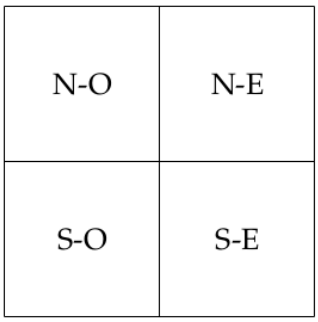
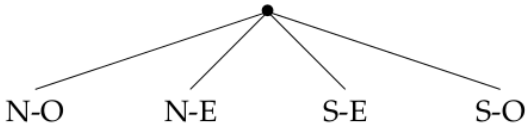
Projet de programmation C
Compression d'images avec des quadtrees

L'objectif de ce projet est de développer une application graphique pouvant encoder, décoder et compresser avec ou sans perte des images en utilisant des quadtrees.

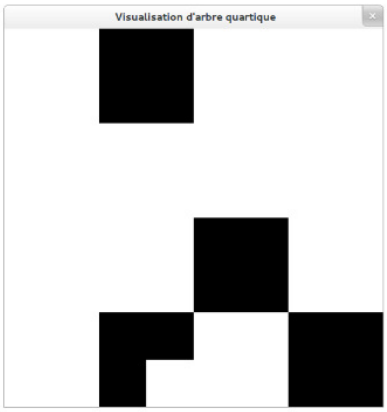
Codage d'image par arbre quartique (quadtree)

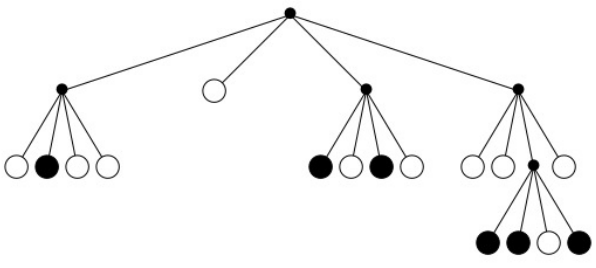
Une image peut être décrite récursivement par zones en la découpant systématiquement par quatre lorsque l'on souhaite avoir plus de précision dans une zone. Ainsi, l'image sera représentée par un arbre dont les noeuds internes auront chacun quatre fils et dont les feuilles contiendront une couleur. Pour une image en noir et blanc (2 couleurs uniquement), seulement deux types de feuilles suffisent.





Voici un exemple complet de correspondance arbre image :





Pour éviter toute considération concernant la taille et la profondeur (possiblement immense) de l'arbre quartique, on se limitera à des images carrées de 512 pixels de côté. Un arbre naïf complet représentant une image pixel par pixel aura ainsi au plus 262144 feuilles et 87381 noeuds internes.

Approcher une image avec un quadtree grandissant

Prenons une image 512 x 512 soit en couleur (rgba 4 octets) ou bien noire et blanche (1 bit uniquement). On fait la moyenne des pixels apparaissant dans l'image et on obtient ça :

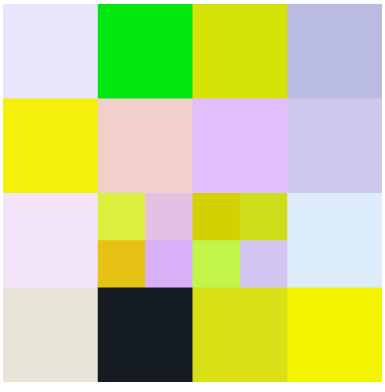
moyenne de tous les pixels de l'image

Ensuite, on coupe l'image en quatre et on refait la moyenne des couleurs sur chacune des quatre zones :

quatre zones avec leurs couleurs moyennes

On itère maintenant le procédé suivant : on cherche la zone la plus loin de la réalité (en calculant la somme des erreurs par rapport à la réalité de chaque zone), puis on partage cette zone en 4 avec 4 nouvelles couleurs moyennes. On s'arrête lorsque toute les zones sont parfaites.

