
# L'algorithme de Hu-Tucker

Projet M1 ANDROIDE

Benjamin Decker & Nathan Guetteville

May 2025

# Contents

# 1 Introduction

L'algorithme de Hu-Tucker est un algorithme de compression qui permet de construire un arbre binaire de recherche alphabétique. Pour rappel, un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci. Un arbre binaire de recherche alphabétique possède de plus la caractéristique que toutes ses feuilles sont ordonnées par leur ordre d'apparition dans la séquence de construction de l'arbre.
Les possibilités offertes par l'implémentation de cet algorithme sont multiples. En effet, la plus évidente est la compression de dictionnaires mais il est également possible d'utiliser cet algorithme pour répondre à certains enjeux en interaction humain machine. Par exemple, l'implémentation d'outils permettant à des personnes en situation de handicap d'utiliser leur ordinateur avec le seul mouvement de leurs yeux.

# 2 Cahier des charges

Le but de ce projet est de comprendre le fonctionnement et d'implémenter l'algorithme de Hu-Tucker en Python. Pour cela, nous procèderons par étapes :

1. Implémentation de l'algorithme de Huffman dont l'algorithme de Hu-Tucker est dérivé afin de mieux comprendre son fonctionnement.
2. Implémentation d'une version naïve de l'algorithme de Hu-Tucker
3. Optimisation de l'algorithme
4. Évaluation et comparaison de performance des différentes versions de l'algorithme

Il s'agit là du travail essentiel à réaliser mais d'autres pistes d'approfondissement s'offrent à nous :

- Comparaison de performance entre notre implémentation de l'algorithme et d'autres implémentations déjà existantes notamment dans d'autres langages.
- Généralisation de l'algorithme à des arbres n-aires
- ...

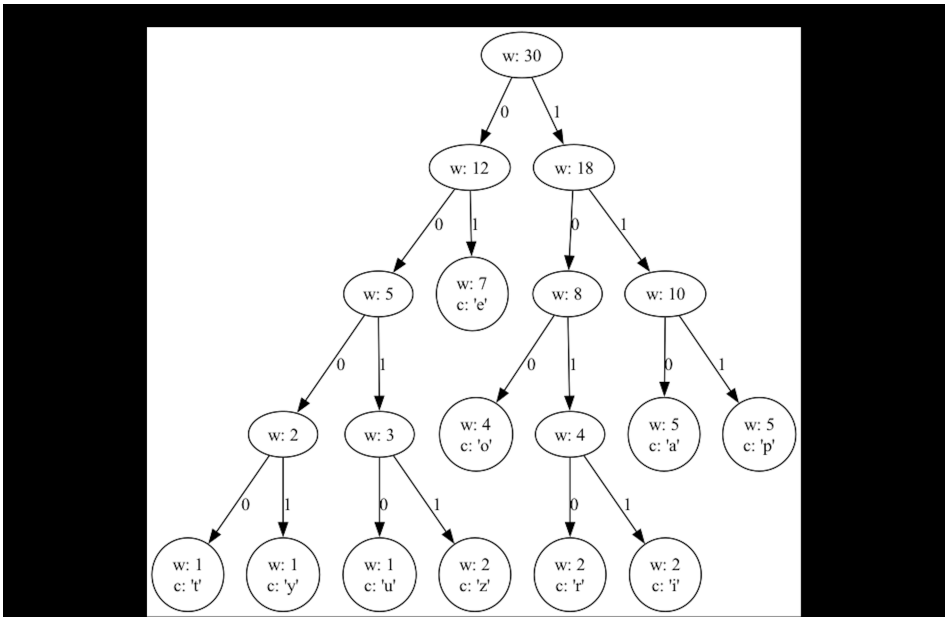
# 3 Généralités

Les algorithmes présentés ici sont des algorithmes de compressions s'appuyant sur les arbres binaires comme structure de données principale. Nous avons implémenté cette structure en utilisant la programmation orientée objet dans le fichier *tree_struct.py*. De plus, dans le fichier *utils.py* nous avons implémenté les méthodes permettant d'initialiser la séquence initiale et les occurences des caractères utilisés par l'algorithme de Huffman et de Hu-Tucker. Les algorithmes sont eux implémentés dans les fichiers $huffman.py$, $hu\_tucker\_naive.py$ et $hu\_tucker\_opt.py$. Enfin, les fichiers *display_tree.py* et *benchmark.py* servent pour les tests. Le premier permet d'afficher les arbres intermédiaires et l'arbre final généré par les deux algorithmes et le second permet de tester leur performance sur un corpus de textes contenu dans le dossier *corpus_tests*.

