Projet Arbres et compression d'image

par Aliman ASSAWA et Elio BTEICH

(GROUPE 584I)

I) Introduction

Dans le cadre d'un projet autour de l'algorithmique et des structures de données pour la Licence Informatique 3 à l'Université de Nantes. Nous avons dû,par groupe de deux personnes, implémenter des algorithmes de compression d'image au format Portable Graymap Format (PGM), pour réduire la taille d'un fichier représentant une image, au prix d'une baisse de qualitée. Ce projet a pour nous été un réel défi, car il nous a fallu respecter un cahier de charges bien précis tout en ayant pour objectif d'avoir un algorithme le plus optimal possible. Toutefois, ce projet s'est révélé être une belle aventure qui nous a permis de mettre en pratique toutes les notions abordées dans notre cursus.

Dans ce rapport, Vous trouverez d'une part, une brève présentation du cahiers des charges et des contraintes posées par nos encadrées de TP. Et d'autre part, nous présenterons notre algorithme et mettrons en évidence les les raisonnements, les difficultées et les choix qui nous ont permis d'aboutir à ce résultat!

II) Cahier des charges et contraintes

1) Structure du PGM

Le format PGM est caractérisé par un en-tête (header) suivi des données de l'image. L'en-tête est composé de trois informations essentielles, comme illustré dans l'exemple suivant :

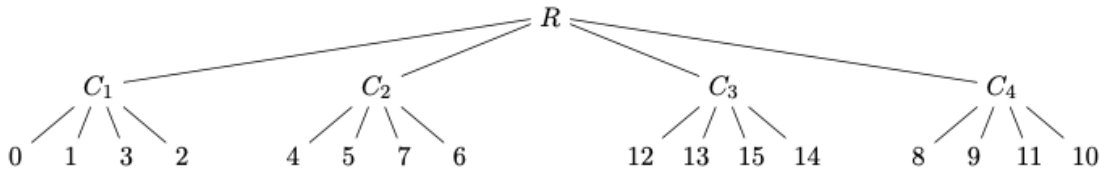
```
P2
# Shows the word "FEEP" (example from Netpbm man page on PGM)
24 7
15
```

Dans l'Exemple 1, la première ligne correspond au "Magic number", qui détermine le format utilisé (P1 pour une bitmap, P2 pour une image en niveaux de gris, P3 pour une image couleur). Dans ce projet, nous travaillerons avec des images en niveaux de gris : les fichiers que vous manipulez commenceront donc toujours par P2. La deuxième ligne est un commentaire. La troisième ligne donne la largeur et la hauteur de l'image, en pixels ; Pour ce projet nous nous contenterons d'une image de taille 2^n × 2^n avec un entier.La quatrième ligne donne la valeur maximale max de luminosité d'un pixel(elle doit être inférieure ou égale à 255). Les données de l'image (ici, le niveau de gris de chaque pixel) sont ensuite présentées comme une série de nombres, obligatoirement compris entre 0 et max et séparés par des espaces et/ou des \n..

2) Représentation des Images avec des Quadtrees

Dans le cadre de ce projet, nous envisageons de représenter les images PGM à l'aide de quadtrees. L'objectif est de compresser ces images en réduisant au maximum l'espace de stockage tout en préservant une qualité d'image acceptable. Les quadtrees, vus en cours, seront utilisés pour cette représentation.
Un quadtree est une structure arborescente qui divise l'espace en quatre quadrants, d'où son nom. Dans le contexte des images, chaque nœud de l'arbre représente une zone rectangulaire de l'image. Les feuilles de l'arbre contiennent des informations sur la luminosité des pixels dans la région correspondante.
Voici un exemple d'image PGM ainsi que le quadtree correspondant:

```
P2
4 4
15
0   1   4   5
2   3   6   7
8   9  12  13
10 11  14  15
```



3) Diminution de la représentation en mémoire

A. Compression Lambda

La première méthode de compression consiste à regrouper quatre feuilles ayant le même parent sous la forme d'une seule feuille au niveau supérieur. Cela permet de réduire le nombre d'éléments dans l'arbre.
La compression Lambda s'effectue en calculant la moyenne logarithmique de luminosité (Λ) des quatre feuilles d'une brindille.
La luminosité de la nouvelle feuille résultante est la valeur de Λ arrondie à l'entier le plus proche.

