BUT-Info.3 2023-2024
Module "3 en 1"
Projet n°2

Format de compression d'image numérique

de l'Image au `QuadTree` et inversement

L'objectif de cette première partie est de réorganiser les pixels de l'image sous une forme hiérarchique et tirer parti de certaines corrélations *propres aux images pour tenter de compresser les données.*

Images numériques en données brutes (non compressées)

Il existe un grand nombre de formats de fichiers pour stocker les images numériques (GIF, PNG, JPEG, pour les plus connus). La plupart correspondent à des données compressées et sont structurés de manière précise et complexe.

Pour gérer ces formats, la `libg2x` fournit quelques outils (cf. `<g2x_pixmap.h>`), basés sur la suite logicielle NetPbm [1], convertissant l'image au format brut (non compressé) `PNM` (format brut standard sous environnement `Linux`)

Le format `PNM`

Il se décline en plusieurs sous-formats selon la nature des données :

- PBM (*Portable Bit Map*) pour les images binaires, en donnnées brutes ou ASCII
- PGM (*Portable Gray Map*) pour les images en niveaux de gris, en donnnées brutes ou ASCII
- PPM (*Portable Pix Map*) pour les images en couleur RGB, en donnnées brutes ou ASCII

Dans tous les cas, l'image est représentée par une matrice (appelée `pixmap`) de pixels à nbl lignes et nbc colonnes, chaque pixel étant représenté par son intensité : un entier codé sur n bits pour une image sur nbg = 2^n niveaux de gris ou un vecteur à trois composantes entières (R,V,B, par exemple), chacune étant codée sur n bits, pour une image couleur.

Nous ne nous intéressons ici qu'aux images en niveaux de gris (format PGM) dont la structure est :

- un code (*magic number*), 2 caractères ASCII sur une ligne, indiquant le format [2] : P2 si les données sont écrites en ASCII, P5 si elles sont brutes (écrites en binaire, raw).
- une ou plusieurs lignes de commentaires, commençant toujours par le caractère '#'.
- trois entiers ASCII écrits sur une ou plusieurs lignes, représentant dans l'ordre les nombres de colonnes nbc, de lignes nbl et de niveaux de gris nbg (en général 255, mais pas toujours).
- enfin viennent les données, écrites pixel après pixel, en ASCII ou en binaire selon le code d'en-tête, sans indication de changement de ligne.
  - en mode raw, un pixel est codé sur 8 bits. La taille totale des données est donc (nbc * nbl) octets.
  - en données ASCII, un pixel est représenté par 1, 2 ou 3 caractères suivi(s) d'un espace. La taille totale des données est donc comprise entre (2 * nbc * nbl) et (4 * nbc * nbl), octets.

☞ dans une optique de compression seuls les fichiers en données binaire (code P5) nous intéresseront

---

[1] voir aussi https://linux.die.net/man/3/libpnm et https://en.wikipedia.org/wiki/Netpbm

[2] les autres codes sont P1 ou P4 pour le format PBM, et P3 ou P6 pour le format PPM

Décomposition en `QuadTree` - réorganisation des données

Le principe de base de la représentation en `QuadTree` est de subdiviser récursivement le *pixmap* en quatre et d'associer à chaque nœud la moyenne des intensités de ses fils. Ainsi, la racine de l'arbre représente l'intensité moyenne sur *toute* l'image et les feuilles représentent directement les pixels, comme le montre l'exemple suivant

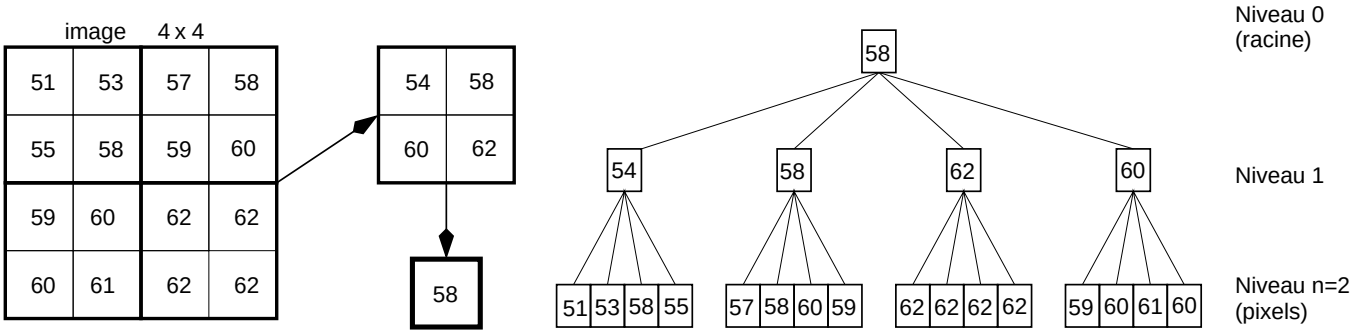

Figure 1: *Exemple de décomposition en* `QuadTree` *pour une image* (4×4)

