SAE 3.2 – Application PIF

Primitive Image Format

BUT Informatique – 2ème année – Groupe 4

Équipe de développement

Youness BOULALAM	`youness`
Algassimou DIALLO	`Diallo-VM-fbleau`
Ayoub ANHDIRE	`anhdire`

Technologie :	Java – Architecture MVC
Dépôt Gitea :	https://grond.iut-fbleau.fr/dialloa/SAE32_2025
Encadrant :	M. Luc Hernandez
Date de rendu :	11 janvier 2025

Sommaire

1 Introduction

Pour cette deuxième SAE du semestre 3, il nous a fallu réaliser **deux** programmes : un convertisseur d'image au format PIF et un visualisateur. Le convertisseur prend en entrée une image PNG (ou tout format supporté par `ImageIO.read()`) via argument ou **JFileChooser**. L'interface affiche les tables de fréquences, codes Huffman et codes canoniques, avec possibilité d'export en PIF.

Le visualisateur affiche une image PIF dans une fenêtre adaptée : image centrée si plus petite, déplaçable à la souris si plus grande.

Ce projet a été réalisé en trinôme en Java, avec architecture MVC, makefile, et interface graphique native. L'ergonomie a été soignée pour une utilisation simple et claire.

FIGURE 1 – Interface du convertisseur

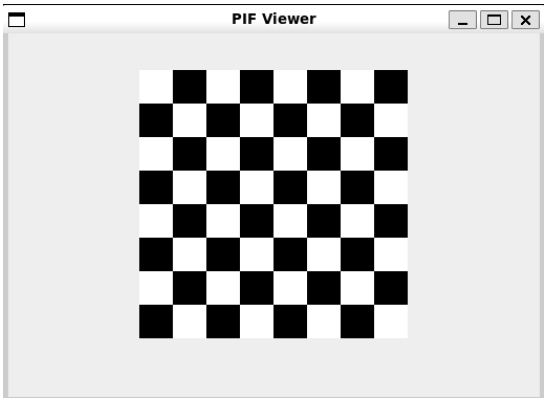


FIGURE 2 – Interface du visualisateur

2 Répartition des tâches

Nom	Tâches effectuées
Algassimou Pellel DIALLO	— Création de la structure générale du projet (Dossiers, UML) — Implémentation de BitOutputStream et BitInputStream — Implémentation complète du contrôleur (ConverterController) — Génération et affichage des fréquences — Travail sur HuffmanTree et HuffmanNode — Intégration de l'interface graphique (ConverterWindow) — Gestion du bouton Export et sauvegarde .pif (thread séparé) — J'ai coder le Makefile dans sa totalité — Implémentation du PIFWriter et PIFReader — Implémentation du contrôleur pour le visualisateur (ViewControleur.java) — Implémentation du visualisateur avec toute les specificité demander dans le sujet. (Gestion de la taille, deplacement avec la souris, etc ...).

Nom	Tâches effectuées
Youness BOULALAM	— Gestion des erreurs et messages utilisateur — Conversion RGBImage ↔ BufferedImage (Viewer et Convertisseur) — Ouverture du fichier .pif via argument ou JFileChooser — Support sur les tâches des coéquipiers
Ayoub ANHDIRE	— Javadoc (Équipe) — Génération des codes Huffman et codes canoniques — Interface d'affichage des fréquences — Test de BitOutputStream — Lecture de l'en-tête — Diagramme de classe

3 Fonctionnalités principales

3.1 Conversion au format PIF

La conversion d'une image vers le format PIF repose sur plusieurs étapes : analyse des composantes RGB, calcul des fréquences, construction de l'arbre de Huffman, création des codes canoniques, puis écriture finale dans un fichier binaire structuré. Cette section présente les mécanismes mis en place et la contribution de chaque membre du groupe.

3.1.1 Contribution de Algassimou Pellel Diallo

Dans ce projet, je me suis surtout occupé de toute la partie qui concerne le fonctionnement général du convertisseur PIF, ainsi que de la coordination entre le traitement et l'interface graphique. Mon rôle a été de faire en sorte que le programme suive un déroulement clair, et compréhensible.