B. Compression Rho

La compression Rho prend en compte l'écart maximal (ε) entre la moyenne logarithmique de luminosité d'une brindille et la luminosité de chacune de ses feuilles. Si
ε est trop élevé, la brindille est compressée pour éviter une perte d'information significative.
La compression Rho s'arrête dès que le taux de compression atteint un seuil spécifié (ρ). Le taux de compression est défini comme le ratio entre le nombre de nœuds du quadtree courant et le nombre de nœuds du quadtree initial.

Les sections suivantes du rapport détaillent l'implémentation de ces algorithmes dans le cadre du projet, en fournissant des exemples et des analyses de complexité.

III) Utilisation du programme

1. Installation

Afin de pouvoir faire tourner notre programme de compression d'images sur votre ordinateur, il faut vous assurer que votre système soit compatible par Java 1.7 ou une version ultérieure.Il est toutefois indépendant de toute IDE et il est possible de le lancer en ligne de commande dans un terminale

2. Execution

Il est possible d'exécuter notre programme de deux manières différentes.

D'une part, de manière non-intéractive. Cela se fait en l'invoquant avec en paramètre le nom d'un fichier PGM et un entier ρ. exemple :

```
java Compression flower.pgm 90
```

Le programme lance immédiatement les deux compressions et génère deux fichiers txt contenant les quadtree de l'images résultant des deux compressions respectives elle génère aussi les images compressées au format pgm.L'états des compression ,ainsi que les statistiques sont aussi affichées dans le terminal :

```
PROCESSING LAMBDA COMPRESSION:
LAMBDA COMPRESSION COMPLETED!
Le programme a mis 0.005 secondes à s'exécuter.
Custom Compression Result:
Initial Nodes: 221233
Final Nodes: 221233
PROCESSING RHO COMPRESSION:
RHO COMPRESSION COMPLETED!
Le programme a mis 0.021 secondes à s'exécuter.
Custom Compression Result:
Initial Nodes: 221233
Final Nodes: 199109
```

D'autre part, il est possible de lancer le programme en mode interactif. Il suffit de ne passer aucun argument en paramètre de la fonction main. Dans ce cas, le programme demande à l'utilisateur de choisir une image à compresser parmi une liste d'images possibles.

```
leviassawa@Levis-Mac-mini quadtree-pmg-compression % java Compression
1. train.pgm
2. flower_small.pgm
3. flower.pgm
4. lighthouse_big.pgm
5. lighthouse.pgm
6. tree_big.pgm
7. tree.pgm
8. test.pgm
Enter the number before the file you want to compress:
```

Après avoir choisi une Image à compressée , l'utilisateur doit ensuite choisir quelle compression appliquer.

```
Choose compression by entering the number before (1 or 2):
1. Lambda compression
2. RHO compression
```

Dans le cas où l'utilisateur choisit la compression rho, le programme demande d'entrer le facteur de compression.

```
PROCESSING RHO COMPRESSION:
Please Enter the factor ρ for the RHO compression:
```

Ensuite, si tout se déroule bien ,le programme génère un fichier txt ,contenant le quadtree correspondant à l'image compressée, et il génère une image au format pgm.

```
PROCESSING LAMBDA COMPRESSION:
COMPRESSION COMPLETED!
Le programme a mis 0.013 secondes à s'exécuter.
Custom Compression Result:
Initial Nodes: 221233
Final Nodes: 221233
```

