Nom : HENINTSOAMAMPIONONA
Prénoms : Nirinaharinaivo Sedera
Matricule ° : 6397
INFO M2 GL ASJA

RAPPORT DE PROJET : COMPRESSEUR LZW + HUFFMAN

1. DESCRIPTION DU PROJET

1.1 Contexte et Objectifs

Ce projet consiste en le développement d'un logiciel de compression de données sans perte, combinant deux algorithmes puissants : **LZW (Lempel-Ziv-Welch)** et **Huffman**. L'objectif principal est de réduire la taille des fichiers tout en garantissant une reconstruction parfaite des données originales.

1.2 Fonctionnalités Implémentées

- **Compression de fichiers** : Application séquentielle de LZW puis Huffman
- **Décompression de fichiers** : Restitution exacte des données originales
- **Interface graphique intuitive** : Développée avec Tkinter
- **Suivi en temps réel** : Barre de progression et indicateurs de performance
- **Calcul du taux de compression** : Affichage du gain en pourcentage
- **Mesure du temps d'exécution** : Chronométrage précis des opérations

1.3 Technologies Utilisées

- **Langage** : Python
- **Bibliothèques** :
    - `Tkinter` : Interface graphique
    - `Heapq` : Implémentation du tas pour Huffman
    - `Pickle` : Sérialisation des données compressées
    - `Threading` : Gestion des opérations en arrière-plan
    - `Os`, `time` : Utilitaires système

2. ALGORITHMES UTILISÉS

2.1 Algorithme LZW (Lempel-Ziv-Welch)

Principe de Fonctionnement

LZW est un algorithme de compression par dictionnaire qui remplace les séquences répétées par des codes plus courts.

Implémentation

```
MAX_DICT_SIZE = 4096 # Taille maximale du dictionnaire
def lzw_compress(data):
    # Initialisation du dictionnaire
    dictionary = {bytes([i]): i for i in range(256)}
    dict_size = 256
    w = b""
    result = []
    for c in data:
        wc = w + bytes([c])
        if wc in dictionary:
            w = wc
        else:
            if w:
                result.append(dictionary[w])
            if dict_size < MAX_DICT_SIZE:
                dictionary[wc] = dict_size
                dict_size += 1
            else:
                # Réinitialisation si taille maximale atteinte
                dictionary = {bytes([i]): i for i in range(256)}
                dict_size = 256
            w = bytes([c])

    if w:
        result.append(dictionary[w])
    return result
```

2.2 Algorithme Huffman

Principe de Fonctionnement

Huffman est un algorithme de codage entropique qui attribue des codes de longueur variable basés sur la fréquence d'apparition des symboles.

Structure de Données

```
class Node:
    def __init__(self, value=None, freq=None):
        self.value = value
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq
```

