Compression d'images à l'aide d'arbres quaternaires

Programmation avancée en C

Irwin Madet & Jimmy Teillard

17 janvier 2021

Résumé

Ce programme permet de convertir une image, en couleurs ou en noir et blanc, en un arbre quaternaire la représentant. Il est possible de compresser les arbres pour gagner en espace. Le programme est capable d'écrire et de lire des arbres, compressés ou non, et de les afficher selon différentes représentations, dont certaines permettant de se rendre compte des parties grossières et des parties plus détaillées des arbres.

Table des matières

1 Utilisation

1.1 Compilation

Ce programme repose sur des librairies standard et sur la librairie `MLV`. Afin de compiler le programme, il faut avoir téléchargé cette dernière au préalable à l'URL suivante :

`http://www-igm.univ-mlv.fr/~boussica/mlv/`

Une fois la librairie téléchargée et installée, il suffit de compiler le programme avec la commande `make`.

1.2 Exécution

Le programme s'exécute exclusivement en interface graphique. À l'ouverture, la fenêtre affiche une image vide, une case permettant de saisir le chemin vers une image à ouvrir, ainsi qu'un bouton pour fermer le programme.

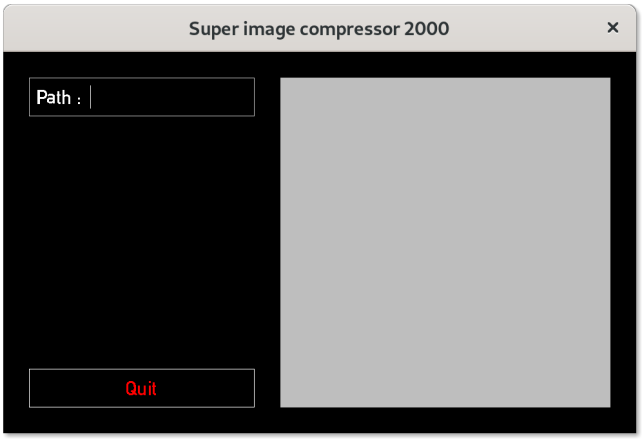
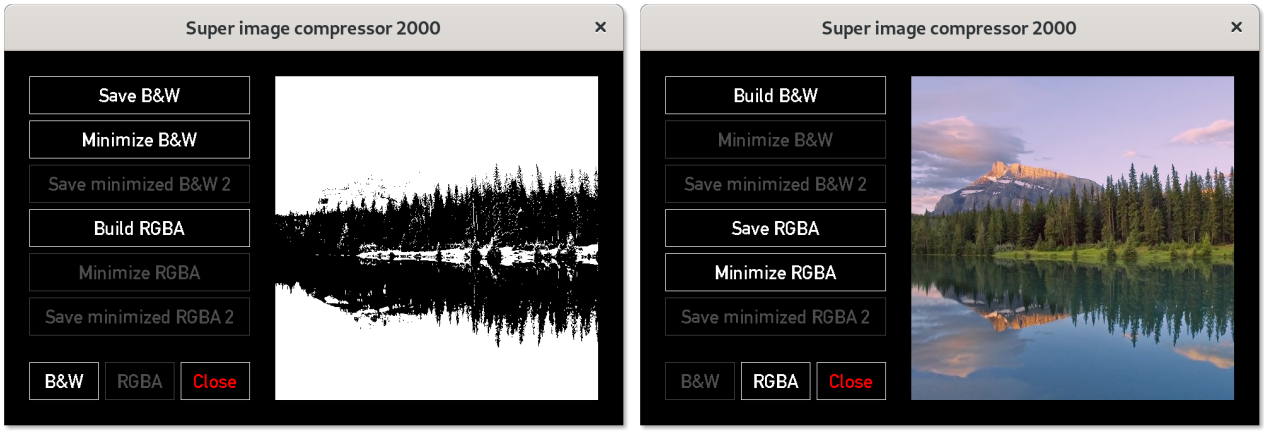