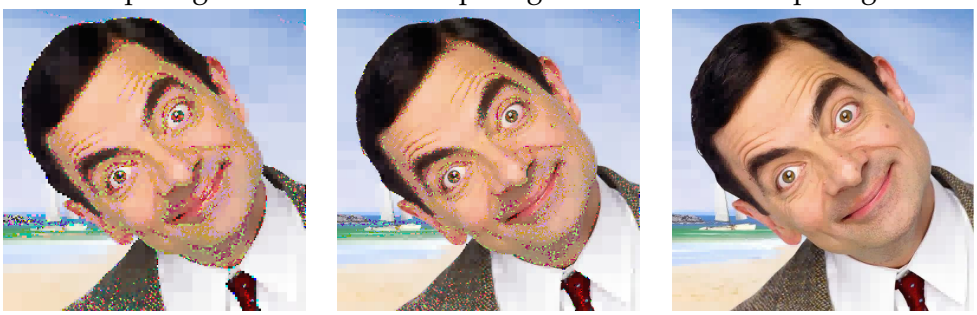
4 partages donc 4 noeuds dans le quadtree



8 partages

16 partages | 32 partages | 64 partages

128 partages | 256 partages | 512 partages



1024 partages | 2048 partages | 4096 partages

On peut facilement utiliser la distance euclidienne pour deux pixels $p_1 = (r_1, g_1, b_1, a_1)$ et $p_2 = (r_2, g_2, b_2, a_2)$ en couleur rgba :

$$dist(p_1, p_2) = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2 + (a_1 - a_2)^2}$$

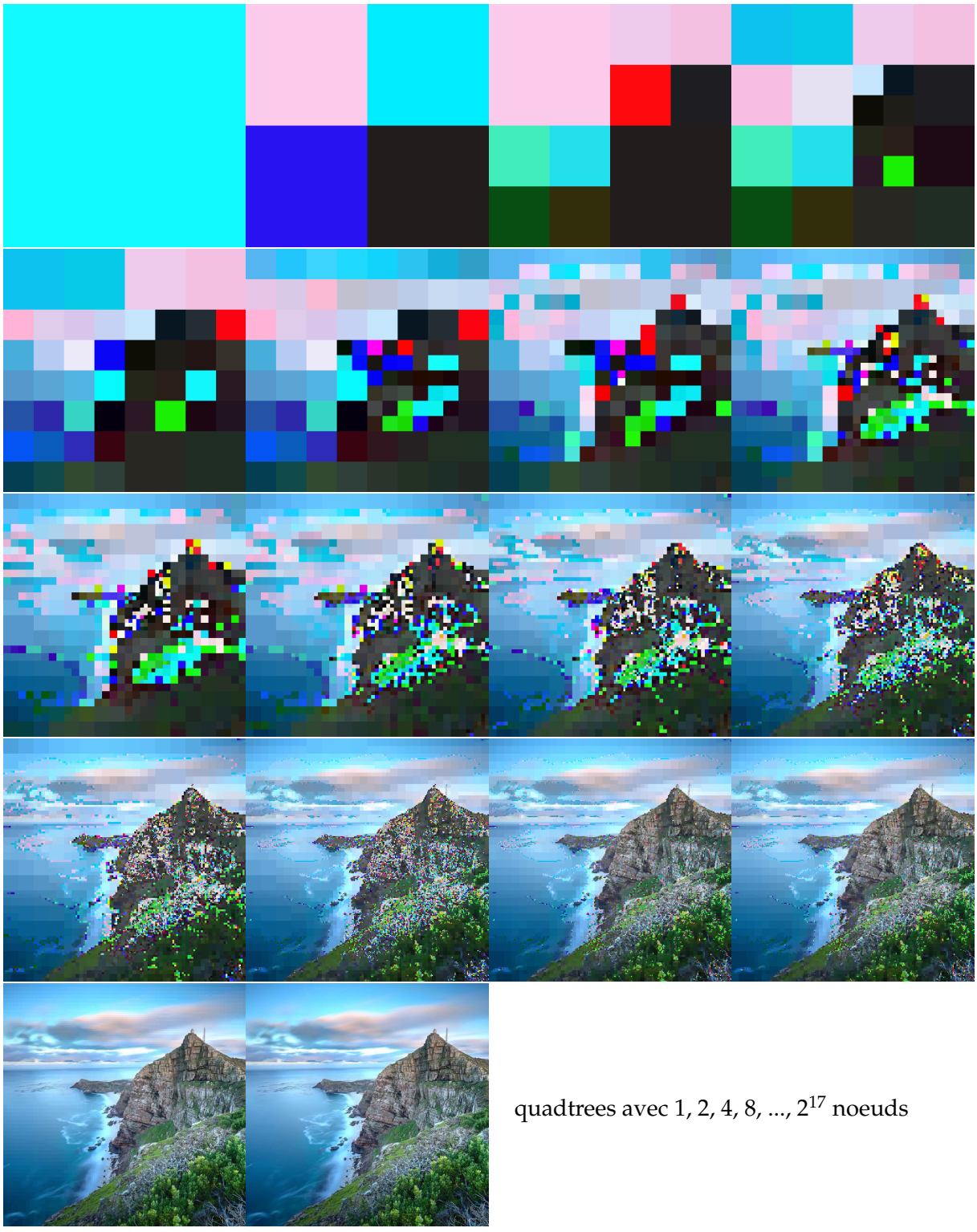
Pour une zone Z composée d'un ensemble de d pixels p_i ayant pour couleur (r_i, g_i, b_i, a_i), on peut lui attribuer une couleur moyenne (r_Z, g_Z, b_Z, a_Z) en calculant les moyennes de chacun des quatre canaux :

