Compression d'images par surfaces unies

Rapport de projet

Lucien Cartier-Tilet

November 28, 2018

Contents

1 Le problème

Le but de ce projet a été de créer un logiciel permettant la compression d'images via la détection de surfaces d'une même couleur, ainsi que de permettre la décompression du fichier ainsi généré à la compression en une image identique à l'image d'origine. Il est à noter que cet algorithme est orienté vers les images de style *comics* ou avec peu de couleurs, d'où l'image test que vous trouverez en annexe qui fut mon image de test principale.

Le lien de l'énoncé original se situe également en annexe. Une documentation du projet sera également disponible en suivant le lien donné également en annexe.

2 Résolution initiale du problème

Afin de résoudre ce problème, je me suis basé sur un algorithme de M. Jean-Jaques Bourdin et l'ai adapté au problème de la manière suivante : pour chaque pixel de l'image, je teste si le pixel courant a une couleur identique à la couleur d'une surface, ou zone, déjà existante. Si une telle zone n'existe pas déjà, alors je la créé puis pour chaque pixel de la ligne courante je me déplace à l'extrême gauche et à l'extrême droite du segment de la même couleur que le pixel courant. Ainsi on obtient un segment de couleur unie, et il est donc possible de maintenant stocker uniquement les limites de ce segment et à l'ajouter à la collection de segments de la zone, qui elle contient les informations de la couleur de la zone. Ensuite, pour chaque pixel en dessus puis en dessous de ce segment, on répète la même procédure. Cela permet de lister toutes les zones contigües et de les ajouter ainsi à la zone courante.

Actuellement, avec l'image de test que l'on peut trouver sur mon dépôt git (lien en annexes) d'une taille de 3582016 octets, j'obtiens un taux de compression de 67% environ avec une image compressée à 2403581 octets.

Pour ce qui est du temps d'exécution, j'exécute le script suivant avec la commande `./run-time.sh 10` (10 étant le nombre d'exécutions que je souhaite effectuer) afin d'obtenir le temps d'exécution à répétition du programme :

```
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "Bad usage of run-time.sh"
  echo "./run-time <number of runs>"
  exit 1
fi
echo "Starting $1 runs of surfaces-unies"
start=$(date +%s.%N)
for i in `seq 1 $1`; do
  ./bin/surfaces-unies -c -i ./img/asterix.ppm
  ./bin/surfaces-unies -u -i output.su
done
end=$(date +%s.%N)
runtime=$(python -c "print(((${end} - ${start}) / ${1}))")
echo "Average runtime was $runtime"
```

Ainsi, j'obtient la moyenne sur dix exécutions d'un temps d'exécution de 7.06 secondes sur un CPU Intel i7-6700HQ (3.500GHz).

3 Améliorations possibles

3.1 Algorithme

3.1.1 Théorie

Il est possible d'améliorer l'algorithme par rapport à celui utilisé initialement en ignorant l'étape de test des pixels supérieurs et inférieurs à un segment testé. En effet, cette étape est un résidu de l'algorithem d'origine qui a pour but de ne détecter qu'une zone dont tous les segments sont contigus à au moint un autre segment de la même forme, le tout ne constituant qu'une seule et unique forme continue. Hors ici ce

dernier point ne nous intéresse pas, et deux formes de la même couleur n'ont pas à être considérées comme étant deux entités différentes. Ainsi, en ignorant cette étape cela permet de passer de manière itérative sur tous les pixels, simplifiant ainsi l'algorithme.

