GODEFROY Théotime
BOTANS Enzo

2023/2024
584J

Compte Rendu Projet

Compression d’image à l’aide de Quadtrees

Algorithmique et Structure de données 3
X31I020

Encadré par :
Irena RUSU-ROBINI

1. Commandes de compilation et exécution sur le terminal

Avant de pouvoir compiler le projet, il faut se situer dans le fichier source « src » comme indiqué ci-dessous.

```
E218557M@S049pc07:~/Tp/L3/Semestre 1/ASD3/projet/projetasd3$ cd src
```

Ensuite, pour compiler le programme il faut écrire la ligne suivante :
javac Quadtree.java

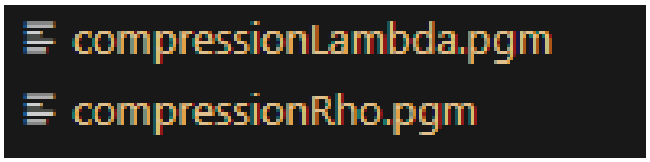
```
E218557M@S049pc07:~/Tp/L3/Semestre 1/ASD3/projet/projetasd3/src$ javac Quadtree.java
```

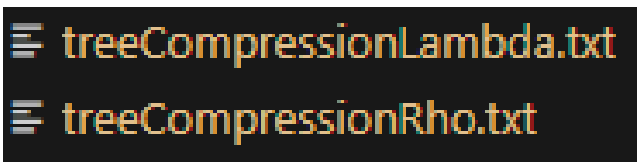
Enfin, on peut lancer le programme avec la commande suivante :
java Quadtree fichier.pgm p

Dans ce tyope d'execution, on peut observer directement les attentes décrites dans le sujet. On retrouve donc :

- Les statistiques des compression Lambda et Rho affichées dans le terminal
- Les fichiers treeCompressionLambda.txt et treeCompressionRho.txt
  Ils contiennent l'écriture de nos Quadtree en forme parenthésée
- Les fichiers CompressionLambda.pgm et CompressionRho.pgm
  Ils représentent l'image après les deux compressions respectives

```
===Compression Lambda===

Hauteur : 8
Luminosite Max : 254
Nombre de Noeuds : 84864
Taux de compression : 25%
========================

====Compression Rho=====

Hauteur : 9
Luminosite Max: 253
Nombre de Noeuds : 186773
Taux de compression : 59.0%
========================
```






```
E218557M@S049pc07:~/Tp/L3/Semestre 1/ASD3/projet/projetasd3/src$ java Quadtree test.pgm 82
```

Il est possible de le lancer sans les paramètres en ligne de commande. Dans ce cas-là, vous sez amener à utiliser un menu interactif particulier.

Tout d'abord, vous pourrez choisir une image à compresser parmi celles se trouvant dans le répertoire courant du programme. Vous n'avez qu'à indiquer dans le terminal son numéro dans la liste.

```
Choisissez parmis la liste d'images PGM dans le repertoire courant celle que vous souhaitez compresser
1. boat.pgm
2. camera.pgm
3. compressionLambda.pgm
4. compressionRho.pgm
5. flower.pgm
6. flower_small.pgm
7. JDD1.pgm
8. JDD2.pgm
9. lighthouse.pgm
10. lighthouse_big.pgm
11. test.pgm
12. test2.pgm
13. train.pgm
14. tree.pgm
15. tree_big.pgm
Entrez le numero de l'image :
```

Ensuite, les statistiques de l'arbre initial s'affichent, donnant donc un aperçu du Quadtree utilisé.

