RAPPORT DU PROJET DE ASD3

MBAYE Serigne Touba & DERROUET Iwan

Décembre 2023

Nantes Université

Table des matières

1 Introduction

Le présent rapport documente le projet que nous avons réalisé dans le cadre du cours **Algorithme et Structure de Données 3**.Ce projet s'articule autour de l'implémentation d'algorithmes de compression d'images basés sur la structure Quadtree. L'objectif principal de ce projet est de développer des algorithmes de compression capables de réduire la taille des fichiers image tout en préservant une qualité visuelle acceptable.
Pour ce faire, nous avons exploré deux approches de compression distinctes : la compression Lambda, axée sur la simplification de la structure de Quadtree, et la compression Rho, qui prend en compte les écarts entre les valeurs des nœuds. Ce rapport détaillera le fonctionnement de nos algorithmes, les choix d'implémentation, les résultats obtenus à travers des exemples concrets. Nous aborderons également les commandes de compilation et d'exécution de notre programme, offrant ainsi une vision complète du projet.

2 Commandes de compilation et d'éxecution

2.1 Compilation

La commande suivante peut être utilisée afin de compiler notre projet :

```
javac *.java
```

2.2 Exécution

Après compilation, notre projet peut s'éxécuter de deux manières différentes :

— `java Main`

exécutera le mode intéractif, l'utilisateur aura alors plusieurs choix afin de poursuivre.
(Petite précision : les images reconnues par le menu, sont les images placées dans le dossier pgm_carres).

— `java Main path/to/image.pgm`

exécutera toutes les étapes de la compression lambda et terminera sont exécution en affichant les statitisques de compressions.

— `java Main path/to/image.pgm rho`

(avec rho un entier) exécutera toutes les étapes de la compression lambda ainsi que de la compression Rho tout en terminant leurs exécution en affichant les statistiques de compressions.

3 une vue globale de notre programmme

3.1 Le constructeur

Le constructeur prends en paramètre un lien vers un fichier pgm. La première étape de l'algorithme est alors de traiter l'entete du fichier (le format, les dimensions, lum max...). Ensuite l'idée de l'algroritme est d'abords de convertir les données des pixel en matrice puis on utilise cette matrice dans une nouvelle fonction recursive qui construira tout l'arbre à partir de cette matrice :

Algorithme 1 : genQuadTree

```
Entrées : Quadtree t, Matrice lumMatrice, Entier x, Entier y, Quadtree parent,
          IntPointeur nbNoeuuds, IntPointeur nbFeuilles
DEBUT
t.parent ← parent
si t.w == 1 et t.h == 1 alors
    // un seul pixel, c'est donc une feuille
    t.lum ← t.lumiMatrice[y][x]
    t.estFeuille ← vrai
    // comptage des noeuds
    nbNoeuds.ajouter(1)   nbFeuilles.ajouter(1)
fin
// ce n'est pas une feuille
// on cree les 4 fils
t.fils ← tableau de 4 Quadtree
pour i ← 1 à 4 faire
    fils[i] = Quadtree(w/2, h/2) // Quadtree vide de dimension w/2 x h/2
fin
// on coupe a chaque fois
genQuadTree(t.fils[1], lumiMatrice, x, y, t, nbNoeuds, nbFeuilles)
genQuadTree(t.fils[2], lumiMatrice, x + w/2, y, t, nbNoeuds, nbFeuilles)
genQuadTree(t.fils[3], lumiMatrice, x + w/2, y + h/2, t, nbNoeuds, nbFeuilles)
genQuadTree(t.fils[4], lumiMatrice, x, y+h/2, t, nbNoeuds, nbFeuilles)
// ensuite on rassemble si les 4 fils s'il contienne la même valeur
si t.estBrindille() et t.filsTousEgaux() alors
    t.estFeuille ← vrai
    lum = fils[0].lum
    pour i ← 1 à 4 faire
        fils[i] ← null
    fin
    // comptage des noeuds
    nbNoeuds.ajouter(-4)
    nbFeuilles.ajouter(-3)
fin
nbNoeuds.ajouter(1) // comptage des noeuds
FIN
```

Cette fonction utilise certaines méthodes de la classe Quadtree que nous avons implémenté :

— Fonction estFeuille
Entrées : Quadtree t
Sortie : Booléen
Sémantique : Renvoi vrai si t est une feuille, faux si t ne l'est pas.

