Rapport Projet ASD3

T.Bersoux et N.Compère

2020

Table des matières

1 Introduction

Dans le cadre de notre enseignement en ASD3, nous avons réalisé ce projet afin de nous familiariser avec les structures de données vues en cours, en particulier les arbres quadratiques. Le sujet traitant d'un problème concret, la compression d'image, a permis d'ancrer les notions théoriques vues en cours. Cela donne l'occasion de mieux comprendre ces notions et le fonctionnements des algorithmes.
De plus, ce projet étant plus conséquent que ceux des autres années, cela nous permet d'avoir un aperçu de la réalisation professionnelle d'un programme informatique.

2 Présentation des algorithmes

2.1 Structures de données

2.1.1 QuadTree

Un QuadTree est représenté par

- Sa racine qui peut être de deux types : une coordonnées (Structure standard Point), ou un pixel (Structure Color). Nous avons fait ce choix, car cela permet de distinguer facilement nœud interne et feuille. En effet, une feuille a une couleur, mais un nœud interne n'en a pas. Ainsi, quand on veut regarder si un QuadTree est une feuille, il suffit de tester si rCouleur n'est pas null. Stocker les coordonnées permet de retrouver facilement le père, et de dessiner l'image représentée pixel par pixel avec beaucoup plus de facilité.
- Ses 4 fils, annotés Nord-Ouest, Nord-Est, Sud-Est, Sud-Ouest. Ce sont eux aussi des QuadTree.
- Un entier nbPixelCote qui représente le nombre de pixel d'un côté du carré représenté par ce QuadTree. Par exemple, pour la racine, cet entier vaudra n (pour une image de 2^n pixels) et pour une feuille, cet entier vaudra 1.
- ImageClone et ImageOriginale qui sont toutes deux au départ la même ImagePNG qui a servie a construire le QuadTree. ImageClone est utilisée pour re-créer l'image représentée par le QuadTree (cf. toPNG). Remarque : stocker les images ainsi veut dire que pour chaque nœud, on stocke deux fois l'image de base, ce qui implique un complexité spatiale élevée. Nous nous en sommes rendons compte assez tardivement dans la réalisation, du projet, et avons décidé de passer outre. Cela est un point d'amélioration possible (cf. conclusion).

2.1.2 QTEcartColor

Un QTEcartColor est une structure simple : elle stocke un nœud (≡ un QuadTree) et un entier, qui représente l'écart colorimétrique. Cette structure est importante pour l'algorithme compressPhi, car cela simplifie beaucoup sa réalisation et sert aussi de "garde-fou" contre des erreurs.

2.2 Algorithmes

2.2.1 QuadTree

QuadTree (ligne 24)

```
QuadTree(Color pixel)
```

```
QuadTree(ImagePNG image,Point coords,int tCote)
```

Présentation

Il y a un constructeur vide, qui est pratique pour faire des copies.
Il y a un constructeur(Color) demandé par le sujet, qui renvoie au "vrai" constructeur suivant qu'on utilise.
Le constructeur principal prend en paramètre l'image, les coordonnées haut gauche et la taille du côté du carré que le QuadTree construit représente. Il est récursif, la condition d'arrêt étant l'appel sur le carré d'un unique pixel.
La compression sans perte s'applique sur chaque nœud lors de sa construction.

Complexité

Un calcul rapide donne le nombre de nœuds $= (4p-1)/3$, avec p le nombre de pixels (Somme partielle de la série géométrique de raison 4), donc on considérera une complexité linaire O(n) avec n le nombre de pixels dès qu'on parcourra l'arbre. On prouve aussi facilement que la hauteur de l'arbre est $log_4(p)$. On réutilisera ces résultats dans le reste du rapport.
Cela signifie, que la classe de ce constructeur est O(n).

ecartColor (ligne 84)

```
int ecartColor()
```

Présentation

On applique juste le calcul donné dans le sujet.

Complexité

O(1)

couleurMoyenne (ligne 111)

```
Color couleurMoyenne(QuadTree noeud)
```

Présentation
Pareillement, on applique le calcul.

Complexité
O(1)

nbFeuilles (ligne 120)

```
int nbFeuilles()
```

Présentation
On récupère récursivement le nombre de feuilles de l'arbre. (Égal au nombre de pixels de l'image avant la compression sans perte)

Complexité
On parcourt tout l'arbre, donc linéaire en nombre de noeuds ≡ linéaire en nombre de pixels, O(n).

pereDe (ligne 132)

```
QuadTree pereDe(QuadTree racine, QuadTree recherche)
```

Présentation
On récupère le père du nœud passé en paramètre. On applique l'algorithme de recherche d'élément dans un QuadTree vu en Cours/TD. C'est dans cet algorithme que le fait de stocker les coordonnées simplifie grandement.

Complexité
On parcourt une seule branche, donc linéaire en hauteur de l'arbre, donc $O(\log_4(n))$, n étant le nombre de pixels.

2.2.2 compressDelta

compressDelta (ligne 154)

```
void compressDelta(int delta)
```

Présentation
On applique la méthode de compression Delta vue dans le sujet. On vérifie bien que delta est entre 0 et 255, ensuite on parcourt l'arbre et pour chaque nœud où cela est possible, on calcul l'écart colorimétrique et s'il est inférieur à delta, on supprime les fils de ce nœud et on lui applique leur couleur moyenne. Ce dernier devient une feuille.
Nous avons fait le choix de ne pas rendre cette compression récursive après quelques tests : en effet, la complexité augmente rapidement, pour un gain de compression très faible, et une perte significative de la qualité (EQM). En ne

faisant la compression que sur les feuilles, la qualité reste très bonne, tout en permettant une compression raisonnable. Si l'utilisateur veut une compression plus importante, il peut relancer l'algorithme pour compresser autant de fois que nécessaire. On peut néanmoins implémenter la récursivité facilement (cf. compressPhi).