Puis, vous avez la possibilité de choisir entre deux compressions, un changement d'image, ou simplement quitter le programme.
Changer l'image réinitialise le Quadtree et vous permet de repartir de zéro.

```
=====Arbre initial======

Hauteur : 9
Luminosite Max : 255
Nombre de Noeuds : 245224
======================


Que souhaitez-vous faire ?
1. Compression Lambda
2. Compression Rho
3. Changer d'image et recommencer
4. Quitter le programme
```

```
Quel pourcentage de compression utiliser ?
68

====Compression Rho=====

Hauteur : 9
Luminosite Max: 255
Nombre de Noeuds : 168586
Taux de compression : 68.0%
======================
```

La compression Rho vous demandera le taux de compression voulu à effectuer.
Enfin, dans les cas de compression, on verra s'afficher les nouvelles statistiques du Quadtree après la compression en question.

Après chacune d’entre elles, vous retournez sur le même menu, vous permettant notamment d’enchaîner les compressions de différents types.

2. Vue Globale de notre programme

Méthode compression Lambda :

---

fonction compressLambda(Quadtree Q)

```
Début
|       compressLambdaRecu(q)
|       verifEqui(q)
Fin
```

---

fonction compressLambdaRecu(Quadtree Q)

```
Début
|       Si  alors
|       |       Si Q n’a que des feuilles alors
|       |       |       Q.value = round(exp(0.25 * (Math.log(0.1 + Q.Q1().value +
Math.log(0.1+Q.Q2().value) + Math.log(0.1 + Q.Q3().value) + Math.log(0.1 + Q.Q4().value))))
|       |       |       Q.Q1 = null
|       |       |       Q.Q2 = null
|       |       |       Q.Q3 = null
|       |       |       Q.Q4 = null
|       |       Fin Si
|       Sinon
|       |       Si( Q.Q1.value!=-1 ) alors
|       |       |       Q1.compressLambda()
|       |       Fin Si
|       |       Si( Q.Q2.value!=-1 ) alors
|       |       |       Q2.compressLambda()
|       |       Fin Si
|       |       Si( Q.Q3.value!=-1 ) alors
|       |       |       Q3.compressLambda()
|       |       Fin Si
|       |       Si( Q.Q4.value!=-1 ) alors
|       |       |       Q4.compressLambda()
|       |       Fin Si
|       Fin Si
Fin
```

La méthode de compressionLambda est utilisée dans le contexte de la compression d'images au format PGM pour réduire la taille des fichiers en exploitant les propriétés des quadtree.

Dans cette méthode, on représente l'image sous forme de quadtree, une structure de données arborescente où chaque nœud représente un bloc de pixels. La compression Lambda se base sur la simplification de ces blocs en fusionnant les nœuds voisins qui ont des valeurs similaires.

Ici, les valeurs des quatre blocs adjacents peuvent être regroupés en un seul bloc avec une valeur moyenne alculée à partir de ces blocs. Il faut obligatoirement que les quatres blocs soient des feuilles. Cette formule de calcul est la suivante :

$$\Lambda = \exp\left(\frac{1}{4}\sum_{i=1}^{4}\ln(0.1+\lambda_i)\right)$$

Ce processus de fusion réduit le nombre de nœuds dans l'arbre, ce qui permet de stocker l'image de manière plus compacte. En effet, au lieu de stocker chaque pixel individuellement, la méthode Lambda regroupe les pixels similaires pour réduire l'espace mémoire nécessaire tout en préservant une qualité d'image acceptable.

Complexité temporelle de la compression Lambda :

Pour évaluer la complexité temporelle de ces fonctions, examinons les parties significatives du code :

La première partie de la fonction compressLambdaRecu effectue une vérification de conditions et des opérations constantes telles que des affectations et des comparaisons. Ces parties ne dépendent pas de la taille de l'entrée, donc elles sont de complexité constante O(1).

La récursion se produit lors des appels récursifs Q1.compressLambda(), Q2.compressLambda(), Q3.compressLambda(), et Q4.compressLambda(). Chaque appel récursif explore un sous-ensemble du quadtree. Dans le pire des cas, chaque nœud de l'arbre sera visité une fois, ce qui signifie qu'il y aura un total de n appels pour un quadtree de taille n. Par conséquent, la complexité dépend du nombre total de nœuds dans le quadtree.