— Fonction estBrindille
Entrée : t Quadtree
Sortie : Booléen
Sémantique :
Renvoi vrai si t est une brindille, faux si t ne l'est pas Une brindille est un Quadtree dont tous ces fils sont des feuilles.

— Fonction filesTousEgaux
Entrée : Quadtree t
Sortie : Booléen Prérequis : t est une brindille
Sémantique : Renvois vrais si toutes les valeurs lum de ces feuilles sont égales, faux sinon

Etude de la complexité : (Rappels : n = nbNoeud)
Les 4 fonctions citées précédemment sont en temps constant : $\mathcal{O}(1)$

3.2 Export en PGM

La procédure qui convertit un Quadtree en image possède la même structure que le constructeur, on ecrit d'abords l'entête puis (après avoir reconstruit la matrice des lum) on écrit la matrice dans le fichier pgm.

3.3 Autres structures implémentées :

Par choix d'implémentation, nous avons dû implémenté deux autres structures :

— IntPointeur
La première est juste un pointeur sur un entier afin de pouvoir passer des entiers par référence dans les fonctions.

— Paire
La deuxième structure est une Paire d'objet de type quelconque qui sera utilisé dans la fonction de compression rho

4 Détails du fonctionnement des méthodes décrites en 3.2.1 & 3.2.2 et leur complexités

4.1 L'algorithme de compression Lambda

La fonction compressLambda ne prends pas de paramètre, cependant, dans l'algorithme que nous avons implémenté, nous voulons aussi enregistrer le nombre de nœuds final après compression, ce nombre est stocké dans un pointeur appelé nbNC (Nombre de Nœuds Courrant) :

Algorithme 2 : compressLambda

Entrées : Quadtree t
DEBUT
IntPointeur nbNC
nbNC ← IntPointeur(t.nbNoeudCourrant)
t.compressLambdaRec(nbNC)
t.nbNoeudCourrant ← nbNC.getValeur()
FIN

La fonction qui réalise donc la fonction de compression lambda est donc la fonction compressLambdaRec qui prends en paramètre le compteur de nœuds :

Algorithme 3 : compressLambdaRec

```
Entrées : Quadtree t, IntPointeur nbNC
DEBUT
// si le nœud considéré est une feuille, pas de compression à faire
si !t.estFeuille alors
    si t.estBrindille() alors
        // Compression de la brindille actuelle
        t.lum ← Arrondir(t.calculMoyenneBrindille())
        t.estFeuille ← true
        pour i ← 1 à 4 faire
            t.fils[i] ← null
        fin
        nbNC.ajouter(-4) // mise à jour du nombre de noeud
    sinon
        // le nœuds n'est pas une brindille, il faut compresser ces fils
        pour i ← 1 à 4 faire
            si t.fils[i] != null et !t.fils[i].estFeuille alors
                // Compression récursive des sous-quadtree
                t.fils[i].compressLambdaRec(nbNC)
            fin
        fin
        // ensuite on préserve le Quadtree
        si t.estBrindille() et t.filsTousEgaux() alors
            t.lum ← t.fils[0].lum
            t.estFeuille ← true
            pour i ← 1 à 4 faire
                t.fils[i] ← null
            fin
            nbNC.ajouter(-4) // mise à jour du nombre de nœud
        fin
    fin
fin
FIN
```

Cette fonction utilise une seule fonction non définie plus tôt :

Fonction calculMoyenneBrindille
Entrée : Quadtree t
Sortie : Réelle
Prérequis : t est une brindille
Sémantique : Renvoi la moyenne logarithme des valeurs lum de ces feuilles

