Compte rendu de Projet
IF112 - Projet d'Informatique

Filière Électronique

Semestre 6

Besson Thomas

Encadrant : R. Giraud

Table des matières

Sujet du Projet d'Informatique

1 Introduction

Ce projet a pour objectif d'implémenter en langage C un système de compression d'images au format PPM (binaire `P6`) à l'aide de l'algorithme de Huffman.

La compression de Huffman est notamment utilisée dans l'algorithme de compression d'images JPEG. A noter que dans le format JPEG, la compression se fait dans un espace fréquentiel permettant d'avoir un gain en mémoire important au détriment d'une certaine qualité visuelle (compression avec perte).

Les images sont compressées en binaire avec une en-tête personnalisé (P7) et peuvent être reconstruites sans perte.

2 Programme global

Dans un premier temps, j'ai construit une version naïve du compresseur, puis une version optimisée exploitant la redondance des intensités.

Structure du projet

Le projet sera structuré avec un seul fichier qui reprend des fonctions vues dans les TP précédents ainsi que le contenu du projet et d'un Makefile :

`projet.c` : Ce fichier contient les fonctions du programme.

`Makefile` : Automatisation de la compilation du projet.

Choix d'implémentation

Le programme est divisé en plusieurs modules activables définis dans le Makefile.
Cela permet de tester séparément :

- la compression naïve
- l'étude statistique
- la compression Huffman
- la décompression
- la génération d'un histogramme visuel

J'ai utilisé des structures simples pour représenter les images et les arbres de Huffman. Le Makefile a été rendu dynamique grâce à une variable `INPUT` permettant de choisir le fichier image sans modifier le code.

Mon système d'exploitation étant Windows, le Makefile diffère légèrement pour le nettoyage des fichiers. Par exemple, la commande `del /f /q naif.exe 2>nul || exit 0` permet de supprimer silencieusement le fichier `naif.exe`, même s'il n'existe pas, sans générer d'erreur ni interrompre l'exécution de make.

3 Mise en place

3.1 Structure du code

Pour commencer, reprenons les structures et fonctions d'enregistrement des images au format PPM écrites des précédents TP. Nous utiliserons donc : `struct color`, `struct picture`.

Chaque pixel est représenté par la structure `color` contenant trois composantes (red, green, blue). L'image est une structure `picture` composée de sa taille (width, height) et d'un tableau dynamique de pixels.

Ces programmes permettent de sauvegarder et de charger une image au format PPM : `picture new_pic`, `void save_pic`, `picture load_pic`.

3.2 Création d'une image de test synthétique

Pour la suite du projet, une image de 256×256 pixels a été créée. Elle est composée de trois bandes verticales rouge, verte et bleue, puis elle est sauvegardée (image.ppm) dans le dossier images.

```c
void create_image()
{
   int width = 256, height = 256;
   picture pic;
   pic.width = width;
   pic.height = height;
   pic.pixels = malloc(sizeof(color)*width*height);

   color r = {255, 0, 0};
   color g = {0, 255, 0};
   color b = {0, 0, 255};

   for (int y = 0; y < height; y++)
   {
      for (int x = 0; x < width; x++)
      {
         if (x < width / 3)
            pic.pixels[y * width + x] = r;
         else if (x < 2* width / 3)
            pic.pixels[y * width + x] = g;
         else
            pic.pixels[y * width + x] = b;
      }
   }
   save_pic(pic, "images/image.ppm");
   free(pic.pixels);
}
```

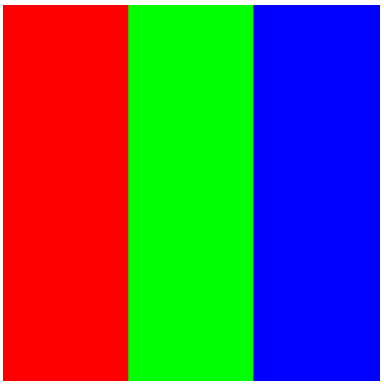


FIGURE 1 – create_image et image.ppm

3.3 Écriture du fichier et compression naïve

```c
typedef struct HuffmanTable
{
   char huffman[256][256];
} HuffmanTable;
```

La structure `HuffmanTable` encapsule une table de codage contenant les codes binaires associés à chaque intensité (0 à 255).

```c
void naive_codes(HuffmanTable* table)
{
   for (int i = 0; i < MAX_INTENSITY; ++i)
   {
      for (int bit= 7; bit>= 0; --bit)
      {
         table->huffman[i][7-bit]=((i>>bit)&1)?'1':'0';
      }
      table->huffman[i][8]= '\0';
   }
}
```

La fonction `naive_codes` génère, pour chaque intensité de pixel (0 à 255), une représentation binaire sur 8 bits sous forme de chaîne de caractères. Cette méthode, dite "naïve", attribue à chaque valeur son code binaire brut, sans compression ni optimisation. Elle permet de comparer plus facilement l'efficacité du codage Huffman par la suite.