Structure du `QuadTree` : *figure imposée*

- Pour une image de taille $2^n \times 2^n$, le `QuadTree` aura $(n+1)$ niveaux. Chaque niveau $(0 \le k \le n)$ sera stocké de manière contigüe dans un tableau de taille 4^k. Le niveau 0 correspond à la racine et le niveau n (feuilles) aux pixels de l'image (mais réorganisés).

  ☞ cette structure sera détaillé dans l'annexe suivante précisant les détails de l'interface.

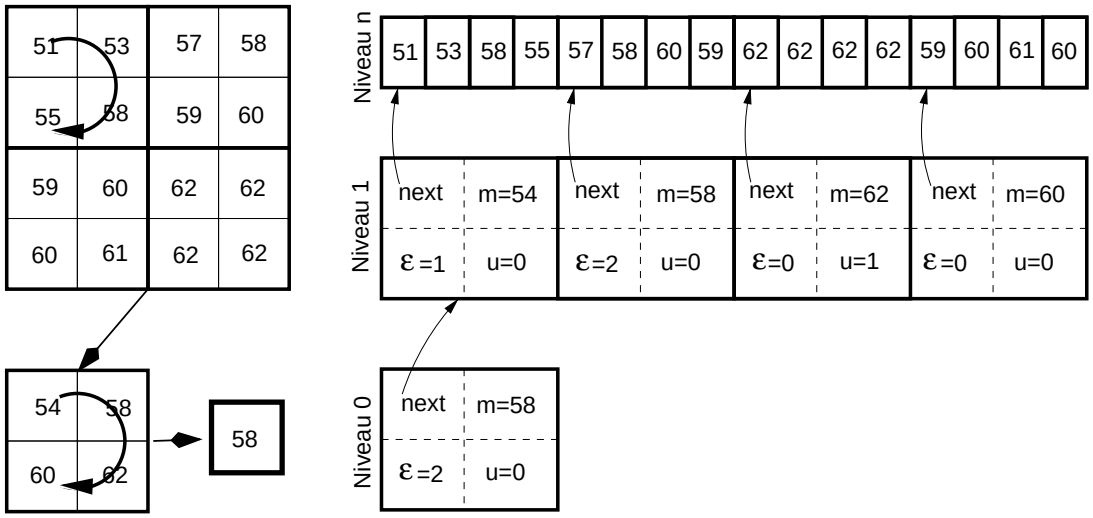

Figure 2: *Structure de* `QuadTree` *pour une image* (4×4)

- Le `QuadTree` sera construit de manière *ascendante* (parcours en profondeur de type *suffixe*), en commençant par le niveau n, qui ne contient que les valeurs des pixels de l'image, réorganisés par groupes de 4 voisins, comme sur la *fig.2*.

  Le sens de lecture des 4 voisins (ici, sens des aiguilles d'une montre) est arbitraire mais devra être le même à chaque niveau, pour tout codeur *et* tout décodeur ☞ il est donc **imposé** par le standard

  Pour cela, on commencera par *charger* l'image de manière standard dans une matrice (type `G2Xpixmap`). Cette matrice temporaire servira à la prévisualisation de référence (mode graphique) ou sera détruite une fois le niveau n du `QuadTree` constitué (mode 'terminal').

- Les niveaux $(n-1)$ à 0 seront un peu plus complexes. Chaque nœud contiendra la valeur moyenne m (entière) de ses quatre fils, ainsi que d'autres champs utiles (u et ϵ) qui seront détaillés par la suite. La *figure 2* montre l'exemple pour une image (4×4) à 3 niveaux.

Informations complémentaires pour les blocs uniformes

On remarque qu'avec la structure hiérarchique précédente, on est très loin de pouvoir compresser quoi que ce soit puisque le volume des données a, au contraire, considérablement augmenté. En effet, si une image de taille $2^n \times 2^n$ occupe un volume de 4^n octets, les données de *moyennes* du `QuadTree` correspondant occupent à elles seules un volume de $\sum_{k=0}^{n} 4^k$ octets.