Enfin, Le programme demande à l'utilisateur s'il souhaite compresser à nouveau la nouvelle image générée! Dans le cas où l'utilisateur répond oui , on réitère le même schéma que dans les parties supérieures sinon le programme s'arrête.
Voici un exemple complet du déroulement de notre programme en ligne de commande:

```
leviassawa@Levis-Mac-mini quadtree-pmg-compression % java Compression
1. train.pgm
2. flower_small.pgm
3. flower.pgm
4. lighthouse_big.pgm
5. lighthouse.pgm
6. tree_big.pgm
7. tree.pgm
8. test.pgm
Enter the number before the file you want to compress:
1
Choose compression by entering the number before (1 or 2):
1. Lambda compression
2. RHO compression
1
PROCESSING LAMBDA COMPRESSION:
COMPRESSION COMPLETED!
Le programme a mis 0.016 secondes à s'exécuter.
Custom Compression Result:
Initial Nodes: 268305
Final Nodes: 268305
do you want to compress this image one more time (1 or 2):
1. yes
2. no
1
Choose compression by entering the number before (1 or 2):
1. Lambda compression
2. RHO compression
2
PROCESSING RHO COMPRESSION:
Please Enter the factor ρ for the RHO compression:
50
COMPRESSION COMPLETED!
Le programme a mis 2.895 secondes à s'exécuter.
Custom Compression Result:
Initial Nodes: 268305
Final Nodes: 200201
do you want to compress this image one more time (1 or 2):
1. yes
2. no
2
leviassawa@Levis-Mac-mini quadtree-pmg-compression %
```

IV) Implémentation de l'effeuillage complet (lambda)

L'algorithme de compression Lambda utilise une approche simple qui consiste à parcourir l'arbre quadtree et à appliquer la compression Lambda sur les brindilles (twigs). Voici l'algorithme en pseudo-code :

PROCÉDURE effeuillerArbre(arbre)

```
   PARCOURIR_ARBRE(arbre, NOEUD)
      SI estBrindille(NOEUD) ALORS
         compresserBrindille(NOEUD)
      FIN SI
   FIN PARCOURIR_ARBRE
FIN PROCÉDURE

PROCÉDURE compresserBrindille(brindille)
   SI brindille.n'a pas de feuilles comme enfants ALORS
      RETOUR
   FIN SI

   sommeLogLuminosite = 0
   POUR chaque feuille DANS les enfants de la brindille FAIRE
      sommeLogLuminosite += ln(0.1 + luminosite(feuille))
   FIN POUR

   moyenneLogLuminosite = exp(sommeLogLuminosite / nombreEnfants(brindille))
   luminositeResultante = arrondir(moyenneLogLuminosite)

   // Remplacer la brindille par une feuille avec la luminosité résultante
   remplacerParFeuille(brindille, luminositeResultante)
FIN PROCÉDURE
```

Complexité:

Le parcours complet dans l'arbre pour trouver les brindilles est O(n) avec n le nombre de nœuds.
La compression Lambda a une complexité constante pour chaque brindille (O(1)).
Donc en total notre compression lambda complète aura une complexité de O(n).

V) Implémentation de la Compression Dynamique Rho

Dans la phase initiale de conception de la compression dynamique Rho, nous avons envisagé plusieurs approches pour parcourir et compresser les brindilles de l'arbre. L'une de ces approches initiales consistait à utiliser une liste pour stocker les brindilles ordonnées par leurs écarts maximaux (ε).

Approche Initiale avec une Liste

L'idée initiale était de maintenir une liste triée linéairement, où chaque élément contient une brindille et son ε. Cela aurait permis de parcourir la liste pour trouver rapidement la brindille avec le plus petit ε. Cependant, une telle approche aurait conduit à des complexités linéaires pour certaines opérations comme par exemple la recherche du minimum de la liste dans une liste ordonnée, ou l'insertion dans l'ordre d'une nouvelle brindille, ce qui aurait pu compromettre les performances dans certains cas.

Transition vers l'Arbre AVL

Nous avons alors décidé de passer à l'utilisation d'une structure de données plus avancée, l'Arbre Binaire de Recherche Équilibré (AVL).

Avantages de l'AVL par rapport à la Liste
La transition vers l'AVL a été motivée par plusieurs avantages :

- ➔ Complexité améliorée : L'AVL offre des opérations d'insertion et de recherche en O(log(n)), ce qui améliore significativement la complexité par rapport à une liste linéaire.
- ➔ Accès rapide : Un AVL équilibré permet un accès rapide à la brindille avec le plus petit ε, optimisant ainsi le processus de compression dynamique.
- ➔ Préservation de l'équilibre : L'AVL assure un équilibre constant, évitant ainsi les dégradations de performances liées à une structure déséquilibrée.