Représentation de l'image avec RGBImage Lors du chargement d'une image, nous devons la stocker sous une forme qui soit simple à manipuler. Nous avons décidé d'utiliser une classe **RGBImage** qui représente l'image comme un tableau 2D de pixels.

Pourquoi un tableau 2D ? Parce que cela offre plusieurs avantages :

— **Accès direct** : grâce aux coordonnées (x, y), et aussi la mémoire est contiguë O(1) et cache-friendly, on peut accéder rapidement à n'importe quel pixel.
— **Exploitation** : parcourir le tableau est simple (deux boucles imbriquées) et efficace.
— **Compatibilité** : les valeurs de pixels stockées peuvent directement être converties en `BufferedImage` pour l'affichage.

Chaque pixel stocke trois valeurs entières : rouge, vert et bleu (RGB), chacune entre 0 et 255.

J'ai entendu parler d'une valeur alpha mais on ne l'a pas utilisée dans le projet.

Organisation du contrôleur et déroulement de la conversion J'ai mis en place la structure du contrôleur, qui sert de lien entre le traitement et l'affichage. C'est lui qui décide

de l'ordre des opérations et de la façon dont les informations sont envoyées à la fenêtre du convertisseur. La conversion s'appuie sur cinq méthodes principales :

— `loadImage` : charge l'image choisie et la transforme en structure RGB exploitable ;
— `computeFrequencies` : analyse tous les pixels et crée les tableaux de fréquences ;
— `computeHuffman` : génère les trois arbres de Huffman (un pour R, un pour G, un pour B) ;
— `computeCanonical` : crée les codes canoniques utilisés pour la compression ;
— `saveAsPIF` : écrit le fichier final (déclenche l'écrivain) (`.pif`) avec l'en-tête, les tables et les bits encodés.

Le contrôleur gère aussi le comportement selon les arguments donnés par l'utilisateur. S'il fournit deux chemins en ligne de commande, alors la conversion et la sauvegarde se font semi-automatiquement (le user déclenche quand même la sauvegarde via le bouton). Sinon, l'utilisateur passe par des `JFileChooser`.

BitOutputStream et BitInputStream Une autre partie importante de mon travail a été l'utilisation et l'adaptation de deux classes essentielles : **BitOutputStream** et **BitInputStream**. Elles servent à manipuler les données bit par bit. Même si dans la théorie ce ne sont pas vraiment des décorateurs, comme me l'a expliqué M. Florant Madeleine (merci à lui), elles fonctionnent quand même comme une couche au-dessus des flux classiques.

Ces classes permettent d'écrire et de lire des bits de manière précise, ce qui est indispensable pour un format comme le PIF. Par exemple, les codes Huffman ne mesurent pas toujours un multiple de 8, donc on doit absolument travailler au niveau du bit. C'est grâce à ces classes que la compression finale est propre et sans gaspillage.

Le casse-tête des threads et le blocage de la fenêtre Une difficulté importante que j'ai rencontrée concerne la sauvegarde du fichier. Au début, quand j'appelais `saveAsPIF` directement depuis le bouton "Exporter", l'interface se figeait complètement. La fenêtre devenait impossible à fermer, impossible à bouger, et parfois même Windows indiquait que le programme "ne répond pas". La seule manière de tout arrêter était d'utiliser le gestionnaire de tâches.

Au début, je ne comprenais pas d'où venait le problème. J'ai donc fait plusieurs tests, et j'ai remarqué que le blocage apparaissait exactement au moment où l'écriture du fichier PIF commençait. C'est en cherchant sur StackOverflow et Reddit que j'ai compris que Swing utilise un seul thread pour gérer toute l'interface graphique. Dès que ce thread est occupé par une opération longue, tout le programme se bloque.

Écrire un fichier PIF peut prendre du temps, surtout pour de grandes images. Le thread graphique ne pouvait donc plus s'occuper de la fenêtre.