# 4 Algorithme de Huffman

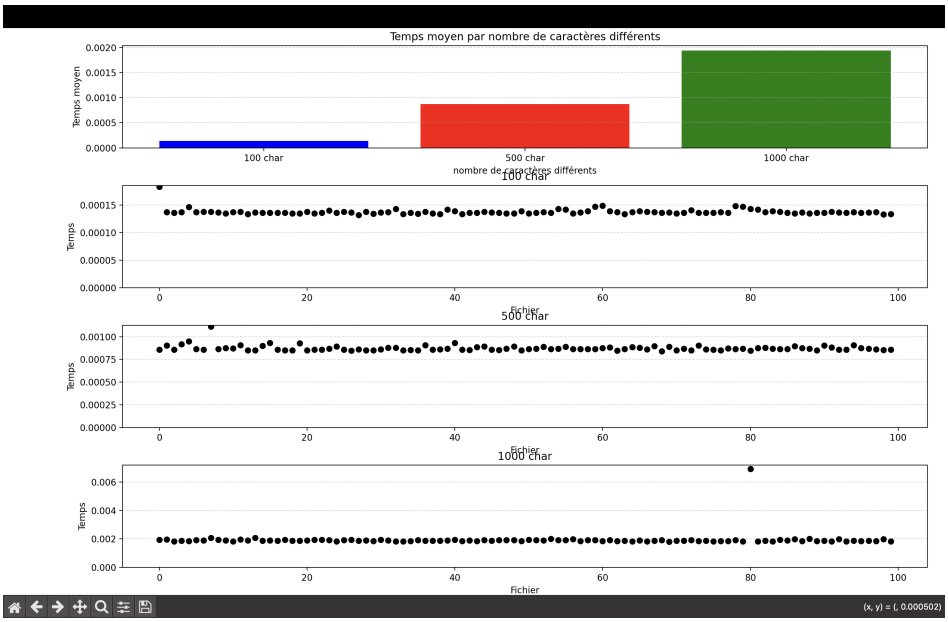
L'algorithme de Huffman est un algorithme de compression qui permet de construire un arbre binaire de codage optimal en fonction des fréquences d'apparition des symboles. Il minimise le coût total du codage en associant aux symboles les plus fréquents les codages les plus courts et garantit qu'aucun codage n'est le préfixe d'un autre, facilitant le décodage. De plus, le codage de Huffan s'approche de l'entropie. C'est à dire que pour une source $X$ d'entropie de Shannon $H(X) = -\sum p_i log_2(p_i)$ qui correspond aux longueurs idéales des mots, on a $H(X) \leq L < H(X)+1$, avec $L$ la longueur moyenne d'un mot obtenu par le codage de Huffman. En effet, $L = \sum l_i p_i$ avec $l_i$ est la longueur (en bits) du code attribué au symbole $i$. Or Huffman utilise des entiers donc chaque code $l_i \geq \lceil -log_2(p_i) \rceil$ et donc $-\sum p_i log_2(p_i) \leq \sum p_i l_i < \sum p_i (log_2(p_i) + 1)$.

Cela montre que Huffman est au plus 1 bit au-dessus par symbole que l'entropie mais qu'il ne peut pas faire mieux.




Arbre obtenu par l'algorithme de Huffman avec une fréquence d'apparition des symboles (5;2;7;2;1;1;1;2;4;5)

Nous avons implémenté deux méthodes pour la construction de l'arbre de Huffman dans le fichier $huffman.py$. La première, $build_huffman_tree$ fusionne succécivement les paires de noeuds de poinds le plus faible jusqu'à ce qu'il n'en reste qu'un mais utilise la fonction $min$ de la bibliothèque python standard ce qui lui fait perdre en efficacité puisqu'elle a une complexité en $O(n^2)$. La seconde, $build_huffman_tree2$, fait de même mais en utilisant un tas binaire permettant d'avoir une complexité en $O(nlog(n))$. Enfin, la fonction $huffman_code$ permet de générer le codage de la séquence initiale à partir de l'arbre créer précédemment. On peut voir grâce au benchmark que la complexité obtenue est conforme à nos attente :



