Huffman à dictionnaire adaptatif, on n'a besoin d'aucune information sur le texte à transmettre, ni de l'alphabet, ni des fréquences des lettres. En effet, ce système permet de construire le code au fur et à mesure de la transmission des symboles, n'ayant aucune connaissance initiale de la distribution de la source, qui permet un codage et une adaptation à l'évolution des conditions dans les données.

Néanmoins, si une telle procédure présente l'avantage de pouvoir être codée en temps réel sans nécessité d'informations externes ni d'un partage de dictionnaire, elle présente l'inconvénient d'être ruinée si elle comporte ne serait-ce qu'une seule erreur de transmission. Par conséquent, elle implique de développer des procédures pour la détection et la correction d'éventuelles erreurs.

De plus, le codage adaptatif nécessite plus de puissance de calcul qu'un codage semi-adaptatif, mais en contrepartie, il permet souvent d'avoir un meilleur taux de compression.

Les principales implémentations de ce codage sont les algorithmes FGK et Vitter.

## V.2 – Algorithme FGK

L'algorithme FGK doit son nom aux initiales de Newton Faller (1973), Robert Gray Gallager (1978) et Donald Knuth (1985), trois chercheurs ayant successivement contribué à son élaboration et à son perfectionnement.

Cet algorithme construit et met à jour l'arbre de compression au fur et à mesure de la compression des données. Pour les caractères qui n'ont pas encore été rencontrés, le système réserve les nœuds « Not Yet Transmitted » (NYT).

Si le caractère est déjà dans l'arbre, on renvoie son chemin binaire. Sinon, on envoie le chemin binaire menant au nœud NYT puis son code binaire brut (ajout d'un bit, ou alors de son code ASCII suivant les conventions). On augmente ensuite la fréquence de ce caractère de 1, puis on actualise l'arbre. Si l'arbre de Huffman ne respecte plus ses propriétés (notamment la propriété selon laquelle plus les nœuds sont proches de la racine, plus ils sont fréquents), on le recalcule, ce qui implique de changer des nœuds ou des branches entières pour restaurer l'équilibre.

## V.3 – Propriétés et exemples de codage FGK

La variante classique du codage FGK consiste à ne procéder aux modifications de l'arbre que lorsque la fréquence d'un nœud (noté 1) devient strictement supérieure à celle d'autres nœuds de l'arbre. Dans ce cas-là, par souci d'efficacité et de simplicité du codage, seuls deux nœuds vont être intervertis : le nœud 1 va prendre la place de celui qui, parmi les nœuds qui sont moins fréquents que lui, possède le codage binaire le plus court.
En outre, pour cette variante, on initialisera systématiquement le NYT = 0 et on mettra toujours la lettre à gauche (ajouter le suffixe 0 au NYT), et le nouveau NYT à droite (ajouter le suffixe 1 au NYT).
Par conséquent, on observe bien qu'un tel codage va avoir un arbre qui s'allonge de plus en plus (avec des chiffrés très longs pour les caractères les moins fréquents), ce qui ne correspond pas à un codage optimal en terme de compression, mais qui est néanmoins astucieux et assez intéressant en terme de compression et de transmission de données sans avoir la nécessité de donner un dictionnaire au destinataire du message.

Par exemple, pour chiffrer le message ABBCDBAAAA, on emploie les étapes suivantes :
- NYT = 0 au début
- A = 00, NYT = 01
- B = 010, NYT = 011
- B = 010, NYT = 011
- On se rend compte que freq(B)=2 > freq(A)=1. Par conséquent, B et A inversent leurs codages (par la suite B = 00 et A = 010)
- C = 0110, NYT = 0111
- D = 01110, NYT = 01111
- B = 00, NYT = 01111
- A = 010, NYT = 01111
- A = 010, NYT = 01111
- A = 010, NYT = 01111
- On se rend compte que freq(A)=4 > freq(B)=3. Par conséquent, A et B inversent leurs codages (par la suite A = 00 et B = 010).
- A = 00, NYT = 01111

Par conséquent, le message chiffré final sera 00/010/010/0110/01110/00/010/010/010/00 soit 00010010011001110000100100100 0.