$$\begin{cases} r_Z = \frac{1}{d} \sum\limits_{pixel(p_i) \in Z} r_i, & g_Z = \frac{1}{d} \sum\limits_{pixel(p_i) \in Z} g_i \\ b_Z = \frac{1}{d} \sum\limits_{pixel(p_i) \in Z} b_i, & a_Z = \frac{1}{d} \sum\limits_{pixel(p_i) \in Z} a_i \end{cases}$$

Pour une zone Z, on définit l'erreur commise dans le quadtree comme il suit :

$$\text{Erreur}(Z) = \sum_{pixel(p) \in Z} dist(\text{couleur réelle de p}, (r_Z, g_Z, b_Z, a_Z))$$

Voici un autre example, encore pour les puissances de deux mais avec une image de paysage contenant un grand nombre de détails.

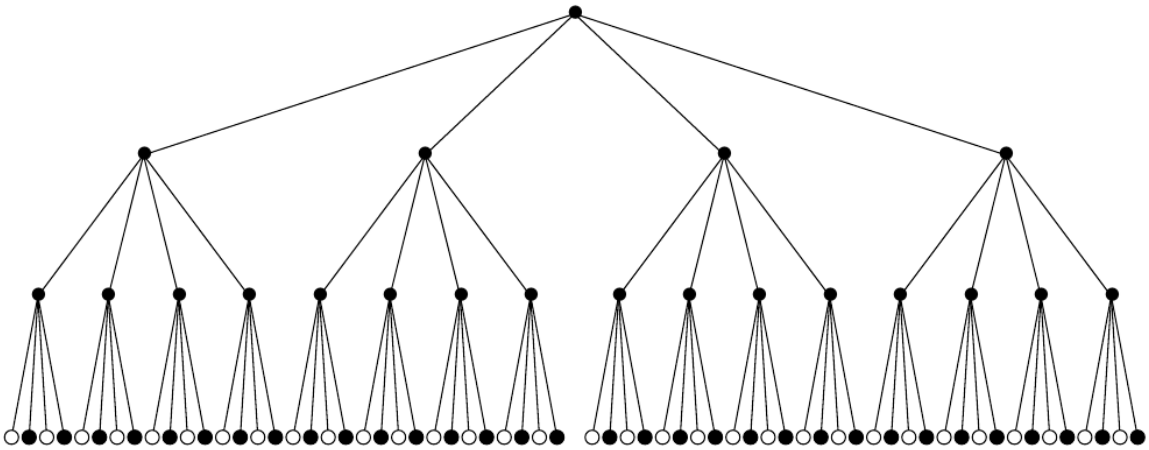


quadtrees avec 1, 2, 4, 8, ..., 2^{17} noeuds

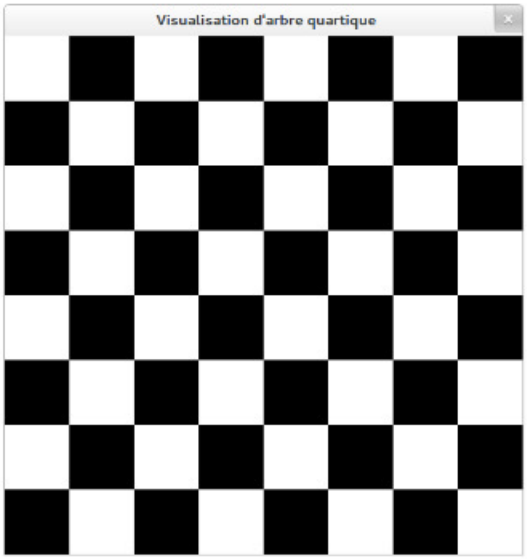
Si on arrête ce processus quand toutes les zones sont parfaites, on a encodé l'image avec un quadtree **sans perte**. Si le processus d'approximation a été arrêté avant, alors l'image construite est différente de la réalité et donc on a quelques pertes. Ce processus d'approximation va donc de l'image grossière vers l'image fine et fidèle. Le prochain processus fait l'inverse...