Pour résoudre ce problème, j'ai complètement déplacé la sauvegarde dans un thread séparé. J'ai créé une classe dédiée qui hérite de `Thread`. L'exécution se fait ainsi :

Désolé je cite un peu mon code mais c'est plus simple pour expliquer :

```
ThreadSauvegardePIF thread = new ThreadSauvegardePIF(this,
   fichierSortie);
thread.start();
```

À partir de ce moment-là, la fenêtre n'a plus jamais été bloquée. J'ai rajouté un dialogue de progression, pour éviter que l'utilisateur ne se sente un peu perdu pendant le mini laps de temps que cela prend, et un message de confirmation apparaît lorsque la sauvegarde est

terminée. Ce problème m'a beaucoup appris, car je n'avais jamais réalisé que Swing reposait autant sur un thread unique. Cela m'a permis de comprendre pourquoi certaines opérations doivent absolument être effectuées en arrière-plan.

Mise en place de la fenêtre du convertisseur Enfin, j'ai organisé la fenêtre principale pour afficher les différentes étapes du traitement : aperçu de l'image, fréquences, codes Huffman et codes canoniques.

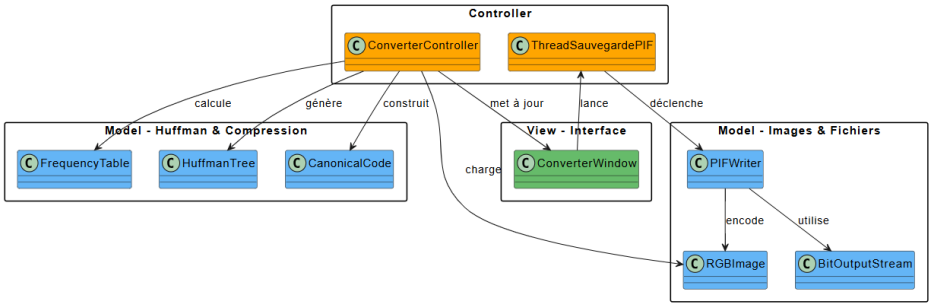


FIGURE 3 – Architecture MVC du convertisseur - Flux de conversion (Bleu : Model, Orange : Controller, Vert : View)

3.1.2 Contribution de Ayoub Anhdire

Un *arbre binaire* est une structure abstraite composée de nœuds dont la principale contrainte est qu'un nœud doit avoir au maximum deux enfants :

— un enfant gauche,
— un enfant droit.

Comment l'arbre d'Huffman est construit ? Au préalable, pour construire l'arbre binaire de Huffman, il nous faut calculer les fréquences pour les composantes R, G et B. Après avoir calculé ces fréquences, pour construire l'arbre d'Huffman, il nous faut prendre les feuilles avec les plus basses fréquences, c'est-à-dire les plus proches de zéro : en l'occurrence, il faut en prendre deux et à partir de ces deux feuilles, on crée un nouveau nœud qui a comme fréquence la somme des feuilles correspondantes. Lorsqu'il ne reste plus qu'une feuille et qu'on ne peut plus appliquer ce principe : alors cela signifie que c'est la racine de l'arbre. Elle est censée avoir la plus grande fréquence que les feuilles de départ. Autrement dit, plus la fréquence est grande, plus le code Huffman associé sera court.

Essayons d'expliquer ce principe avec un diagramme objet et un petit dessin de l'arbre. Prenons l'exemple le plus simple avec simplement deux feuilles, voici le diagramme objet basé sur notre code et un dessin de l'arbre de ce diagramme objet :

