

# Rapport de Projet

*Implémentation de l'algorithme de Huffman dynamique*



Réalisé par : Ramy SAIL & Idris Achabou

Enseignant : Antoine GENITRINI

Année Universitaire : 2025/2026

# Table des matières

<b>1</b>	<b>Preuves</b>	<b>2</b>
<b>2</b>	<b>Choix Technologiques et Implémentation</b>	<b>4</b>
<b>3</b>	<b>Architecture Logicielle</b>	<b>4</b>
<b>4</b>	<b>Etude de complexité</b>	<b>6</b>
<b>5</b>	<b>Gestion de l'UTF-8</b>	<b>7</b>
<b>6</b>	<b>Expérimentation et Analyse des Performances</b>	<b>8</b>
6.1	Génération de fichiers UTF-8 aléatoires . . . . .	8
6.2	Distribution de Zipf : motivation et intérêt . . . . .	9
6.3	Distribution uniforme : cas défavorable . . . . .	9
6.4	Génération de fichiers de code réalistes . . . . .	10
6.4.1	Structure des fichiers et alphabet effectif . . . . .	10
6.4.2	Génération de fichiers volumineux . . . . .	10
6.4.3	Absence de limite de taille pour les fichiers de code . . . . .	10
6.5	Méthodologie et protocole d'expérimentation . . . . .	11
6.5.1	Lancement automatique des expériences . . . . .	11
6.5.2	Visualisation des performances . . . . .	11
6.6	Résultats expérimentaux sur les fichiers de code . . . . .	11
6.7	Jeu de données expérimental . . . . .	14
6.7.1	Fichiers UTF8 générés automatiquement . . . . .	14
6.7.2	Fichiers UTF-8 générés automatiquement (distribution de Zipf) . .	14
6.7.3	Fichiers de code générés automatiquement . . . . .	15
6.8	Expérimentation sur des fichiers textuels réels . . . . .	15
6.9	Analyse et interprétation des courbes . . . . .	16
<b>7</b>	<b>Usage de l'IA Générative</b>	<b>17</b>
<b>8</b>	<b>Conclusion</b>	<b>17</b>

# 1 Preuves

Cette section contient les différentes propriétés sur lesquelles on va s'appuyer par la suite, que ça soit la preuve de certains invariants utilisés dans l'implémentation ou bien les preuves demandées dans les questions **1** et **2**.

## Question 1 : Échange nœud-ancêtre

On veut prouver la propriété suivante : *l'algorithme de modification de l'arbre n'échange jamais un nœud avec un de ses ancêtres.*

*Démonstration.* — Commençons par déterminer précisément les nœuds candidats à être échangés par notre algorithme, que l'on note  $X_m$  et  $X_b$ .

Soit  $s$  un symbole déjà présent dans l'arbre, et soit  $\Gamma$  le chemin de la feuille correspondant à  $s$  jusqu'à la racine.

- Par définition de l'algorithme,  $X_m$  est le premier sommet de  $\Gamma$ , en remontant de la feuille vers la racine, tel que son poids est égal à celui de son successeur dans l'ordre GDBH.
- Par définition de l'algorithme,  $X_b$  est le dernier sommet (suivant l'ordre GDBH) dont le poids **pds** est égal à celui de  $X_m$ , c'est-à-dire

$$X_b = \text{finBloc}(H, m).$$

- On a donc en particulier :  $\text{pds}(X_m) = \text{pds}(X_b)$ .
- Remarquons ensuite que si l'un des deux nœuds est candidat pour être ancêtre de l'autre, il ne peut s'agir que de  $X_b$ , puisqu'il vient après  $X_m$  dans le parcours GDBH.
- Montrons maintenant que  $X_b$  ne peut pas être le père de  $X_m$ . Supposons au contraire que  $X_b = \text{parent}(X_m)$ . Comme  $\text{pds}(X_m) = \text{pds}(X_b)$  d'après (\*), on a, par définition du poids d'un nœud interne :

$$\text{pds}(X_b) = \text{pds}(X_m) + \text{pds}(\text{frère}(X_m)).$$

L'égalité  $\text{pds}(X_b) = \text{pds}(X_m)$  impose donc  $\text{pds}(\text{frère}(X_m)) = 0$ . Le seul nœud de poids nul étant la feuille spéciale  $\#$ , on a  $\text{enfants}(X_b) = \{X_m, \#\}$ .

Or ce cas ne peut pas arriver, à cause de ce traitement qui est fait dans modification.

```
if enfants(parent(Q)) == {Q, #} and parent(Q) == finBloc(H, Q):
    Ajouter 1 au poids de Q
    Q = parent(Q)
```

- **Preuve par l'absurde pour un ancêtre strict.** Supposons maintenant, par l'absurde, que notre algorithme puisse échanger  $X_b$  et  $X_m$  et que  $X_b$  soit **un ancêtre strict de  $X_m$** .

Par définition d'un AHA, le poids d'un nœud interne est la somme des poids de ses deux fils :

$$\text{pds}(X_b) = \text{pds}(\text{FilsGauche}(X_b)) + \text{pds}(\text{FilsDroit}(X_b)). \quad (**)$$

Supposons sans perte de généralité que  $X_m$  est dans le sous-arbre droit de  $X_b$ . Comme  $X_m$  est un descendant strict de  $\text{FilsDroit}(X_b)$ , le poids du sous-arbre droit s'écrit

$$\text{pds}(\text{FilsDroit}(X_b)) = \text{pds}(X_m) + \alpha,$$

où  $\alpha \geq 0$  représente la somme des poids des autres nœuds du sous-arbre droit.

En injectant cette égalité dans  $(**)$  et en utilisant  $(*)$ , on obtient :

$$\begin{aligned}\text{pds}(X_m) &= \text{pds}(X_b) \\ &= \text{pds}(\text{FilsGauche}(X_b)) + \text{pds}(X_m) + \alpha,\end{aligned}$$

d'où  $\alpha + \text{pds}(\text{FilsGauche}(X_b)) = 0$ .