Problème présenté:

Au début on avait pensé à implémenter un AVL de brindille tout simple avec chaque noeud qui contient l'epsilon et la brindille associé (une référence au noeud du quadtree) mais comme on le sait déjà, un AVL ne peut pas contenir plusieurs noeud qui ont le même epsilon vu que c'est l'epsilon la clé de l'AVL, donc une petite modification a été requise pour pouvoir résoudre ce problème, c'est mettre une liste de référence vers des noeuds de Quadtree à la place de juste un seul noeud, et on a gardé l'attribut epsilon, donc la liste contiendrait toute les brindilles du quadtree qui possède ce même epsilon.

L'algorithme de compression dynamique Rho vise à comprimer activement les twigs d'un quadtree en utilisant une approche basée sur la luminosité. Lorsqu'une twig est identifiée comme compressible, elle est ajoutée à un arbre de twigs et sera compressée ultérieurement par une compression Lambda. Après chaque compression Lambda, l'algorithme vérifie si le parent de la twig compressée est également devenu compressible, calculant son epsilon et l'ajoutant à une structure de données AVL dédiée. Cette structure stocke les twigs compressibles en fonction de leur variation de luminosité (epsilon). Le processus se répète jusqu'à ce qu'un seuil global de compression (ρ) soit atteint.

Lors de la prochaine compression, l'algorithme sélectionne la twig compressible avec le plus petit epsilon dans l'AVL, favorisant celles qui ont la plus faible différence de luminosité. La compression de la twig est suivie de vérifications du parent et de la gestion des enfants en fonction de la similarité des valeurs des enfants du parent compressé. La complexité totale de l'algorithme est analysée, avec une attention particulière portée à l'implémentation d'une structure AVL pour stocker efficacement les twigs compressibles.

L'AVL de brindilles suit les principes fondamentaux des arbres binaires de recherche équilibrés, avec une implémentation utilisant les classes `TwigAVLTree` et `TwigAVLNode`. Les opérations d'insertion, de suppression et de recherche maintiennent l'équilibre de l'arbre, assurant une complexité logarithmique. Cette implémentation permet de stocker efficacement les twigs, assurant une recherche rapide lors des compressions futures.

Pseudo code de l'algorithme de compression RHO:

PROCÉDURE DetecterBrindillesCompressibles()
Créer un nouvel arbre AVL de brindilles appelé "twigs"
Appeler DetecterBrindillesCompressibles_ avec la racine de l'arbre QuadTree comme argument
FIN PROCÉDURE

```
PROCÉDURE DetecterBrindillesCompressibles_(noeud)
   SI le "noeud" n'est pas nul ET n'est pas une feuille ALORS
      SI le "noeud" est une racine de brindille ALORS
         Calculer epsilon en utilisant la méthode Util.calculateEpsilon(noeud)
         Insérer (epsilon, noeud) dans l'arbre AVL "twigs"
      SINON
         POUR chaque enfant DANS le "noeud" FAIRE
            Appeler récursivement DetecterBrindillesCompressibles_(enfant)
         FIN POUR
      FIN SI
   FIN SI
FIN PROCÉDURE



PROCÉDURE RhoCompressTree(ρ)
   DetecterBrindillesCompressibles()
   RhoCompressTree_(ρ)
   Réinitialiser l'arbre AVL "twigs" à nul
FIN PROCÉDURE

PROCÉDURE RhoCompressTree_(ρ)
   initial_nodes_number ← NombreDeNœudsDansLArbre()
   minTwig ← TrouverLaPlusPetiteBrindille(twigs.getRoot())
   ratio ← 1.0

   TANT QUE minTwig N'EST PAS NULL ET (ratio * 100 > ρ) FAIRE
      LambdaCompressTwig(minTwig.DernièreBrindille())

      nœudParent ← minTwig.DernièreBrindille().getParent()
      SupprimerLaBrindille(twigs, minTwig.getEpsilon())
      nbNœuds ← nbNœuds - 4

      TANT QUE nœudParent N'EST PAS NULL ET
TousLesEnfantsSontÉgaux(nœudParent) FAIRE
         nœudParent.setValeur(nœudParent.getValeurEnfant(0))
         DétruireEnfants(nœudParent)
         nbNœuds ← nbNœuds - 4
         nœudParent ← nœudParent.getParent()
      FIN TANT QUE

      SI nœudParent N'EST PAS NULL ET nœudParent est une racine de brindille
ALORS
         epsilon ← Util.calculateEpsilon(nœudParent)
         InsérerDansBrindilles(epsilon, nœudParent)
      FIN SI

      minTwig ← TrouverLaPlusPetiteBrindille(twigs.getRoot())
      ratio ← (double)nbNœuds / (double)initial_nodes_number
   FIN TANT QUE
FIN PROCÉDURE
```