FIGURE 1 – Programme à l'ouverture

Une fois une image chargée, selon son type et ses possibilités, différents boutons sont proposés. Une image au sens propre du terme (formats `.png`, `.jpg`, ...) permet de générer un arbre noir et blanc et un arbre en couleur. Les deux types d'arbres sont ensuite compressibles. Cependant, si un fichier contenant un arbre est ouvert, les possibilités sont plus minces. Par exemple il n'est pas possible de générer un arbre d'un autre type que celui ouvert. Il n'est pas non plus possible d'enregistrer un arbre non-compressé si l'arbre ouvert est en réalité un graphe compressé.



(a) Génération d'un arbre binaire

(b) Génération d'un arbre RGBA

FIGURE 2 – Programme après la génération d'un arbre à partir d'une image

Deux boutons, situés au bas de la fenêtre, permettent de basculer l'affichage entre l'arbre binaire et l'arbre RGBA. Un clic droit sur un des ces boutons permet de faire ressortir la compression de l'image sur l'affichage. Si une image est ouverte, un clic sur la zone d'affichage permet d'afficher cette dernière. Enfin, un message s'affiche sur la zone d'affichage lorsque le programme travaille intensément pour faire patienter l'utilisateur.

2 Fonctionnement

2.1 Construction des arbres

Les arbres sont construits selon une méthode récursive sans perte. L'algorithme est lancé sur l'intégralité de l'image et tant que l'erreur sur la couleur moyenne de la zone est strictement positive, l'algorithme se relance récursivement sur 4 sous-zones. L'arbre final peut contenir de la compression si une zone de l'image est strictement de la même couleur, il n'est cependant pas possible de perdre de l'information avec cette méthode.

En ce qui concerne la construction des arbres binaires (en noir et blanc), un seuillage de l'image est effectué pour déterminer si une couleur est plutôt claire ou plutôt sombre.

2.2 Lecture et écriture

2.2.1 Arbres non-compressés

Les arbres non-compressés sont écrits en bit-à-bit selon un encodage simple. Pour les arbres binaires, un nœud interne est représenté par un 0, suivi des représentations de ses quatre fils. Une feuille, quant à elle, est représentée par un 1 suivi de sa valeur (0 ou 1). Le tout forme un flux ininterrompu de bits que l'on peut aisément découper. En commençant par le début, si le bit est à 1, alors on lance récursivement la reconnaissance sur le prochain bit, sinon on recrée la feuille correspondante.

Les arbres RGBA sont semblables aux arbres binaires à la différence que les feuilles sont encodées avec quatre octets représentant les quatre composantes de la couleur : rouge, vert, bleu, alpha.

2.2.2 Graphes compressés

Les graphes compressés sont très différents des arbres non-compressés, et en particulier sur le fait qu'un arbre peut être l'enfant de plusieurs nœuds différents. Pour résoudre ce problème, on donne à chaque nœud un identifiant unique qui sert à le référencer aux endroits nécessaires. Ainsi, un nœud interne est représenté par son identifiant, suivi des identifiants de ses quatre fils.

Concernant les feuilles, si le graphe est un graphe binaire, la feuille est représentée par son identifiant unique suivi de sa valeur (0 ou 1). Cependant, pour les feuilles de graphes RGBA, le format est légèrement adapté pour ne pas confondre une feuille avec un nœud. Ainsi, les feuilles des graphes RGBA sont représentées par leur identifiant unique, suivi de la lettre **f**, elle-même suivie des quatre composantes de la couleur.

2.2.3 Graphes compressés en bit-à-bit

Afin d'aller plus loin, et d'économiser de l'espace pour le stockage des graphes compressés, nous avons mis au point un encodage bit-à-bit des graphes compressés. Tout commence par la mesure de la taille de l'arbre pour déterminer le plus grand identifiant unique. Une fois cette information déterminée, un octet en début de fichier est réservé pour représenter le nombre de bits n représentant un identifiant unique.

