Projet 2
Compression d'images
Bitmap

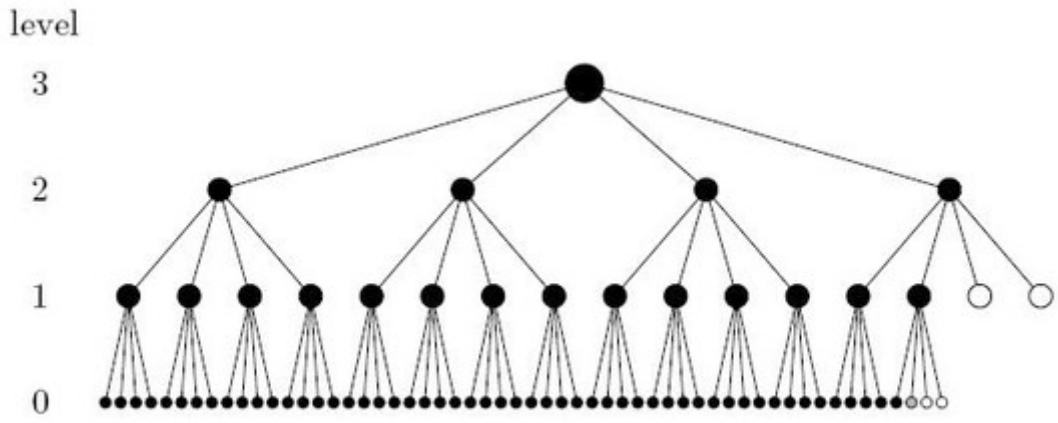
I. Introduction

Les images bitmap ont pour rôle de représenter une image via un ensemble de pixels et sont plus communément appelées images pixelisées. Le soucis majeur de ces images est leurs poids. Afin d'alléger le poids de ces dernières des compressions ont étaient inventées afin d'améliorer la qualité et le poids. Le but de ce projet est d'étudier les différentes techniques de compression ainsi que leurs représentation à travers le langage C++. Une image a était représentée lors de ce projet via une structure QuadTree où chaque feuille représente un pixel de l'image. La principale problématique étant la compression de ces dîtes images on implémentera deux méthodes, une compression delta et une compression phi. Deux classes sont implémentées, une classe ImagePNG de la bibliothèque lodepng et notre classe QuadTree définie ci-dessous.

II. QuadTree

Nos QuadTrees utilisés sont des arbres enracinés 4-aire complet, c'est-à-dire chaque nœuds interne a 4 et uniquement 4 fils. Les feuilles, nœuds ayant aucun fils, stock les couleurs des pixels d'une image. La taille d'une image est donc équivalente au nombre de feuilles d'un QuadTree. Les nœuds internes ont pour couleur la moyenne de leurs 4 fils.

Notre classe QuadTree a pour attribut privée un Noeud _racine, représentant la racine de l'arbre ainsi qu'un unsigned _taille qui représente la taille de l'arbre. On utilise une structure privée Noeud composée d'un pointeur vers un Noeud père, un tableau de 4 Noeud* fils et une Couleur. Un exemple d'arbre quaternaire est donnée ci-dessous.



Arbre quaternaire – source: Wikipedia/MD6

L'interface de notre classe est composée de plusieurs méthodes gérant l'affichage,

l'import/export et les compressions.

Nous allons expliquer ci-dessous l'implémentation de ces méthodes ainsi que les fonctions facilitatrices privées qui ont été utilisées lors de l'implémentation de ces dernières.

i. Constructeur

Notre constructeur a pour rôle de créer un arbre «vide» c'est à dire un arbre sans fils. Nous initialisons donc l'attribut _taille à 0 et son pointeur vers un nœud père à nullptr.

ii. Destructeur

Le destructeur lui supprime tout les nœuds récursivement, tant que les pointeurs vers fils ne sont pas égale à nullptr, c'est-à-dire tant qu'un nœud a des enfants on continue donc de supprimer. On appelle dans une boucle for sur les fils notre première fonction facilitatrice, deleteNoeud, qui supprime tout les noeuds récursivement avec des appels sur tout les fils.

iii. Importer

Cette méthode encode une image donnée en paramètre dans le QuadTree. Image qui doit être carré. La première chose à faire ici est de vérifier si l'arbre n'est pas déjà «remplis», si c'est le cas on supprime le contenu récursivement un peu comme pour le destructeur et on initialise sa nouvelle _taille à 0. Après quelques manipulations de taille on appelle récursivement une fonction facilitatrice, creerDescendants, qui prends comme paramètres l'adresse d'un nœud,un entier tailleNoeud, deux entiers x et y représentant les coordonnées des pixels et une image, ici l'image est toujours la même bien sûr.