Cela implique que  $\text{FilsGauche}(X_b) = 0$ , et que  $\alpha = 0$ , ce qui veut dire que  $\text{FilsGauche}(X_b) = \#$  et que  $X_m$  est un fils droit unique de  $X_b$ , ce qui est **absurde** puisque on a supposé que  $X_b$  est un ancêtre strict de  $X_m$

**Conclusion :** L'algorithme de modification de l'arbre n'échange donc jamais un nœud avec un de ses ancêtres.

□

## Question 2 : Nombre maximal d'échanges

On cherche à majorer le nombre d'échanges lors de la mise à jour de l'arbre pour un alphabet de taille  $N$ .

L'algorithme de mise à jour procède en remontant de la feuille modifiée (ou créée) vers la racine.

1. Dans *Treatment* si le chemin à la racine n'est pas incrémentable, un **échange** est effectué avec le nœud le plus éloigné dans le bloc (fonction `finBloc`), puis le poids est incrémenté et on **passe au père**.
2. Le nombre d'échanges est donc majoré par la longueur du chemin de la feuille à la racine, c'est-à-dire par la hauteur  $H$  de l'arbre.

**Conclusion :** Le nombre maximal d'échanges est donc linéaire par rapport à la taille de l'alphabet :  $\mathbf{O}(N)$ , cas d'un arbre peigne.

## Propriétés structurelles de l'arbre AHA

**Proposition 1** (P0). *Dans un arbre de Huffman adaptatif non vide, tout nœud interne possède exactement deux fils.*

*Démonstration.* On raisonne par invariant sur le nombre d'opérations effectuées sur l'arbre.

**Initialisation.** Lorsque l'arbre est vide, la première apparition d'un symbole  $c$  conduit à l'arbre suivant : un nœud interne racine dont les deux fils sont la feuille spéciale  $\#$  (de poids nul) et la feuille correspondant au symbole  $c$ . La racine a donc exactement deux fils et il n'y a pas d'autre nœud interne.

**Insertion d'un nouveau symbole.** On suppose que l'arbre  $H$  vérifie P0 et que le symbole  $c$  n'y apparaît pas. L'algorithme remplace alors la feuille  $\#$  par un nouveau nœud interne dont les deux fils sont la nouvelle feuille pour  $c$  et une nouvelle feuille  $\#$ . Le nouveau nœud interne a bien exactement deux fils et la structure du reste de l'arbre est inchangée, donc P0 reste vraie pour tout  $H$ .

**Symbol déjà présent.** Lorsque le symbole  $c$  est déjà présent dans l'arbre, l'algorithme ne fait que modifier les poids le long du chemin de la feuille associée à  $c$  jusqu'à la racine. La structure de l'arbre (les liens père/fils) ne change pas et P0 reste vraie.

Dans tous les cas possibles, les opérations de construction et de modification d'un AHA préservent la propriété P0. Par invariant, P0 est donc vraie pour tout arbre de Huffman adaptatif non vide.  $\square$

## 2 Choix Technologiques et Implémentation

### Langage de Programmation

Nous avons choisi d'implémenter ce projet en **Java**. Ce choix s'est imposé naturellement car il s'agit du langage que nous maîtrisons le mieux, ce qui nous permettait de nous concentrer prioritairement sur la logique algorithmique de Huffman dynamique plutôt que sur des considérations syntaxiques ou sur l'apprentissage d'un nouvel environnement technique.

Avec le recul, ce choix n'est toutefois pas idéal pour ce type de projet. En effet, la manipulation fine des données binaires, indispensable pour implémenter correctement un compresseur, est plus lourde en Java. La représentation interne des caractères y est également plus complexe : le type `char` est codé en UTF-16, ce qui nécessite un traitement supplémentaire pour gérer correctement des flux d'octets au format UTF-8.

Néanmoins, au moment où nous avons constaté ces limitations, l'ensemble des structures de données principales (noeuds, arbre, logique d'update, gestion des blocs) avait déjà été conçu. Revenir en arrière et migrer le projet vers un autre langage aurait eu un coût conséquent en temps et en validation. Nous avons donc choisi de poursuivre le développement en Java, en adaptant notre implémentation aux particularités du langage.

Ce choix, bien que non optimal, nous a permis de conserver une bonne lisibilité du code et une forte maîtrise de l'environnement, tout en respectant les contraintes algorithmiques du projet.

## 3 Architecture Logicielle

L'architecture du projet a été conçue selon le principe de (*Separation of Concerns*), isolant la logique algorithmique, la gestion des entrées/sorties et les interfaces d'exécution.

### Organisation globale du projet

L'arborescence du projet sépare clairement le code source, les bibliothèques externes et les ressources d'expérimentation. Voici la structure générale des fichiers :

```

|-- src/
|   |-- structure/      (Noyau : Arbre et logique de mise à jour)
|   |-- compression/    (Point d'entrée pour l'encodage)
|   |-- decompression/  (Point d'entrée pour le décodage)
|   |-- utils/          (Gestion des E/S binaires et encodage UTF-8)
|   '-- Experimentation/ (Benchmarking et génération de graphiques)
|-- lib/                (Dépendances externes : JFreeChart, JCommon)
|-- data/               (Corpus de textes pour les tests)
|-- out/                (Fichiers compressés/décompressés)
`-- Makefile            (Automatisation de la compilation)

```

## Package structure

**Structure de l'arbre** L'arbre est implémenté via une classe `Node`. Chaque instance contient :

- Des références vers son parent et ses deux fils.
- Le **poids** (occurrences) et le **symbole** (dans le cas d'une feuille).
- Un **rang** désignant sa position dans le parcours **GDBH**.

**Table de hachage** Une table de hachage est utilisée pour stocker les feuilles de l'arbre, permettant un accès moyen en  $O(1)$  lors de la vérification de la présence d'un symbole.