Les nœuds sont encodés d'une façon qui mélange les arbres non-compressés et les graphes compressés. Le nœud est représenté par son identifiant unique, codé sur n bits, suivi d'un bit à 0 s'il s'agit d'un nœud interne, ou à 1 s'il s'agit d'une feuille. Pour les nœuds internes, la suite consiste en la représentation des identifiants uniques des quatre fils, chacun sur n bits. Pour les feuilles, la suite consiste en la représentation des quatre composantes de la couleur, chacune sur un octet, ou de la valeur de la feuille sur un bit pour les graphes binaires.

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

3 0
4 1
```

(a) Encodage d'un graphe compressé (23 octets)

```
00000011
000 0 001 001 001 001
001 0 010 010 010 010
010 0 011 100 011 100
011 1 0
100 1 1
```

(b) Encodage d'un graphe compressé en bit-à-bit (9 octets)

FIGURE 3 – Comparaison d'encodage du même graphe minimisé

Cet encodage bit-à-bit est, en moyenne, trois fois plus léger que l'encodage "naïf".

2.3 Méthodes de compression

Le module qui sert à la minimisation des `QuadTree` se situe dans les fichiers `compressor.c` et `compressor.h`. Il est, comme son nom l'indique, consacré à réduire au maximum la taille prise par un `QuadTree` pour représenter une image, et ce au prix d'une perte (optionnelle).

Ainsi, la fonction globale utilisée est `void minimizeQuadTree(QuadTree *tree, float distErr)` (en deux variantes, une pour les arbres RGBA, et une autre pour les arbres en noir et blanc (aka bin)). Cette fonction reçoit donc un arbre à minimiser avec une distance d'erreur allouée (qui peut être égale à 0 dans le cas où l'on ne veut pas de perte de qualité dans l'image), et enchaîne différents algorithmes de minimisation qui résulteront alors en un graphe orienté.

2.3.1 Calcul de distance entre deux arbres

Pour certains des algorithmes qui seront expliqués par la suite, il est nécessaire de pouvoir calculer la distance entre deux arbres. Ainsi, une fonction récursive que nous appellerons ici `dist` en découle.

Dans le cas où :

— les deux arbres sont des feuilles, alors nous calculons leur distance en terme de couleur :

$$dist(T1, T2) = distColor(T1, T2) \tag{1}$$

$$distColor(p1, p2) = \sqrt{(r1 - r2)^2 + (g1 - g2)^2 + (b1 - b2)^2 + (a1 - a2)^2} \tag{2}$$

— seulement T1 est une feuille, alors il faut faire la moyenne des distances entre T1 et chaque fils de T2 :

$$dist(T1, T2) = \sum_{i=1}^{4} \frac{dist(T1, \text{fils numéro i de } T2)}{4} \tag{3}$$

— seulement T2 est une feuille, alors il faut faire la moyenne des distances entre chaque fils de T1, et T2 :

$$dist(T1, T2) = \sum_{i=1}^{4} \frac{dist(\text{fils numéro i de } T1), T2)}{4} \tag{4}$$

— aucun des deux arbres n'est une feuille, alors il faut faire la distance entre chacun des fils de T1 et T2 :

$$dist(T1, T2) = \sum_{i=1}^{4} \frac{dist(\text{fils numéro i de } T1), \text{fils numéro i de } T2)}{4} \tag{5}$$

Cette distance sera alors comparée à la distance d'erreur donnée à plusieurs reprise.

2.3.2 Réduction naïve

La fonction de réduction naïve ne produit pas de graphe, son rôle principal est de préparer le travail des autres algorithmes pour les accélérer en effectuant des moyennes. Pour cela, nous lui donnons donc un arbre à réduire, la distance d'erreur, ainsi qu'un buffer qui servira de poubelle à libérer à la fin du programme (C'est un tableau qui connaît sa taille et possède des fonctions d'allocations et de libérations de mémoire). Le but est de parcourir récursivement les nœuds de l'arbre de haut en bas. À chaque nœud rencontré, nous calculons alors la couleur moyenne de ce nœud en tant qu'arbre avec l'algorithme récursif suivant :