Cette fonction a pour rôle de créer tout les descendants de la racine. A chaque appel elle créé les 4 fils du nœud passé en paramètre et s'appelle 4 fois sur chacun des fils en se déplaçant sur l'image tant que notre tailleNoeud n'est pas égale à 1 c'est-à-dire tant qu'on est pas arrivé aux feuilles. Sans oublier qu'à chaque appel on stock les 4 couleurs des fils dans un vector pour faire la moyenne du nœud en paramètre via la fonction moyenne de la classe ImagePNG. Lorsque tailleNoeud = 1 la récursivité s'arrête on a donc fini l'importation de l'arbre et on écrit donc pour chaque feuille sa couleur via lirePixel(x,y) appelé sur l'image importée.

iv. Exporter

Pour constater des compressions appliquées à des arbres, il faut exporter les images. Notre méthode exporter renvoie une image associé au contenu du QuadTree, nous créons donc une instance d'ImagePNG de taille 2^_taille x 2^_taille, qui sera retournée par la méthode exporter après appel de notre fonction parcoursDescendants.

parcoursDescendants fonctionne également récursivement, elle a pour paramètre un pointeur vers Noeud, un entier tailleNoeud, deux entiers x et y représentant les coordonnées des pixels et une image. Tant que le nœud passé en paramètre a des fils on rappelle la fonction sur tous les fils avec modifications des paramètres x et y pour bien modifier tout les pixels de l'image qui sera renvoyé lorsqu'on atteindra les feuilles. Arrivé

aux feuilles on peut enfin écrire sur l'image les pixels avec les couleurs associées aux feuilles grâce à deux boucles for. Un peu comme quand on rempli une matrice par exemple.

v. Compression delta

Voici maintenant les compressions. Tout d'abord notre compression delta, effectué grâce à la méthode compressionDelta de la classe QuadTree. Ayant pour contrainte de ne pas modifier l'interface publique de la classe nous avons créé 3 fonctions : elagage, testLuminance et estFeuille.

Dans notre méthode compressionDelta nous appelons uniquement la fonction elagage, fonction récursive prenant comme paramètre un pointeurs vers Noeud et un unsigned delta. Cette fonction test si les fils des fils d'un nœud sont des feuilles et si ce n'est pas le cas alors ça rappelle la même fonction sur les fils du nœud passé en paramètre. Ca permet d'atteindre les nœuds. Lorsque ce n'est pas le cas on appelle testLuminance sur le nœud pour élaguer ou non les fils selon la différence de luminance maximum. Dans testLuminance on cherche le maximum de différence de luminance, stocké dans notre variable seuil puis on le compare avec delta, lorsqu'il est inférieur ou égale à delta alors la compression est possible et on élague les fils du nœud en question puis on remonte dans la hiérarchie de l'arbre. estFeuille est une fonction booléenne qui renvoie vrai lorsqu'un nœud est une feuille, faux sinon. On vérifie uniquement sur un seul fils vu qu'on travaille sur des arbres complets, cela nous permet de passer d'une complexité temporelle O(n) à un temps constant O(1).

vi. Compression Phi

III. Complexité spatiale et temporelle

La complexité spatiale et temporelle est une problématique importante, l'implémentation de nos méthodes ont été réfléchis afin d'obtenir la meilleur complexité possible. Voici celles qu'on a réussi à obtenir.

Méthodes	***Complexité Temporelle***
QuadTree()	constant
~QuadTree()	θ(_taille)
afficher()	θ(_taille)
Importer()	θ(H*L)
exporter()	θ(_taille²)
compressionDelta	Θ(H*L,delta,pixels de l'image)
compressionPhi()	Θ(?)