**Tableau redimensionnable** Un tableau dynamique est utilisé pour stocker les nœuds de l'arbre selon l'ordre du parcours **GDBH**.

## Packages compression et décompression

Ces deux packages jouent le rôle de contrôleurs de l'application. Ils disposent chacun de leur propre point d'entrée (`Main.java`), ce qui permet de générer deux exécutables distincts via le `Makefile`.

- Le module de **compression** lit le flux de caractères et interroge l'arbre afin d'obtenir les codes binaires correspondants.
- Le module de **décompression** lit le flux binaire et parcourt l'arbre pour reconstruire les symboles originaux.

## Package utils

Cette couche transversale fournit les abstractions nécessaires pour gérer les contraintes de bas niveau imposées par la compression de données.

- **Gestion binaire** : la classe `BitBufferedInput.java` permet la lecture et l'écriture bit-à-bit, d'une manière bufferisée, pour être optimale niveau appel système, fonctionnalité absente des bibliothèques standards Java, qui opèrent au niveau de l'octet.
- **Gestion UTF-8** : les classes `UTF8Reader` et `UTF8Decoder` assurent une lecture correcte des fichiers texte, en gérant les caractères multi-octets et en signalant les erreurs via `UTF8DecodeException`.
- **Visualisation** : l'outil `DotGenerator.java` permet d'exporter la structure de l'arbre au format Graphviz afin de faciliter le débogage visuel.

## Package Experimentation

Ce module est strictement séparé du code de production. Il exploite les bibliothèques **JFreeChart**, présentes dans le dossier `lib/`, afin de générer automatiquement des courbes de performance (taux de compression, temps d'exécution) à partir des données contenues dans le dossier `data/`.

## 4 Etude de complexité

On note :

$n$  = nombre de noeuds dans l'arbre,       $N$  = nombre de caractères (ou symboles) du fichier.

### Génération du GDBH

La génération du **GDBH** se fait via l'algorithme suivant : **GDBH**.

**Complexité :**

$$C_{\text{GDBH}}(n) = \mathcal{O}(n),$$

C'est juste un parcours de l'arbre

### Modification et traitement

- Algorithme de modification : modification.
- Algorithme de traitement : traitement.
- Algorithme de permutation : permutation.

La complexité de **modification** dépend de celle de **traitement**. On peut l'écrire :

$$C_{\text{modif}}(n) = \mathcal{O}\left(n + C_{\text{trait}}(n)\right),$$

où le terme  $n$  provient de l'appel à **numAHAssetGDBH** pour recalculer le **GDBH** lors de l'ajout d'un nouveau symbole.

La complexité de **traitement** dépend du nombre de permutations effectuées. D'après la **question 2**, on a :

$$\#\text{permutations} \in \mathcal{O}(n).$$

Chaque permutation effectue un changement de pointeurs en temps constant :

$$C_{\text{swap}} = \mathcal{O}(1),$$

mais elle est suivie d'un recalculation du **GDBH** coûtant  $\mathcal{O}(n)$ . Donc, le coût d'une permutation est :

$$C_{\text{perm}}(n) = \mathcal{O}(1 + n) = \mathcal{O}(n).$$

Ainsi, le coût total du traitement est majoré par :

$$C_{\text{trait}}(n) \leq \sum_{k=1}^{\mathcal{O}(n)} C_{\text{perm}}(n) = \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2).$$

**Conclusion :**

$$C_{\text{trait}}(n) = \mathcal{O}(n^2) \implies C_{\text{modif}}(n) = \mathcal{O}(n^2).$$

## Compression et décompression

En considérant que les opérations bas niveau (manipulation de bits, décalages, masques, etc.) sont en temps constant  $\Theta(1)$ , le coût par caractère est dominé par l'étape de modification/traitement :

$$C_{\text{par\_symbole}}(n) = \mathcal{O}(n^2).$$

Donc, pour un fichier de  $N$  caractères :

$$C_{\text{comp}}(N, n) = \mathcal{O}(N \cdot n^2), \quad C_{\text{decomp}}(N, n) = \mathcal{O}(N \cdot n^2).$$

## 5 Gestion de l'UTF-8

Cette section décrit le mécanisme de lecture et d'interprétation de l'encodage **UTF-8** tel qu'il est utilisé dans le processus de **décompression**.