— Si le noeud est une feuille, alors sa couleur est retournée.
— Si le noeud n'est pas une feuille, alors la moyenne des couleurs de ses fils est retournée.

Une fois cette moyenne récupérée, nous comparons la distance entre le nœud courant et la feuille issue de la couleur moyenne du nœud, avec la distance d'erreur. Si elle est inférieure, alors nous réduisons le nœud en le donnant à la poubelle et en le remplaçant par la feuille issue de la couleur moyenne.

Dans le cas d'un arbre en noir et blanc, puisque l'on ne veut pas de minimisation à perte (au risque de perte complètement la clarté de l'image), cet algorithme n'est pas lancé, car si la distance d'erreur est de 0 alors il n'a aucun effet.

2.3.3 Minimisation des feuilles dupliquées

Il n'est pas rare qu'une image contienne plusieurs pixels de la même couleur, et ceux d'autant plus après la réduction naïve (dans le cas d'une image en noir et blanc, il n'y a que deux couleurs...). Dans un arbre, cela fait donc potentiellement un très grand nombre de feuilles étant structurellement identiques, qui prennent beaucoup d'espace mémoire.

Cet algorithme a donc pour but de relier dans la mémoire chaque feuille de la même couleur à la même adresse mémoire. Il prend donc en entrée l'arbre à minimiser, et deux buffers : une poubelle et un cache (pour garder en mémoire les feuilles rencontrées de manière structurellement unique). Le parcours de l'arbre se fait récursivement de haut en bas. À chaque feuille rencontrée, on regarde si elle est dans le cache, si c'est le cas on la place dans la poubelle et on remplace son adresse par celle en cache.

2.3.4 Recherche de sous-arbres

L'étape finale est la recherche de sous-arbres similaires. En suivant la même idée que les minimisations des feuilles dupliquées, nous pouvons imaginer qu'une image a plusieurs zones qui sont similaires, résultant en plusieurs sous-arbres structurellement similaires. Cette fois-ci, le parcours de l'arbre se fait récursivement des feuilles jusqu'à la racine, par conséquent la fonction commence par un appel récursif sur tout les fils du nœud courant. L'arbre à minimiser est donné en entrée, ainsi qu'une distance d'erreur, un cache et une poubelle.

À chaque nœud rencontré, nous parcourons le cache et faisons des comparaisons :
— Si la hauteur de l'arbre i dans le cache est supérieure à celle de l'arbre courant, alors il n'est pas la peine de faire des tests supplémentaires, et le nœud courant est ajouté au cache.
— Sinon, si la distance entre l'arbre i dans le cache et l'arbre courant est supérieure à la distance d'erreur, alors on considère que l'arbre courant n'est pas encore en cache et qu'il faut l'y ajouter.
— Sinon, cela signifie que l'arbre i du cache est similaire à l'arbre courant, par conséquent nous pouvons le mettre dans la poubelle et remplacer son adresse mémoire par celle mise en cache.

Une fois les différentes phases de la minimisation accomplies, il ne reste donc plus qu'a libérer les différentes poubelles et leur contenu.

3 Description du programme

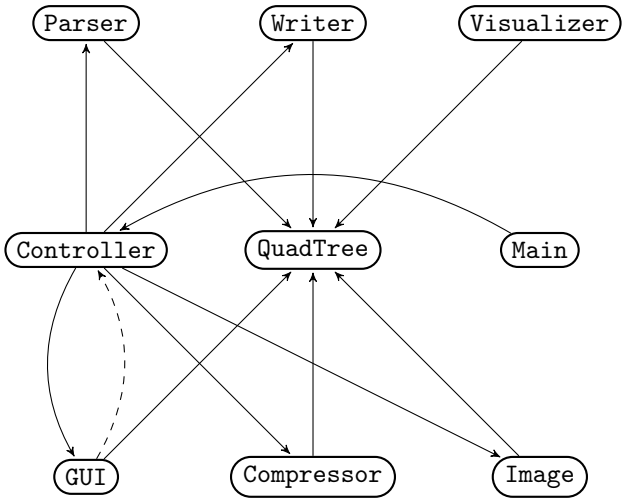


FIGURE 4 – Graphe des dépendances

3.1 Module `QuadTree`

Ce module se charge de déclarer les structures représentant les arbres quaternaires ainsi que plusieurs fonctions permettant d'interagir avec ces derniers. En plus des arbres, ce module déclare une structure qui représente un buffer pour stocker des nœuds.

Le module déclare les structures suivantes :