Méthodes publiques du QuadTree

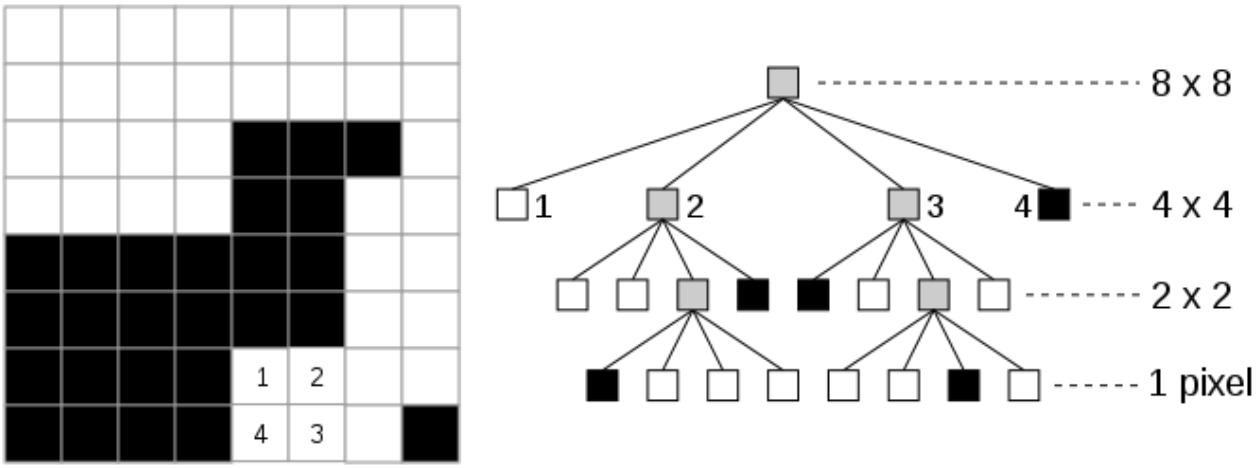
IV. Mesures expérimentales.

Nous avons 4 méthodes de la classe QuadTree dont la complexité est inconnue, le but de nos tests est de retrouver expérimentalement les complexités théoriques énoncées ci-dessus. Tout au long des mesures nous négligerons les images de taille 4096x4096 pixels et 8192x8192 car impossible à mesurer. La méthode afficher est en O(n), les test sur les images png fournis nous l'ont confirmé.

Commençons par les compressions.

1. Compression delta

La compression delta a pour but de compresser une image par élagage des feuilles en autorisant ou non un certain niveau de perte, renseigné par le delta en question. Le principe de compression se base sur la différence de luminance maximum entre un nœud et ses fils. Si la différence de luminance maximum est inférieur ou égale à delta, alors on peut élaguer les fils du nœud en question. Via ce procédé nous pouvons choisis de réduire drastiquement ou non le poids d'une image via la variation du delta. Par exemple une compression sans perte est possible avec delta = 0 comme ci-dessous.



Compression delta=0 (source Wikipedia/QuadTree)

A ce stade là des tests plusieurs choix s'offrent à nous. Calculer le temps d'exécution de la fonction selon le delta, la taille de l'image ou encore l'uniformité des pixels de l'image. Nous nous limiterons donc à ces paramètres lors des mesures de temporelles.

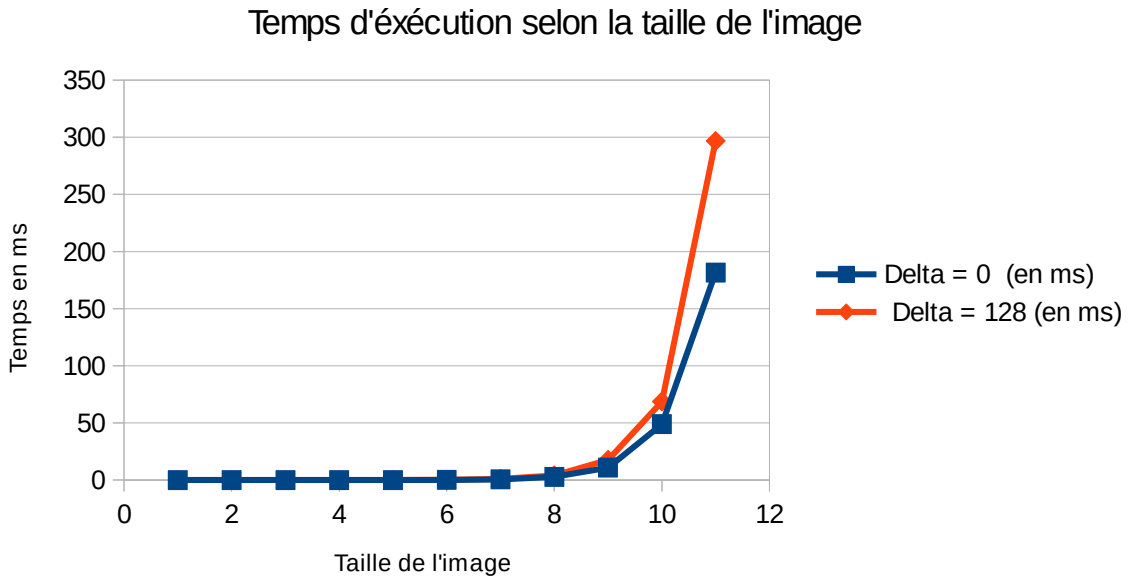
a) Variation de la taille de l'image

Nous utilisons lors de ces expérimentations des images de pixels uniformément aléatoires dont la taille varie afin de vérifier si c'est un paramètre déterminant ou non. On suppose à ce stade des tests que les pixels n'influent pas sur la compression.