Dans le cas où le quadtree est équilibré et que chaque niveau est divisé en quatre parties égales, la hauteur de l'arbre sera log4(n), où n est le nombre total de nœuds. Ainsi, dans le pire des cas, la complexité temporelle de cet algorithme est O(n), où n est le nombre total de nœuds dans le quadtree.

Cependant, la complexité réelle peut varier en fonction de la structure du quadtree et de la manière dont les valeurs sont distribuées. Si le quadtree est mal équilibré ou s'il n'y a pas beaucoup de nœuds "feuilles" (nœuds sans enfants), la complexité peut être inférieure à O(n). Mais dans tous les cas, elle restera au moins linéaire en fonction du nombre total de nœuds dans le quadtree.

Méthode Compression Rho :

---

fonction compressRho(Quadtree Q, double p)

```
Variables :
        nbN : Entier
        noeudsACons : Entier
        noeudsASuppr : Entier
        noeudsCompress : Tableau de Quadtree
        epsilonCompress : Tableau de réels
        cheminCompress : Tableau de chaîne de caractères
Début
|       Si (p<0 ou p>100) alors
|       |       Afficher « Erreur, Valeur > 100 ou < 0 »
|       Sinon
|       |       nbN ← Calcul du nombre de nœuds du Quadtree
|       |       noeudsACons ← Calcul  du nombre de nœuds à conserver
|       |       noeudsASuppr ← nbN – noeudsACons
|       |       noeudsCompress ← Stockage des nœuds compressibles
|       |       epsilonCompress ← Stockage des epsilon des nœuds compressibles
|       |       cheminCompress ← Stockage des chemins des nœuds compressibles
|       |       Si  noeudsACons > 5 alors
|       |       |       Tant que nbASuprr > 0 faire
|       |       |       |       compressRhoIte(Q, noeudsASuppr, noeudsCompress,
epsilonCompress, cheminCompress)
|       |       |       Fin Tant que
|       |       Sinon
|       |       |       Tant que nbASuprr > 4 faire
|       |       |       |       compressRhoIte(Q)
|       |       |       Fin Tant que
|       |       |       compressLambdaRecu(Q)
|       |       Fin Si
|       Fin Si
Fin
```

---

fonction compressRhoIte (Quadtree Q, Entier nASuppr, Tableau de Quadtree nCompress, Tableau de réels epsCompress, Tableau de chaîne de caractères chCompress)

```
Variables :
        cheminReduit : chaîne de caractères
        Qparent : Quadtree


Début
        cheminReduit ← chCompress[0] (Ici, on stockera la chaîne sans son premier
caractère)
        Qparent ← getParent(Q, cheminReduit)
        nASuppr ← nASuppr – 4
        On retire le premier élément de nCompress, epsCompress et chCompress
        Si QParent n’est pas une feuille alors
                Si QParent n’a que des feuilles alors
                        On insert QParent dans nCompress, son chemin dans chCompress et
son epsilon dans  epsCompress
```

```
                Fin Si
        Fin Si
Fin
```

La compression Rho ne suit pas le schéma de la compression précédente. Elle se base tout d'abord sur un pourcentage de nœuds maximum à conserver. Grâce à cela, on peut retrouver le nombre de nœuds à supprimer pour avoir notre résultat.

Tout d'abord, il faudra analyser le Quadtree et stocker les nœuds compressibles dans un tableau en ordre croissant. Ensuite, il faudra retenir l'indice de cette insertion pour insérer le Epsilon et le chemin dans deux autres tableaux aux mêmes indices. La sauvegarde du chemin de ce nœud nous permettra de retrouver son père en très peu de temps, améliorant donc la complexité temporelle de notre programme. De ce fait, il est possible de retrouver le plus petit Epsilon en tant constant.

Après avoir effectuer une compression, on regarde si le parent du nœud compressé est compressible, si c'est le cas, on l'insère de la même manière que les autres nœuds. Cette méthode va donc compresser les nœuds jusqu'à ce que le nombre de nœuds à supprimer soit à zéro.