3.1.2 Modifications apportées

La modification apportée au code source est relativement simple étant donné qu'il suffit de commenter les deux dernières boucles `for` de la fonction `addPixelToSelectedZone` du fichier `compress.c`. Cela bloque l'exécution de la fonction créant les segments à partir des pixels au dessus et en dessous du segment courant.

```
   }
   darrayPushBack(t_zone->segments,
                  newSegment((uint32_t)right_limit, (uint32_t)left_limit));
-  for (; left_limit <= right_limit; ++left_limit) {
-    addPixelToSelectedZone(t_img, t_idx + t_img->sizeX, t_zone);
-  }
-  for (; left_limit <= right_limit; ++left_limit) {
-    addPixelToSelectedZone(t_img, t_idx - t_img->sizeX, t_zone);
-  }
+  /* for (; left_limit <= right_limit; ++left_limit) { */
+  /*   addPixelToSelectedZone(t_img, t_idx + t_img->sizeX, t_zone); */
+  /* } */
+  /* for (; left_limit <= right_limit; ++left_limit) { */
+  /*   addPixelToSelectedZone(t_img, t_idx - t_img->sizeX, t_zone); */
+  /* } */
 }
```

3.1.3 Résultats

Avec dix exécutions successives, sur le même processeur avec les mêmes arguments et le programme également compilé avec les options d'optimisation '-O3' également, j'obtiens une vitesse d'exécution moyenne à la compression/décompression successives de 7.23s, soit une perte de 2.35% du temps d'exécution d'origine. J'explique cela par le fait que pour chaque nouveau pixel non-traité, il est nécessaire de trouver la zone correspondant à la couleur du nouveau pixel, ce qui en soit prend du temps alors qu'avec la méthode d'origine, on connaît déjà la zone et on n'a qu'à explorer les pixels de couleur unie, sachant que là où on a le plus de chance de trouver des pixels de couleur identique sur des images de type comics est au niveau des pixels adjascents au pixel actuel. Ainsi, cette piste s'est révélée non concluante, et je ne continuerai pas dans cette direction.

3.2 Sortir du concept d'image avec des lignes et colonnes

3.2.1 Théorie

L'algorithme actuel considère le fichier d'entrée comme étant une image en deux dimensions, limitant ainsi les segments au début d'une ligne de pixels même si le pixel précédent dans le vecteur dans lequel ces derniers sont stockés est de la même couleur, divisant ainsi des segments en plusieurs segments inutilement. En ignorant ces fins de lignes pour ne procéder qu'à l'analyse des pixels comme ne faisant partie que d'une ligne unique permettrait d'éviter ces ajouts inutiles de segments, et ainsi économiser un peu d'espace avec le fichier compressé. Cependant, je ne pense pas que cette solution soit réellement efficace du fait du peu de cas où ce problème se pose.

3.2.2 Modifications apportées

Les deux boucles de la fonction `addPixelToSelectedZone` cherchant les segments ont été modifiées ainsi :

```
for (right_limit = t_idx; right_limit < img_size; ++right_limit) {
  current_pixel = darrayGet(t_img->pixels, (size_t)right_limit);
  if (!sameColor(current_pixel, t_zone)) {
```

```
    break;
  }
  (*current_pixel).visited = 1;
 }

for (left_limit = t_idx; left_limit > 0; --left_limit) {
  current_pixel = darrayGet(t_img->pixels, (size_t)left_limit);
  if (current_pixel->visited || !sameColor(current_pixel, t_zone)) {
    break;
  }
  (*current_pixel).visited = 1;
 }
```

Cette modification permet d'ignorer les retours à la ligne et ainsi potentiellement économiser de l'espace lors de l'écriture de fichier compressés.

3.2.3 Résultats

Hélas le gain d'espace n'est pas flagrant, et cela s'explique par le faible nombre de lignes de pixels de l'image, 1500 dans ce cas, et du fait qu'économiser deux `uint32_t` par ligne est une amélioration mineure, permettant dans le cas de l'image `asterix.ppm` d'économiser uniquement 12 kilo octets. La différence sur le fichier compressé de 2.3 méga octets d'origine est donc négligeable. Je ne continuerai donc pas de travailler sur cette piste.

3.3 Passer à de la compression à pertes

3.3.1 Théorie

Il existe toujours la possibilité de ne plus faire que de la compression sans pertes mais de faire également de la compression avec pertes, en acceptant en argument un taux de tolérance quant à la similarité des couleurs entre elles. Ainsi, cela éliminera certaines couleurs de l'image et économisera ainsi des zones disposant d'un petit nombre de zones ou de segments en les assimilant aux couleurs qui leur sont proches. L'inconvénient est qu'avec ce paramètre, il sera impossible de restituer l'image d'origine à l'identique.