```
typedef struct s_quadtree_rgba
{
    struct s_quadtree_rgba *northWest;
    struct s_quadtree_rgba *northEast;
    struct s_quadtree_rgba *southWest;
    struct s_quadtree_rgba *southEast;
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
} * QuadTreeRGBA;

typedef struct s_quadtree_bin
{
    struct s_quadtree_bin *northWest;
    struct s_quadtree_bin *northEast;
    struct s_quadtree_bin *southWest;
    struct s_quadtree_bin *southEast;
    unsigned char b;
} * QuadTreeBin;

typedef struct s_quadtree_rgba_buffer
{
    QuadTreeRGBA *buffer;
    size_t bufferSize;
} * QuadTreeRGBABuffer;

typedef struct s_quadtree_bin_buffer
{
    QuadTreeBin *buffer;
    size_t bufferSize;
} * QuadTreeBinBuffer;
```

3.2 Module `Visualizer`

Ce module a permis, lors de la phase de développement, de représenter les arbres graphiquement au format `pdf` grâce à l'utilitaire `graphviz`. Il a beaucoup servi lors des tests de minimisation sur de petits arbres pour se rendre compte de son efficacité.

3.3 Module `Writer`

Ce module se charge d'écrire les arbres non-compressés ainsi que les graphes compressés dans des fichiers. Il définit plusieurs méthodes utiles pour l'écriture bit-à-bit.

3.4 Module `Parser`

Ce module se charge de lire les arbres non-compressés ainsi que les graphes compressés dans des fichiers. Il définit plusieurs méthodes utiles pour la lecture bit-à-bit.

3.5 Module `Gui`

Ce module se charge de dessiner tous les éléments graphiques sur la fenêtre. Il est étroitement lié au module `Controller` pour l'interaction de l'utilisateur avec les différents éléments graphiques.

3.6 Module `Controller`

Ce module permet d'interagir avec l'interface graphique au moyen de méthodes permettant d'effectuer des actions simples.