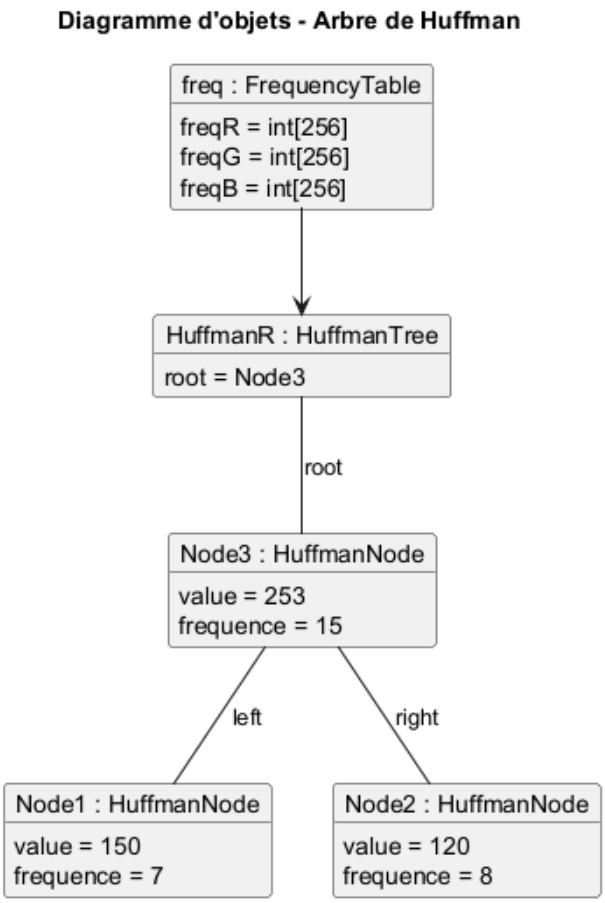


FIGURE 4 – Diagramme Objet - Construction de l'arbre Huffman

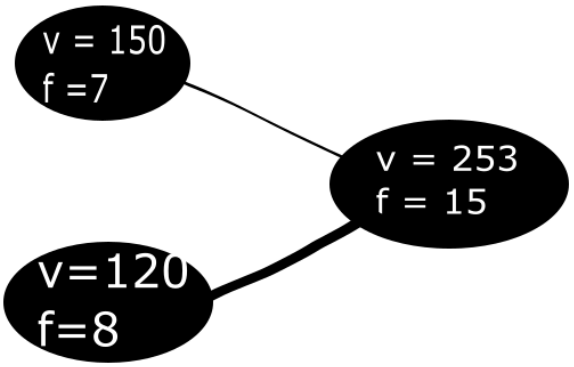


FIGURE 5 – Dessin Arbre Huffman - Basé sur le diagramme objet

Puisqu'il y a trois composantes R, G et B, il est censé avoir 3 arbres Huffman mais pour simplifier la compréhension, nous n'en avons fait qu'un seul : celui de la composante Rouge. Expliquons le diagramme objet : nous avons un objet *freq* de la classe **FrequencyTable**. Cette classe permet d'initialiser les trois tableaux de fréquences (R, G et B) et ces tableaux sont passés en argument dans le constructeur de classe **HuffmanTree**.

Après ça on peut remarquer la présence de deux feuilles *Node1* et *Node2*. Elles ont comme fréquence respective 7 et 8. Comme répété plus haut, pour faciliter la compréhension, nous avons choisi que deux feuilles. Voici la base de notre arbre. On remarque la présence d'un nœud *Node3* avec comme fréquence, il a été obtenu en faisant la somme des deux nœuds *Node1* et *Node2* et la fréquence obtenue est 15, donc 7+8. C'est la seule feuille qui reste dans notre arbre : on en conclut que c'est la racine de notre arbre, comme en témoigne l'attribut root de la classe **HuffmanTree**.

Après avoir compris le principe de comment construire l'arbre Huffman, comment générer les codes Huffman ? Nous avons codé cela de manière récursive : si on saute vers un fils gauche on ajoute 0 et si on saute vers un fils droit on ajoute 1 : les codes sont enregistrés dans des dictionnaires : **Map<Integer,String>**. Pour chaque feuille, pour obtenir son code, on parcourt l'arbre de la racine jusqu'à la feuille.

La question qu'on se pose désormais : c'est est-ce que cette solution est optimale ? La réponse est oui ! Pourquoi ? Comme on a placé les symboles fréquents près de la racine, la moyenne des longueurs de tous les codes est minimale. De plus, les codes sont différents, puisqu'aucun code n'est le début d'un autre code, donc il n'y a pas d'erreur possible à la lecture. Même si certains

symboles ont la même fréquence et que l'arbre peut être légèrement différent, la longueur moyenne reste toujours la plus courte possible. C'est pour cela que la génération des codes à partir de l'arbre de Huffman est optimale : aucun autre code ne peut donner une longueur moyenne plus courte pour les mêmes symboles.