Autre exemple, pour chiffrer CABAS_A, on emploie les étapes suivantes :
- NYT = 0 au début
- C = 00, NYT = 01
- A = 010, NYT = 011
- B = 0110, NYT = 0111
- A = 010, NYT = 0111
- On se rend compte que freq(A)=2 > freq(C)=freq(B)=1. Par conséquent, la lettre A va être intervertie avec celui qui possède le plus petit codage parmi B et C (il s'agit de C car son codage binaire est de longueur 2, contre 4 pour celui de B). Par conséquent, pour la suite A = 00 et C = 010.
- S = 01110, NYT = 01111
- _ = 011110, NYT = 011111
- A = 00, NYT = 011111

Par conséquent, le message chiffré final est donc : 000100110010011110011110 00

## V.4 – Algorithme Vitter

L'algorithme Vitter, créé par Jeffrey Vitter en 1987, est une amélioration de l'algorithme FGK qui est devenu une méthode de référence pour la compression adaptative. En effet, à force d'échanges basiques, l'arbre peut parfois devenir très « haut » et déséquilibré (car chaque nouveau caractère allonge l'arbre vers le bas, ce qui conduit l'algorithme à être sous-optimal), ce qui créé des chemins longs, et donc des codes binaires moins efficaces.

L'amélioration est donc de mettre une règle de tri beaucoup plus stricte, à savoir qu'une lettre est prioritaire sur un nœud, à fréquence égale, pour forcer l'arbre à rester le plus « large » et équilibré possible. Par conséquent, si une lettre gagne en fréquence, elle glisse pour passer automatiquement devant tous les embranchements qui ont le même poids qu'elle (au lieu de simplement échanger sa place avec le nœud situé au-dessus d'elle).

Grâce à cette règle de priorité stricte, l'arbre s'élargit au lieu de s'allonger et les codes générés seront toujours très proches de l'optimum absolu.

## V.5 – Comparaison des codages adaptatifs FGK et Vitter

On va à présent étudier la transmission du message ABBCD par codage adaptatif, en utilisant la convention selon laquelle la lettre part à gauche (bit=0) comme dans la convention canonique du codage semi-adaptatif :

- Au début, comme aucun caractère n'a été rencontré, on a seulement NYT qui vaut 0
- Lorsqu'on rencontre A, on crée A = 00 (fréquence = 1) et NYT = 01

- Lorsqu'on rencontre B, on garde A = 00 (fréquence = 1) mais on crée B = 010 (fréquence = 1) et NYT = 011
- Lorsqu'on rencontre B, on passe à fréquence = 2 pour B, qui est supérieur à fréquence = 1 pour A. On inverse donc A et B, pour obtenir : B = 00 (fréquence = 2), A = 010 (fréquence = 1) et NYT = 011
- Lorsqu'on rencontre C, on garde B = 00 (fréquence = 2), A = 010 (fréquence = 1) et on crée C = 0110 (fréquence = 1) et NYT = 0111
- Lorsqu'on rencontre D, cela permet de mettre en valeur les différences entre les algorithmes FGK et Vitter.
  Pour FGK, on obtient B = 00 (fréquence = 2), A = 010 (fréquence = 1), C = 0110 (fréquence = 1), D = 01110 (fréquence = 1) et NYT = 01111. Néanmoins, la fréquence du nœud 011 (comprenant C de fréquence 1 et D de fréquence 1) passe à 2, ce qui est supérieur à fréquence = 1 pour A. On inverse donc le nœud et A pour avoir A = 011 et le nœud à 010 qui donne : C = 0100 (fréquence = 1), D = 01010 (fréquence = 1) et NYT = 01011. Puis on réalise que la fréquence du nœud 01 (comprenant A, C et D) passe à 3, ce qui est supérieur à fréquence = 2 pour B. On inverse donc le nœud et B pour avoir B = 01 et le nœud à 00 qui donne : A = 001, C = 0000, D = 00010 et NYT = 00011.
  Pour Vitter, on obtient B = 00 (fréquence = 2), A = 010 (fréquence = 1), C = 0110 (fréquence = 1), D = 01110 (fréquence = 1) et NYT = 01111. Néanmoins, la fréquence du nœud 011 (comprenant C de fréquence 1 et D de fréquence 1) passe à 2, donc on va pouvoir créer une branche parallèle à B (branche de code 1), et faire glisser A vers le haut. Par conséquent, on garde B = 00 (fréquence = 2), mais on bénéficie d'un glissement pour obtenir A = 01 (fréquence = 1), une nouvelle branche avec C = 10 (fréquence = 1), D = 110 (fréquence = 1) et NYT = 111.