On aura donc :

$$\begin{aligned} \text{huffman}[0] &= \text{"00000000"} \\ \text{huffman}[1] &= \text{"00000001"} \\ \text{huffman}[2] &= \text{"00000010"} \\ &\dots \\ \text{huffman}[255] &= \text{"11111111"} \end{aligned}$$

Compression et taille de fichier

La fonction `compress_img_naif` effectue une compression naïve d'une image couleur au format PPM. Chaque composante (r, g, b) des pixels est remplacée par son code binaire de 8 bits préalablement défini dans la table Huffman. Le fichier résultant est écrit dans un format personnalisé HPPM contenant les dimensions de l'image, la table de codage, puis les données compressées.

La fonction `file_size` est utile pour mesurer précisément les effets de la compression, elle permet de donner le taux de compression ainsi que le gain de taille en octets après un simple rapport dans le `main()`.

```c
long file_size(const char* filename) {
   struct stat st;
   if (stat(filename,&st) != 0)
   {
      perror("Impossible de lire la taille du fichier");
      return -1;
   }
   return st.st_size;
}
```

Taille fichier PPM original : 196623 octets
Taille fichier compresse : 262159 octets
Taux de compression : 133.33%
Gain de taille : -33.33%

Je précise ici que Windows affiche une valeur arrondie au Ko supérieur 196 623 octets → soit environ 193 Ko car pour Windows 1Ko = 1024 octets.

On remarque que la taille du fichier compressé avec le codage naïf (`image_naif.hppm`, 257 Ko) est supérieure à celle de l'image d'origine (`image.ppm`, 193 Ko) soit un gain négatif de -33,33%. Cela s'explique par l'ajout de la table de codage dans le fichier de sortie, ainsi que par le fait que les codes naïfs utilisent toujours 8 bits, sans réelle compression.

Taille minimale ?

On peut se demander quelle serait la taille minimale obtenue par cette compression, pour cela créons une image de 1x1 pixel.

```c
void create_image_1x1()
{
    int width = 1, height = 1;
    picture pic;
    pic.width = width;
    pic.height = height;
    pic.pixels = malloc(sizeof(color) * width * height);

    color red = {255, 0, 0};
    pic.pixels[0] = red;

    save_pic(pic, "images/image_1x1.ppm");
    free(pic.pixels);
}
```

■ (échelle x10 pour l'affichage ici), et testons la compression de celle-ci :

Taille fichier PPM original : 14 octets
Taille fichier compresse : 65550 octets
Taux de compression : 468214.29%
Gain de taille : -468114.29%

La taille minimale d'un fichier HPPM généré avec ce format est d'environ 65 Ko, même pour une image de 1×1 pixel. En effet, la compression vient rajouter une table de $256 * 256 \approx 65536$ octets à notre fichier HPPM. Cela montre que ce format de compression n'est pas optimisé.

Decompression

La fonction `decompress_img_naif` lit un fichier HPPM généré par compression naïve. Elle commence par extraire l'en-tête contenant les dimensions de l'image, puis recharge la table de codage stockée dans le fichier. Ensuite, elle lit les données des pixels compressés (sous forme binaire codée sur 8 bits) et les reconvertit directement en composantes (r, g, b), sans effectuer de décodage complexe.

Cette approche fonctionne parce que les codes utilisés sont naïfs, c'est-à-dire qu'ils conservent exactement la même valeur que les intensités d'origine. Ainsi, il n'y a aucune altération de l'image lors de la compression puis décompression : chaque pixel est reconstitué fidèlement.