Complexité temporelle de la compression Rho :

Insertion dichotomique dans nCompress :

La complexité de l'insertion dichotomique considérée comme O(log n) pour la recherche de la position et potentiellement jusqu'à O(n) pour le décalage des éléments, n étant le nombre d'éléments du tableaux.
Le comptage du nombre de nœuds est en O(m), m étant le nombre de nœuds.
On insère des éléments x fois le nombre d'éléments à supprimer.
Le reste de nos opérations se font en tant constants.

Donc on obtient une complexité de (p * log (q)), p étant le nombre de nœuds à supprimer divisé par 4 (car on soustrait 4 à ce nombre à chaque compression) et q étant la taille du tableau de nœuds compressibles au début.

Fonction QtoString(Quadree Q)

```
Début
|       Si(Q. != null) alors
|       |       Si(Q.Q1 != null) alors
|       |       |       Afficher("(")
|       |       |       Q.Q1.QtoString()
|       |       |       Afficher(" ")
|       |       Fin Si
|       |       Si(Q.Q2 != null) alors
|       |       |       Q.Q2.QtoString()
|       |       Fin Si
|       |       Si(Q.value = -1) alors
|       |       |       Afficher(" ")
|       |       Sinon
|       |       |       Afficher(Q.value)
```

```
|       |       Fin Si
|       |       Si(Q.Q3 != null) alors
|       |       |       Q.Q3.QtoString()
|       |       Fin Si
|       |       Si(Q.Q4 != null) alors
|       |       |       Afficher(" ")
|       |       |       Q.Q4.QtoString()
|       |       |       Afficher(")")
|       |       Fin Si
|       Fin Si
Fin
```

---

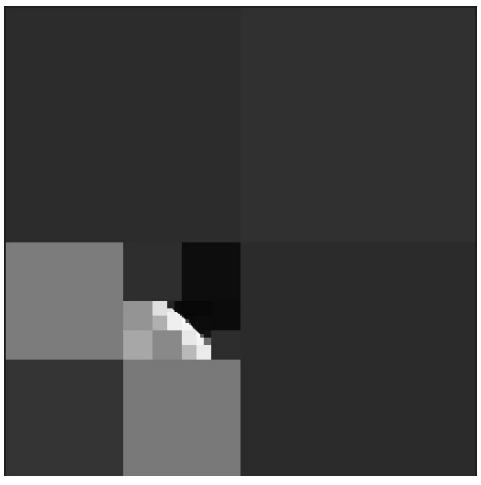
fonction Epsilon (Quadtree q) : réel

```
Variables:
        gamma,G1,G2,G3,G4 : réels

Début
|       gamma ← exponentielle(0,25*(log(0,1 + Q.Q1.value) + log(0,1 + Q.Q2.value) +
|       log(0,1 + Q.Q3.value) + log(0,1 + Q.Q4.value)))
|       G1 ← |gamma - Q.Q1.value|
|       G2 ← |gamma - Q.Q2.value|
|       G3 ← |gamma - Q.Q3.value|
|       G4 ← |gamma - Q.Q4.value|
|       retourner max(G1,G2,G3,G4)
Fin
```

3. Exemples de résultats

Compression Rho sur tree_big.pgm à :

50 % **1 %** **0,1 %**



Compression Rho sur boat.pgm à :