Les codes canoniques et leur logique Un *code canonique* est une version basée sur les codes Huffman : la longueur de chaque code Huffman est préservée mais les codes sont réorganisés de manière en commençant par les codes les plus courts. On commence par trier les codes initiaux par longueur du code puis par valeur. Les nouveaux codes s'obtiennent ainsi : le premier est rempli de zéro, le deuxième commençant par 1 et rempli de zéro à droite mais bien faire attention à ce que la longueur ne soit pas dépassée, le troisième commençant par 11 et ainsi de suite jusqu'à avoir réalisé cela, jusqu'à ce que toutes les valeurs aient un code canonique.

Pour cela, la démarche que nous avons employée est celle-là : on récupérait les entrées (des dictionnaires en l'occurrence) des codes Huffman afin de les trier, puis on a trié la liste avec un **Comparator** que nous avons implémenté : on compare d'abord par la longueur des codes (longueur de la valeur dans le dictionnaire) ou sinon on trie par rapport à la valeur de la clé. Puis on fait une boucle qui parcourt toute la liste, on attribue un code canonique à chaque symbole qu'on ajoute dans un dictionnaire : **Map<Integer,String>**.

Pourquoi les codes canoniques au lieu des codes Huffman ? Pour le décodage d'un fichier au format PIF, le fait de stocker l'arbre d'Huffman prendrait énormément de place et de temps mais on peut restituer ces codes Huffman grâce aux codes canoniques. Il nous suffit juste de connaître la longueur des codes et l'ordre des symboles pour pouvoir les reconstituer. On garde la même longueur que les codes initiaux, donc la compression reste optimale.

Le résumé de ces deux principes L'algorithme de Huffman sert à coder les symboles avec des codes plus courts pour les symboles fréquents et plus longs pour les rares, ce qui permet de gagner de l'espace. Les codes sont optimaux, puisqu'aucun code n'est le début d'un autre, donc on peut les lire sans erreur. Les codes canoniques sont une version plus simple des codes Huffman : ils gardent la même longueur pour chaque symbole mais suivent d'autres contraintes qui les diffèrent des codes initiaux. Cela permet de stocker moins de données et de décoder plus vite, tout en gardant la même efficacité que Huffman.

3.2 Visualisateur au format PIF

Le visualisateur a pour rôle de lire un fichier au format `.pif` et de reconstruire l'image d'origine. Pour cela, il ne récupère pas directement les codes Huffman, ni l'arbre utilisé lors de la compression. Le fichier `.pif` ne contient qu'une information minimale : les longueurs des codes canoniques pour chaque symbole (un symbole étant ici une valeur de couleur entre 0 et 255). À partir de ces longueurs, le visualisateur reconstruit entièrement les codes puis les arbres nécessaires au décodage.

Le point d'entrée principal est la méthode `decodePifFile()` qui orchestre l'ensemble du processus : lecture de l'en-tête, récupération des tables de longueurs, reconstruction des codes canoniques, construction des arbres et décodage des pixels.

3.2.1 Structure complète du fichier PIF

Le fichier `.pif` est organisé de la manière suivante :

1. **En-tête** (4 octets) :
   — Largeur de l'image sur 16 bits,
   — Hauteur de l'image sur 16 bits.
2. **Tables de longueurs** (768 octets) :
   — 256 longueurs pour le Rouge,
   — 256 longueurs pour le Vert,
   — 256 longueurs pour le Bleu.
3. **Données compressées** (taille variable) :
   — Flux de bits contenant les codes Huffman canoniques de chaque pixel,
   — Les pixels sont encodés ligne par ligne (R, G, B pour chaque pixel).

3.2.2 La forme des tables de codes dans le visualisateur

Juste après l'en-tête du fichier (largeur et hauteur, chacune sur 16 bits), le fichier PIF contient trois tables :

— 256 longueurs pour la composante Rouge,
— 256 longueurs pour la composante Verte,
— 256 longueurs pour la composante Bleue.

Chaque longueur est stockée sur 8 bits. Cela donne au total :

$$256 \times 3 = 768 \text{ octets de longueurs}$$

La méthode `readHeader()` lit d'abord la largeur et la hauteur de l'image (chacune sur 16 bits), puis `readCanonicalTables()` parcourt les trois tables de 256 valeurs pour stocker les longueurs dans les tableaux `lenR`, `lenG` et `lenB`.

Ces longueurs correspondent à celles des codes canoniques générés pendant la compression. Aucun code Huffman ni aucun arbre n'est sauvegardé. Le visualisateur doit tout reconstruire à partir de ces seules informations.