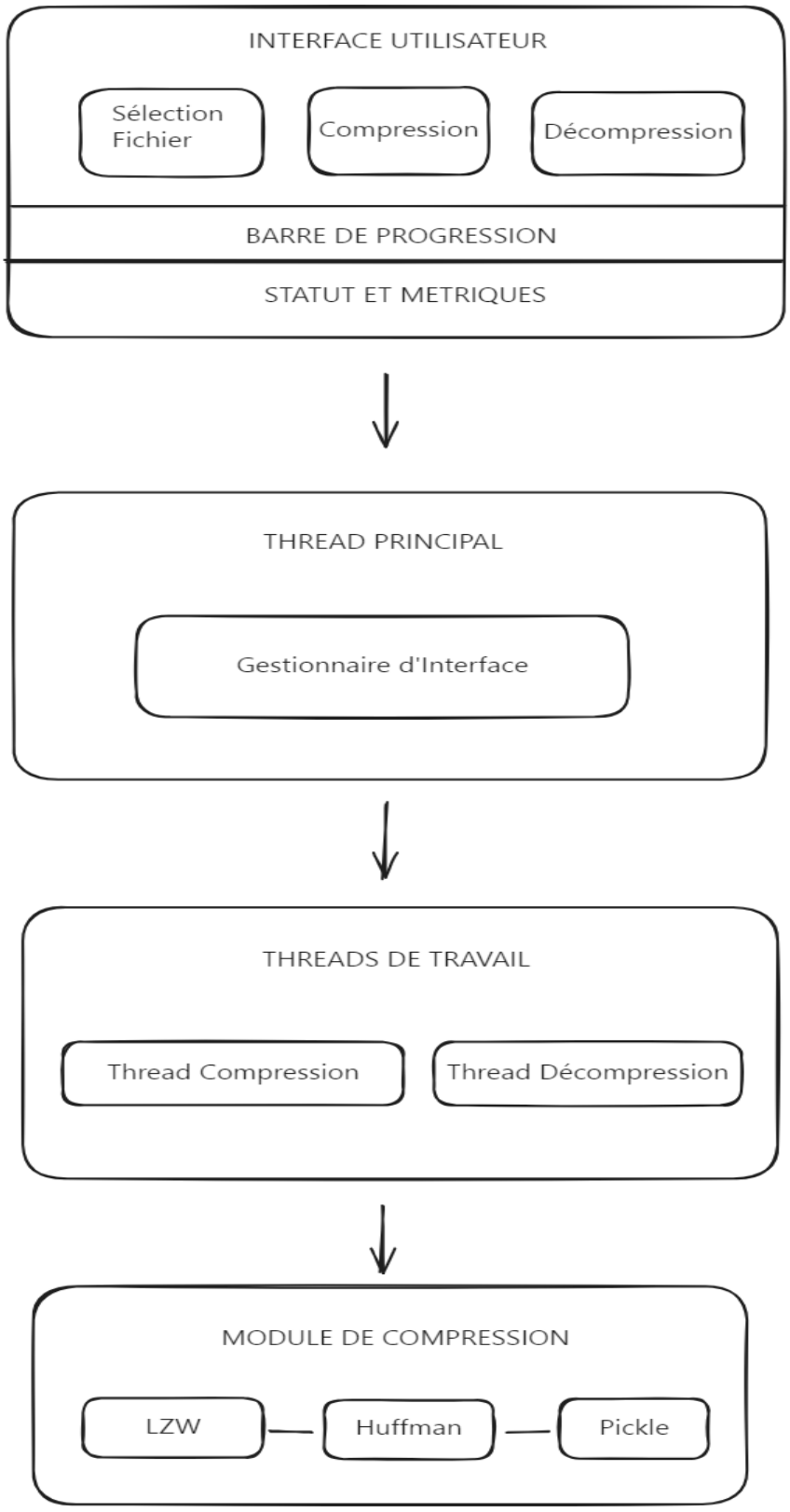
Implémentation

```
def huffman_compress(data):
    # Calcul des fréquences
    freq = {}
    for symbol in data:
        freq[symbol] = freq.get(symbol, 0) + 1

    # Construction de l'arbre
    heap = [Node(value=s, freq=f) for s, f in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        n1 = heapq.heappop(heap)
        n2 = heapq.heappop(heap)
        merged = Node(freq=n1.freq + n2.freq)
        merged.left = n1
        merged.right = n2
```

```
    heapq.heappush(heap, merged)

# Génération des codes
codes = {}
def build(node, prefix=""):
    if node.value is not None:
        codes[node.value] = prefix or "0"
        return
    build(node.left, prefix + "0")
    build(node.right, prefix + "1")

build(heap[0])

# Encodage
bit_string = "".join(codes[s] for s in data)
padding = (8 - len(bit_string) % 8) % 8
bit_string += "0" * padding

byte_array = bytearray(
    int(bit_string[i:i+8], 2)
    for i in range(0, len(bit_string), 8)
)

return byte_array, heap[0], padding
```