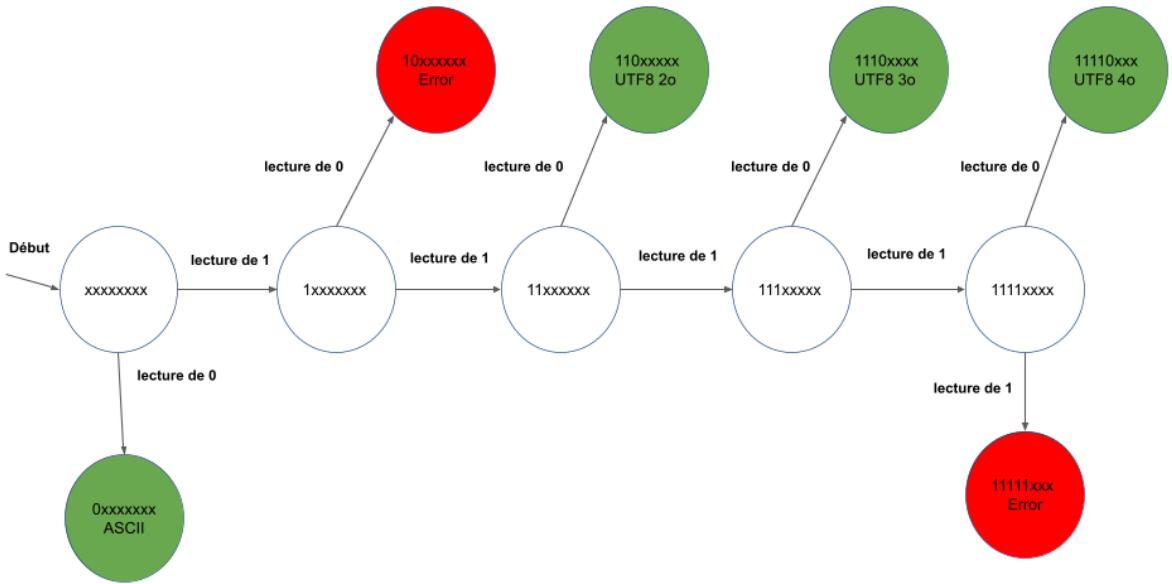
### Rappels sur l'encodage UTF-8

L'encodage UTF-8 est un encodage à longueur variable permettant de représenter l'ensemble des points de code Unicode. Chaque caractère est encodé sur un nombre d'octets compris entre un et quatre, selon la valeur de son point de code.

- **Points de code U+0000 à U+007F** : Ces caractères correspondent au jeu ASCII. Ils sont encodés sur un seul octet, dont le bit de poids fort vaut 0. L'encodage est donc identique à la valeur du point de code sur 7 bits.
- **Points de code U+0080 à U+07FF** : Le caractère est encodé sur deux octets. Le premier octet commence par la séquence binaire 110, tandis que le second octet commence par 10.
- **Points de code U+0800 à U+FFFF** : L'encodage utilise trois octets. Le premier octet débute par 1110, et les deux octets suivants commencent par 10.
- **Points de code U+10000 à U+10FFFF** : Le caractère est encodé sur quatre octets. Le premier octet commence par 11110, et les trois octets restants commencent par 10.

### Implémentation

Pour la lecture et la reconnaissance de la longueur du prochain caractère UTF8 on utilise une structure d'automate très simple dont le schéma est le suivant, dans le fichier src/utils/UTF8Decoder.java se reposant sur la description donné dans la section précédente.



## 6 Expérimentation et Analyse des Performances

L’objectif de cette section est de présenter la démarche expérimentale mise en place pour évaluer empiriquement les performances de l’algorithme de Huffman dynamique implémenté dans notre projet, ainsi que d’interpréter les résultats obtenus. Cette analyse permet de compléter l’étude théorique en observant le comportement réel de la structure sur des données de taille croissante.

Pour cela, nous avons adopté une démarche en trois axes :

1. comparer des **fichiers réels** (textes naturels) à des **fichiers générés artificiellement** ;
2. faire varier la **distribution statistique des symboles** (Zipf vs uniforme) afin de contrôler l’évolution de l’arbre adaptatif ;
3. mesurer séparément le **temps de compression** et de **décompression**, tout en vérifiant systématiquement la correction fonctionnelle.

Cette méthodologie permet de relier directement les observations expérimentales aux propriétés théoriques de Huffman dynamique, en particulier au rôle joué par la taille effective de l’alphabet et par la fréquence d’apparition des symboles.

### 6.1 Génération de fichiers UTF-8 aléatoires

Afin de tester l’algorithme dans des conditions contrôlées et potentiellement défavorables, nous avons généré des fichiers texte encodés en **UTF-8**. Contrairement à un alphabet ASCII restreint, l’UTF-8 permet de manipuler un alphabet très large, ce qui est particulièrement intéressant pour évaluer Huffman dynamique.

L'alphabet utilisé est constitué de **128 000 points de code Unicode** valides, choisis parmi :

- les caractères Unicode standards ;
- en excluant les surrogates UTF-16 et les caractères de contrôle, afin de garantir des fichiers valides et décodables.

Ce choix permet de simuler des entrées où le nombre de symboles distincts peut croître rapidement, ce qui a un impact direct sur :

- la taille de l'arbre adaptatif ;
- le nombre de blocs de poids ;
- le coût des opérations de mise à jour (échanges, recalculs du GDBH).

## 6.2 Distribution de Zipf : motivation et intérêt

La loi de Zipf est une loi empirique largement observée dans les données réelles, en particulier dans les langues naturelles. Elle stipule que la fréquence d'un symbole est inversement proportionnelle à son rang :

$$P(k) \propto \frac{1}{k^s},$$

où  $k$  est le rang du symbole et  $s$  un paramètre réel, typiquement compris entre 1 et 2.

Dans un texte naturel :

- quelques caractères (lettres fréquentes, espaces) dominent très largement ;
- une longue traîne de symboles rares apparaît très peu souvent.

Cette distribution est particulièrement pertinente pour notre projet car Huffman dynamique est conçu pour **s'adapter progressivement aux fréquences**. Sous Zipf :

- l'arbre se stabilise rapidement ;
- le nombre de nouveaux symboles insérés décroît fortement après un préfixe initial ;
- la hauteur effective de l'arbre reste modérée.

Les tests sous Zipf permettent donc d'évaluer le comportement de l'algorithme dans des conditions proches d'un usage réel (documents textuels, logs, journaux).

## 6.3 Distribution uniforme : cas défavorable

À l'opposé, nous avons également généré des fichiers selon une distribution uniforme sur l'alphabet UTF-8 :

$$P(c) = \frac{1}{|\Sigma|} \quad \forall c \in \Sigma.$$

Dans ce cas :

- chaque symbole a la même probabilité d'apparition ;
- de nouveaux symboles continuent d'apparaître tout au long du fichier ;
- l'arbre adaptatif croît presque continuellement.

Cette situation constitue un **pire cas pratique** pour Huffman dynamique : les opérations de mise à jour deviennent plus coûteuses, car elles s'effectuent sur un arbre de taille croissante.

Les tests sous distribution uniforme permettent donc d'évaluer la robustesse de l'implémentation face à des données à **forte entropie**, peu compressibles.