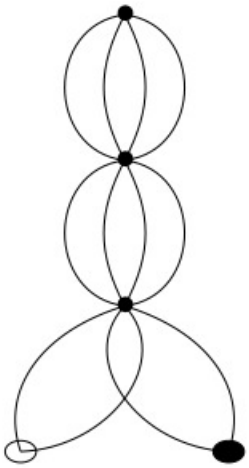
Minimisation des quadtrees

Voici un quadtree complet modélisant un échiquier. Il n'y a ici aucune perte, les 64 cases carrées de l'échiquier sont représentées.



On remarque rapidement qu'il y a des structures et notamment beaucoup de redondances dans cet arbre. Et puis un arbre, en langage C, c'est principalement des pointeurs, des chemins... Ainsi, on peut facilement compresser un arbre en faisant pointer certains fils vers le même sous-arbre pour n'en garder qu'un seul exemplaire en mémoire. Au final, le quadtree suivant encode complètement l'échiquier.





Au final, il n'y a plus que 3 nœuds internes et seulement deux feuilles différentes. (Pour une image en couleur de type (r,g,b,a), il y aura probablement bien plus de feuilles différentes.) L'arbre minimisé représente toujours l'échiquier sans aucune perte. On va voir que la minimisation automatique (en utilisant des algos programmés) est toutefois assez compliquée.

Minimisation sans perte :

Les algos pour la minimisation sont plus compliqués. Il en existe quelques uns relativement simples.

- (Simple) Tant qu'il existe un noeud à hauteur 1 ayant ses quatres enfants de la même couleur : on remplace ce noeud par un de ses enfants.

- (Simple) Ne garder qu'une seule feuille par couleur. Les images en noir et blanc n'auront que deux feuilles (2 mallocs seulement, le reste du quadtree (plein d'autres mallocs) ne sera composé que de noeuds internes).
- (Difficile) Fonction de recherche et comparaison de sous-arbres. Si un sous-arbre apparaît plusieurs fois, alors on libère le second et on rattache les deux pères vers le premier qui sera le seul gardé en mémoire.

Malheureusement, un algorithme simple ne compressera pas autant le damier que dans l'exemple. Seul des algorithmes rusés peuvent rechercher des motifs et redondances évoluées. Le traitement des images en couleur (r,g,b,a) rajoute encore davantage de complexité.

Minimisation avec pertes :

C'est ici que les Mozarts de la compression s'expriment. Il s'agit clairement de la partie la plus compliquée de ce projet. Pour compresser, il faut faire des concessions mais on souhaite faire les concessions qui ne choqueront pas l'oeil humain. Un programme informatique et déterministe ne prend pas en argument la résolution de l'oeil humain (du moins, il n'y a pas de type C pour ça...). Il faudra donc faire un nombre d'essais sûrement important sur une banque d'images de préférence très variées.

En première approximation, il faut tenter de définir une distance sur tout couple d'arbres. Dans un couple d'arbre, les deux arbres n'ont pas forcement la même profondeur et la même forme, mais il faut pouvoir tout comparer. Une fonction récursive peut faire l'affaire :

Pour comparer deux quadtrees T_1 et T_2 :

- Si T_1 et T_2 sont des feuilles :
  On retourne la distance entre les deux couleurs

$$dist(T_1, T_2) := dist(couleur(T1), couleur(T_2))$$

- Si T_1 est une feuille mais pas T_2 :
  On retourne la somme des quarts des distances de T_1 vers chaque fils de T_2

$$dist(T_1, T_2) := \sum_{i=1}^{4} \frac{dist(T1, \text{fils numéro i de } T_2)}{4}$$

- Si T_2 est une feuille mais pas T_1 :
  On retourne la somme des quarts des distances de T_2 vers chaque fils de T_1

$$dist(T_1, T_2) := \sum_{i=1}^{4} \frac{dist(\text{fils numéro i de } T_1, T_2)}{4}$$