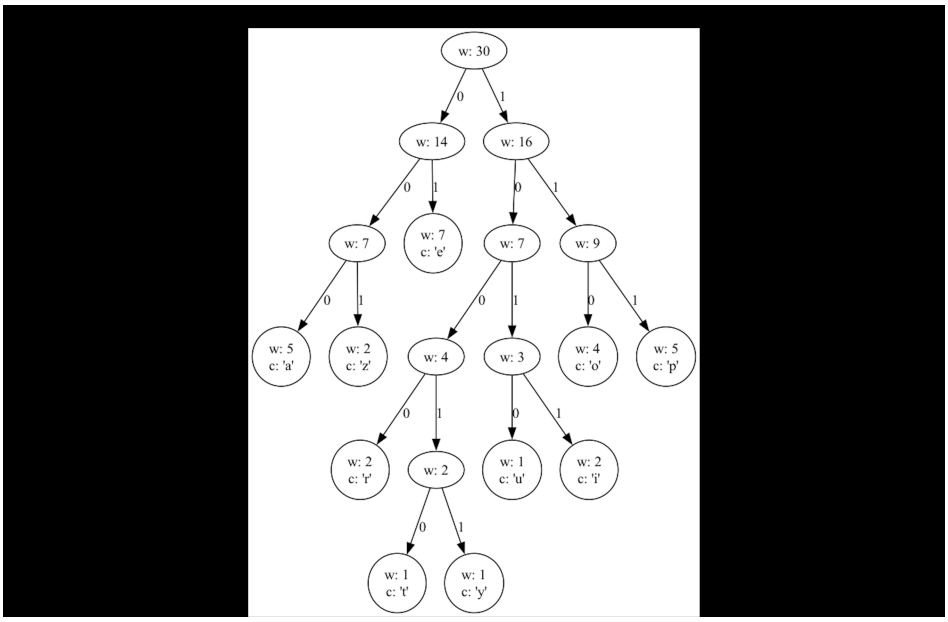

# 5 Algorithme de Hu-Tucker

## 5.1 Implémentation naïve

L'algorithme de Hu-Tucker est un algorithme de compression qui construit un arbre binaire de codage optimal pour un alphabet ordonné tout en respectant l'ordre d'insertion des symboles. Il est similaire à l'algorithme de Huffman mais avec une contrainte d'ordre. Il fonctionne en trois grande phases.
L'implémentation naïve de l'algorithme est assez facile à réaliser à l'instinct. Pour la phase 1, on utilise la fonction *combination* qui à chaque itération forme le noeud parent des deux noeuds non séparés par un noeud externe de poids combiné le plus faible. Cette sélection est effectuée par la fonction *select_args_nodes_to_merge* qui parcours la liste de noeuds plusieurs fois, ce qui entame l'efficacité de l'algorithme. En effet, cette fonction a une complexité en $O(n^2)$ avec $n$ le nombre de symboles dans la séquence initiale. C'est ici qu'on a le plus de temps à gagner car on teste dans le pire des cas la compatibilité de toutes les paires de noeuds de la séquence. Le noeud parent est ensuite placé à la position du premier de ses fils, et le deuxième est retiré de la liste, assurant ainsi la terminaison de la phase. En effet, à la fin, il ne reste plus que le noeud racine de l'arbre, contenant toute l'arborescence.

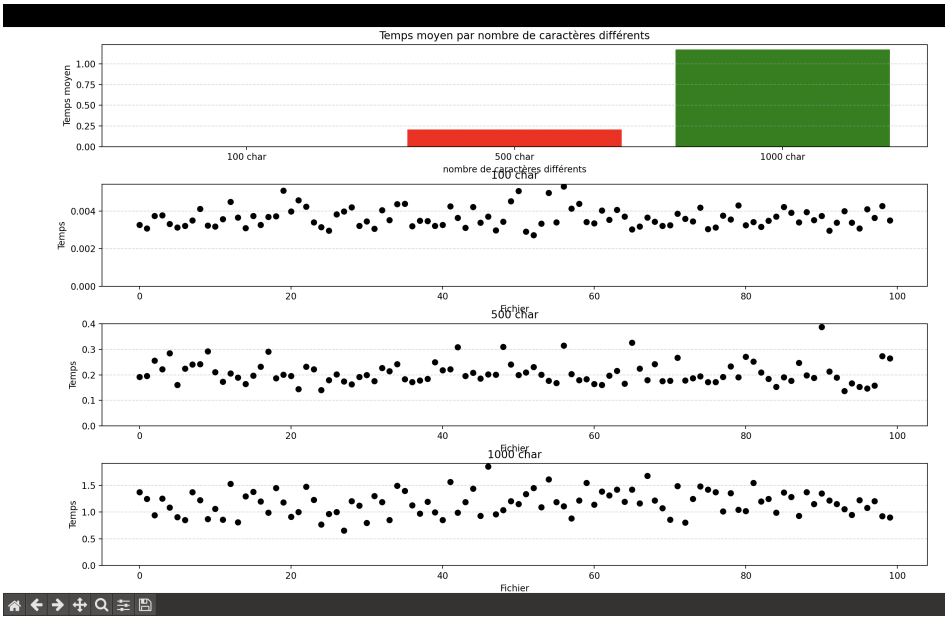
La phase 2 est la phase la plus simple, et elle ne change pas selon l'efficacité de l'algorithme. La fonction *level_assignment* effectue un parcours en profondeur sur l'arbre obtenu en phase 1 et affecte les niveaux adéquats à ses feuilles. Cette phase a donc une complexité en $O(n)$ avec $n$ le nombre de noeuds grâce à l'utilisation d'un dictionnaire pour retrouver une feuille en temps $O(1)$.