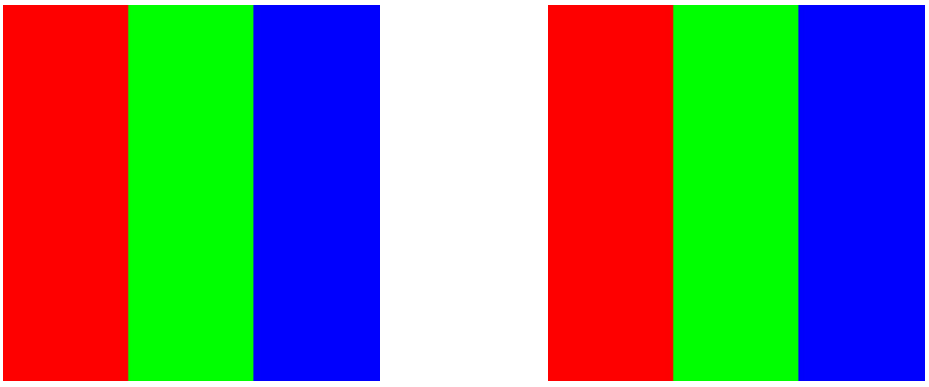


FIGURE 2 – image.ppm et image_naif_reconstruit.ppm

4 Calculs et visuel

4.1 Histogramme

La fonction `histogram_img` calcule un histogramme normalisé des intensités de couleur $i \in [0, 255]$ pour une image. Pour chaque pixel, elle incrémente les fréquences des composantes rouge, verte et bleue dans un tableau de 256 cases. La constante `MAX_INTENSITY` est définie ici pour simplifier nos codes et la compréhension par la suite :

```c
#define MAX_INTENSITY 256
```

Ensuite, elle divise chaque valeur par le nombre total de composantes (3 par pixel) pour obtenir une distribution de probabilité. Le tableau retourné représente donc la fréquence relative d'apparition de chaque intensité dans l'image.

Plus un symbole est présent dans la source, plus il est intéressant de lui donner un code court, et inversement. Pour évaluer cette redondance, on doit donc calculer le nombre d'occurrences ou l'histogramme des intensités.

```c
float* histogram_img(color* pixels, int total_pixels)
{
   float* hist = calloc(MAX_INTENSITY, sizeof(float));
   for (int i = 0; i < total_pixels; ++i)
   {
      hist[pixels[i].r]++;
      hist[pixels[i].g]++;
      hist[pixels[i].b]++;
   }
   float total_values = 3.0f * total_pixels;
   for (int i = 0; i < MAX_INTENSITY; ++i)
   {
      hist[i] /= total_values;
   }
   return hist;
}
```

on retrouve donc pour `image.ppm` :

$$hist[0] \approx 0.6667$$
$$hist[255] \approx 0.3333$$

Ce résultat est cohérent, pour $hist[0]$ on a 2 composantes sur 3 qui sont nulles pour chaque pixel ($\frac{2}{3} \approx 0.6667$) et $hist[255]$ une seule composante par pixel est à 255 ($\frac{1}{3} \approx 0.3333$).

Ainsi on peut afficher graphiquement notre histogramme :

```c
void histogram_ppm(float* hist, const char* filename)
{
   int width = 256, height = 256;
   picture img;
   img.width= width;
   img.height= height;
   img.pixels= malloc(sizeof(color) *width* height);

   color white = {255,255,255};
   color black = {0,0,0};

   for (int i = 0; i < width * height; i++)
   {
      img.pixels[i] = white;
   }
   for (int x =0; x< 256; x++)
   {
      int h = (int)(hist[x] * height);
      for (int y = height- 1; y >= height- h; y--)
      {
         img.pixels[y *width + x] = black;
      }
   }
   save_pic(img, filename);
   free(img.pixels);
}
```

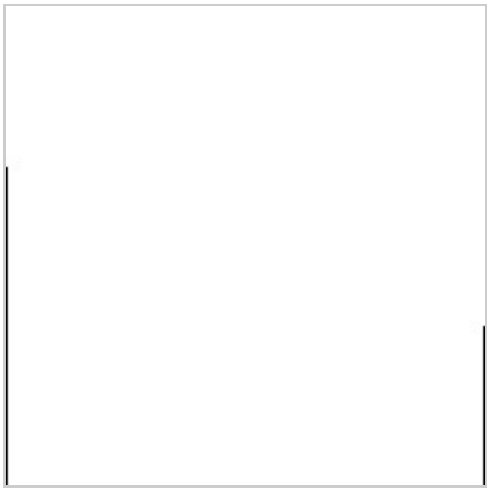


FIGURE 3 – save_histogram_ppm et image_histogram.ppm

4.2 Entropie

En théorie du codage, cette mesure reflète la variabilité de l'information contenue dans une source composée de plusieurs symboles. Elle se calcule selon la formule suivante :