- Si T_1 et T_2 ne sont pas des feuilles :
  On retourne la somme des quarts des distances de chaque paire de fils de T_1 et T_2

$$dist(T_1, T_2) := \sum_{i=1}^{4} \frac{dist(\text{fils numéro i de } T_1, \text{fils numéro i de } T_2)}{4}$$

Une fois que l'on a une fonction distance qui fonctionne, il faut rechercher dans le quadtree des paires de sous-arbres ayant une distance minimale (c'est algorithmiquement très lourd : naïvement quadratique sur le nombre de nœuds). Pour trouver ces couples, il va falloir ainsi raffiner pour éviter des doubles boucles `for` sur tous les nœuds puis un tri. Une heuristique intéressante et possible est l'utilisation de la couleur moyenne. Un sous-arbre définit une zone et on peut lui associer une couleur moyenne comme dans la première partie de ce projet. En réfléchissant, vous verrez que deux arbres proches doivent avoir une couleur moyenne proche. Par contraposition, deux arbres donnant une couleur moyenne éloigné ne peuvent être proches.

Quand on trouve un couple de sous-arbres distincts à distance zéro, on minimise sans perte. Sinon, il faut tenter de trouver le couple d'arbres les plus proches, puis libérer le second et rattacher les deux pères vers l'unique exemplaire gardé en mémoire. On peut répéter ce procédé tant que l'oeil humain n'est pas choqué par ce changement sur l'image. Ici, seul des simulation vous diront jusqu'où vous pouvez aller. C'est probablement la manière la plus fine de compresser.

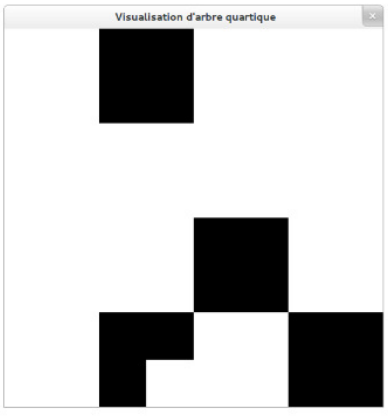
Format d'encodage

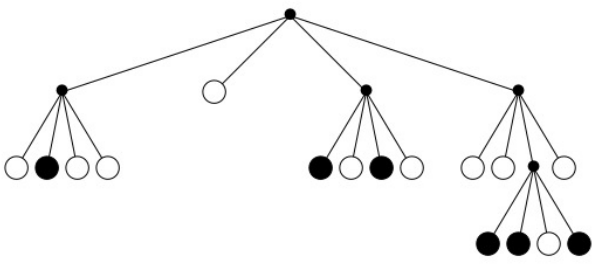
Nous allons définir quatre types de format pour sauvegarder les images construites sous forme de quadtree (minimisé ou non). Pour les quadtree en noir et blanc, on nommera ces fichier avec l'extension `.qtn` alors que pour la couleur, ce sera `.qtc`. Ces fichiers seront à ouvrir, lire et écrire en bit à bit (bien voir le cours associé...).

On placera dans ces fichiers un parcours en profondeur préfixe du quadtree (forcément non minimisé) avec les règles suivantes :

- Les nœuds internes seront codés par un bit à 0.
- Pour les images monochromes, les feuilles seront codées par le bit 1 suivi d'un bit indiquant sa couleur, 1 pour noir et 0 pour blanc.
- Pour les images en couleurs, les feuilles seront codées par le bit 1 suivi de 4 octets décrivant les valeurs de r, g, b, et a.

En reprenant le premier exemple noir et blanc de cet énoncé :



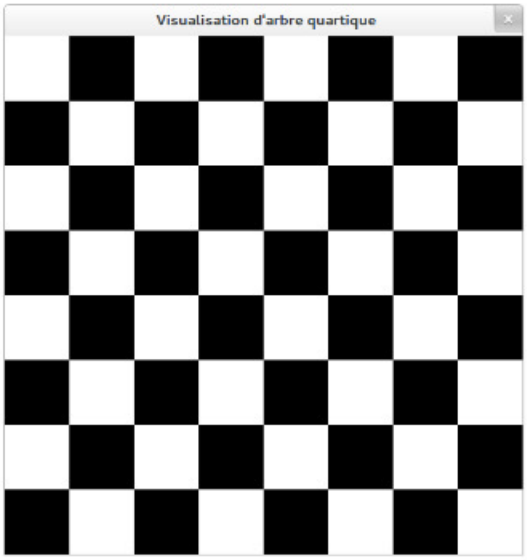


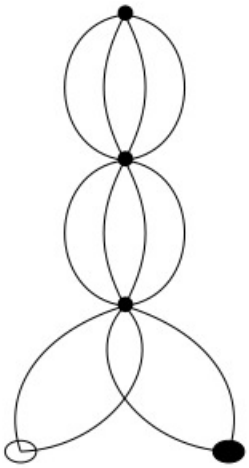
La donnée de sauvegarde sera la suivante :
00101110101001110111001010011111011 10

qui peut être lue, de manière structurée, comme il suit :
0 (0 (10 11 10 10) (10) 0 (11 10 11 10) 0 (10 10 0 (11 11 10 11) 10))

