RAPPORT DE PROJET : SYSTÈME DE COMPRESSION LOSSLESS

1. Description du programme

Le programme **SMART-COMPRESSOR** est une solution logicielle de compression de données sans perte (*lossless*). Développé en Python, il est conçu pour traiter des volumes de données importants (fichiers > 100 Mo) en combinant deux approches complémentaires : la réduction de dictionnaire et le codage statistique.

L'objectif principal est de réduire l'espace de stockage tout en garantissant une reconstruction à l'identique (bit à bit) des données originales, vérifiée par une empreinte cryptographique SHA-256.

2. Algorithmes utilisés

A. LZ77 (Lempel-Ziv 1977)

Cet algorithme de compression par dictionnaire utilise une "fenêtre glissante".

- Pour traiter 100 Mo rapidement, nous avons implémenté une **Table de Hachage** (Hash Chains) pour indexer les triplets d'octets. Cela réduit la complexité de O(N×W) à presque O(N), permettant de trouver des correspondances instantanément sans scanner toute la mémoire.

B. Codage de Huffman

Une fois les répétitions traitées par LZ77, le codage de Huffman intervient pour optimiser la taille des symboles restants.

- Il construit un arbre binaire basé sur la fréquence d'apparition des caractères.
- Les caractères fréquents sont codés sur peu de bits, les caractères rares sur plus de bits.

3. Architecture du système

Le système est découpé en modules pour respecter les principes de programmation propre (*Clean Code*) :

- **main.py** : Interface utilisateur, gestion des menus et pilotage des tests.
- **Codec.py** : Orchestrateur de la compression/décompression. Il assure la sérialisation binaire via le module “struct” pour garantir un fichier “.bin” compact.
- **algorithms/** : Dossier contenant les implémentations pures de `lz77.py` et `huffman.py`.
- **utils.py** : Fonctions utilitaires (génération de données, calcul de Hash SHA-256, vérification des contraintes de taille).

4. Résultats expérimentaux

4. Résultats expérimentaux et Validation

Cette section détaille le protocole de test, les mesures effectuées et la validation de l'intégrité des données.

4.1. Protocole de test et Guide d'utilisation

- **Lancement :** Via un terminal avec la commande **python main.py.**
- **Navigation :** Un menu propose deux scénarios :
    - **Option 1 (Génération automatique) :** Crée un fichier **test_data.txt**. C'est le test "étalon" pour vérifier la gestion de volumes massifs et de répétitions cycliques.
    - **Option 2 (Fichier existant) :** Permet de tester des fichiers réels (ex: `enwik8`, `data.csv`).
- **Gestion des contraintes (Sécurité 100 Mo) :** Conformément au cahier des charges, la fonction **verifier_taille()** intercepte tout fichier inférieur à 100 Mo.

4.2. Déroulement technique du traitement

Chaque test suit un pipeline de validation strict pour garantir un résultat *lossless* :

1. **Hachage initial :** Calcul du SHA-256 original.
2. **Compression hybride :** Passage dans LZ77 (fenêtre glissante indexée), sérialisation binaire compacte via **struct**, puis encodage de Huffman.
3. **Décompression automatique :** Inversion immédiate du processus (Huffman puis LZ77) pour restaurer le fichier.
4. **Vérification finale :** Comparaison des hashs SHA-256 (Original vs Restauré).

4.3. Analyse des mesures obtenues

Les tests réalisés sur un fichier de 105 Mo généré (Option 1) présentent les indicateurs suivants :

Métrique	Valeur mesurée
Taille du fichier avant compression	**105.00 Mo**
Taille du fichier après compression	**1.25 Mo**
Taux de compression (Gain)	**98.81 %**
Temps de calcul (Compression)	86.83 secondes
Temps de décompression	41.88 secondes
Vérification de l'intégrité	**RÉUSSIE (Hash SHA-256 identique)**

La Garantie de l'Intégrité (SHA-256)

Pour valider le caractère **"Lossless"** (sans perte) du système, nous avons intégré une vérification par empreinte cryptographique **SHA-256**.

- **Principe :** Le SHA-256 génère une signature unique de 64 caractères hexadécimaux pour un fichier donné. La moindre modification d'un seul octet dans le fichier modifierait radicalement cette signature.
- **Interprétation des résultats :** Dans nos tests, le Hash obtenu avant la compression est strictement identique au Hash obtenu après la décompression (f1dff143...).

4.4. Analyse et Discussion

Les résultats démontrent que l'architecture hybride remplit ses objectifs :

- Le gain de **98.81%** prouve l'efficacité de l'indexation par table de hachage dans LZ77 pour détecter les redondances sur de gros volumes.
- La vitesse de décompression, nettement supérieure à celle de la compression, confirme le comportement asymétrique classique et optimisé des algorithmes de type LZ.
- L'identité parfaite des empreintes SHA-256 valide la robustesse de la chaîne de traitement binaire.

5. Analyse et discussion

L'analyse des résultats démontre la robustesse du système :

1. **Efficacité hybride :** Le gain exceptionnel de 98% sur les données générées prouve que LZ77 capture parfaitement les structures répétitives. Sur des fichiers réels (comme le CSV ou enwik8 qu’on a utilisé), le gain reste significatif (33% à 70%), validant la complémentarité avec Huffman.
2. **Performance :** L'utilisation de **“struct.pack”** au lieu de “**pickle”** a été déterminante. Sans cette optimisation, le fichier compressé était plus lourd que l'original à cause du surpoids des objets Python.
3. **Fiabilité :** La symétrie parfaite entre les processus de compression et de décompression (ordre inversé des algorithmes) assure la réversibilité totale du système.

Conclusion

Le projet remplit tous les critères du cahier des charges : gestion de gros volumes, gain d'espace significatif et garantie de restauration intégrale des données.