$$E = -\sum_{i=0}^{255} p_i \cdot \log_2(p_i)$$

où $p_i \in [0, 1]$ représente la probabilité d'apparition de l'intensité i, c'est-à-dire le rapport entre le nombre d'occurrences de cette intensité et le nombre total de pixels.

```c
float entropy(float* hist) {
    float E = 0.0f;
    for (int i = 0; i < MAX_INTENSITY; ++i) {
        if (hist[i] > 0.0f) {
            E -= hist[i] * log2f(hist[i]);
        }
    }
    return E;
}
```

Pour `image.ppm` on retrouve $E = 0.9183$ bits. Ce qui est cohérent, car avec $p_0 = \frac{2}{3}$ et $p_{255} = \frac{1}{3}$:

$$E = -(\frac{2}{3} * \log_2(\frac{2}{3}) + \frac{1}{3} * \log_2(\frac{1}{3})) \approx 0.9183$$

Cela reflète un faible degré de variabilité dans la répartition des intensités : certaines valeurs sont beaucoup plus fréquentes que d'autres, donc la source est peu imprévisible.

Si l'image était parfaitement aléatoire (toutes intensités équiprobables), l'entropie serait 8 bits. Ici, comme l'entropie est bien inférieure à 8, il existe un fort potentiel de compression.

5 Codage de Huffamn

Dans cette section, nous présentons le processus d'implémentation du codage de Huffman dans le but de compresser efficacement un fichier image. La démarche consiste d'abord à construire l'arbre de Huffman à partir des fréquences d'apparition des intensités de pixels, afin d'associer à chacune un code binaire optimal. Ces codes sont ensuite utilisés pour encoder l'image, en remplaçant chaque intensité par sa représentation binaire correspondante dans le fichier de sortie.

5.1 Construction de l'arbre de Huffman

Pour compresser efficacement une image à l'aide du codage de Huffman, il est nécessaire de construire un arbre binaire permettant d'assigner à chaque intensité un code binaire unique, plus court pour les valeurs fréquentes, et plus long pour les valeurs rares. L'implémentation proposée repose sur deux étapes principales : la création des nœuds de base et la construction itérative de l'arbre.

L'algorithme fourni dans le sujet :

```
void build_huffman_tree(float* histogram, Node** root) {
Node* nodes[256]; //
// Creation des n_noeuds pour chaque intensite presente (value, freq, etc.)
// Construction de l arbre de Huffman
while(n_noeuds > 1) {
// Recherche des 2 plus petites frequences
// Creation d un nouveau noeud fusionnant les deux plus petits
// Remplacement des deux anciens noeuds par le nouveau
nodes[min1] = new_node;
nodes[min2] = nodes[n_noeuds-1]; // Deplacer le dernier element
n_noeuds--;
}
//return le dernier noeud
}
```

La fonction commence par analyser l'histogramme des intensités pour identifier les intensités présentes dans l'image :

```
for (int i = 0; i < MAX_INTENSITY; i++)
{
   if (histogram[i]> 0.0f)
   {
      int freq = (int)(histogram[i] * 1000000);
      nodes[n_nodes++] = create_node((unsigned char)i, freq, NULL, NULL);
   }
}
```

Seules les intensités ayant une fréquence non nulle sont converties en feuilles (un nœud sans enfants) de l'arbre.
Les probabilités sont multipliées par 1 000 000 pour éviter d'utiliser des flottants dans le tri et les comparaisons. Cela reste proportionnel, ce qui suffit pour construire un arbre correct.
Chaque feuille contient la valeur de l'intensité et sa fréquence approximative.

La phase suivante consiste à construire l'arbre en fusionnant récursivement les deux nœuds les moins fréquents :

```
while (n_nodes > 1)
{
   // recherche des deux plus petites frequences
   // fusion des deux noeuds
   Node* merged = create_node(0, freq1 + freq2, left, right);
}
```

À chaque itération, les deux nœuds avec les fréquences les plus basses sont sélectionnés. Ces deux nœuds sont fusionnés en un nouveau nœud interne, dont la fréquence est la somme des deux, il devient un parent et est inséré à la place de l'un des anciens nœuds. L'autre ancien nœud est remplacé par le dernier du tableau pour maintenir une structure compacte.