Les quadtrees minimisés ne sont plus des arbres mais des graphes (des graphes orientés acycliques enracinés). Un parcours ne permet plus de les caractériser sans redondance d'information. Or, quand on fait de la compression, la redondance d'information est l'ennemi numéro un. L'idée est alors d'encoder ça comme un graphe (quoi, vous n'avez pas eu de cours sur les graphes ??? Mouhahahaha, souffrance dans la programmation n'est que volupté!).

On donne alors un unique numéro à chaque noeud interne (en commençant par 0 pour la racine) et on fait un fichier qui décrit chaque noeud. Reprenons le damier minimisé à outrance :





Si la racine a pour numéro 0, 1 pour le noeud interne en dessous et 2 pour le troisième noeud interne le plus bas, le fichier devra alors être de la forme :

```
0 1 1 1 1
1 2 2 2 2
2 3 4 3 4
3 0
4 1
```

Il y a donc 5 items au total. Les nœuds 3 et 4 sont des feuilles suivies par une couleur (0 blanc et 1 pour le noir). Le nœud 0 a ses quatre fils qui pointent vers le nœud 1, le nœud 1 a ses quatre fils qui pointent vers le nœud 2, les fils du nœud 2 pointent alternativement vers la feuille blanche et la feuille noire.

Pour les images en couleurs en (r,g,b,a), seul l'encodage des feuilles change. Il faut être capable de savoir qu'on a une feuille et donc que les informations qui suivent la feuille sont des niveau de couleur et pas d'autre fils. On codera, par exemple, une feuille rouge portant le numéro 125 dans le quadtree de la manière suivante :

```
125f 255 0 0 255
```

Ces codages de graphes acycliques sont très très loin d'être optimaux et pourraient faire largement augmenter la taille des images encodées. Parmi les mauvais choix, ce codage n'est pas binaire, il n'utilise que les caractères : '0', '1', '2', ... , '9', ' '(espace), 'f' et '\n' sur les 128

caractères du code ascii. Toutefois, il n'y a pas de redondance d'information si le graphe est bien minimisé.

Pour mesurer la vraie capacité de votre programme à minimiser les images, il sera ainsi intéressant de comparer :

- Un zip d'une image de base (jpg, png, ...)
- Un zip de la sauvegarde binaire du quadtree (Le zip ne devrait pas compresser beaucoup)
- Un zip de la sauvegarde sémantique d'un graphe minimisé (Le zip va pallier au mauvais choix de la représentation)

Les fichiers de sauvegarde de graphes minimisés en noir et blanc seront des fichiers texte d'extension `gmn` alors que les fichiers de sauvegarde de graphes minimisés en couleurs seront d'extension `gmc`.

Description du programme

Votre programme doit être capable d'ouvrir des images de taille 512 par 512 d'extension semblable à ce qu'est capable de faire la libMLV (png, jpg, gif, tiff, pcx, tga, etc). L'affichage de l'image de base est toujours importante pour apprécier le point de départ des calculs de votre projet.

Dans votre interface, quelques boutons cliquables seront appréciés :

- Un bouton lançant l'approximation sous forme de quadtree
- Un bouton pour faire une sauvegarde binaire noir et blanc (l'utilisateur est responsable du clic)
- Un bouton pour faire une sauvegarde binaire en rgba (pour l'utilisateur à cliquer)
- Un bouton pour lancer la minimisation
- Un bouton pour faire une sauvegarde de graphe minimisé noir et blanc (l'utilisateur est responsable du clic)
- Un bouton pour faire une sauvegarde de graphe minimisé en rgba (l'utilisateur est responsable du clic)
- Un bouton pour ouvrir une image à partir de son nom. Votre programme devra alors interpréter l'extension du fichier pour savoir comment produire un affichage de l'image (qtn, qtc, gmn, gmc, reste du monde = jpeg, png, etc ...)