Processus de Compression Dynamique Rho :

Détections des Brindilles Compressibles :

Un parcours complet de l'arbre est fait en détectant les brindilles et en les ajoutant dans l'arbre de brindilles.

Compression :

Une compression lambda va être effectué sur chaque brindille détectée en partant de celle qui a le plus petit epsilon, et vu que l'AVL de brindilles est un arbre de liste de brindilles donc il pourrait y avoir plusieurs brindilles dans la liste (n'oublions pas que ces brindilles ont le même epsilon) donc on prend la dernière brindille, on la compresse avec la compression lambda et on supprime la brindille de la liste. Après avoir compresser la brindille, on vérifie si son parent est devenu lui aussi une brindille, si c'est le cas, on vérifie si ses fils ont les mêmes luminosités, si c'est le cas on réduit directement à un seul noeud pour respecter la définition de l'AVL, sinon on calcule l'epsilon de la brindille et on la rajoute dans l'AVL en tant que nouvelle brindille détectée et prête à être compresser.

Répétition jusqu'à la condition d'arrêt:

Le processus se répète jusqu'à ce qu'un critère de compression global, défini par un seuil (ρ), soit atteint. De nouvelles brindilles compressibles peuvent être ajoutées à l'AVL durant le processus.

Complexité de l'Algorithme :

- **Compression Lambda :**
  - C'est un simple calcul mathématique de complexité temporelle constante donc O(1).
- **Détection des Brindilles Compressibles :**
  - Parcours linéaire de tous les nœuds dans l'arbre (O(n)).
  - Insertion dans l'AVL avec une complexité de O(log(n)).
  - Complexité totale : O(nlog(n)).
- **Compression Rho :**
  - Recherche dans l'AVL pour la brindille avec le plus petit epsilon (O(log(n))).
  - Compression de la brindille (O(1)).
  - Suppression de la brindille de l'AVL (O(1)).
  - Complexité totale : O(log(n)).

La complexité totale de l'algorithme Rho, combinant la détection des brindilles compressibles et la compression Rho, est donc de O(nlog(n)) + O(log(n)) donc O(nlog(n)). La complexité peut être optimisée dans des scénarios favorables, mais la pire complexité est conservée pour l'analyse.

VI) Implémentation en JAVA

Dans cette section nous verrons plus en détails les différentes Classes de notre programme, les méthodes implémentées et nous discuterons de nos choix d'implémentations.

- ***Class Compression :***
  Cette Classe contient la fonction Main du Programme. Elle lance soit le Menu principal, soit le Menu par défaut en fonction de si des arguments sont passés à l'appel du programme.

❖ ***Class DefaultMenu***
Cette classe permet d'exécuter le programme de manières non-interactif.Elle est appelé par le programme principale Lorsque des arguments sont passés en entrés de de la fonction main.Elle se charge d'appliquer les deux compressions, de stocker les quadtrees correspondant au images compressés dans des fichiers txt , et de générer les nouvelles images compressés au format pgm. Ainsi que d'afficher les statistiques des deux compressions.

❖ ***Class MainMenu***
Cette classe est appelée par la fonction principale lorsque l'utilisateur lance le programme en mode intéractif. Elle gère l'interaction entre l'utilisateur et le programme. Elle affiche une liste d'images et demande à l'utilisateur de choisir parmi elle une image à compresser. Ensuite elle lance le Menu de compression.