Ce choix permet d'obtenir un fichier plus compact qu'un format brut comme le BMP. En revanche, il ne peut pas rivaliser avec des formats déjà hautement compressés tels que PNG ou JPG. D'ailleurs, au début du projet, je pensais que notre fichier `.pif` serait toujours plus léger que tout autre format, mais je me suis rendu compte en testant et en cherchant sur le Web (Wikipedia, etc.) que PNG et JPG sont beaucoup plus optimisés que ce qu'on fabrique ici. En revanche, par rapport à un fichier BMP, notre fichier PIF est bel et bien plus léger.

La méthode `isPIFFile()` permet de vérifier qu'un fichier est valide avant de tenter le décodage : elle contrôle l'existence du fichier, son extension `.pif` et une taille minimale de 772 octets (4 octets d'en-tête + 768 octets de tables).

3.2.3 Reconstruction des codes canoniques

À partir des longueurs lues, la méthode `rebuildCanonical()` reconstruit les codes selon le principe suivant :

1. On récupère chaque symbole dont la longueur est non nulle.
2. On trie les couples (symbole, longueur) à l'aide d'un `ComparateurEntreeCanonique` :
   — d'abord par longueur croissante,
   — puis par valeur du symbole.
3. On génère les codes dans cet ordre :
   — le premier code d'une longueur donnée est rempli de zéros,
   — les suivants sont obtenus en incrémentant un compteur binaire,
   — lorsque la longueur augmente, on décale le compteur avec l'opération `code «= (length - previousLength)` pour l'aligner correctement.

Le résultat est stocké dans une `Map<String, Integer>` où la clé est le code binaire sous forme de chaîne et la valeur est le symbole correspondant.

3.2.4 Choix de l'arbre plutôt que du dictionnaire

Pour décoder les données compressées, j'aurais pu utiliser un simple dictionnaire où chaque code binaire serait associé à son symbole. Cette approche aurait été plus simple à implémenter : il suffirait d'accumuler les bits lus et de vérifier à chaque étape si la chaîne obtenue existe dans le dictionnaire.

Cependant, j'ai finalement choisi d'utiliser un **arbre de décodage**, même si cela était plus difficile à coder. Ce choix s'explique par plusieurs raisons :

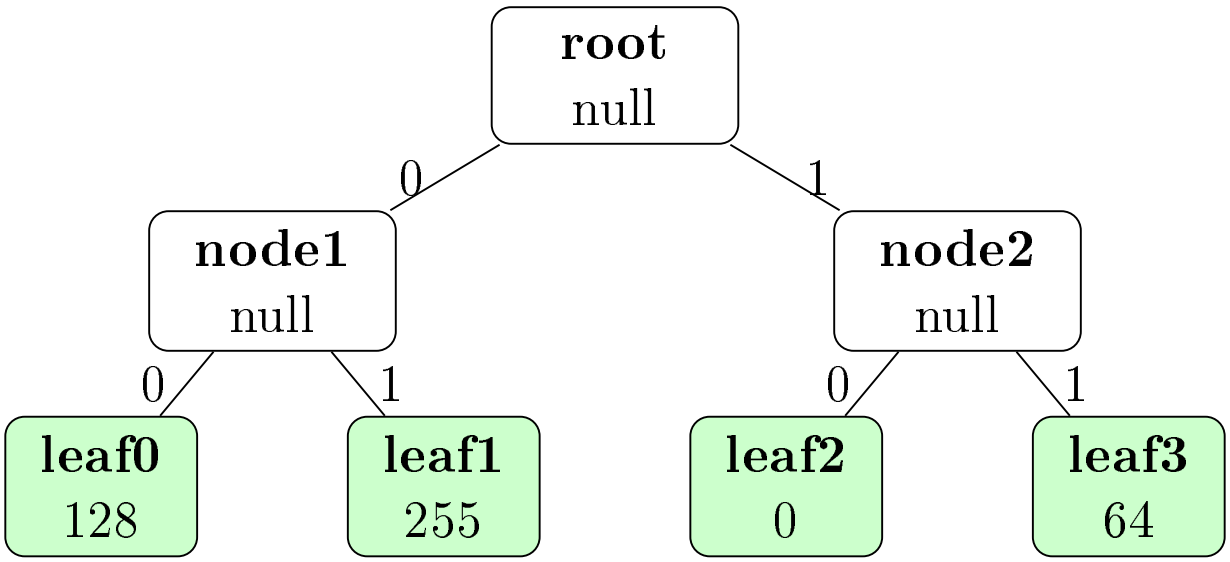
— **Performant** : Avec un dictionnaire, il faudrait accumuler les bits lus et tester à chaque étape si la chaîne obtenue correspond à un code existant. Cette approche nécessite de nombreuses recherches dans le dictionnaire et n'est pas efficace.
— **Parcours DFS** : L'arbre permet un parcours où chaque bit lu (0 ou 1) détermine directement si l'on descend à gauche ou à droite. Dès qu'on atteint une feuille, on a trouvé le symbole sans aucune recherche supplémentaire.
— **Approche standard** : En regardant plusieurs vidéos explicatives sur YouTube concernant le décodage Huffman, j'ai constaté que toutes utilisaient un arbre de décodage plutôt qu'un dictionnaire. Cela m'a conforté dans l'idée que cette approche est la méthode de référence pour ce type de problème.