Pour FGK, on observe bien que la structure de l'arbre n'est pas du tout optimale car, si les caractères les plus fréquents montent naturellement vers le haut de l'arbre, cela se fait au détriment des caractères les moins fréquents qui sombrent dans de longs chiffrés. Par exemple, dans notre cas, les plus longs font 5 bits.
A contrario, pour Vitter on conserve bien une structure optimale car aucun code ne dépasse 3 bits.

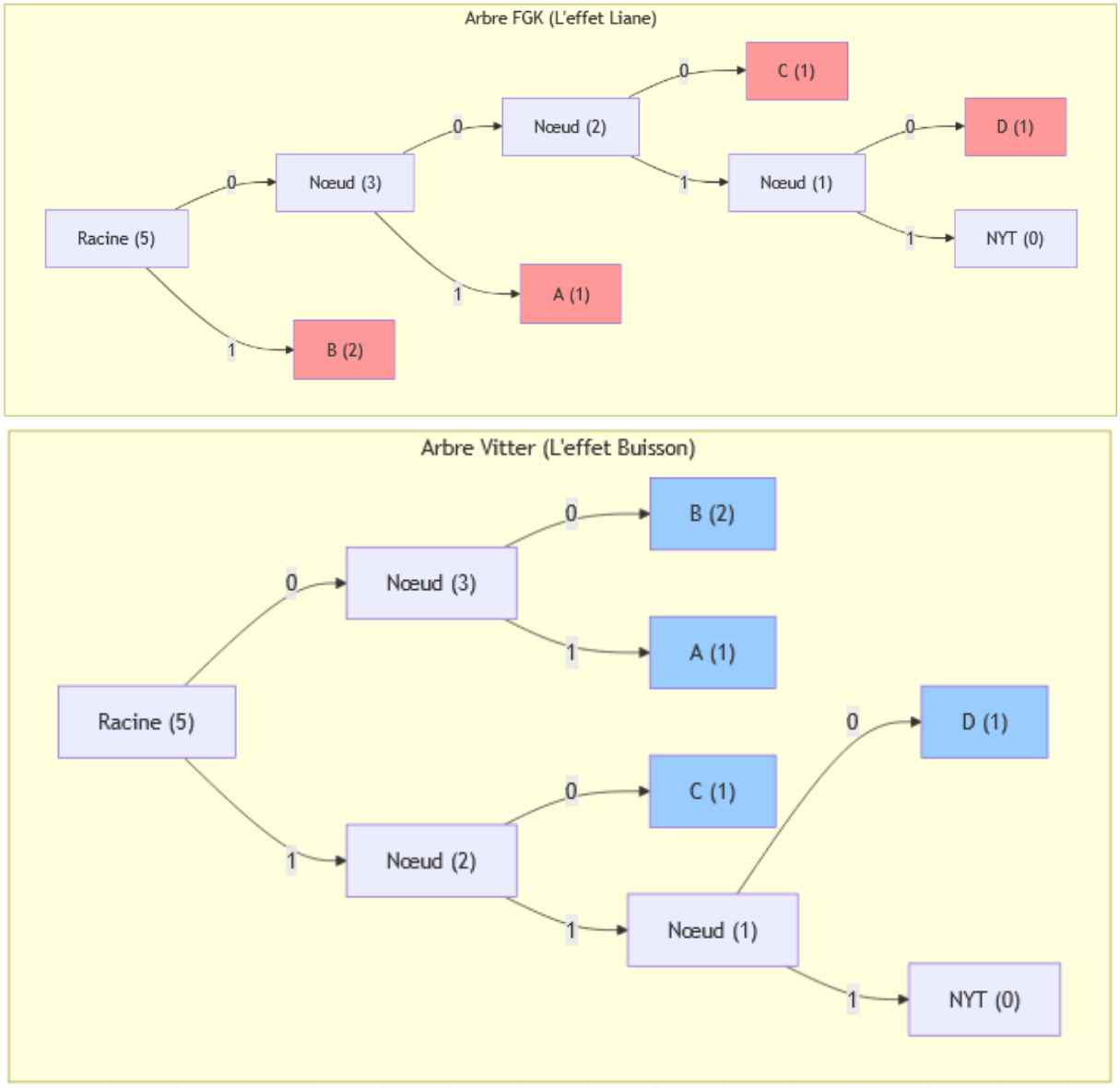


*Figure 5 - Comparaison des arbres des algorithmes FGK et Vitter (source : jaudi)*

# VI – Analyse de l'outil du site Dcode

## VI.1 – Fonctionnement de l'outil

A présent, nous allons analyser un peu plus en détail l'outil « Codage de Huffman » du site Dcode.