3.3.2 Modifications apportées

Seuls trois fichiers ont eu à être modifiés : `main.c`, `compress.h` et `compress.c`. Voici ci-dessous les modifications apportées à chacun de ces fichiers (sortie de la commande `git diff`) :

```
diff --git a/src/compress.c b/src/compress.c
index 191265a..25e43aa 100644
--- a/src/compress.c
+++ b/src/compress.c
@@ -5,6 +5,8 @@

 #include "compress.h"

+int tolerance;
+
 /**
  *  Cette fonction permet d'évaluer si le pixel passé en argument est éligible à
  *  la zone passée également en argument.
@@ -14,10 +16,17 @@
  *  \return Valeur booléenne, `1` si le pixel est éligible, `0` sinon
  */
 int32_t sameColor(Pixel *t_pixel, Zone *t_zone) {
-  return (t_pixel->red == t_zone->red && t_pixel->green == t_zone->green &&
-          t_pixel->blue == t_zone->blue)
```

```
-             ? 1
-             : 0;
+  int diff_red, diff_green, diff_blue;
+  if(tolerance == 0) {
+    return (t_pixel->red == t_zone->red && t_pixel->green == t_zone->green &&
+            t_pixel->blue == t_zone->blue)
+               ? 1
+               : 0;
+  }
+  diff_red = (abs((int32_t)t_zone->red - (int32_t)t_pixel->red) * 100) / 255;
+  diff_green = (abs((int32_t)t_zone->green - (int32_t)t_pixel->green) * 100) / 255;
+  diff_blue = (abs((int32_t)t_zone->blue - (int32_t)t_pixel->blue) * 100) / 255;
+  return ((diff_red + diff_green + diff_blue) / 3) <= tolerance;
 }

 /**
@@ -173,13 +182,15 @@ void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones) {
  *  \param[in] t_input_file Nom/chemin du fichier `.ppm` d'entrée
  *  \param[in] t_output_file Nom/chemin du fichier `.su` de sortie
  */
-void compress(const char *t_input_file, const char *t_output_file) {
+void compress(const char *t_input_file, const char *t_output_file,
+              int32_t t_tolerance) {
   Image *img;
   darray *zones;
   FILE *output_file;
   if (!t_output_file) {
     t_output_file = DEFAULT_COMPRESSED_NAME;
   }
+  tolerance = t_tolerance;
   img = newImage();
   imageLoadPPM(t_input_file, img);
   output_file = get_file(t_output_file, "wb");
diff --git a/src/compress.h b/src/compress.h
index 9b2f832..50f7c25 100644
--- a/src/compress.h
+++ b/src/compress.h
@@ -23,6 +23,7 @@ void write_segments(FILE *t_output, darray *t_segments);
 /// Écrit les données compressées dans le fichier de sortie
 void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones);
 /// Compresse l'image d'entrée
-void compress(const char *t_input_file, const char *t_output_file);
+void compress(const char *t_input_file, const char *t_output_file,
+              int32_t tolerance);

 #endif /* SRC_COMPRESS_H_ */
diff --git a/src/main.c b/src/main.c
index 8ba5bee..4676ca7 100644
--- a/src/main.c
+++ b/src/main.c
@@ -24,18 +24,21 @@
  */
 void help(int t_exit_code) {
   puts("Usage:\n"
-       "surfaces-unies -i path [-o path] [-options]\n\n"
+       "surfaces-unies -i path [-o path] [--options] [-t 0-100]\n\n"
        "The default action is to compress the mandatory input image to a .su\n"
```

```
        "file saved in the current directory.\n"
        "The input image MUST be saved in the ppm format.\n"