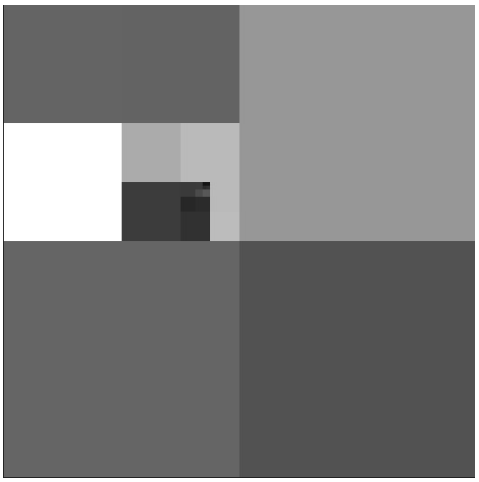
50 %

```
=====Arbre initial======

Hauteur : 9
Luminosite Max : 255
Nombre de Noeuds : 280709
========================

===Compression Lambda===

Hauteur : 8
Luminosite Max : 255
Nombre de Noeuds : 77549
Taux de compression : 27%
========================

====Compression Rho=====

Hauteur : 9
Luminosite Max: 255
Nombre de Noeuds : 140353
Taux de compression : 50.0%
========================


real    0m1,406s
```

1 %

```
=====Arbre initial======

Hauteur : 9
Luminosite Max : 255
Nombre de Noeuds : 280709
========================

===Compression Lambda===

Hauteur : 8
Luminosite Max : 255
Nombre de Noeuds : 77549
Taux de compression : 27%
========================

====Compression Rho=====

Hauteur : 9
Luminosite Max: 255
Nombre de Noeuds : 2805
Taux de compression : 1.0%
========================


real    0m1,642s
```

0,1 %



```
=====Arbre initial======

Hauteur : 9
Luminosite Max : 255
Nombre de Noeuds : 280709
========================

===Compression Lambda===

Hauteur : 8
Luminosite Max : 255
Nombre de Noeuds : 77549
Taux de compression : 27%
========================

====Compression Rho=====

Hauteur : 7
Luminosite Max: 236
Nombre de Noeuds : 29
Taux de compression : 0.01%
========================


real    0m1,548s
```

Les variations de temps ici ne sont pas très significatives. Néanmoins, sur des images plus grandes, on remarquera de plus grosses variations. Celles-ci sont très correctes et ne représentent que quelques secondes.

On retrouvera des temps d'exécution pouvant aller jusqu'à 25 secondes pour une image de 1024*1024 avec une compression à 0,1 %.

La complexité temporelle de notre programme en général est très satisfaisante et nous permet de compresser nos images en peu de temps.

4. Documentation

Pour documenter notre projet, nous avons choisi d'utiliser Javadoc. On peut remarquer que chaque attribut ou fonction de notre programme possède un commentaire juste au-dessus. Grâce à celui-ci, après s'être situé dans le dossier racine du projet, on peut générer un dossier de documentation complet en utilisant la commande suivante :

javadoc -d docs src/*.java

Cette documentation est déjà disponible dans notre dossier docs.

Pour explorer cette documentation, il faut ouvrir le fichier suivant :

index.html

Ce fichier met à disposition tout ce dont l'utilisateur a besoin pour comprendre et explorer le code de manière plus simple et organisé.

Voici un exemple (petite partie) de ce fichier index :

Field Summary

Fields

Modifier and Type	Field	Description
static java.util.ArrayList<java.lang.String>	cheminComp	
static java.util.ArrayList<java.lang.Double>	epsilonComp	Variable statique ArrayList des Epsilons correspondant aux noeuds compressibles lors de la compression Rho
static int	hmax	
static int	nbASuppr	Valeur statique Utilisee pour la suppression de noeuds lors de la compression Rho
static java.util.ArrayList<Quadtree>	noeudComp	Variable statique ArrayList des noeuds compressibles lors de la compression Rho
Quadtree	Q1	Premier noeud du Quadtree courant
Quadtree	Q2	Deuxième noeud du Quadtree courant
Quadtree	Q3	Troisième noeud du Quadtree courant
Quadtree	Q4	Quatrième noeud du Quadtree courant
int	value	La valeur du noeud (value = -1 -> branche | value !

Constructor Summary

Constructors

Constructor	Description
Quadtree()	Constructeur d'une instance de Quadtree avec 4 noeuds inexistants Constructeur d'une feuille de Quadtree
Quadtree(int v, Quadtree q1, Quadtree q2, Quadtree q3, Quadtree q4)	Constructeur d'une instance de Quadtree