Tout d'abord, Dcode implémente seulement la version semi-adaptative du codage de Huffman. En outre, il n'utilise pas du tout la convention du codage de Huffman canonique (et par conséquent, cet outil n'est pas adapté si vous devez décrypter des énigmes utilisant la convention canonique du codage).

En effet, quand plusieurs nœuds ont la même fréquence : le premier nœud rencontré (index le plus petit) avec le poids minimal devient enfant gauche, le deuxième nœud rencontré avec le même poids devient enfant droit. C'est donc l'ordre d'insertion initial (l'ordre des caractères) qui détermine quelle lettre sera à gauche ou à droite, donc qui aura un code commençant par 0 ou par 1.

Cet outil utilise le même chiffrement que celui décrit précédemment. Il prend en argument une chaîne de caractères, calcule les fréquences des différents caractères dans la chaîne fournie, et construit ensuite l'arbre selon la procédure expliquée ci-dessus.

## VI.2 – Premier exemple : entrée ABCDEFGHIJKLMNOPQRSTUVWXYZ

Par exemple, pour l'entrée ABCDEFGHIJKLMNOPQRSTUVWXYZ, toutes les fréquences des caractères sont identiques et égales à 1/26. Par conséquent comme $2^4 \leq 26 \leq 2^5$ et que toutes les fréquences sont égales, un certain nombre de lettres va être représenté sur 4 bits, et le reste sur 5 bits. En effet, comme la fréquence des lettres est égale, l'algorithme cherche juste à minimiser la somme des longueurs des bits (il s'agit d'un cas très particulier du codage de Huffman).

L'algorithme de Dcode va à présent regrouper les nœuds les moins fréquents pour construire l'arbre. Sa convention est ici de commencer par le début de la chaîne. Il regroupe A et B sous le même nœud (paquet 1), puis C et D (paquet 2)... jusqu'à regrouper S et T (paquet 10).
A présent, il dispose de 10 paquets de deux lettres (de fréquence 2/26) et de 6 paquets de 6 lettres uniques (de fréquence 1/26), soit un total de 16 paquets. On vérifie alors que tous les paquets d'un même niveau sont compris entre une fréquence $f_0$ et son double $2 \times f_0$, ce qui correspond au fait que le dictionnaire est optimal pour la chaîne de caractères donnée en entrée. Tous les paquets sont représentés sur 4 bits, et les lettres uniques seront donc codées sur 4 bits, tandis que les lettres doubles auront 4 bits + 1 bit de suffixe (0 ou 1) soit 5 bits au total.
On peut se convaincre de la véracité de ce raisonnement en inversant deux lettres dans cet alphabet (par exemple RBCDEFGHIJKLMNOPQASTUVWXYZ inverse juste les rôles et codages de R et de A)

## VI.3 – Deuxième exemple : entrée AABCDEFGHIJKLMNOPQRSTUVWXYZ

Pour l'entrée AABCDEFGHIJKLMNOPQRSTUVWXYZ, on a 25 lettres de fréquence 1/27 et une lettre (le A) de fréquence 2/27.
De manière analogue à précédemment, l'algorithme regroupe les plus basses fréquences (en commençant arbitrairement par le début de l'alphabet). Il regroupe B et C sous le même nœud (paquet 1), puis D et E (paquet 2)... jusqu'à regrouper T et U (paquet 10). A présent, il dispose de 10 paquets de deux lettres (de fréquence 2/27) et de 6 paquets comprenant 6

lettres uniques (5 de fréquence 1/27 et le A de fréquence 2/27), soit un total de 16 paquets. On vérifie alors que tous les paquets d'un même niveau sont compris entre une fréquence $f_0$ et son double $2 \times f_0$, ce qui correspond au fait que le dictionnaire est optimal pour la chaîne de caractères donnée en entrée. Tous les paquets sont représentés sur 4 bits, et les lettres uniques seront donc codées sur 4 bits, tandis que les lettres doubles auront 4 bits + 1 bit de suffixe (0 ou 1) soit 5 bits au total.