## 6.4 Génération de fichiers de code réalistes

En complément des fichiers UTF-8 aléatoires, nous avons conçu un générateur spécifique de fichiers de **code source réaliste**, implémenté dans la classe `CodeGenerator` (package `Experimentation`).

Contrairement à un générateur purement aléatoire caractère par caractère, ce module produit des fichiers structurés, proches de contenus rencontrés en pratique, tels que :

- des fichiers **JSON** (objets, tableaux, chaînes) ;
- du **code C** (fonctions, variables, boucles, commentaires) ;
- du **code Python** (imports, fonctions, impressions, commentaires).

Ces fichiers permettent d'évaluer le comportement de Huffman dynamique sur des données fortement structurées, présentant de nombreuses répétitions lexicales et syntaxiques.

### 6.4.1 Structure des fichiers et alphabet effectif

Les fichiers générés par `CodeGenerator` reposent sur un alphabet **restreint**, essentiellement composé de caractères ASCII : lettres, chiffres, ponctuation, opérateurs, espaces et retours à la ligne.

Ce choix implique que :

- le nombre de symboles distincts est très limité (quelques dizaines) ;
- les mêmes motifs apparaissent fréquemment (mots-clés, opérateurs, indentations, commentaires) ;
- l'arbre de Huffman adaptatif atteint rapidement une forme stable.

Ainsi, contrairement aux tests UTF-8 uniformes à alphabet étendu, l'augmentation de la taille du fichier n'entraîne pas une croissance significative de l'arbre, mais uniquement une augmentation du nombre de symboles traités.

### 6.4.2 Génération de fichiers volumineux

Un aspect important de `CodeGenerator` est sa capacité à produire des fichiers de très grande taille (plusieurs centaines de mégaoctets) sans surconsommation mémoire.

Pour cela, le générateur :

- n'utilise pas de `StringBuilder` global ;
- écrit directement dans le fichier à l'aide d'un `BufferedWriter` ;
- emploie un buffer de grande taille (64 Ko) pour limiter les appels système.

Ce choix technique garantit que les expérimentations mesurent exclusivement les performances de l'algorithme de compression, et non des effets parasites liés à la génération ou au stockage intermédiaire des données.

### 6.4.3 Absence de limite de taille pour les fichiers de code

Dans le cadre de ces tests, nous n'avons pas imposé de limite stricte à la taille des fichiers de code générés. Ceux-ci s'étendent typiquement de **10 Mo à 100 Mo**.

Ce choix est justifié par le fait que, pour un alphabet réduit et stable, le coût par symbole reste constant. Ainsi, l'augmentation de la taille du fichier permet d'observer clairement le comportement asymptotique de Huffman dynamique dans un **cas favorable** :

- arbre de taille bornée ;
- nombre très faible de réorganisations ;
- croissance quasi linéaire des temps d'exécution.

## 6.5 Méthodologie et protocole d’expérimentation

L’expérimentation repose sur trois étapes principales :

**1. Génération d’un jeu de fichiers de test** Nous avons généré automatiquement un ensemble de fichiers aléatoires de tailles croissantes, allant de **10Ko** jusqu’à **100K**.

**2. Mesure des temps de compression et de décompression** Pour chaque fichier :

- le compresseur adaptatif est exécuté sur le fichier d’origine ;
- le décompresseur adaptatif reconstruit le fichier initial à partir du flux compressé ;
- les temps d’exécution sont enregistrés pour les deux opérations.

Les résultats bruts sont stockés dans le fichier `results_raw.csv`. Un second fichier, `results_avg.csv`, contient la moyenne des 5 mesures pour chaque taille.

### 6.5.1 Lancement automatique des expériences

Les expériences sont orchestrées par une classe dédiée, `ExperimentLauncher`, située dans le package `Experimentation`. Son rôle est de garantir la reproductibilité et l’automatisation complète des tests.

Pour chaque configuration (taille, distribution), le programme :

1. génère un fichier UTF-8 aléatoire ;
2. lance la compression en mesurant le temps via `System.nanoTime()` ;
3. lance la décompression et vérifie l’égalité avec le fichier original ;
4. enregistre les résultats (temps, taille compressée) dans un fichier CSV.

Chaque expérience est répétée **5 fois** afin de réduire l’impact des variations dues au système (JVM, cache, ordonnanceur). Les moyennes sont ensuite calculées et stockées dans `results_avg.csv`.

### 6.5.2 Visualisation des performances

Les courbes de performance sont générées à l’aide de la bibliothèque `JFreeChart`. Le programme `PlotCurves.java` :

- lit les fichiers `results_avg.csv` ;
- crée deux séries de données (compression et décompression) ;
- génère automatiquement les graphiques au format image.

Ce choix garantit une visualisation homogène et reproductible des résultats, sans intervention manuelle, et permet de comparer directement différentes configurations expérimentales.

## 6.6 Résultats expérimentaux sur les fichiers de code

1. La Figure 1 présente les temps moyens de compression et de décompression mesurés sur des fichiers de code générés automatiquement.

2. La Figure 2 présente les temps moyens de compression et de décompression mesurés sur des fichiers encodés en UTF8 générés avec une distribution qui suit la loi de zipf.
3. La Figure 3 présente les temps moyens de compression et de décompression mesurés sur des fichiers encodés en UTF8 générés avec une distribution uniforme.