❖ ***Class CompressionMenu***
Cette classe est appelée par le Menu principal .Elle représente un menu interactif permettant à l'utilisateur de choisir entre deux méthodes de compression (Lambda ou RHO) pour un objet QuadTree représentant une image. Elle permet aussi la saisie du facteur de compression dans le cas ou la compression rho est choisie par l'utilisateur. La classe utilise un gestionnaire de fichiers (FileManager) pour sauvegarder le QuadTree dans un fichier texte après la compression. Elle génère une nouvelle image compressée au format pgm. Elle prend aussi en charge l'affichage des statistiques de la compression dans le terminal.

❖ ***Class FileManager***
Cette classe contient toutes les méthodes statiques que nous utilisons pour manipuler la lecture et l'écriture des fichiers.

Intéressons nous aux méthodes de cette classe qui sont très utiles dans la compression de notre algorithme.

- **Méthode isGoodFormat**
  La méthode isGoodFormat vérifie si le format d'image est correct en fonction du numéro magique, de la largeur et de la hauteur. Elle retourne true si le format est correct et false sinon. Elle imprime également des messages d'erreur explicites en cas de problème.

- **Méthode loadImage**

Cette méthode lit le contenu d'un fichier image et retourne un objet Pgm qui a pour attribut un tableau 2D d'entiers ,représentant l'image et la luminosité maximale de l'image. Elle vérifie également si le format est correct en utilisant la méthode isGoodFormat.

Complexité: la méthode Load Image a une complexité linéaire car si nous avons n valeurs de luminosité nous accédons à notre grille n fois pour les insérer.

- **Méthode SaveImage**
  La méthode SaveImage prend un objet QuadTree et un nom de fichier en paramètre et enregistre l'image représentée par l'arbre quadtree dans le fichier spécifié. Elle utilise un objet FileWriter pour écrire les données dans le fichier. Afin d'écrire les différentes valeurs de luminosités dans le fichier, la méthode saveImage() fait appelle a une autre méthode createGrilleTemp() qui a pour but , grâce au quadtree passer en paramètre de générer un Tableau 2D d'entier contenant les valeurs de luminosité de l'image. Ensuite , à partir du tableau 2D, la méthode saveImage() dans un temps linéaire écrit les différentes valeur dans le fichiers. Enfin , il faut préciser que c'est cette méthode qui est ensuite appelé par la méthode toPgm() dans la classe Quadtree.

- **Méthode CreateGrilleTemp**
  Elle a pour but de générer un tableau 2D d'entier à partir du quadtree passer en paramètres.Il faut noter que le quadtree correspond a une image compresser, donc une valeur luminosité contenu dans une feuille peut correspondre à une zone plus ou moins grande de l'image.Ainsi il faut pouvoir Insérer cette valeur aux bonnes indices dans notre grilles.Pour ce faire, La méthode createGrilleTemp() parcourt récursivement le quadtree et le tableau 2D.pour parcourir le tableau elle utilise les coordonnée de la case de début(haut gauche) et la case de fin du tableau(bas droit).Ainsi à chaque fois que la fonction accède à l'un des fils d'un noeud, elle joue sur les coordonées en paramètres afin de travailler sur la zone du tableau correspond au noeuds fils.

  Complexité: Si on note n comme la taille de la région (nombre total d'éléments dans la grille), la complexité totale dépendra du nombre de niveaux de profondeur dans l'arbre QuadTree. Si l'arbre a une profondeur maximale de d, la complexité totale serait généralement de l'ordre de O(nd), où n est la taille de la région et d est la profondeur maximale de l'arbre.

Remarques Générales

- Le code utilise la classe Scanner pour lire et interagir avec l'utilisateur, ce qui est une bonne pratique pour l'entrée utilisateur.
- Les messages d'erreur sont informatifs et aident à déboguer les éventuels problèmes.
- Les méthodes sont bien décomposées et respectent le principe de responsabilité unique.
- La gestion des erreurs d'entrée utilisateur est bien traitée.