Le schéma de compression proposé ici repose sur 3 critères :

- La valeur d'un nœud interne (autre qu'une feuille) représentant la moyenne de ses quatre fils, on peut se contenter de ne coder que trois de ses fils, le dernier pouvant être reconstruit par interpollation :

  $$m = \frac{x_1 + x_2 + x_3 + x_4}{4} \quad \Rightarrow \quad x_4 = 4m - (x_1 + x_2 + x_3)$$

  Le nombre de valeurs à stocker est alors $\left(1 + \sum_{k=1}^{n} (4^k - 4^{k-1}) = 4^n\right)$ comme pour l'image brute.

- `la valeur d'erreur` ϵ : le fait de travailler avec des valeurs entières va provoquer des erreurs d'arrondi lors des calculs de moyennes (division entière). Ainsi, à la reconstruction, la valeur *interpollée* à un niveau `k` risque d'être fausse et cette erreur se propagera aux niveaux `p > k`

  Si l'on veut assurer une compression sans perte (le fichier après décodage est *exactement* le même que l'original), il faut prendre en compte les erreurs introduites au codage par le calcul de la moyenne et qui se répercuteront au décodage sur l'interpollation de la valeur du quatrième fils.

  Il faudra donc ajouter aux nœuds internes du `QuadTree` un champ `ε` pour coder l'erreur. Celle-ci est facilement quantifiable puisque, si $m = (x_1 + x_2 + x_3 + x_4)/4$ (division entière), alors $\epsilon = (x_1 + x_2 + x_3 + x_4)\%4$ (où % représente l'opérateur *modulo*). Ainsi, l'erreur ϵ ne peut prendre qu'une seule des quatre valeurs 0, 1, 2 ou 3. **Donc ϵ est codable sur 2** bits.

- `le bit d'uniformité u` : lorsqu'un bloc est uniforme à un niveau `k`, il l'est à tous les niveaux `p > k`. Ainsi, si un nœud représente un bloc uniforme, le sous-arbre dont il est la racine ne contient aucune information supplémentaire : il pourra être éliminé au moment du codage.

  Il faudra donc prévoir, pour chaque nœud autre qu'une feuille, un champ booléen `u` (codable sur 1 bit), précisant si le bloc représenté est uniforme ou non.

  Le processus de compression n'aura donc pas besoin de descendre dans le sous-arbre correspondant, et lors de la décompression, on reconstruira directement un bloc uniforme à la bonne taille.

☞ Avec l'image `4x4` précédente, la structure de `QuadTree` obtenue serait donc :

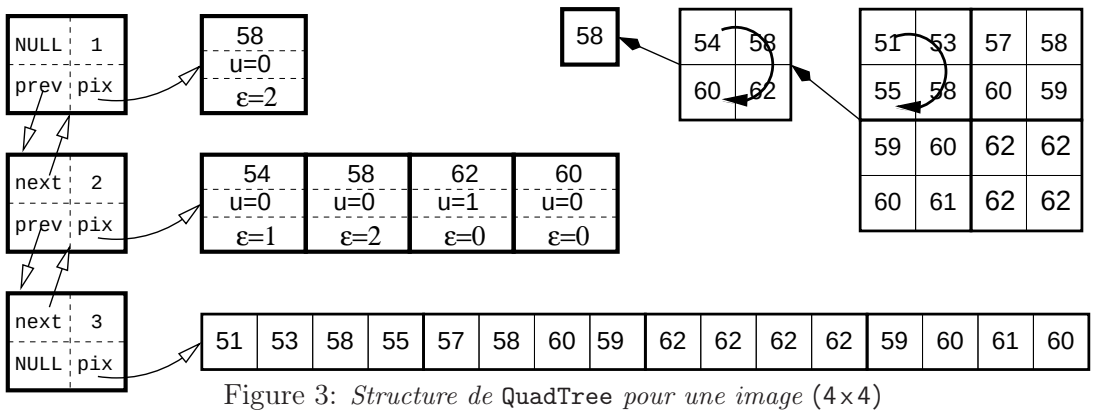

Figure 3: *Structure de* `QuadTree` *pour une image* (`4x4`)

Schéma de codage compressif

C'est bien sûr l'exploitation des blocs uniformes qui va surtout permettre de *compresser* l'image. Cela suppose évidemment que l'image à traiter en contienne suffisamment, sinon les données codées à partir du `QuadTree` risquent d'être tout de même plus volumineuses que les données originales (c'est un risque inhérent à tout schéma de compression sans perte.)