Une fois qu'il ne reste qu'un seul nœud, celui-ci devient la racine de l'arbre, qui contient toute la structure nécessaire pour encoder l'image.

```
*root = nodes[0];
```

5.2 Construction de la table de codage

La fonction `generate_huffman_codes` parcourt récursivement l'arbre de Huffman pour générer les codes binaires associés à chaque intensité (0 à 255) et les stocke dans `HuffmanTable`.
Chaque chemin dans l'arbre (gauche → 0, droite → 1) devient une chaîne de caractères représentant le code binaire d'une intensité donnée.

```
void generate_huffman_codes(Node* root, HuffmanTable* table, char* buffer, int depth)
{
   if (!root) return;
   if (root->left ==NULL && root->right== NULL)
   {
      buffer[depth] ='\0';
      strcpy(table->huffman[root->value], buffer);
      return;
   }
   buffer[depth]='0';
   generate_huffman_codes(root->left, table, buffer, depth+ 1);

   buffer[depth]='1';
   generate_huffman_codes(root->right, table, buffer, depth+ 1);
}
```

Cette ligne :

```
strcpy(table->huffman[root->value], buffer);
```

copie la chaîne buffer dans la case correspondant à la valeur de l'intensité (0 à 255).

Le choix d'inclure la bibliothèque `<string.h>`, qui fournit la fonction `strcpy`, permet de manipuler facilement les chaînes de caractères. Cela simplifie grandement la construction et la sauvegarde des codes binaires dans la table de Huffman. Cette fonction construit donc tous les chemins possibles dans l'arbre, et ainsi tous les codes.

Test sur image.ppm

Nous testons à présent cette fonction sur notre image d'entrée afin de générer les codes Huffman associés à chaque intensité. Le résultat obtenu est le suivant :

Intensité 0 : 1
Intensité 255 : 0

Ce qui nous confirme que l'intensité 0 est la plus fréquente avec donc un code plus court, et que l'intensité 255 a un code plus long car elle est moins fréquente.

Calcul de la longueur moyenne des code

La fonction `moy_length` permet de calculer la longueur moyenne pondérée des codes Huffman générés, en prenant en compte les fréquences d'apparition des intensités dans l'image.

```
float moy_length(float* histogram, HuffmanTable* table)
{
   float total = 0.0f;
   for (int i = 0; i< MAX_INTENSITY; ++i)
   {
      if (histogram[i]> 0.0f)
      {
         int length=strlen(table->huffman[i]);
         total+= histogram[i]* length;
      }
   }
   return total;
}
```

Ce qui nous donne :

Longueur moyenne des codes : 1.0000 bits

Cette valeur est très proche de l'entropie (pour rappel $E = 0.9183$) Ce qui signifie que mon arbre de Huffman est optimal pour cette distribution.

De plus, chaque composante (r, g, b) peut être encodée en 1 bit de moyenne dans notre `image.ppm`.

6 Compression de Huffman

La fonction `compress_img_huffman` permet de sauvegarder une image compressée au format HPPM à l'aide d'un codage de Huffman appliqué pixel par pixel.

Ce mécanisme permet de compresser les données en tenant compte de la fréquence des intensités, tout en produisant un format lisible et déchiffrable.

```
void compress_img_huffman(picture pic, const char* filename,
    HuffmanTable* table)
{
   FILE* file = fopen(filename, "wb");
   if (!file)
   {
      perror("Erreur ouverture fichier .hppm");
      exit(1);
   }
   fprintf(file, "P7 %d %d 255 ", pic.width, pic.height);
   for (int i = 0; i < MAX_INTENSITY; ++i)
   {
      fwrite(table->huffman[i], sizeof(char), MAX_INTENSITY,
          file);
   }
   unsigned char buffer = 0;
   int bit_count = 0;

   for (int i = 0; i< pic.width * pic.height; ++i)
   {
      unsigned char values[3] =
      {
         pic.pixels[i].r,
         pic.pixels[i].g,
         pic.pixels[i].b
      };
      for (int c = 0; c < 3; ++c)
      {
         char* code = table->huffman[values[c]];

         for (int j = 0; code[j] != '\0'; ++j)
         {
            buffer <<= 1;
            if (code[j] == '1')
            {
               buffer |= 1;
            }
            bit_count++;
            if (bit_count== 8)
            {
               fputc(buffer, file);
               bit_count= 0;
               buffer= 0;
            }
         }
      }
   }
    if (bit_count > 0)
   {
      buffer <<= (8 - bit_count);
      fputc(buffer, file);
   }
   fclose(file);
}
```

Cette fonction `compress_img_huffman` a un fonctionnement similaire à la compression naïve, pour l'ouverture du fichier et l'écriture de l'en-tête.