        "Options available:\n"
-       "-h --help\n\tdisplay the current message\n"
-       "-i --input\n\tpath to the input file (MANDATORY)\n"
+       "-h --help\n\tDisplay the current message\n"
+       "-i --input\n\tPath to the input file (MANDATORY)\n"
        "-o --output\n"
-       "\tpath to the output file (if the file already exists, it will be\n"
+       "\tPath to the output file (if the file already exists, it will be\n"
        "\toverwritten)\n"
-       "-c --compress\n\tcompress the input file\n"
-       "-u --uncompress\n\tuncompresses the input file to the output file.");
+       "-t --tolerance\n"
+       "\tColor tolerance for lossy compression. By default at 0 for lossless\n"
+       "\tcompression, at 100 will consider every color to be the same.\n"
+       "-c --compress\n\tCompress the input file\n"
+       "-u --uncompress\n\tUncompresses the input file to the output file.");
   exit(t_exit_code);
 }

@@ -49,9 +52,10 @@ void help(int t_exit_code) {
  *  supplémentaires aux fonctions.
  */
 struct Argres {
-  char *input;   /*!< Nom du fichier d'entrée */
-  char *output;  /*!< Nom du fichier de sortie */
-  char compress; /*!< Le fichier d'entrée doit-il être compressé ? */
+  char *input;       /*!< Nom du fichier d'entrée */
+  char *output;      /*!< Nom du fichier de sortie */
+  char compress;     /*!< Le fichier d'entrée doit-il être compressé ? */
+  int32_t tolerance; /*!< Tolérance en pourcentage des différences de couleur */
 };
 typedef struct Argres Argres;

@@ -72,6 +76,12 @@ void get_args(Argres *t_args, const int *const t_c) {
   case 'o': (*t_args).output = optarg;  break;
   case 'c': (*t_args).compress = 1;  break;
   case 'u': (*t_args).compress = 0; break;
+  case 't':
+    (*t_args).tolerance = atoi(optarg);
+    if(t_args->tolerance < 0 || t_args->tolerance > 100) {
+      help(ARGERROR);
+    }
+    break;
   case '?':
   default: help(ARGERROR);
   }
@@ -95,16 +105,18 @@ Argres process_args(const int t_argc, char *t_argv[]) {
   res.input = NULL;
   res.compress = 1;
   res.output = NULL;
+  res.tolerance = 0;
   while (1) {
     int option_index = 0;
     static struct option long_options[] = {
         {"help", no_argument, NULL, 'h'},
```

```
         {"input", required_argument, NULL, 'i'},
         {"output", required_argument, NULL, 'o'},
+        {"tolerance", required_argument, NULL, 't'},
         {"compress", no_argument, NULL, 'c'},
         {"uncompress", no_argument, NULL, 'u'},
         {NULL, 0, NULL, 0}};
-    int c = getopt_long(t_argc, t_argv, "hi:o:cu", long_options, &option_index);
+    int c = getopt_long(t_argc, t_argv, "hi:o:t:cu", long_options, &option_index);
     if (c == -1)
       break;
     get_args(&res, &c);
@@ -129,8 +141,8 @@ int main(int argc, char **argv) {
     fprintf(stderr, "ERROR: no input file.");
     help(ARGERROR);
   }
-  if(argresults.compress) {
-    compress(argresults.input, argresults.output);
+  if (argresults.compress) {
+    compress(argresults.input, argresults.output, argresults.tolerance);
   } else {
     uncompress(argresults.input, argresults.output);
   }
```