En phase 3, la fonction *recombination* repart de la séquence de feuilles initiale et selectionne tour à tour les paires de noeuds de niveau le plus élevé selon la phase 2 et les itérations précédentes. Ceci grâce à la fonction *select_args_nodes_to_merge_levels* qui, dans laquelle on fait bien attention à ce que les deux noeuds selectionnés soient consécutifs dans la séquence. Comme dans la phase 1, on construit le noeud parent de niveau inférieur et de poids le cumul de ceux de ses enfants, et on le place à la position du premier de ceux-ci et le deuxième est retiré de la liste. Il ne reste à la fin plus que le noeud racine de l'arbre d'HU-Tucker, dont il ne nous reste plus qu'à étiqueter les branches avec des 0 et des 1 pour obtenir notre code. La fonction *select_args_nodes_to_merge_levels* parcours une fois la liste de niveaux des feuilles et a donc un complexité en $O(n)$ et elle est appelée $n-1$ fois dans la fonction *recombinaison* ce qui donne à la phase 3 une complexité en $O(n^2)$. Cette phase est tout comme la première optimisable en utilisant ici une file et une pile.




Arbre final obtenu par l'algorithme de Hu-Tucker avec la même séquence initiale que l'exemple de Huffman. On remarque ici que l'ordre des symboles a été respecté (azertyuiop) mais qu'il a une hauteur supérieure à l'arbre obtenu par Huffman.

Au final, puisque les phases 1 et 3 ont une complexité en $O(n^2)$, la version naïve de l'algorithme de Hu-Tucker aussi. Ce résultat est cohérent avec les observations obtenues grâce au benchmark. En effet, en moyenne on observe une multiplication du temps de calcul d'environ 25 entre les fichiers de 100 et de 500 caractères et d'un peu plus de 4 entre ceux de 500 et 1000 caractères.



## 5.2 Implémentation efficace

Nous venons de voir que nous pouvions gagner en efficacité sur les phases 1 et 3 de l'algorithme en changeant les structures de données utilisées. Nos implémentations améliorées se trouvent dans le fichier *hu_tucker_opt.py*.
Pour la phase 1, le but est de ne plus tester toutes les paires de noeuds fusionnables mais de comparer la somme des poids des noeuds deux à deux adjacents, de sélectionner ceux dont la somme est minimale et de les fusionner en un nouveau noeud. On réitère alors ce principe jusqu'à ce qu'il n'en reste qu'un seul. Pour faire cela efficacement, on peut utiliser des tas binaires. On initialise $n-1$ tas qui contiennent chacun la somme des poids de deux noeuds adjacents, on appelle ces tas les HPQs. Et on initialise également un autre tas qui joue un rôle de liste de priorité contenant les HPQs précédemment initialisées. On peut alors sortir de la MPQ la HPQ qui contient les noeuds dont la somme est minimale. On supprime ces noeuds de toutes les HPQs auxquelles ils participent et on fusionne ces HPQs pour en faire une nouvelle et on supprime les HPQs fusionnées de la MPQ. On fusionne également les noeuds en un nouveau que l'on insère dans la nouvelle HPQ qu'on insère elle dans la MPQ.
En théorie, grâce aux opérations sur les tas (*heappop* et *heappush*) qui s'exécutent en temps $O(log(n))$, on obtient au final une complexité de $O(nlog(n))$ pour la phase 1. Cependant, notre implémentation de cette version de l'algorithme n'atteind pas cette complexité car certaines de nos opérations sont en $O(n)$ comme la suppression des éléments des HPQs ou les opérations sur les listes. Pour résoudre ça nous aurions notamment dû ajouter des sauvegardes d'indices dans les HPQs, ce que nous n'avons pas réussi à faire à temps. Au final, notre implémentation optimisée de la phase 1 a toujours une complexité en $O(n^2)$.
Pour ce qui est de la phase 3, on utilise maintenant une pile et une file. On commence par remplir la file avec les feuilles et leur niveau puis tant que la pile à une taille inférieure à 2 on défiles les élement pour les empiler. Ainsi, si les deux derniers élément de la pile ont le même niveau, on défile un élément pour l'empiler au dessus des deux derniers de la pile. Sinon, les dépile et on empile un nouveau noeud dont le poids est la somme des poids des deux noeuds dépilés puis on empile ce nouveau noeud au niveau d'au dessus. On répète cette opération jusqu'à ce que la file soit vide ou que la pile n'ai qu'un élément. Les opérations sur les piles et les files comme *pop*, $popleft$, *append* et $appendleft$ s'exécutent en temps constant $O(1)$. De ce fait, cette nouvelle implémentation de la phase 3 a une complexité en $O(n)$.
Nous avons réalisé à la fin du projet que dans certains cas, la fonction *recombinaison* n'avait pas le comportement attendu. Nous n'avons pas réussi à déterminer qu'elles étaient les conditions exactes de ces echecs. Ci-joint, veuillez trouver le pseudo code de la fonction :

