Master 1 STL — Année 2025/2026

Algorithmique Avancée — Devoir de Programmation

Huffman Dynamique (FGK/Vitter)

Rapport + rendu logiciel (compression/décompression)

Auteurs : Maëlys Mistretta, Ayoub Salhi

Date : Décembre 2025

Dépôt (structure) : `core/`, `encoder/`, `decoder/`, `utils/`, `tests/`, `io/`.
Langage : Python 3.12.2.
Lien Github : Algav Huffman Dynamique

Table des matières

1 Introduction et contexte

Ce projet s'inscrit dans le devoir d'Algorithmique Avancée (Master 1 STL, Sorbonne Université, 2025/2026) et consiste à implanter un **compresseur** et un **décompresseur** basés sur l'algorithme de **Huffman dynamique** (Huffman adaptatif, FGK/Vitter).

Contrairement à Huffman statique, l'arbre n'est pas construit à partir d'une table de fréquences connue à l'avance : il **évolue au fur et à mesure** du flux. L'enjeu principal est que le compresseur et le décompresseur modifient l'arbre **de manière strictement identique**, afin de partager implicitement les mêmes codes à chaque instant.

Pour ce projet, nous avons décidé de mettre en commun notre travail avec github, afin d'avoir facilement accès aux modifications réalisées et pour garder une trace de tout les changements. Le lien vers le github avec tous les documents est disponible dans l' en-tête.

1.1 Principe (NYT / échappement)

On maintient un arbre de Huffman adaptatif (AHA) contenant une feuille spéciale **NYT** (*Not Yet Transmitted*, notée # dans l'énoncé). Pour chaque symbole lu :

- si le symbole est **connu** : on émet son code courant (chemin racine → feuille) ;
- sinon : on émet le code du **NYT**, puis le symbole sous forme brute, et on l'insère dans l'arbre ;
- dans les deux cas : on met à jour l'arbre (échanges + incrémentations) jusqu'à la racine.

1.2 Choix d'implantation : symbole = octet

Les fichiers d'entrée sont en UTF-8, mais notre implémentation traite le flux au niveau **octet** : un symbole Huffman est un entier dans $\{0, \ldots, 255\}$. Ainsi, un caractère UTF-8 multi-octets (accents, emoji, etc.) devient une suite de symboles. Ce choix garantit un alphabet borné et une décompression simple, au prix d'une compression parfois moins efficace sur certains textes.

2 Architecture logicielle (dépôt)

Le dépôt est structuré par responsabilité :

- `core/` : nœuds, arbre et mise à jour FGK/Vitter ;
- `encoder/` : compression ;
- `decoder/` : décompression ;
- `utils/` : lecture/écriture bit-à-bit ;
- `tests/` et `io/` : jeux de tests, résultats expérimentaux.

3 Format I/O binaire et conventions de bits

Conformément à l'énoncé, l'écriture/lecture se fait par octets (8 bits) en lisant/écrivant les bits **du poids fort vers le poids faible**. Si le nombre total de bits n'est pas multiple de 8, on complète le dernier octet par des 0 à droite.

3.1 En-tête de taille (64 bits)

Le fichier `.huff` commence par **64 bits** représentant la taille (en octets) du fichier original. Le décompresseur s'arrête après avoir reconstruit exactement ce nombre d'octets, ce qui permet d'ignorer le padding final.

4 Réponses aux questions

Question 1 — L'algorithme n'échange jamais un nœud avec un ancêtre

Dans notre code, avant tout échange nous testons explicitement la condition :

```
if leader is not q_node and not is_ancestor(leader, q_node): swap_nodes(...)
```

Ainsi, un échange avec un ancêtre est interdit par construction.

D'un point de vue théorique (algorithme FGK/Vitter), un échange avec un ancêtre inverserait une relation parent/enfant, casserait la structure binaire de l'arbre et rendrait incohérente la numérotation hiérarchique GDBH. Les échanges sont donc restreints à des nœuds de même bloc et sans relation d'ascendance, ce que garantit explicitement notre implémentation.

Question 2 — Nombre maximal d'échanges lors d'une modification

Une modification (mise à jour après lecture d'un symbole, c'est-à-dire d'un *octet*) est effectuée dans `core/update_algorithm.py` par une remontée depuis la feuille correspondant au symbole jusqu'à la racine. À chaque itération, l'algorithme : (i) recherche le *chef de bloc* (nœud de même poids avec l'identifiant GDBH maximal), (ii) effectue éventuellement un **unique échange**, puis (iii) incrémente le poids et remonte vers le parent.

Borne sur le nombre d'échanges. Notons k le nombre de symboles *distincts déjà rencontrés* dans le flux. L'arbre de Huffman dynamique contient alors k feuilles correspondant aux symboles, ainsi qu'une feuille spéciale NYT, soit $k+1$ feuilles au total.

Un arbre binaire plein possédant L feuilles contient $2L-1$ nœuds. Dans notre cas, $L = k+1$ et le nombre total de nœuds n vérifie :

$$n = 2(k+1) - 1 = 2k + 1 \leq 511.$$

On traite 8 bits à chaque modification, donc le nombre d'éléments dans l'alphabet est borné par 255, ce qui implique que le nombre total maximal de nœuds est $255 \times 2 + 1 = 511$.