Les boutons de sauvegarde devront générer les fichiers idoines dans un répertoire `img` placé dans le répertoire racine de votre projet.

Modularité

Votre projet doit organiser ses sources sémantiquement. Votre code devra être organisé en modules rassemblant les fonctionnalités traitant d'un même domaine. Un module sans entêtes pour le `main`, un module pour la représentation des quadtrees et un module graphique

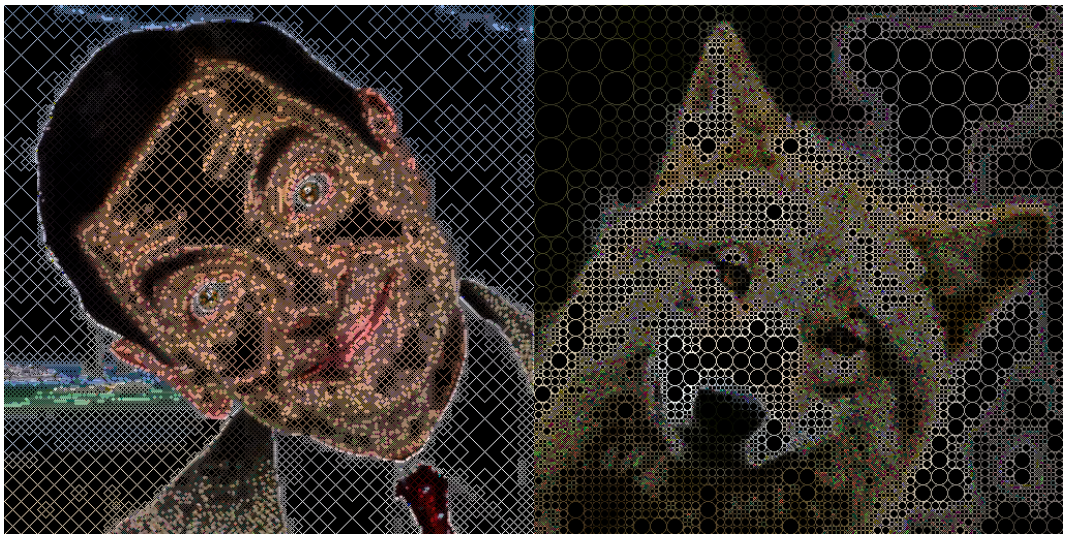
semblent être un minimum. La partie graphique devrait vraisemblablement être très importante et on peut facilement imaginer la subdiviser.

Votre projet devra être compilable via un `Makefile` qui exploite la compilation séparée. Chaque module sera compilé indépendamment et seulement à la fin, une dernière règle de compilation devra assembler votre exécutable à partir des fichiers objets en faisant appel au linker. Le flag -lMLV est normalement réservé aux modules graphiques ainsi qu'à l'assemblage de l'exécutable (sinon, c'est que votre interface graphique dégouline de partout et que vos sources sont mal modulées).

Pour aller plus loin

Pour les plus récalcitrants d'entre vous, toute amélioration sera considérée comme un plus pour l'évaluation. Voici quelques suggestions possibles de raffinements pour ce projet. Ces propositions sont graduées avec un indice de difficulté entre parenthèses.

- (Pour enfant) Rajouter un mode dans votre application permettant la visualisation progressive des images générées. Par exemple, cet énoncé présente souvent les images générées pas à pas pour des puissances de 2. On peut imaginer la présence d'une option (bouton graphique peut-être) pour changer le taux de rafraichissement (la vitesse au final) de l'image générée petit à petit. Un affichage du nombre de noeuds dans l'arbre en construction est un plus.
- (Pour adulte) Faire de la minisation avec pertes tout en gardant des images fidèles à la réalité.
- (Pour adulte) Proposer un encodage bit à bit des graphes, monochromes et/ou en couleurs, qui économise la taille du fichier. Faire une comparaison pour voir l'économie de votre encodage sur des images d'exemples.
- (Pour adulte sérieux) Après le milieu d'un segment, définir le milieu de deux quadtrees. Lors d'une minimisation avec pertes, on ne garde qu'un seul sous-quadtree à partir des deux quadtrees les plus proches. Définissez donc le quadtree milieu de deux quadtrees et remplacez par le milieu les deux quadtrees d'origine.
- (Pour grand enfant) Vous apportez des preuves de l'ouverture, par vos programmes, d'images en bit à bit encodées en quadtree par d'autres binômes. Vous apportez des preuves que d'autres groupes ont pu ouvrir des images encodées par vos soins.
- (Pour adulte sérieux) Vous apportez des preuves de l'ouverture, par vos programmes, d'images minimisées encodées en graphes par d'autres binômes. Vous apportez des preuves que d'autres groupes ont pu ouvrir vos images encodées sous forme de graphes minimisés par vos soins.
- (Pour enfant) Vous fournissez une banque d'images sympathiques ouvrables par vos programmes.
- (Pour enfant joueur) Générer des images trop stylées comme le remplacement des rectangles par des cercles, ce qui permet de visualiser où le quadtree est précis et où le quadtree est grossier...