3.2.5 Construction de l'arbre de décodage

Une fois les codes reconstruits sous forme de chaînes de bits, la méthode `buildDecodageTree()` crée un arbre binaire composé de nœuds `DecodeNode`. Le parcours suit la règle suivante :

— `0` signifie descendre à gauche (`current.left`),
— `1` signifie descendre à droite (`current.right`).

Lorsqu'on arrive au dernier bit d'un code, la méthode crée une feuille contenant le symbole associé via `new DecodeNode(null, null, symbol)`. Ce symbole est la valeur d'une composante (entre 0 et 255). Ce procédé est répété pour les trois composantes : Rouge, Vert et Bleu. On obtient ainsi trois arbres distincts (`trieR`, `trieG`, `trieB`).



Code	Valeur
00	128
01	255
10	0
11	64

FIGURE 6 – Diagramme Objet – Arbre de décodage Huffman

3.2.6 Décodage des pixels

Une fois les arbres construits, la méthode `decodePixels()` lit le reste du fichier bit par bit grâce au flux `BitInputStream`. Pour chaque pixel :

1. On appelle `decodeSymbole(in, trieR)` pour parcourir l'arbre Rouge jusqu'à tomber sur une feuille et obtenir la valeur du rouge.
2. On appelle `decodeSymbole(in, trieG)` pour l'arbre Vert.
3. On appelle `decodeSymbole(in, trieB)` pour l'arbre Bleu.

À la fin, les trois valeurs retrouvées permettent de créer un objet `Pixel` qui est placé dans l'image via `image.setPixel(x, y, pixel)`. L'ensemble des pixels donne l'image complète sous forme de `RGBImage`.

La méthode `decodeSymbole()` effectue ce travail pour une seule composante : elle parcourt l'arbre avec `in.readBit()` jusqu'à ce que `current.isLeaf()` soit vrai, puis retourne `current.value`. Grâce à la propriété des codes canoniques (aucun code n'est préfixe d'un autre), le décodage est fiable et ne provoque aucune ambiguïté.

3.2.7 Pourquoi cette méthode fonctionne

Cette approche fonctionne parce que :

— la longueur de chaque code suffit pour reconstruire la même structure qu'un arbre Huffman,
— les codes canoniques forment un ensemble non ambigu (pas de code préfixe),
— le décodage bit par bit suit un chemin déterministe dans l'arbre,
— l'utilisation de l'arbre plutôt que du dictionnaire rend le décodage plus efficace,

— seules les informations vraiment nécessaires sont stockées dans le fichier.

Même si le fichier PIF n'est pas aussi compact que les formats modernes, il reste nettement plus léger qu'un fichier BMP tout en étant suffisamment simple pour que l'on puisse reconstruire l'image seulement avec la table des longueurs.