Nous faisons tout d'abord varier la taille des images pour le temps d'exécution avec delta = 0 et delta = 128. Voici un tableau représentant les valeurs trouvées ainsi qu'une représentation graphique de ces dernières.

Image (en pixels)	***Delta = 0 (en ms)***	***Delta = 128 (en ms)***
Image 2x2	0.001	0.001
Image 4x4	0.001	0.001
Image 8x8	0.002	0.005
Image 16x16	0.006	0.015
Image 32x32	0.027	0.055
Image 64x64	0.1	0.2
Image 128x128	0.566	1.091
Image 256x256	2.833	4.136
Image 512x512	10.895	17.611
Image 1024x1024	48.855	68.71
Image 2048x2048	181.636	296.8

Variation de la taille de l'image pour la méthode compressionDelta



Comme nous pouvons le constater les valeurs augmentent selon la taille de l'image. Pour des images relativement petites(inférieur à 512x512) le temps d'exécution est globalement le même mais dès qu'on franchit ce cap il augmente énormément. On peut constater également des valeurs différentes pour delta=0 et delta=128 et qu'il influe donc lui aussi le

temps d'exécution de la compression.

b) *Variation de delta et des pixels*

Les deux derniers facteurs qui pourraient influer sur le temps d'exécution sont le delta et l'uniformité des pixels. L'uniformité des pixels est un facteur totalement légitime car si on y pense plus les pixels sont identiques et plus la différence maximum de luminance se rapproche potentiellement de 0 et donc plus d'élagation. Cela a sans doute son rôle à jouer.

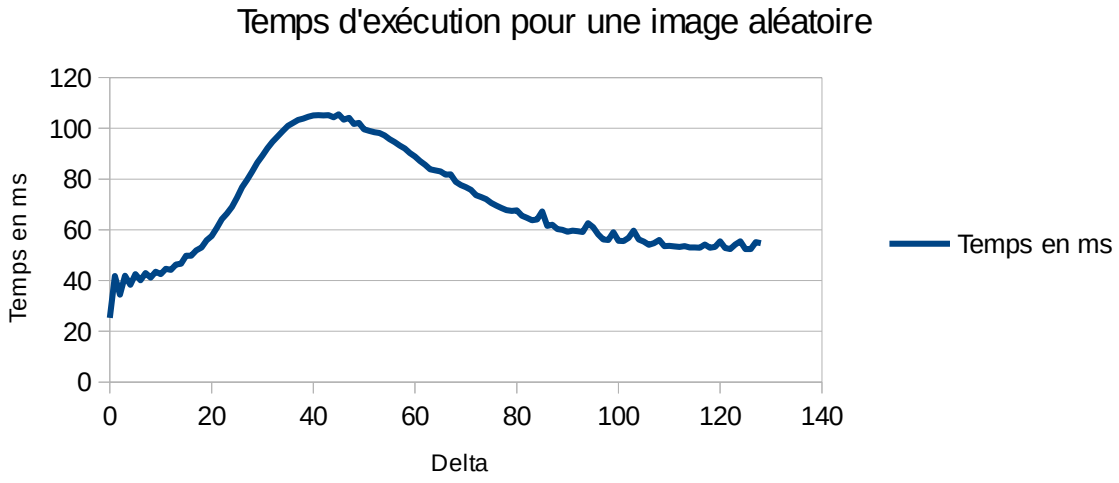
On effectue les mesures sur une image dont les pixels sont placés aléatoirement et sur une image d'une couleur uniforme.

Nous choisissons donc une taille d'image fixe, ici on utilise deux images de taille 1024x1024 pixels.

Delta	***aléatoire(ms)***	***uniforme (en ms)***
0	25.017	44.438
25	44.546	44.462
50	52.041	47.951
75	82.163	44.464
100	82.144	44.478
128	59.486	44.467

Via ces mesures nous constatons deux choses, que le delta n'influe pas le temps d'exécution sur une image uniforme(où tout les pixels sont les mêmes) et que sur une image aléatoire les valeurs ne sont pas très significatives.

Nous refaisons donc un test sur la même image mais cette fois-ci nous faisons varier delta un à un pour avoir une meilleur allure. 130Lignes étant un peu trop, nous ferons uniquement une représentation graphique des mesures.



Variation de delta sur une image aléatoire 1024x1024

Cette fois-ci nous avons une meilleur appréciation des valeurs. On retrouve globalement les 6valeurs de l'expérience précédentes mais cette fois ci on peut mieux les interpréter.