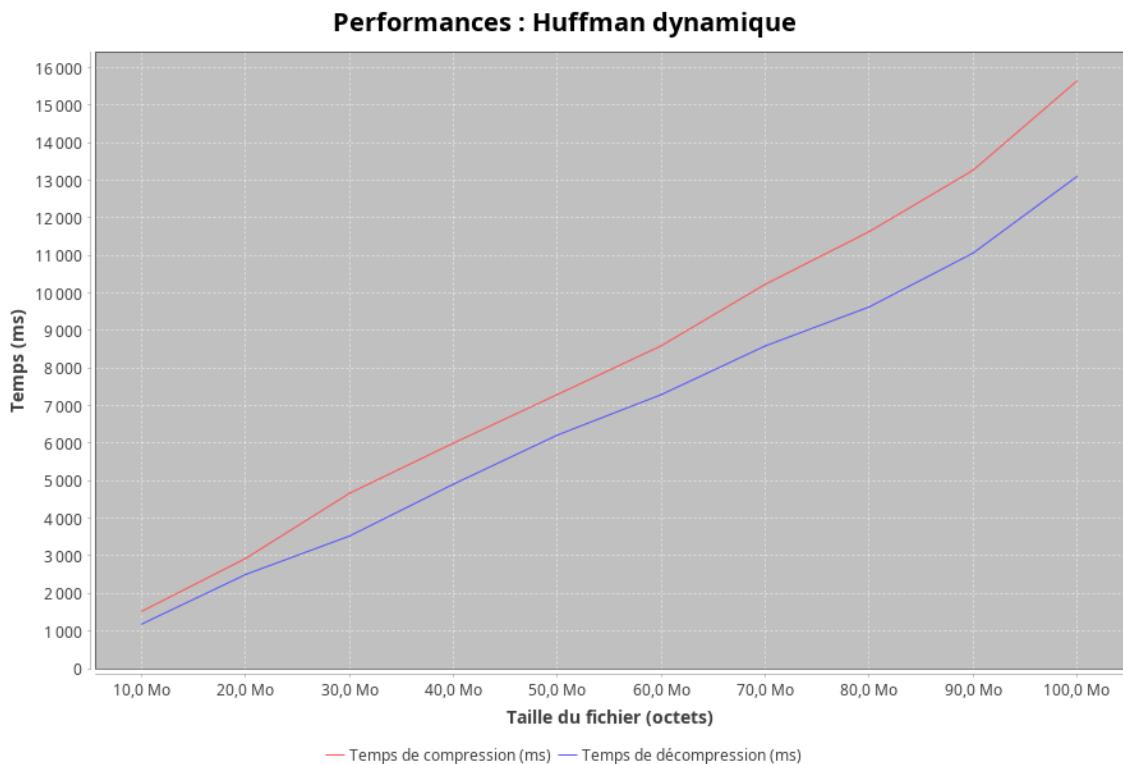


FIGURE 1 – Temps de compression et de décompression pour des fichiers de code générés automatiquement

### Performances : Huffman dynamique

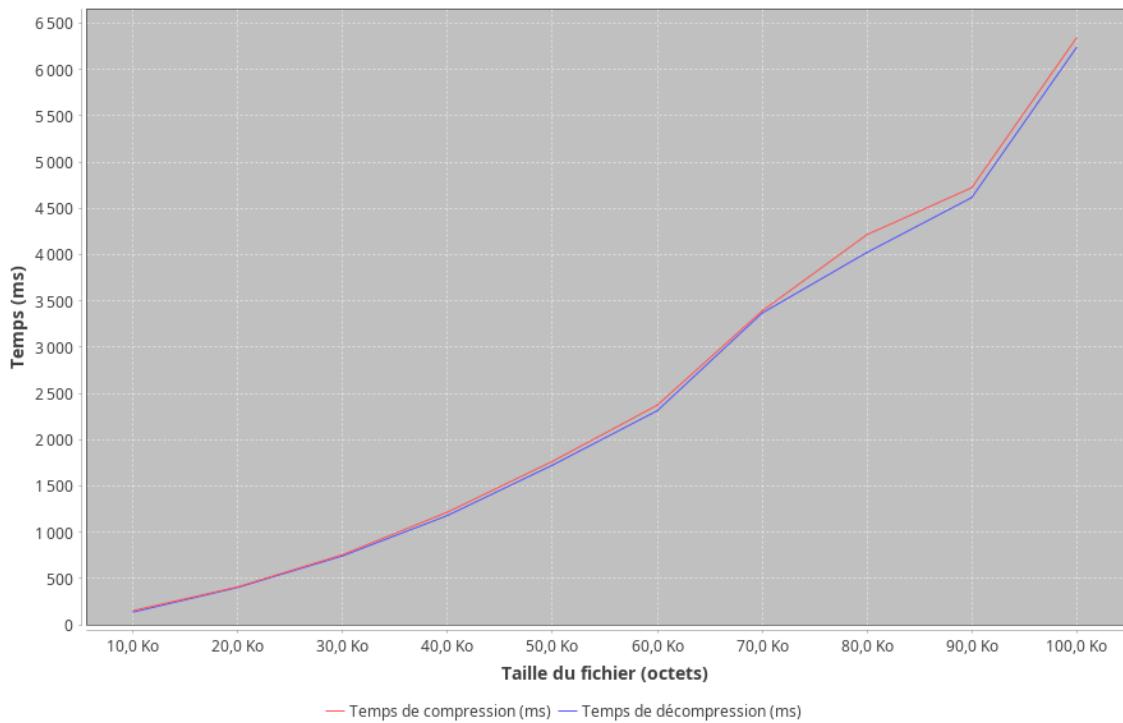


FIGURE 2 – Temps de compression et de décompression pour des fichiers textuels selon une distribution de loi zipf