FIGURE 7 – Diagramme Objet – Arbre de décodage Huffman

4 Makefile

4.1 Gestion des dépendances

Le projet utilise un **Makefile** pour automatiser la compilation. Les classes interdépendantes sont compilées ensemble :

```
$(BIN)/$(PKG_CONV)/ConverterController.class \
$(BIN)/$(PKG_CONV)/ConverterWindow.class \
$(BIN)/$(PKG_CONV)/ExportButtonListener.class \
$(BIN)/$(PKG_CONV)/ThreadSauvegardePIF.class: \
    $(SRC)/$(PKG_CONV)/ConverterController.java \
    $(SRC)/$(PKG_CONV)/ConverterWindow.java \
    $(SRC)/$(PKG_CONV)/ExportButtonListener.java \
    $(SRC)/$(PKG_CONV)/ThreadSauvegardePIF.java
    $(JAVAC) -d $(BIN) -sourcepath $(SRC) $^
```

4.2 Génération des JARs

Le Makefile génère deux JARs exécutables :

— `pifConverter.jar` : le convertisseur d'images
— `pifViewer.jar` : le visualisateur d'images PIF

4.3 Commandes disponibles

Commande	Description
`make`	Compile tout le projet (équivalent à `make all`)
`make runnotjar-conv ARGS="..."`	Lance le convertisseur sans créer de JAR
`make runnotjar-view ARGS="..."`	Lance le visualisateur sans JAR
`make jar-conv`	Génère `pifConverter.jar`
`make jar-view`	Génère `pifViewer.jar`
`make jar`	Génère les deux JARs
`make run-conv ARGS="..."`	Exécute le convertisseur via JAR
`make run-view ARGS="..."`	Exécute le visualisateur via JAR
`make doc`	Génère la Javadoc dans `docjava/`
`make clean`	Supprime `build/`, JARs et documentation

4.4 Exemple d'exécution

```
# Conversion d'une image
make run-conv ARGS="image.png output.pif"

# Visualisation d'un fichier PIF
make run-view ARGS="output.pif"

# Nettoyage
make clean
```

5 Conclusion

5.1 Youness BOULALAM

Dans ce projet, j'ai pu, contrairement au précédent, échanger avec mes collaborateurs afin de rendre la meilleure version possible du projet et d'avoir une vue d'ensemble de celui-ci.

Sans vous mentir, le Java n'est pas vraiment ma tasse de thé, mais lorsqu'on est assisté, on peut plus facilement comprendre et moins rester bloqué sur des concepts qui peuvent nous démotiver, voire nous dégoûter du projet.

Pour conclure, je souhaite remercier mes camarades Algassimou et Ayoub, ainsi que vous, M. Hernandez, de nous avoir permis de réaliser ce projet et de le mener à bien.

5.2 Algassimou DIALLO

Pour conclure, ce projet m'a beaucoup apporté, même s'il m'a posé plusieurs casse-têtes, notamment avec le gel de la fenêtre lors de la sauvegarde et la gestion des flux binaires. J'ai dû comprendre d'où venaient ces problèmes et chercher des solutions propres, comme le déplacement de la sauvegarde dans un *thread* dédié.

J'ai aussi découvert la complexité d'un vrai projet Java : les arbres, les codes canoniques, le décodage bit par bit, mais aussi la structure générale du programme et les dépendances lors

de la compilation. Malgré les difficultés, j'ai apprécié le travail, car chaque blocage m'a permis de progresser et de mieux comprendre ce que je faisais. Au final, ce projet a été une bonne expérience et m'a réellement aidé à monter en compétence.

5.3 Ayoub ANHDIRE

Pour conclure, j'ai bien aimé ce projet en général, j'ai pris du plaisir à coder en Java d'autant plus que j'affectionne la programmation orientée objet. Ce projet m'a permis d'augmenter mes compétences techniques, plus précisément dans la compréhension de structures abstraites notamment les dictionnaires ou encore les arbres. J'ai pu travailler d'autres notions comme la récursivité où j'avais du mal à comprendre la logique mais grâce à ce projet, j'ai pu m'améliorer.

J'ai pu aussi développer mes qualités de communication avec mes camarades : chacun a joué un rôle où il sait qu'il va perfectionner et la communication a été un enjeu majeur dans cette SAE car lorsque quelqu'un était bloqué, il faisait signe et ne restait pas tout seul sans avancer. En conclusion, ce projet a été pour moi une expérience enrichissante.