Complexité
On parcourt tout l'arbre (sauf les feuilles), donc de classe O(n), n le nombre de pixels de l'image.

2.2.3 compressPhi

compressPhi est divisée en plusieurs sous fonction. Concrètement, on fait une liste de piles d'écarts colorimétriques (256 piles, pour des écarts colorimétriques arrondis à l'entier allant de 0 à 255), on mets tous les nœuds internes ayant des feuilles comme fils. Ainsi, pour traiter celui avec le plus petit écart colorimétrique, il suffit de dépiler la première pile non vide.
Pour ce plus petit, on supprime ses fils, on lui donne la couleur moyenne de ces derniers, et ensuite on vérifie si son père est devenu éligible à la compression. Si c'est le cas, on l'ajoute à la liste de piles.

La compression se termine lorsque le nombre de feuilles restantes est inférieure ou égale à phi.
Au départ, nous ne faisions pas la compression de cette manière, on utilisait une liste de QTEcartColor au lieu d'utiliser une liste de pile. Cela marchait, et avait l'avantage d'avoir une complexité spatiale un peu plus réduite. Mais la complexité temporelle était bien trop élevée, puisque il fallait insérer les nœuds à leur place dans une liste pouvant parfois contenir plusieurs millions d'éléments.

initPile (ligne 179)

```
LinkedList<Stack<QTEcartColor>> initPile()
```

Présentation
On initialise juste la liste avec 256 piles vides.

Complexité
O(1)

addListePile (ligne 190)

```
void addListePile(LinkedList<Stack<QTEcartColor>> liste, QTEcartColor aAjouter)
```

Présentation
On ajoute le QTEcartColor dans la pile qui correspond à son écart colorimétrique.

Complexité
Comme LinkedList.get() est linéaire en nombre d'éléments, on a O(256)≈O(1), car on aura toujours 256 piles dans la liste.

depilerPlusPetit (ligne 199)

```
QTEcartColor depilerPlusPetit(LinkedList<Stack<QTEcartColor>> liste)
```

Présentation
Retourne l'élément en haut de la première pile non vide : ainsi, cela va retourner le QTEcartColor avec le plus petit écart colorimétrique de toute la liste. En cas d'égalité, c'est donc le dernier ajouté qui va être retourné (Aucune contrainte dans le sujet).

Complexité
Comme pour addListePile, le pire cas est celui où tous les QTEcartColor ont un écart colorimétrique de 256, et dans ce cas on devra aller au bout de la liste. En pratique, on a peu de chance d'aller si loin dans la liste. Et dans tous les cas, cela reste O(1).

remplirListe (ligne 214)

```
void remplirListe(LinkedList<Stack<QTEcartColor>> liste)
```

Présentation
On parcourt tout l'arbre pour remplir la liste de piles de tous les QTEcartColor construits à partir des noeuds éligibles à la compression, en le mettant à leur place.

Complexité
O(n) avec n nombre de noeuds / nombre de pixels.

compressPhi (ligne 234)

```
void compressPhi(int phi)
```

Présentation
Utilise les fonctions précédentes pour réaliser la compression telle que décrite auparavant ; Tant que le nombre de feuilles est supérieur à phi, on dépile le plus petit QTEcartColor, on le compresse, et on vérifie son père. Cette compression là est donc récursive.

Complexité
Il y a autant de tour de "tant-que" que de feuilles à supprimer. De plus, chaque tour appelle la fonction pereDe(), qui est dans $O(\log_4(n))$, avec n le nombre de pixels. On a donc pour cette fonction une classe de complexité de $O(m\log_4(n))$, avec m le nombre de feuilles à supprimer, et n le nombre de pixels.

2.2.4 toPNG (ligne 272)

```
ImagePNG toPNG()

void toPNG(ImagePNG aModifier)
```

Présentation
La première fonction est celle demandée par le sujet. La seconde est la version qui prend en paramètre la copie d'image enregistrée dans l'arbre, sur laquelle on re-dessine l'image pixel par pixel en parcourant l'arbre.

Complexité
O(n) avec n le nombre de pixels de l'image.

2.2.5 toString

```
String toString()
```

Présentation
Parcourt l'arbre, pour chaque feuille affiche sa couleur en hexadécimal.

Complexité
O(n) avec n le nombre de noeuds/le nombre de pixels.

2.3 Main

On a crée des fonctions simples qui appellent les fonctions de QuadTree et de ImagePNG. Ainsi, chacune de ces fonctions statiques est utilisée dans la fonction main(). Cela permet au main d'être relativement clair, et modulable. Pour améliorer le confort utilisateur, le programme se met en pause pendant 1300ms après chaque fonction, ainsi l'utilisateur à le temps de constater si il y a eu un message d'erreur ou de réussite.

3 Conclusion

Nous sommes plutôt satisfait de notre programme. En effet, les complexités temporelles sont basses, le code fonctionne comme il faut. Nous avons testé avec des images faisant 4096*4096 et toutes les fonctions se font instantanément. Il y a des pistes d'amélioration, comme créer une structure spéciale pour la racine afin de sauvegarder l'image une unique fois au lieu de la sauvegarder pour chaque nœud, et ainsi réduire la complexité spatiale. On pourrait également améliorer la lecture de fichiers pour ne pas être limités au dossier pngs/. Une option pourrait aussi être implémentée pour laisser à l'utilisateur le choix du nombre de passages de la compression Delta.
Avec beaucoup plus de temps, une interface graphique pourrait être implémentée, avec aperçu de l'image originale à côté de l'image compressée.