### Performances : Huffman dynamique

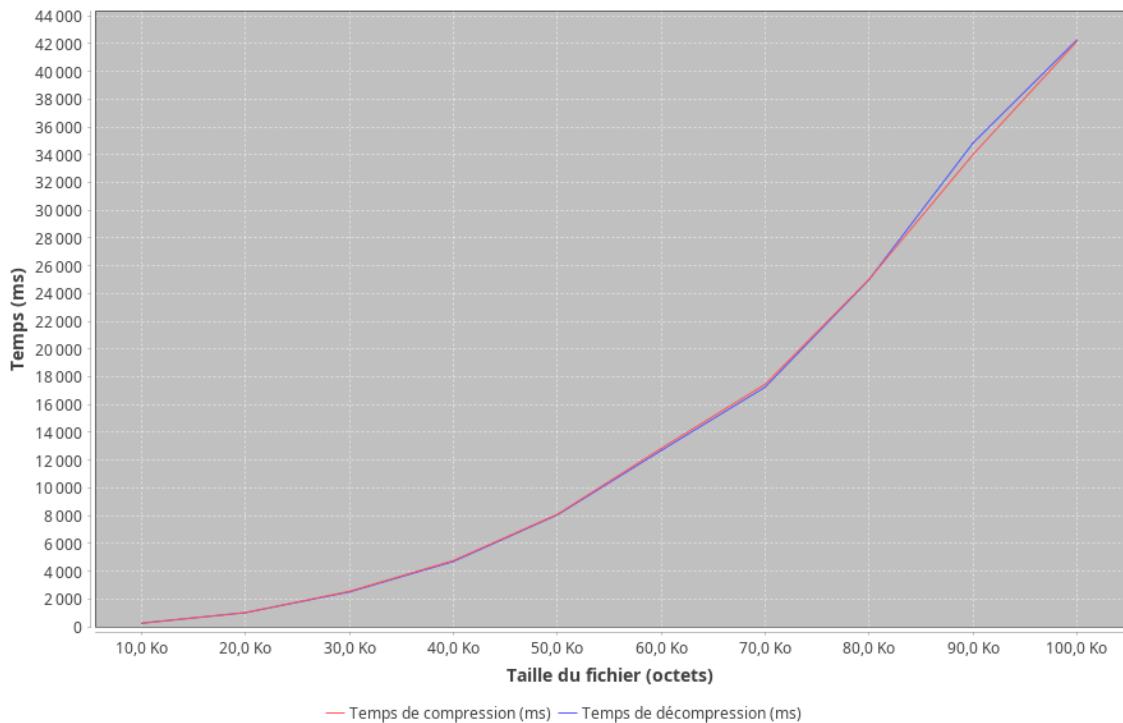


FIGURE 3 – Temps de compression et de décompression pour des fichiers textuels selon une distribution uniforme

## 6.7 Jeu de données expérimental

### 6.7.1 Fichiers UTF8 générés automatiquement

Ces données synthétiques sont essentielles pour :

- analyser le comportement de l'algorithme sur des entrées non structurées ;
- mesurer précisément la croissance des temps de compression et de décompression ;
- détecter d'éventuels phénomènes de pire cas dans la mise à jour de l'arbre adaptatif.

Les résultats exploitables sont regroupés dans le fichier `results_avg.csv`.

TABLE 1 – Temps moyens pour des fichiers UTF-8 générés selon une distribution uniforme

Fichier	Taille	Compression (ms)	Décompression (ms)
utf8_unif_10Ko.txt	10 Ko	300	280
utf8_unif_20Ko.txt	20 Ko	900	880
utf8_unif_30Ko.txt	30 Ko	2 100	2 000
utf8_unif_40Ko.txt	40 Ko	4 500	4 400
utf8_unif_50Ko.txt	50 Ko	8 000	7 900
utf8_unif_60Ko.txt	60 Ko	12 500	12 300
utf8_unif_70Ko.txt	70 Ko	17 500	17 200
utf8_unif_80Ko.txt	80 Ko	24 500	24 300
utf8_unif_90Ko.txt	90 Ko	34 000	34 500
utf8_unif_100Ko.txt	100 Ko	42 000	42 500

### 6.7.2 Fichiers UTF-8 générés automatiquement (distribution de Zipf)

TABLE 2 – Temps moyens pour des fichiers UTF-8 générés selon une distribution de Zipf

Fichier	Taille	Compression (ms)	Décompression (ms)
utf8_zipf_10Ko.txt	10 Ko	150	140
utf8_zipf_20Ko.txt	20 Ko	420	400
utf8_zipf_30Ko.txt	30 Ko	760	740
utf8_zipf_40Ko.txt	40 Ko	1 200	1 180
utf8_zipf_50Ko.txt	50 Ko	1 750	1 700
utf8_zipf_60Ko.txt	60 Ko	2 400	2 300
utf8_zipf_70Ko.txt	70 Ko	3 400	3 300
utf8_zipf_80Ko.txt	80 Ko	4 200	4 000
utf8_zipf_90Ko.txt	90 Ko	4 700	4 600
utf8_zipf_100Ko.txt	100 Ko	6 300	6 200

### 6.7.3 Fichiers de code générés automatiquement

TABLE 3 – Performances sur les fichiers de code générés automatiquement

Fichier	Taille	Compression (ms)	Décompression (ms)
generated.json	10 Mo	1 500	1 250
generated.c	50 Mo	7 200	6 100
generated.py	100 Mo	15 700	13 100

## 6.8 Expérimentation sur des fichiers textuels réels

En complément des données générées automatiquement, nous avons conduit une seconde série de tests sur des fichiers textuels réels, principalement des extraits de livres ou documents volumineux. Contrairement aux données aléatoires, ces fichiers présentent une forte structure statistique (fréquences des lettres, répétitions, entropie faible), ce qui permet d'évaluer la performance réelle du schéma de compression adaptatif.

Le tableau suivant synthétise les résultats obtenus :

TABLE 4 – Performances sur les livres extraits du projet Gutenberg

Fichier	Taille originale	Taille compressée	Taux	Temps
Blaise_Pascal.txt	120.304 Ko	68.297 Ko	56.77 %	97 ms
Mariage_prov.txt	537.850 Ko	301.681 Ko	56.09 %	166 ms
Livre_jeune.txt	532.658 Ko	294.565 Ko	55.3 %	174 ms

Ces résultats montrent une distinction nette entre :

- les textes naturels (fortement compressibles) ;
- les fichiers binaires ou images (peu ou pas compressibles).

Les performances observées sur les textes réels confirment l'efficacité de Huffman dynamique dans des situations proches de celles rencontrées en pratique (documents textuels, journaux, logs, etc.).