Etude de la complexité : (Rappels : n = nbNoeud)
La fonction calculMoyenneBrindille est en $\mathcal{O}(1)$
D'où, la procédure compressLambda passe une fois dans tous les nœuds du quadtree, cet algo est donc en $\mathcal{O}(n)$

4.2 L'algorithme de compression Rho

Afin de gagner en complexité, tout d'abords on calcule toutes les valeurs des epsilons (y compris ceux des nœuds interne que l'on peut connaitre en faisant remonter les moyenne), ensuite on stocke toutes les paires de epsilon avec leur brindille respectifs dans un tableau, on trie le tableau par ordre croissant des epsilon, et pour fini on parcourt le tableau et on compresse toutes les brindille (Il faut aussi faire attention si le père de la brindille doit aussi être compressé ou non, d'où l'importance du pointeur vers le père)

Algorithme 4 : compressRho

```
Entrées : Quadtree t, Entier rho
DEBUT
Entier nbBS, indiceTab
Tableau de paires (epsilon, Quadtree) tabEpsilon
IntPointeur nbNC

nbBS ← 0
indiceTab ← 0
nbNC ← IntPointeur(nbNoeudInitial)
tabEpsilon ← Paire[this.nbNoeudInitial - this.nbFeuilles]

compressRhoInitEpsilonTab(tabEpsilon, IntPointeur(), IntPointeur())
tabEpsilon ← t.tri(tabEpsilon)

tant que (nbNC.getValeur() / t.nbNoeudInitial) * 100.0 > rho
et indiceTab < t.nbNoeudInitial − t.nbFeuilles faire
    // Si le noeud courrant est un brindille, on le compresse
    si tabEpsilon[indiceTab].second.estBrindille() alors
        tabEpsilon[indiceTab].second.lum ←
          Arrondir(tabEpsilon[indiceTab].second.calculMoyenneBrindille())

        tabEpsilon[indiceTab].second.estFeuille ← true
        pour i ← 0 à 4 faire
            tabEpsilon[indiceTab].second.fils[k] ← null
        fin

        nbNC.ajouter(-4) // Comptage des noeuds

        // Appel de la compression du père
        tabEpsilon[indiceTab].second.parent.compressRhoParent(
          tabEpsilon[indiceTab].premier, t.nbNoeudInitial, nbNC, rho)
    fin
    indiceTab += 1 // On passe au noeud suivant
fin
t.nbNoeudCourrant ← nbNC.getValeur() FIN
```

Faisant partie du fonctionnement de la compression Rho, il est nécessaire de détailler la procédure compressRhoParent.
Cette procédure va tenter de compresser le père d'une brindille qui vient d'être compressée. Tout d'abords on vérifie que c'est bien une brindille, Si c'est le cas, si tous ces fils sont égaux on le compresse automatiquement (afin de préserver le quadtree), Si ce n'est pas le cas on, on regarde si on n'a pas dépassé le taux de compression rho, si on ne la pas dépassé, on vérifie bien que la brindille a déjà été rencontrée (sa valeurs d'epsilon est inférieure aux dernier epsilon rencontrée dans le tableau), si c'est le cas on le compresse et on fait de même pour son père (si ce n'est pas la racine).

Algorithme 5 : compressRhoParent

Entrées : Quadtree t, réelle epsilonCourrant, Entier nbNI, Entier nbNC, Entier rho
Prérequis : t est le père d'une brindille compressée
DEBUT
si $t.estBrindille()$ *et* $(t.filsTousEgaux()$ *ou*
$(t.calculerEcartMaxEpsilon() \leq epsilonCourrant$ *et*
$(nbNC.getValeur()\ /\ nbNI) * 100.0 > rho))$ **alors**

```
    // Compression de la brindille
    t.lum ← Arrondir(t.calculMoyenneBrindille())
    t.estFeuille ← true
    pour i ← 1 à 4 faire
        t.fils[k] ← null
    fin

    nbNC.ajouter(-4) // comptage des nœuds

    // Appel de la compression du père
    si t.parent != null alors
        t.parent.compressRhoParent(epsilonCourrant, nbNI , nbNC, rho)
    fin
fin
FIN
```