**Hu-Tucker Algorithm Phase III, Recombination**

*Initialize*

$Q[N] = T[N]$

S=empty

*Main Loop*

**while** ( !(($Q$ is empty) and (($size\_of(S)$) == 1))))

  **if** (($size\_of(S)$) $\geq 2$) and (the top 2 elements have the same levels)

  **then**

    $level_{top} = level\_of\_element(pop(S))$

    Combine those two elements of the queue $Q$, whose levels
      are at the top of the stack.

    Remove the element from the top of the stack: $pop(S)$

    Remove the element from the top of the stack: $pop(S)$

    Push an element with $level = level_{top} - 1$ on the top of the stack:
      $push(S.level_{top} - 1)$

  **else**

    Remove an element from the front of the queue $Q$
      and push it on the top of the stack $S$: $push(S, pop(Q))$

## 6 Implémentation des tests

Les tests de performance des différents algorithmes sont implémentés dans le fichier *benchmark.py* et utilisent les fichiers textes dans le dossier $corpus_tests$. Ce corpus contient un total de 300 textes chacun ayant un certains nombre de caractères distincts (100, 500 ou 1000, 100 textes par taille). Un fichier dans le sous dossier 100_*char* peut avoir une taille supérieure à 100 caractères, cela signifie qu'il contient 100 caractères différents mais certains peuvent se répéter. Tous les textes ont été générés aléatoirement selon une loi de probabilité uniforme sur les caractères selectionnés. Dans le fichier *benchmark.py* on trouve une fonction par algorithme à tester. Elles sont toutes écrites de la même façon afin de garantir une homogénéité dans les calculs de performance. L'ouverture et la fermeture des fichiers textes ainsi que le calcul des occurences de caractères et la création de la séquence initiale ne sont pas comptabilisés. Seules les fonctions directement présentes dans les fichiers des algorithmes sont testées afin d'évaluer leur efficacité. Les résultats sont inscrits dans des fichiers csv (un par fonction de test) dans lesquels on retrouve à chaque ligne, l'ensemble des temps de calcul pour les fichiers d'une certaine taille. De plus, les résultats sont stockés dans différents tableaux afin de calculer des moyennes et de générer des graphiques pour comparer les résultats entre eux.

## 7 Conclusion

Ce projet nous a permis de nous intéresser aux algorithmes de compression utilisant un arbre binaire, tels que Huffman et Hu-Tucker. Si le premier est plus efficace, le deuxième à la particularité de conserver l'ordre de la séquence d'origine, ce qui constitue un avantage non négligeable. Une implémentation naïve nous a permis de nous failiariser avec l'algorithme, avant d'implémenter une version semi-optimale s'appuyant sur des techniques moins gourmandes, et une large batterie de tests permettant de s'assurer des propriétés de l'algorithme. Avec un peu plus de travail, nous aurions pu implémenter nos propres structures de données permettant d'atteindre la complexité optimale de l'algorithme. L'étape suivante théorique de notre travail serait,

après une comparaison avec d'autres implémentations existant dans d'autres lagages, d'étudier la possibilité de développer une version de cet algorithme permettant d'utiliser des arbres non binaires pour la compression, ou encore de packager notre travail pour pouvoir l'intégrer dans la bibliothèque de Python.

# 8 Bibliographie

Hu-Tucker alogorithm for building optimal alphabetic binary search trees, Sashka Davis, 1998
A Method for the Construction of Minimum-Redundancy Codes, David A. Huffman, 1952
Bayesian Information Gain, Julien Gori