❖ *Classe Util*

La classe Util est une classe utilitaire qui fournit des méthodes pour les opérations liées à QuadTree ainsi que des calculs numériques spécifiques. On y trouve des méthodes tel que :

calculateAvgLogLuminosity:

Cette méthode calcule et retourne la luminosité logarithmique moyenne pour un nœud QuadTree.

calculateEpsilon:

Description : Cette méthode calcule et retourne la valeur epsilon maximale pour un nœud QuadTree.

isPowerOfTwo:

Description : Vérifie si un nombre donné est une puissance de deux.
Fonctionnement : Utilise une opération de bits pour déterminer si le nombre est une puissance de deux.
Utilisation : Principalement utilisé pour des opérations liées à la taille des images QuadTree.

getMax:

Description : Trouve et retourne la valeur maximale d'un tableau d'entiers.
Fonctionnement : Parcourt le tableau et maintient la valeur maximale rencontrée.
Utilisation : Utilisé pour obtenir la valeur maximale à partir d'un tableau d'entiers, probablement utilisé dans le contexte du traitement de l'image.

Constante Déclarée :

UNDEFINED_LOG_LUMINOSITY : Constante représentant une valeur de luminosité logarithmique indéfinie.

❖ ***Classe QuadTreeNode***

- **Constructeur**

Les constructeurs de la classe QuadTreeNode permettent la création de nœuds avec ou sans valeur initiale, et ils initialisent correctement les attributs.

- **Méthode areChildrenEqual**

La méthode areChildrenEqual vérifie si les enfants d'un nœud sont égaux, ce qui est utilisé dans le processus de compression.

- **Méthodes createChildren et destroyChildren**

Les méthodes createChildren et destroyChildren gèrent la création et la destruction des enfants d'un nœud, respectivement.

- **Méthodes isLeaf et isTwigRoot**

Les méthodes isLeaf et isTwigRoot vérifient si un nœud est une feuille ou la racine d'une brindille, ce qui est utile dans le processus de compression.

D'autres méthodes utilitaires, telles que **printChildren**, sont présentes pour faciliter le débogage.

❖ ***Classe QuadTree***

Avant toute chose, il est convenable de mettre en évidence la logique que nous utilisons pour construire notre Quadtree.
La construction du quadtree se fait en divisant récursivement l'image en quadrants jusqu'à atteindre une taille spécifique (par exemple, une image carrée de taille 2^n * 2^n).
Maintenant intéressons nous aux méthodes de cette classe.

Constructeur QuadTree :

Cette méthode prend le chemin local d'une image comme argument. Elle utilise la classe FileManager pour charger l'image depuis le fichier spécifié, puis crée un nouveau QuadTree avec une racine initiale. La méthode appelle ensuite constructQuadtree pour construire le QuadTree à partir de l'ensemble d'images 2D.

Méthode constructQuadtree :

Cette méthode est responsable de la construction récursive du QuadTree. Elle prend un nœud QuadTree, une représentation 2D de l'image, et les indices de début et de fin de la portion d'image à traiter. La construction se fait de manière récursive en subdivisant l'image en quarts jusqu'à ce que des feuilles de twig soient identifiées. Ensuite, elle vérifie si les valeurs des feuilles sont identiques, auquel cas elle définit la valeur du nœud actuel à cette valeur commune. Si les valeurs des feuilles sont différentes, elle crée les enfants correspondants avec les valeurs appropriées. La méthode gère également le cas où les enfants du nœud actuel ont des valeurs égales, optimisant ainsi la structure du QuadTree en consolidant ces nœuds égaux en un seul nœud.

- **Méthode toString**

La méthode toString génère une représentation parenthésée de l'arbre quadtree, ce qui est utile pour le débogage et la compréhension visuelle de la structure de l'arbre.

VII) Conclusion:

En conclusion de notre projet de compression d'images en format PGM, nous avons relevé le défi avec succès en créant des algorithmes répondant aux exigences spécifiques du cahier des charges. Notre objectif principal était de réduire la taille des fichiers tout en acceptant une légère perte de qualité visuelle.

Ce projet a été une opportunité précieuse pour appliquer nos connaissances théoriques en algorithmique et structures de données dans un contexte pratique. Il a également renforcé notre capacité à prendre des décisions éclairées pour résoudre des problèmes concrets.

En fin de compte, notre algorithme a atteint les objectifs fixés, démontrant notre capacité à concevoir des solutions efficaces. Nous sommes fiers de cette réalisation.