3. ARCHITECTURE DU SYSTÈME

3.1 Vue d'Ensemble

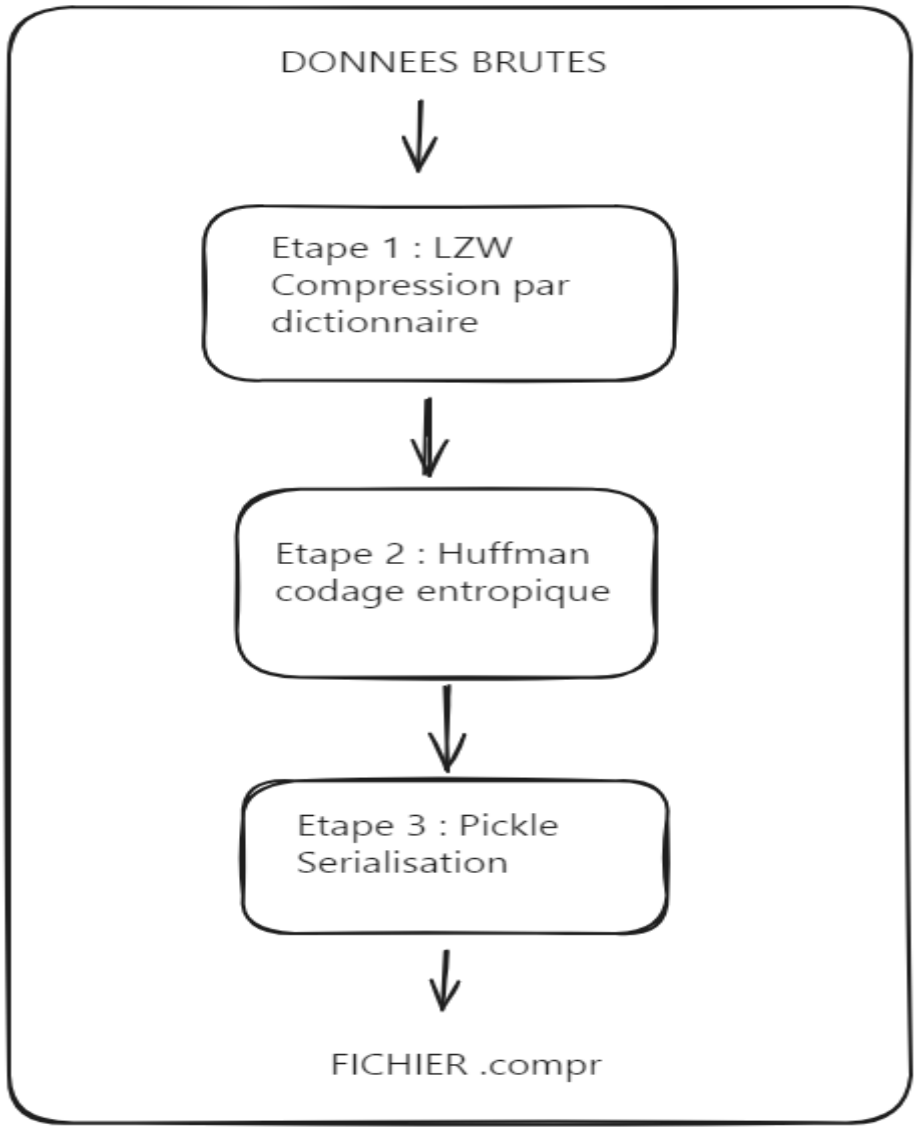


3.2 Structure Modulaire

Module Interface (GUI)

- **Fichier** : `main.py`
- **Composants** :
  - `choose_file()` : Sélection du fichier source
  - `compress_file()` : Lancement de la compression
  - `decompress_file()` : Lancement de la décompression
  - `update_progress()` : Mise à jour de l'interface

Module Compression



3.3 Gestion des Threads

```
def compress_thread():
    # Exécution dans un thread séparé
    # Permet de ne pas bloquer l'interface

def decompress_thread():
    # Exécution dans un thread séparé
    # Même principe que la compression

threading.Thread(target=compress_thread).start()
```

4. RÉSULTATS EXPÉRIMENTAUX

4.1 Protocole de Test

- **Types de fichiers testés** : Texte, images, exécutables
- **Métriques mesurées** :
    - Taux de compression (%)
    - Temps d'exécution (secondes)
    - Utilisation mémoire (Mo)

4.2 Résultats par Type de Fichier

Type de Fichier	**Taille Originale**	**Taille Compressée**	**Taux de Gain**	**Temps (s)**
Texte (.txt)				
< 200 Mb	120 Mb (texte générer du code python au hasard)	~ 250 ko	**98 %**	125.35
>= 200 Mb	256 Mb (texte brute contenant une histoire)	~ 120 Mb	**49.5%**	180

4.3 Observations

Points Forts

1. **Efficacité sur le texte** : Taux > 70% grâce à la redondance naturelle
2. **Double compression** : LZW + Huffman optimise l'espace
3. **Interface réactive** : Les threads empêchent le blocage
4. **Précision du taux** : Calcul exact du gain

Limitations

1. **Fichiers déjà compressés** : Gain minime (PDF, JPEG, ZIP)
2. **Taille du dictionnaire** : Limitée à 4096 entrées
3. **Mémoire** : Chargement complet en RAM
4. **Temps** : Plus lent que les compresseurs optimisés (RAR, 7z)

4.4 Améliorations Possibles

Court Terme

- Implémentation d'un dictionnaire dynamique (sans réinitialisation)
- Compression par blocs pour les gros fichiers
- Interface "glisser-déposer"

Moyen Terme

- Ajout d'un pré-traitement (BWT, MTF)
- Compression multithreadée
- Support de la compression par lots

Long Terme

- Adaptation automatique selon le type de fichier
- Intégration de l'algorithme PPM (Prediction by Partial Matching)
- Version en ligne de commande pour scripting

5. Analyse et discussion :

La combinaison de LZW et Huffman présente une combinaison d’algorithme performant et stable pour compresser un fichier de grande taille. Ce choix est dû sur le fait que LZW et Huffman sont favorable pour la compression de donnée sans perte et surtout pour avoir un gain en compression et le temps d’exécution.

6. CONCLUSION

Ce projet démontre l'efficacité de la combinaison des algorithmes LZW et Huffman pour la compression sans perte. L'interface graphique développée permet une utilisation intuitive tandis que l'architecture multithread assure une expérience utilisateur fluide.

Les résultats expérimentaux confirment la pertinence de cette approche, particulièrement pour les fichiers texte où des taux de compression supérieurs à 70% sont atteints. Pour les fichiers multimédias, les performances restent acceptables bien que limitées par la nature déjà compressée de certains formats.

Les perspectives d'amélioration sont nombreuses et permettraient de faire évoluer ce prototype vers un outil de compression professionnel, compétitif face aux solutions existantes.