## VI.4 – Troisième exemple : entrée AAAAAAAAAAABCDEFGHIJKLMNOPQRSTUVWXYZ

Puis, pour l'entrée AAAAAAAAAAABCDEFGHIJKLMNOPQRSTUVWXYZ, on a 25 lettres de fréquence 1/36 et une lettre (le A) de fréquence 11/36.
Comme $2^5 \leq 36 \leq 2^6$, tous les caractères de fréquence 1/36 seront codés sur 5 ou 6 bits, et, pour A, comme $2^1 \leq 36/11 \leq 2^2$, il sera donc codé sur 2 bits.
De manière analogue à précédemment, l'algorithme regroupe les plus basses fréquences (en commençant arbitrairement par le début de l'alphabet). Il regroupe B et C sous le même nœud (paquet 1), puis le paquet 1 avec D (paquet 2), puis E et F (paquet 3), puis G et H (paquet 4)... jusqu'à regrouper Y et Z (paquet 12). A présent, on a 12 au même niveau (compris entre $f_0$ et $2 \times f_0$ comme mentionné précédemment), 11 de fréquence 2/36 et 1 de fréquence 3/36 (et $2 \times 2/36$ est inférieur à la fréquence de A, qui ne rentre donc toujours pas dans l'arbre).
On regroupe ces 12 paquets (deux à deux) en 6 paquets de fréquence comprise 4/36 et 5/36 (et $2 \times 4/36$ est inférieur à la fréquence de A, qui ne rentre donc toujours pas dans l'arbre).
On regroupe ces 6 paquets en 3 paquets de fréquence : deux de fréquence 8/36, un de fréquence 9/36, auquel on ajoute A de fréquence 11/36, soit un total de 4 paquets. On vérifie alors que tous les paquets d'un même niveau sont compris entre une fréquence $f_0$ et son double $2 \times f_0$, ce qui correspond au fait que le dictionnaire est optimal pour la chaîne de caractères donnée en entrée.
Tous les paquets sont représentés sur 2 bits, et A est donc directement codé sur 2 bits, les autres lettres uniques seront donc codées sur 2+3=5 bits, tandis que les lettres doubles du paquet 1 (B et C) auront 2+3+1=6 bits au total.
On vérifie bien ici la propriété que $f(A) > 2^4 \times f(B) = 2^4 \times f(C) = ... = 2^4 \times f(Z)$ se traduit bien par le fait que les caractères les moins fréquents ont un codage de 4 bits plus long que le caractère le plus fréquent (ici A).

## VI.5 – Quatrième exemple : entrée EASINTRLUODCMPVGFQHBXJYZKW

L'entrée EASINTRLUODCMPVGFQHBXJYZKW (classée par fréquences de langue française) se ramène simplement à une permutation de l'alphabet du cas 1 (avec X, J, Y, Z, K et W représentés par 4 bits et tout le reste des lettres représentées par 5 bits).

# Bibliographie

Pour ceux parmi vous qui sont désireux d'approfondir ce système, voici certains des nombreux articles que j'ai consultés (en avril 2026) pour écrire ce cours de cryptographie :

Ressources en français :
https://fr.wikipedia.org/wiki/David_Albert_Huffman
https://fr.wikipedia.org/wiki/Codage_de_Huffman
https://liora.io/codage-de-huffman-tout-savoir
https://www.dcode.fr/codage-huffman-compression
http://obligement.free.fr/articles/algorithmes_compression_donnees_huffman.php
https://www1.ucsc.edu/currents/99-00/10-11/huffman.html

Ressources en anglais :
https://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf
https://en.wikipedia.org/wiki/David_A._Huffman
https://en.wikipedia.org/wiki/Huffman_coding
https://www.dcode.fr/huffman-tree-compression
https://www.ittc.ku.edu/~jsv/Papers/Vit87.jacmACMversion.pdf
https://www2.cs.duke.edu/csed/curious/compression/adaptivehuff.html#tree
http://paper.ijcsns.org/07_book/200901/20090145.pdf