La boucle de mise à jour parcourt exactement les nœuds situés sur le chemin "feuille → racine", de longueur p (profondeur de la feuille modifiée). À chaque niveau, l'algorithme peut effectuer **au plus un échange**. Le nombre total d'échanges S lors d'une modification vérifie alors :

$$S \leq p + 1 \leq \mathrm{h}.$$

Ainsi, le **nombre maximal d'échanges par mise à jour** est en $O(h)$, où h est la hauteur de l'arbre. Dans le cas d'un arbre dégénéré (pire cas), on a la hauteur h qui est égale au nombre de symboles distincts déjà rencontrés, c'est-à-dire k, ce qui nous donne $O(k)$. Cette borne est volontairement pessimiste, l'arbre de Huffman dynamique n'étant pas garanti équilibré.

Nombre d'échanges vs temps d'exécution. Le nombre d'échanges ne correspond pas directement au temps d'exécution d'une mise à jour. Dans notre implémentation actuelle, la recherche du chef de bloc est réalisée par la fonction `find_block_leader`, qui effectue un **parcours en largeur (BFS) complet** de l'arbre, de coût $O(n)$. Cette recherche est appelée à chaque niveau de la remontée.

Le coût total d'une mise à jour est donc :

$$T_{\text{update}} = O(h \cdot n) + O(n) = O(hn),$$

où le terme $O(n)$ supplémentaire correspond à la renumérotation finale des identifiants GDBH (`renumber_tree`), également réalisée par un parcours BFS.

Piste d'amélioration. Cette complexité peut être réduite en évitant les parcours globaux répétés de l'arbre. En maintenant des structures de données permettant un accès direct aux blocs de poids (par exemple des listes ou dictionnaires indexés par poids), la recherche du chef de bloc pourrait être réalisée en $O(1)$ ou $O(\log n)$. Le coût d'une mise à jour serait alors ramené à $O(h)$ (ou $O(h \log n)$), sans modifier la borne sur le *nombre* d'échanges.

Question 3 — Fonction `lecture(fichier.bin)` : binaire → chaîne de bits

Nous avons implémenté `utils/bitreader.py` de la façon suivante :

1. lecture du fichier en mode `rb`
2. conversion de chaque octet en représentation binaire sur 8 bits, concaténée en une chaîne de `'0'` et `'1'`.

Question 4 — Fonction `ecriture(fichier_chaine.txt, fichier.bin)` : bits → binaire

Nous avons implémenté `utils/bitwriter.py` comme ceci :

1. lecture de la chaîne de bits
2. padding à droite pour obtenir un multiple de 8
3. conversion par blocs de 8 bits en octets
4. écriture en mode `wb`

Le résultat est lisible par `lecture`.

Question 5 — Structures de données et complexités

Arbre dynamique : classe `DynamicHuffmanTree` contenant :
— `root` : pointeur vers la racine ;
— `NYT` : pointeur vers la feuille NYT ;
— `symbol_nodes` : dictionnaire `symbole` → `feuille`.
Complexités (avec h la hauteur de l'arbre) :
— test "symbole connu ?" : $O(1)$ via `symbol_nodes` ;
— obtention d'un code : $O(h)$ (remontée feuille → racine) ;
— mise à jour (remontée + échanges) : $O(hn)$ (explications dans la question 2) ;
— renumérotation GDBH : BFS $O(n)$.

Question 6 — Structure arborescente et fonction `finBloc`

Structure :
— `LeafNode` (symbole ou NYT) ;
— `InternalNode` (deux enfants) ;
— `NodeBase` (poids, parent, identifiant GDBH).

finBloc : dans notre code, l'équivalent opérationnel de `finBloc(H,m)` est `find_block_leader(tree, node)` : on cherche, parmi les nœuds de même poids, celui d'identifiant GDBH maximal (fin du bloc). Cette recherche est réalisée par parcours BFS de l'arbre.

Question 7 — Tests compression/décompression

Compression Pour la compression, nous avons suivi le processus suivant :

1. création d'un arbre vide
2. lecture du fichier UTF-8 en format binaire
3. écriture de la taille du fichier en binaire sur 64 bits dans un fichier temporaire

4. boucle sur chaque octet lu du fichier :
   — ✓ si le symbole lu existe déjà, on récupère son chemin
     ✓ sinon on récupère le chemin menant du noeud NYT, puis on le concatène au code binaire du symbole lu
   — écriture du chemin du symbole dans le fichier temporaire
   — mise à jour de l'arbre
5. appel à la fonction écriture() pour recopier le contenu du fichier temporaire dans le fichier de sortie .huff

Décompression La décompression suit la même logique :

1. création d'un arbre vide
2. lecture de la taille du fichier dans les 64 bits d'en-tête
3. lecture du fichier .huff avec la fonction lecture()
4. boucle sur les bits de la séquence récupérée :
   — ✓ si on a le chemin menant du nœud NYT, on récupère les 8 bits suivants
     ✓ sinon, on a le chemin menant à une feuille, et on récupère son symbole
   — écriture en binaire du symbole dans le fichier de sortie
   — mise à jour de l'arbre

Remarque : avec notre approche utilisant une en-tête contenant la taille dans le fichier compressé, nous ne sommes pas en mesure de décompresser correctement le fichier .huff fourni sur moodle.