Le visage d'un inconnu avec des croix... Un chien avec des cercles...

Un panda qui a trop joué à la Playstation...Un homme très fier de son chapeau...

Conditions de développement

Le but de ce projet est moins de pondre du code que de développer le plus proprement possible. C'est pourquoi vous développerez ce projet en utilisant un système de gestion de versions git. Le serveur à disposition des étudiants de l'Université est disponible à cette adresse : `https://forge-etud.u-pem.fr/`. Comme nous attendons de vous que vous le fassiez sérieusement, votre rendu devra contenir un dump du fichier de logs des opérations effectuées sur votre projet, extrait depuis le serveur de gestion de versions ; afin que nous puissions nous assurer que vous avez bien développé par petites touches successives et propres (commits bien commentés), et non pas avec un seul commit du résultat la veille du rendu. L'évaluation tient compte de la capacité du groupe à se diviser équitablement le travail.

Remarques importantes :

- L'intégralité de votre application doit être développée exclusivement en langage C (la documentation technique dynamique fait évidemment exception). Toute utilisation de code dans un autre langage (y compris C++) vaudra 0 pour l'intégralité du projet concerné.

- En dehors des bibliothèques standard du langage C et de la libMLV, il est interdit d'utiliser du code externe : vous devrez tout coder vous-même.
  Toute utilisation de code non développé par vous-même vaudra 0 pour l'intégralité du projet concerné.
- Tout code commun à plusieurs projets vaudra 0 pour l'intégralité des projets concernés.

Conditions de rendu :

Vous travaillerez en binôme et vous lirez avec attention la Charte des Projets. Il faudra rendre au final une archive `tar.gz` de tout votre projet (tout le contenu de votre projet `git`), les sources de votre application et ses moyens de compilation. Il sera alors crucial de lire des recommandations et conseils d'utilisation de `git` sur la plate-forme `moodle`. Vous devrez aussi donner des droits d'accès à votre chargé de TD et de cours à votre projet via l'interface `redmine`.

Un exécutable devra alors être produit à partir de vos sources à l'aide d'un `Makefile`. Naturellement, toutes les options que vous proposerez (ne serait-ce que –help) devront être gérées avec `getopt` et getopt_long.

La cible `clean` doit fonctionner correctement. Les sources doivent être propres, dans la langue de votre choix, et commentées. C'est bien de se mettre un peu à l'anglais si possible.

Votre archive devra aussi contenir :

- Un fichier `log_dev` correspondant au dump des logs de votre projet (nom des commits, qui? et quand?), extrait depuis le serveur de gestion de versions que vous aurez utilisé.
- Un fichier `makefile` contenant les règles de compilation pour votre application ainsi que tout autre petit bout de code nécessitant compilation (comme les tests par exemple).
- Un dossier `doc` contenant la documentation technique de votre projet ainsi qu'un fichier `rapport.pdf` contenant votre rapport qui devra décrire votre travail. Si votre projet ne fonctionne pas complètement, vous devrez en décrire les bugs.
- Un dossier `src` contenant les sources de votre application.
- Un dossier `include` contenant tous les headers de vos différents modules.
- Un dossier `bin` contenant les fichiers objets générés par la compilation séparée.
- Aucun fichier polluant du type `bla.c~` ou `.%smurf.h%` généré par les éditeurs. Votre dossier doit être propre !
- Si possible, la partie html générée par l'utilitaire `doxygen` à partir de votre application de visualisation. Ceci est optionnel mais tellement plus propre.

Sachant que de nombreux vilains robots vont analyser et corriger votre rendu avant l'oeil humain, le non respect des précédentes règles peuvent rapidement avorter la correction de votre projet. Le respect du format des données est une des choses les plus critiques.