Quelques remarques

- Si, pour un nœud interne donné, on peut se contenter de ne coder que trois des valeurs `m` de ses fils, il est tout de même nécessaire de coder la valeur ϵ (et donc éventuellement le `bit u`) du quatrième fils, comme dans l'exemple de la *fig.3*, pour le quatrième bloc du niveau 1.
- Si le bloc courant est uniforme, l'erreur ϵ commise en calculant sa moyenne est forcément nulle. On en déduit que si l'erreur associée à la moyenne d'un bloc est non nulle, c'est que le bloc n'est pas uniforme.

  Il n'y a donc que dans le cas où $\epsilon = 0$ qu'il est nécessaire de coder le marqueur booléen `u` (1 bit) précisant si le bloc est uniforme ou non ☞ on économise 1 bit sur la plupart[3] des blocs **non uniformes**
- Mais on peut prendre le problème dans l'autre sens : si le bloc est uniforme (`u = 1`), l'erreur ϵ est forcément nulle, donc pas besoin de la coder ☞ on économise 2 bits sur chaque bloc **uniforme**

☞ Selon le cas, on aurait donc intérêt à coder d'abord 2 bits pour ϵ, puis éventuellement 1 bit pour `u`, ou alors, d'abord 1 bit pour `u`, puis si `u = 0`, les 2 bits pour ϵ.

Données d'en-tête

Comme pour tout format d'image, quelques données supplémentaires seront nécessaires au décodeur pour reconstruire l'image initiale.

- Le fichier commencera par un code d'identification (ou `"magic number"`) : ici la chaîne `"Q1\n"`.
- Outre ce code, les seules informations indispensables pour l'instant sont la taille et la profondeur de l'image (`< 256`). Dans le cadre de ce projet, on ne manipulera que des images de taille $2^n \times 2^n$. Il suffira donc de coder la valeur `n`, qui correspond en fait au nombre de niveaux (*profondeur*) de l'arbre.
- `Métadonnées` : en plus des données d'en-tête nécessaires au décodage, le fichier compressé devra contenir, en commentaire, les date et heure de création du fichier et le taux de compression[4] des données image (hors en-tête). Ces commentaires seront, comme dans le cas des images `PGM`, des lignes de texte `ASCII` en nombre quelconque commençant par le caractère spécial `#`.

Écriture des données du `QuadTree` (format imposé)

Avec la structure précédente en choisissant de coder d'abord ϵ, puis `u` si nécessaire, les données du `QuadTree` s'écrivent comme une suite hétérogène de mots de 8 bits (les moyennes `m`), de 2 bits (ϵ) et de 1 bit (`u`).

Sur l'exemple numérique précédent la séquence codée serait donnée par la deuxième ligne de la table suivante. La troisième ligne indique quant à elle la taille, en bits, des valeurs codées. Bien sûr, pour le codage du niveau terminal 0, on ne code plus que des valeurs de moyenne sur 8 bits.

champ	*code*	*com.*	*taille*	m	ϵ	m	ϵ	m	ϵ	m	ϵ	u	ϵ	u	m	m	m	m	m	m	m	m	m
valeurs	"$Q1$"	#.....	2	58	2	54	1	58	2	62	0	1	0	0	51	53	58	57	58	60	59	60	61
tailles	16	?	8	8	2	8	2	8	2	8	2	1	2	1	8	8	8	8	8	8	8	8	8
	en-tête			racine		niveau $n-1$									niveau 0								

Ainsi, l'image `4×4` de l'exemple, qui contient un bloc `2×2` uniforme, sera codée sur 116 bits (hors en-tête), contre 128 pour l'original (`4×4×8`), soit un taux de compression de `116/128 = 90.6%`

Selon le type d'image, les performances pourront être bien meilleures ou bien plus mauvaises !

---

[3] oui, mais sur lesquels ? Quelle est la proportion en moyenne ?