Tests Nous validons la correction par tests de **round-trip** : compression puis décompression, et comparaison **octet à octet** entre l'entrée et la sortie.

Le script `python -m tests.test_roundtrip_basic` exécute ces tests sur les fichiers présents dans `io/input/` et produit un CSV de résultats dans `tests/compression_report.csv`.

Question 8 — Expériences sur texte naturel (ex. Gutenberg)

Nous avons utilisé des textes issus de Project Gutenberg, par exemple `io/input/Blaise_Pascal.txt`, ainsi que `io/input/gutenberg_small_fr.txt`. Les mesures de taille/ratio/temps sont collectées automatiquement.

Question 9 — Expériences sur fichiers aléatoires (distribution contrôlée)

Premièrement nous avons testé `io/input/emojis.txt` qui nous a permet de valider la compression de symboles de plusieurs octets (1 emoji = 4 octets) Nous avons aussi testé :

— `io/input/random_uniform_letters.txt` : distribution quasi uniforme des lettres ;
— `io/input/random_biased_letters.txt` : distribution biaisée (beaucoup de `A`, puis `B`, etc.).

Ces fichiers permettent d'observer l'effet direct de la distribution sur la longueur moyenne des codes.

Question 10 — Expériences sur fichiers "informatifs" non langue naturelle

Nous avons testé :

— `io/input/data_json_sample.txt` (JSON),
— `io/input/code_python_sample.txt` (code Python),
— `io/input/json_like_config.txt` (config).

Question 11 — Analyse expérimentale

Le tableau 1 synthétise les résultats observés (issus de `tests/compression_report.csv`).

TABLE 1 – Résultats expérimentaux

File	Original (B)	Compressed (B)	Ratio	Compress (ms)	Decompress (ms)
gutenberg-small-fr.txt	290	210	0.724	122.053	50.078
json-like-config.txt	170	143	0.841	19.670	19.415
short.txt	3	12	4.000	0.597	0.320
data-json-sample.txt	309	215	0.696	52.093	46.639
code-python-sample.txt	600	407	0.678	89.108	80.924
random-uniform-letters.txt	655	430	0.656	98.079	81.151
english-mixed-case.txt	202	173	0.856	35.935	38.842
emojis.txt	1758	1321	0.751	610.631	582.637
Blaise-Pascal.txt	118026	68594	0.581	38279.061	45327.403
random-biased-letters.txt	407	83	0.204	4.605	6.366

Code couleur :
- vert : ratio < 0.6
- orange clair : ratio < 0.8
- orange foncé : ratio < 1
- rouge : ratio > 1

Analyse :
- **Petits fichiers** : expansion fréquente (en-tête 64 bits + premières occurrences via NYT + padding).
- **Fichiers biaisés** : forte compression car un symbole dominant reçoit un code très court.
- **Uniforme** : gain plus faible car les fréquences sont proches (codes de longueurs proches).
- **Textes naturels** : ratios plausibles (< 1) et relativement stables quand la taille augmente.

5 Format du rendu (scripts + logs)

Deux scripts Bash sont fournis à la racine :
- `./compresser <input.txt> <output.huff>` ;
- `./decompresser <input.huff> <output.txt>`.

Ils appellent `compresser.py` et `decompresser.py`. À chaque exécution, une ligne est ajoutée dans `compression.txt` et `decompression.txt` au format :

`input_path;output_path;input_bytes;output_bytes;ratio;time_ms`.

6 Usage d'IA générative

Nous avons utilisé une IA générative comme outil d'assistance pour :
- clarifier certaines parties de l'énoncé et du cours (FGK/Vitter, propriétés de l'AHA) ;
- aider à la structuration du rapport et à la reformulation de certaines explications ;
- identifier et corriger des erreurs techniques (scripts Bash, encodage UTF-8, gestion des bits).

Analyse critique. Les propositions de l'IA ont systématiquement été vérifiées et adaptées. En effet, une attention particulière a été portée à la distinction entre caractères UTF-8 et octets, ainsi qu'au respect strict des propriétés théoriques de l'algorithme de Huffman dynamique.

Estimation quantitative. La proportion de code généré ou modifié avec l'aide de l'IA est estimée à environ **30 %**.

7 Conclusion

Nous avons implémenté un compresseur et un décompresseur basés sur l'algorithme de Huffman dynamique (FGK/Vitter), opérant au niveau octet et respectant les conventions de bits et le format d'exécution imposés. Les tests de round-trip valident la correction fonctionnelle, et l'étude expérimentale met en évidence l'influence de la distribution des symboles ainsi que l'overhead induit sur les fichiers de petite taille.

Des améliorations futures pourraient inclure une gestion plus efficace des blocs de poids afin de réduire la complexité des mises à jour, ainsi qu'un traitement symbolique au niveau UTF-8 pour améliorer la compression sur des textes multilingues.