On constate que le temps d'exécution n'est pas proportionnel au delta, nous avons une augmentation du temps d'exécution jusqu'à atteindre le pic delta = 50 pour en suite décroître jusqu'à stagner quand le delta dépasse 100. D'après ces résultats il est garanti que le choix du delta a son importance et qu'il peut significativement améliorer le temps d'exécution de la compression.

Le poids des images compressées est jusqu'à 3fois plus léger pour les images compressées avec un delta équivalent à 128, mais la qualité est mauvaise pour beaucoup d'entre elles.

2. *Compression Phi*

3. *Importer*

Notre fonction importer a pour but d'importer une image dans un QuadTree, lors des expérimentations nous cherchons à trouver des facteurs qui pourraient réduire le temps d'exécution de la fonction et retrouver sa complexité temporelle.

Nous faisons ici deux tests, le premier mesure le temps d'exécution d'importer en faisant varier la taille des images, images dont les pixels sont placés aléatoirement, le deuxième test la même chose sur des images de couleur uniforme.

Taille (en pixels)	**Image aléatoire**	**Image uniforme**
Image 2x2	0.004	0.003
Image 4x4	0.007	0.005
Image 8x8	0.02	0.026
Image 16x16	0.066	0.065
Image 32x32	0.26	0.273
Image 64x64	1.055	1.03
Image 128x128	5.116	5.201
Image 256x256	16.941	17.637
Image 512x512	67.165	69.898
Image 1024x1024	269.833	271.793
Image 2048x2048	1077.14	1080.01

Variation des pixels de l'image pour la méthode importer

Comme on peut le constater via ces valeurs le contenu de l'image importe peu, aux imprécisions des mesures près nous avons les même valeurs. Ce n'est donc pas un facteur influant le temps d'exécution de la méthode contrairement à la taille qui augmente énormément dès que l'on dépasse des images 256x256. Sa complexité est en O(n) avec n taille de l'image(noté H*L plus haut).

4. Exporter

Grâce aux mesures expérimentales pour la fonction importer, nous pouvons admettre que les pixels d'une image n'influe pas sur le temps d'exécution lorsqu'il n'y'a aucun calcul vis à vis des couleurs des pixels, nous ferons donc les test d'exporter uniquement sur des arbres contenant des images aléatoires de taille variable.

Voici les résultats trouvés.

Taille (en pixels)	**Temps en ms**
Image 2x2	0.006
Image 4x4	0.005
Image 8x8	0.01
Image 16x16	0.011
Image 32x32	0.041
Image 64x64	0.163
Image 128x128	0.893
Image 256x256	2.767
Image 512x512	11.062
Image 1024x1024	43.618
Image 2048x2048	176.122

La taille est donc un facteur non négligeable lors de l'exportation d'images, plus la taille augmente et plus le temps de de l'exportation augmente. La complexité temporelle est de O(_taille).

V. Problèmes rencontrés

Le premier problème rencontré fut la récursivité, la représentation des arbres ainsi que les méthodes associées ont étaient assez difficile à implémenter, nous n'avons d'ailleurs pas réussis à faire la compression phi. Nous avons passé du temps sur la

méthode exporter ainsi que compressionDelta, les manipulations des coordonnées x et y des pixels ont étaient difficile à visualiser.

Le second problème a était la complexité spatiale, n'ayant était traité que rarement en TD – comparé à la complexité temporelle – ce fut pour nous impossible de retrouver les complexité spatiales des méthodes vu leurs complexités, nous avons donc préféré ne pas les traiter car nous en avons retrouvé aucune. Le dernier problème rencontré fut l'interprétation des résultats afin de retrouver les complexités théoriques, en effet pour certains cas (la méthode compressionDelta par exemple) on a jusqu'à 3 paramètres influant le temps d'exécution, difficile d'en choisir un qui surclasserait les deux autres, nous avons donc par exemple jugé correct de citer les 3.

VI. Conclusion

A travers ce projet nous avons pu étudier les différentes techniques de compressions d'images et étudier leurs efficacités. Pour la compression delta plus le delta est grand et plus la perte de qualité est grande mais plus le poids est réduit, il faut donc choisis le bon compromis selon les images à traiter.
Comme pour quasiment toutes les méthodes de QuadTree, le temps d'exécution est relativement bas lorsqu'il s'agit d'images inférieur à 512x512 pixels, la structure QuadTree est donc performante pour des images de petites tailles et donc de faible poids mais lorsqu'il s'agit d'image très grande les performances s'essoufflent assez vite.

Légende: Bleu pour les variables, vert pour les méthodes/fonctions.