[4] c'est-à-dire le rapport $\frac{\text{(taille des données compressées)}}{\text{(taille des données originales=}4^n\text{)}}$

Schéma de décodage

Les phases de codage et décodage étant jusque là parfaitement symétriques, il est important de les développer en parallèle et de **tester**, à chaque étape, la **réversibilité** des chaînes de traitement : tout ce qui est *codé* doit pouvoir être *décodé*. Pour ces mêmes raison, il est conseillé de conserver un état fonctionnel de chaque étape (archives de versions `{Makefile*|include/|src}`)

Pour reconstituer l'image, le décodeur devra :

- ⓐ lire et évaluer le code d'identification du format et la donnée de profondeur `n` (hauteur de l'arbre).
- ⓑ extraire les données compressées du flux binaire (blocs hétérogènes de `8`, `2` ou `1` `bit(s)`) et, parallèlement, construire les niveaux successifs du `QuadTree`.
- ⓒ recréer l'image `pixmap` originale de taille $(2^n \times 2^n)$ en interpolant les blocs uniformes (remplissage).
- ⓓ visualisation et/ou enregistrement sous un format d'image brut (`PGM`).

Reconstruction du `QuadTree`

Cette opération est le symétrique de la phase d'écriture du codeur. Il faut décomposer le flux entrant en blocs de `p` (8, par défaut) `bits`, suivis éventuellement de blocs de `2` `bits` (les valeurs d'erreurs ϵ, pour les niveaux autres que le niveau terminal), eux mêmes pouvant être suivis de `bits` isolés (champ d'uniformité `u`, lorsque l'erreur ϵ est nulle).

On rappelle que pour le quatrième fils d'un nœud, la valeur de `m` n'a pas été codée, contrairement aux valeurs ϵ et éventuellement `u`. Il faudra donc prendre soin de construire le `QuadTree` en même temps pour être en mesure d'identifier les *quatrièmes fils*.

Pour un tel nœud, il faudra interpoller sa valeur `m` à partir de celles de ses trois frères et de l'erreur ϵ de son père : $x_4 = 4m - (x_1 + x_2 + x_3) + \epsilon$

De même, il faudra veiller à traiter correctement les sous-arbres de blocs uniformes : il doivent être eux-mêmes uniformisés (parcours en profondeur *préfixe*).

Reconstruction de l'image

À partir du `QuadTree` décompressé, la dernière étape consistera à reformer la matrice carrée $2^n \times 2^n$ des pixels, simplement en réorganisant le tabeau contenant le niveau `0` du `QuadTree` (cf. *fig.4*).

Enfin, le *pixmap* ainsi reformé sera affiché et/ou sauvé sur fichier pour reformer une image au format `PGM`. En commentaire de l'image devront figurer les dates et heures de compression (lues dans le fichier compressé) et de décompression.

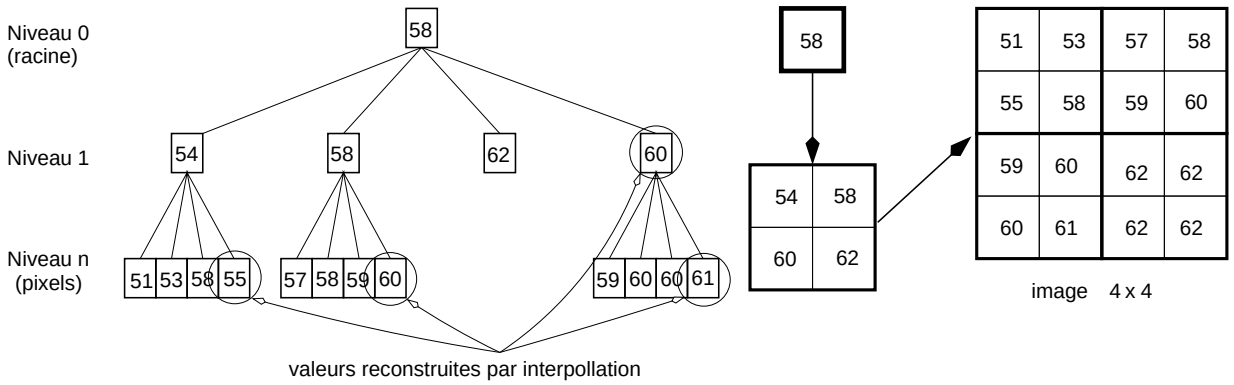

Figure 4: *Décompression de l'image* (4×4) *de la fig.1*

A ce stade, les fichiers de sortie (format `image.qtc`) ne seront que rarement (et peu) compressés. C'est normal. Ce sont les étapes suivantes qui vont corriger ça