Écriture de la table Huffman

```
for (int i = 0; i < 256; ++i)
   {
      fwrite(table->huffman[i], sizeof(
          char), 256, file);
   }
```

chaque ligne de la table huffman contient un code binaire ASCII pour une intensité de 0 à 255 (terminée par 0), codée sur 256 octets.

Encodage des pixels avec Huffman

Chaque pixel a trois composantes : (r, g, b). Pour chaque composante : On récupère son code Huffman.

On écrit chaque bit un par un dans un buffer de 8 bits (unsigned char buffer).

Quand le buffer est plein, on l'écrit dans le fichier.

Écriture finale du buffer

```
if (bit_count > 0)
{
   buffer <<= (8 - bit_count);
   fputc(buffer, file);
}
```

Assure que les derniers bits restants sont écrits, complétés avec des 0 à droite.

Afin de tester l'efficacité de notre algorithme de compression basé sur le codage de Huffman, nous avons appliqué la fonction `compress_img_huffman` sur `image.ppm`.

Taille fichier PPM original : 196623 octets
Taille fichier compresse : 90127 octets
Taux de compression : 45.84%
Gain de taille : 54.16%

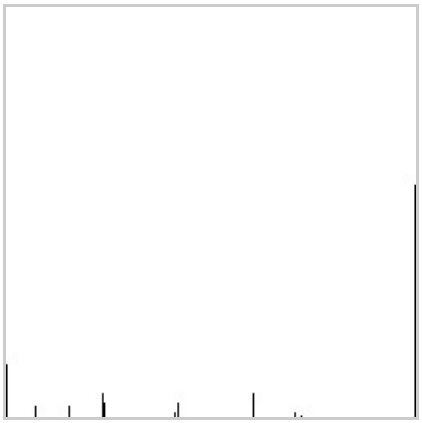
Le fichier compressé occupe moins de la moitié de l'espace disque par rapport au fichier original. Ce gain s'explique par la forte redondance des couleurs dans l'image, ce qui rend le codage de Huffman efficace.
Cependant, avec une image plus complexe ou bruitée, le taux de compression serait probablement moins élevé.

7 Compression et analyse d'une image simple

Prenons une autre image afin de tester et interpréter notre projet :

Calculs et visuel

Entropie : 2.2854 bits
Longueur moyenne des codes : 2.3165 bits
Histogramme sous forme d'une image :



Compression naive

Taille fichier PPM original : 183039 octets
Taille fichier compresse : 248575 octets
Taux de compression : 135.80%
Gain de taille : -35.80%

Compression de Huffman

Taille fichier PPM original : 183039 octets
Taille fichier compresse : 118548 octets
Taux de compression : 64.77%
Gain de taille : 35.23%

Interprétation

L'entropie de l'image est faible, ce qui signifie que certaines intensités sont nettement plus fréquentes que d'autres. Cela rend le codage de Huffman efficace, en attribuant des codes très courts aux valeurs dominantes.

La longueur moyenne des codes Huffman est très proche de l'entropie théorique, ce qui montre que l'arbre de Huffman est bien construit.

Ce résultat montre que, pour une image avec beaucoup de redondance, le codage de Huffman permet une compression significative tout en restant sans perte. Cela illustre que le contenu de l'image est un facteur déterminant dans l'efficacité réelle de la compression.

8 Compression et analyse d'une image complexe

Une autre image, plus complexe, a été testée avec les mêmes algorithmes de compression. Les résultats observés sont les suivants :

Calculs et visuel

Entropie : 7.7502 bits
Longueur moyenne des codes : 7.7806 bits
Histogramme sous forme d'une image :

Compression naive

Taille fichier PPM original : 786447 octets
Taille fichier compresse : 851983 octets
Taux de compression : 108.33%
Gain de taille : -8.33%

Compression de Huffman

Taille fichier PPM original : 786447 octets
Taille fichier compresse : 830416 octets
Taux de compression : 105.59%
Gain de taille : -5.59%

Interprétation

Contrairement au cas précédent, ces compressions ont conduit à une augmentation de la taille du fichier. Plusieurs facteurs peuvent expliquer ces phénomènes :

L'entropie mesurée E est supérieure à 7,7 bits, soit proche du maximum théorique (8 bits pour une image 8 bits). Cela signifie que les intensités sont réparties de manière relativement uniforme, sans dominance claire d'une valeur sur les autres. Il y a donc peu de redondance à exploiter.