3.7 Module `Compressor`

Ce module permet de manipuler les arbres afin de les compresser en les transformant en graphes. C'est de loin le module le plus complexe et le plus élaboré.

3.8 Module `Image`

Ce module se charge de la construction d'arbres à partir d'image. Il comporte toutes les fonctions nécessaires telles que le calcul de distance entre deux couleurs ou le calcule de la couleur moyenne d'une zone d'une image.

4 Difficultés rencontrées

L'avancée du développement du module `Compressor` s'est faite à tâtons et avec de nombreuses refontes. En effet, bien que les premières fonctions nécessaires à l'ensemble du module telles que le calcul des distances se sont dessinées très rapidement, la conception des fonctions de minimisation elles-même ont due être réfléchies longuement en terme de conception.

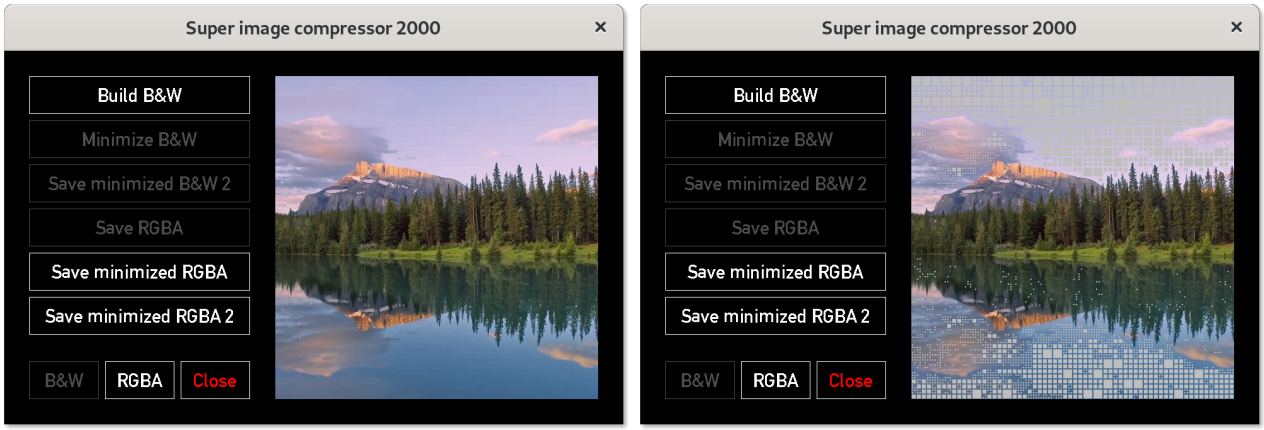
Par exemple, le premier mode de minimisation à été la suppression des feuilles dupliquées de l'arbre. Les premières versions (pour lesquelles peu de réflexion a été consacrée) étaient très redondantes dans les tests. La version noire et blanc était identique à la version RGBA (sans raison valide). Assez rapidement, nous nous sommes donc rendu compte de l'ampleur de ce module et du temps à y consacrer.

Au début, la difficulté a été de gérer un système de cache afin de garder en mémoire les feuilles (et plus tard les nœuds) parcourus. La manipulation de celui-ci était souvent fastidieuse et alourdissait le code en le rendant difficilement debuggable car on utilisais un tableau dynamique directement. La solution d'une structure dédiée à cela, ainsi que des fonctions utilitaires d'allocation sont venues bien après.