Ensuite de même que pour la compression lambda voici le détail des autres fonctions :

— Fonction calculerEcartMaxEpsilon(t : Quadtree) : réelle
Prérequis : t est une brindille
Sémantique : renvoi l'écart maximum entre la moyenne et les valeurs lum de ces fils

— Fonction compressRhoInitEpsilonTab
Entrées : un Quadtree, un tableau de Paire (epsilon, Quadtree), 2 pointeurs sur un entiers
Sémantique :
Construit un tableau contenant toutes les paires de nœuds interne avec leur Valeurs d'epsilon

— Procédure tri
Entrée : un tableau de Paire (epsilon, Quadtree)
Sémantique :
Trie le tableau par ordre croissant des epsilons. Afin de gagner en complexité, nous avons choisi d'implémenter le trie fusion

Etude de la complexité : (Rappels : n = nbNoeud)
La fonction calculerEcartMaxEpsilon ce fait en temps constant : $\mathcal{O}(1)$

La fonction compressRhoInitEpsilonTab parcours une fois tous les nœuds du Quadtree et donc cet algo est en $\mathcal{O}(n)$

La fonction tri, réalise un tri fusion sur les nœuds du Quadtree, on connait alors la complexité $\mathcal{O}(n \ln(n))$

Ainsi, dans la procédure de compression Rho, il nous reste à calculer la complexité de la boucle tant que. Cette boucle consiste à parcourir tous les nœuds de l'arbre, ainsi que leur père. Chaque nœud est alors parcouru au plus 4 fois (car 4 fils mènerons au même père) et donc ce parcours est en $\mathcal{O}$(n)

On en conclut alors que la complexité de la fonction compressRho est en $\mathcal{O}(n \ln(n))$

5 Quelques exemples de résultats obtenus avec notre projet

5.1 Chemin menant à l'aboutissement de nos résultats

```
mbaye@touba-mb14:~/Documents/ASD3/projet-asd-3$ javac Main.java
mbaye@touba-mb14:~/Documents/ASD3/projet-asd-3$ java Main
Choisissez l'image a partir de la banque detecte :
0) tree_big.pgm    1) flower_small.pgm    2) boat.pgm    3) lighthouse.pgm    4) flower.pgm    5) lighthouse_big.pgm

tapez un nombre entre 0 et 9 :
7
Chargement de l'image...

Que voulez vous faire ?
0) compression lambda    1) compression rho    2) revenir en arriere    3) quitter

tapez un nombre entre 0 et 3 :
1

Entrez un taux de compression (rho) :
20
Compression...

Que voulez vous faire ?
0) Exporter en pgm    1) Enregistrer le Quadtree   2) Afficher les statistiques de compression    3) Revenir en arriere

Tapez un nombre entre 0 et 4 :
2

Statistiques de compression :
------------------------------
Nombre de noeud avant compression : 268305
Nombre de noeud apres compression : 53661
Taux de compression : 20.0%

Que voulez vous faire ?
0) Exporter en pgm    1) Enregistrer le Quadtree   2) Afficher les statistiques de compression    3) Revenir en arriere

Tapez un nombre entre 0 et 4 :
0
Saisissez un nom de fichier :
TrainV2.pgm
Enregistrement de l'image...
```

Ceci est un exemple qui montre les étapes de compilation et d'exécution

5.2 Résultat obtenu après application des deux compression

L'image du train avant la compression

Après la compression Lamba

Après la compression Rho de 26%

6 Conclusion

A la fin de notre projet, nous avons observé des résultats significatifs en termes de réduction de la taille des fichiers image tout en préservant une qualité visuelle acceptable.La compression Lambda a démontré son efficacité en éliminant les nœuds inutiles dans l'arbre de Quadtree, tandis que la compression Rho a permis d'atteindre des taux de compression plus élevés en prenant en compte les écarts entre les valeurs des nœuds.