Sur cette image avec peu de redondance, la répartition des intensités annule les gains éventuels de taille et produit une expansion, cependant toujours inférieure à la compression naïve.

Ce cas met en évidence les limites du codage de Huffman : bien que théoriquement sans perte, il n'est pas toujours efficace en termes de taille lorsque l'image est peu redondante.

9 Décompression de Huffman

La fonction `decompress_image_huffman` permet de reconstruire une image à partir d'un fichier compressé au format HPPM, en s'appuyant sur le décodage de Huffman.

Elle a été fournie initialement par l'enseignant, puis adaptée pour correspondre au format d'en-tête et au fichier HPPM généré lors de la compression.

Lecture de l'en-tête

Le fichier HPPM commence par une signature (P7), suivie des dimensions de l'image et de l'intensité maximale. Ces informations sont extraites avec `fscanf`.

Lecture de la table de codage Huffman

La table est lue directement avec `fread`, dans une structure `HuffmanTable`. Puis l'arbre binaire est reconstruit via `build_tree_from_table`.

Décodage bit-à-bit

Chaque octet du fichier est lu puis interprété bit par bit. À chaque bit, on descend dans l'arbre Huffman. Lorsque l'on atteint une feuille (un nœud sans enfants), la valeur correspondante est affectée à une composante couleur du pixel (r, g, b).

Remplissage des pixels

Trois intensités décodées successivement sont regroupées en un pixel (r, g, b). Le processus continue jusqu'à reconstruction complète de l'image.

Ainsi, il n'y a aucune altération de l'image lors de la compression puis décompression : chaque pixel est reconstitué fidèlement.

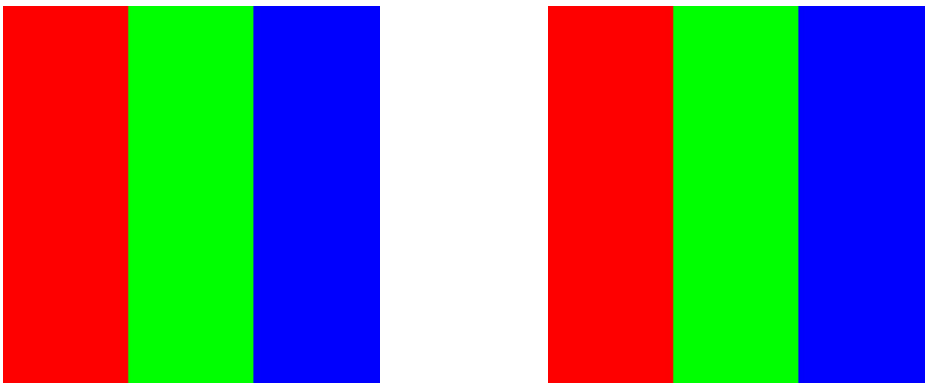


FIGURE 4 – image.ppm et image_huffman_reconstruit.ppm

10 Conclusion

Ce projet de compression d'image en langage C m'a permis d'aborder de manière concrète et progressive plusieurs notions liées au traitement de données, à l'optimisation et à la représentation binaire. À travers l'implémentation d'un compresseur naïf, puis d'un compresseur basé sur l'algorithme de Huffman, j'ai pu comparer deux approches opposées : une méthode simple mais peu efficace, et une méthode plus performante.

Bien que le format naïf entraîne une surcharge importante liée à la table de codage et n'offre aucune compression réelle, il constitue une bonne base de comparaison. À l'inverse, le codage de Huffman s'adapte dynamiquement aux caractéristiques de l'image. Lorsqu'une forte redondance est présente, le gain de taille peut être significatif, comme le montrent les tests réalisés.

Ce projet m'a également permis d'appliquer des concepts liés à l'entropie, aux histogrammes et à la représentation binaire. En testant plusieurs images, j'ai constaté que le contenu visuel joue un rôle essentiel dans l'efficacité des algorithmes de compression, notamment que les méthodes sans perte sont peu adaptées aux images trop riches en détails.

En conclusion, ce projet m'a aidé à mieux comprendre les mécanismes de la compression sans perte et de la manipulation de fichiers binaires.