Le deuxième mode de minimisation a été la réduction par niveaux, c'est-à-dire le remplacement de sous arbres dans la mémoire qui ont le même père et lorsqu'ils sont similaires (en terme de distance). Cette méthode a été remplacée par la suite par la plus importante, lourde et efficace méthode de recherche des sous-arbres, qui, dans le principe, a un fonctionnement similaire à la minimisation des feuilles. Cependant, il nous a fallu beaucoup de temps pour nous rendre compte que des erreurs de segmentation étaient dues, au fait que l'on ajoutait au cache des onuds déjà visités (que l'on visite une nouvelle fois à cause de la réduction), rendant en plus le parcours du cache beaucoup plus long qu'il ne devrait l'être.

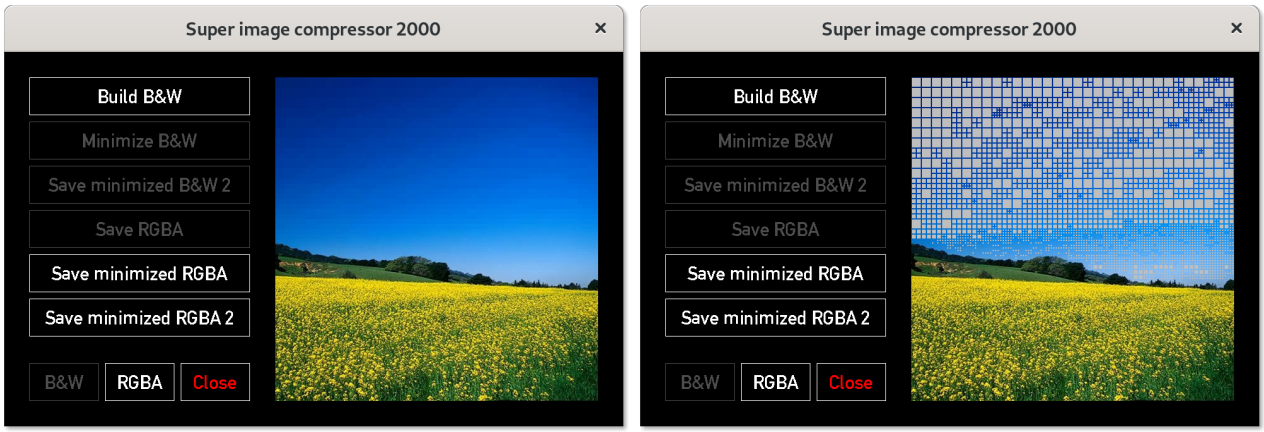
Après avoir corrigé ce problème, la minimisation d'une image nous semblait trop longue, s'en est suivit donc des tentations de réduire ce temps au maximum, pendant longtemps, en vain. La solution a alors été de voir le problème sous un autre angle : réduire naïvement l'arbre en premier lieu avec la distance d'erreur. Ainsi, cela réduit considérablement le nombre de nœuds à parcourir par la suite. Est ensuite venue l'idée de comparer les hauteurs d'arbres afin d'éviter des calculs de distance inutiles.

5 Démonstration



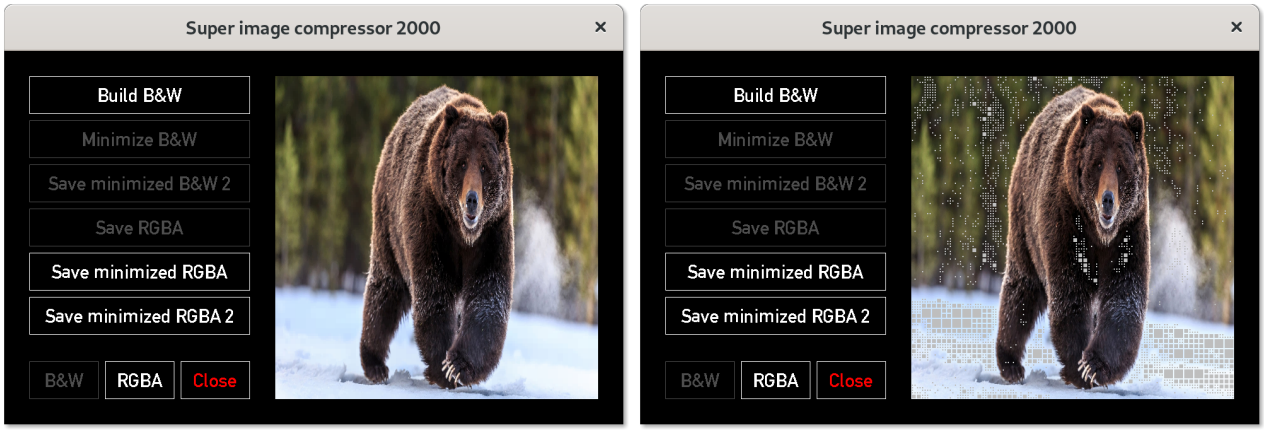
(a) Image compressée (b) Mise en évidence des zones compressées

FIGURE 5 – Compression d'une image d'un lac



(a) Image compressée (b) Mise en évidence des zones compressées

FIGURE 6 – Compression d'une image d'un champs



(a) Image compressée (b) Mise en évidence des zones compressées

FIGURE 7 – Compression d'une image d'un ours[1]

1. Ici, on ne parle pas de la déformation de l'image.