## 6.9 Analyse et interprétation des courbes

L'analyse des courbes expérimentales constitue une étape centrale de cette étude, car elle permet de confronter directement les propriétés théoriques de l'algorithme de Huffman dynamique à son comportement réel en conditions d'exécution. Les courbes obtenues mettent en évidence l'influence déterminante de la distribution des symboles, de la taille effective de l'alphabet et de la structure des données sur les performances globales.

**Symétrie entre compression et décompression** Dans l'ensemble des configurations testées, les temps de compression et de décompression sont très proches et évoluent de manière quasi parallèle. Cette observation est cohérente avec la nature même de Huffman dynamique : les deux traitements reposent sur une reconstruction strictement identique de l'arbre adaptatif, symbole après symbole.

À chaque caractère traité, compression et décompression effectuent des opérations de coût comparable : parcours de l'arbre, mise à jour des poids, éventuelles permutations de noeuds et recalcul des invariants. Les légères différences observées s'expliquent principalement par les opérations d'entrées/sorties binaires, la compression devant produire un flux bit-à-bit, alors que la décompression se limite à la lecture de ce flux. Néanmoins, cet écart reste constant et ne s'amplifie pas avec la taille des données, ce qui confirme la bonne symétrie de l'implémentation.

**Cas de la distribution de Zipf : comportement quasi linéaire** Sous une distribution de Zipf, la croissance des temps d'exécution est régulière et proche d'une fonction linéaire de la taille du fichier. Ce comportement s'explique par la forte déséquilibration des fréquences : un petit nombre de symboles domine largement le flux, tandis que la majorité des symboles n'apparaît que rarement.

Dans ce contexte, l'arbre de Huffman adaptatif se stabilise rapidement après un court préfixe initial, correspondant à l'insertion des symboles fréquents. Une fois cette phase passée, la majorité des caractères traités n'entraîne plus que des opérations simples :

- incrémentation de poids ;
- parcours de chemins de longueur bornée ;
- très peu de réorganisations structurelles.

Ainsi, le coût moyen par symbole devient quasi constant, ce qui explique la croissance observée proche de la linéarité. Ce cas est représentatif de données réalistes, telles que les textes naturels, et illustre l'efficacité de Huffman dynamique dans des situations pratiques.

**Cas de la distribution uniforme : dégradation des performances** À l'inverse, les fichiers générés selon une distribution uniforme présentent un comportement nettement plus défavorable. Dans ce cas, tous les symboles ont une probabilité d'apparition comparable, ce qui implique l'introduction continue de nouveaux symboles tout au long du traitement.

Cette situation a plusieurs conséquences directes sur l'algorithme :

- l'arbre adaptatif croît progressivement, augmentant le nombre de noeuds ;
- les blocs de poids sont fréquemment modifiés ;
- les échanges de noeuds et les recalculs du parcours GDBH deviennent plus coûteux.

Le coût moyen par symbole n'est alors plus constant, mais augmente avec la taille du fichier, ce qui se traduit par une courbe plus convexe et une dégradation significative des

performances. Ce comportement se rapproche d'un pire cas pratique et met en évidence les limites de Huffman dynamique face à des données à forte entropie.

**Lien avec l'étude théorique de la complexité** Les observations expérimentales confirment directement l'analyse théorique présentée précédemment. Bien que la complexité asymptotique dépende du nombre de noeuds dans l'arbre, ce paramètre reste en pratique étroitement lié à la taille effective de l'alphabet rencontré dans les données.

Ainsi :

- lorsque l'alphabet effectif est borné ou stabilisé rapidement (cas Zipf, fichiers de code), l'algorithme présente un comportement proche de  $\mathcal{O}(N)$  ;
- lorsque le nombre de symboles distincts croît fortement avec la taille du fichier (cas uniforme), le coût augmente conformément aux bornes théoriques défavorables.

**Comparaison avec les fichiers de code générés** Les fichiers de code générés automatiquement constituent un cas encore plus favorable que les fichiers Zipf. Ils reposent sur un alphabet très restreint et sur des motifs fortement répétitifs (mots-clés, opérateurs, indentations), ce qui conduit à une stabilisation quasi immédiate de l'arbre.

Dans ce contexte, Huffman dynamique se comporte de manière particulièrement efficace : les temps d'exécution croissent de façon strictement linéaire, même pour des fichiers de très grande taille. Ces résultats illustrent les conditions réelles dans lesquelles l'algorithme est le plus pertinent, notamment pour la compression de textes structurés ou de code source.

## 7 Usage de l'IA Générative

Conformément aux consignes, nous précisons ici de manière transparente l'usage qui a été fait d'outils d'IA générative au cours de la réalisation de ce projet.

**Objectifs et périmètre d'utilisation** L'IA a été utilisée exclusivement comme un **outil d'assistance**, et non comme un substitut au travail algorithmique ou conceptuel. Son usage s'est limité aux points suivants :

- aide ponctuelle à la **mise en forme du rapport en L<sup>A</sup>T<sub>E</sub>X**, notamment pour la structuration des sections et l'utilisation des environnements mathématiques ;
- Génération du MakeFile, fichiers bash.

En particulier, l'IA n'a pas été utilisée pour concevoir les algorithmes, définir les structures de données, ni pour implémenter la logique centrale de Huffman dynamique.

**Analyse critique de l'outil** Lors de son utilisation, nous avons constaté que l'IA peut être très pertinente pour des tâches de nature formelle ou rédactionnelle.

## 8 Conclusion

Ce projet nous a permis d'appréhender la complexité des structures de données dynamiques. Contrairement à Huffman statique, la version adaptative nécessite une rigueur absolue dans la maintenance des invariants de l'arbre à chaque bit traité. Les tests confirment la complexité théorique linéaire, rendant cet algorithme viable pour la compression de flux en temps réel.