3.3.3 Résultats

La nouvelle option `-t` est une option très sensible qui peut profondément changer la qualité de l'image dès des valeurs basses, aux alentours de 15 à 25. Cependant, une nette réduction de la taille de l'image compressée peut être constatée, ainsi qu'une baisse notable du temps de compression de l'image dû aux recherches des zones correspondant aux pixels moins fréquentes, et lorsqu'elles se produisent, la recherche se fait parmi moins de zones. Ainsi, pour différentes valeurs de tolérance, on obtient ces résultats :

tolérance	vitesse d'exécution	taille de l'image compressée
0	7.37s	2.3Mo
5	0.85s	2.0Mo
10	0.82s	2.0Mo
15	0.86s	2.0Mo
20	0.84s	1.9Mo
25	0.85s	1.9Mo
30	0.92s	1.8Mo
40	1.18s	2.1Mo
50	0.90s	1.5Mo
60	1.41s	1.9Mo
70	1.34s	1.8Mo
80	1.40s	1.9Mo
90	1.42s	1.9Mo
100	0.17s	12Ko

On remarque cependant qu'après environ 30% de tolérance de couleur, les temps ont tendance à remonter et les fichiers ont tendance à devenir plus lours, excepté pour la tolérance de 50%. Je suppose qu'il s'agit du fait d'un grand nombre de segments contigus de couleur différente s'alternant rapidement. En tous cas, bien qu'on aie en effet quelques améliorations concernant la taille de l'image compressée, je ne considère personellement pas cette technique comme en vallant la peine du fait d'artéfacts facilement visibles dès les premières valeurs proches de 0, comme on peut le voir sur l'image ci-dessous, compressée avec une tolérance de 5% :

Dans le cas où cela ne serait pas bien visible en version papier, une version digitale de ce document est disponible en ligne, voir le lien donnée en annexes.

4 Annexes

4.1 Liens

Énoncé http://www.ai.univ-paris8.fr/~jj/Cours/Algo/AA18.html#projets (projet n°22)

Dépôt https://labs.phundrak.fr/phundrak/surfaces-unies/

Image de test https://labs.phundrak.fr/phundrak/surfaces-unies/blob/master/img/asterix.ppm

Image compressée avec pertes https://labs.phundrak.fr/phundrak/surfaces-unies/blob/master/img/asterix5p.png

Version digitale de ce document https://phundrak.fr/surfaces-unies/rapport.pdf

Documentation du code source https://phundrak.fr/surfaces-unies/

Version PDF de la documentation du code source https://phundrak.fr/surfaces-unies/refman.pdf

4.2 Image de test

4.3 Code source

main.c

```c
/**
 *   \file main.c
 *   \brief Fichier principal du programme
 *
 *  Ce fichier contient les fonctions principales du logiciel qui ne sont pas
 *  directement liées à la logique et au traitement des données du logiciel,
 *  mais plutôt au traitement des arguments passés au processus et au lancement
 *  des fonctions cœures du programme.
 */

#include "compress.h"
#include "uncompress.h"
#include <getopt.h>
#include <string.h>

/**
 *  \brief Affiche un message d'aide
 *
 *  Affiche un message d'aide pour le logiciel ainsi que son utilisation, puis
 *  termine le processus avec le code de sortie indiqué par l'argument de la
 *  fonction.
 *
 *  \param[in] t_exit_code Code de sortie du processus
 */
void help(int t_exit_code) {
  puts("Usage:\n"
       "surfaces-unies -i path [-o path] [--options] [-t 0-100]\n\n"
       "The default action is to compress the mandatory input image to a .su\n"
       "file saved in the current directory.\n"
       "The input image MUST be saved in the ppm format.\n"
       "Options available:\n"
       "-h --help\n\tDisplay the current message\n"
       "-i --input\n\tPath to the input file (MANDATORY)\n"
       "-o --output\n"
       "\tPath to the output file (if the file already exists, it will be\n"
       "\toverwritten)\n"
       "-t --tolerance\n"
       "\tColor tolerance for lossy compression. By default at 0 for lossless\n"
       "\tcompression, at 100 will consider every color to be the same.\n"
       "-c --compress\n\tCompress the input file\n"
       "-u --uncompress\n\tUncompresses the input file to the output file.");
  exit(t_exit_code);
}

/**
 *  \struct Argres
 *  \brief Résultats du traitement des arguments du processus
 *
 *  Cette structure est utilisée pour consolider ensemble les résultats du
 *  traitement des arguments et les renvoyer en une fois au lieu d'avoir à
 *  utiliser des variables globales ou des pointeurs en arguments
 *  supplémentaires aux fonctions.
 */
struct Argres {
```

```c
  char *input;       /*!< Nom du fichier d'entrée */
  char *output;      /*!< Nom du fichier de sortie */
  char compress;     /*!< Le fichier d'entrée doit-il être compressé ? */
  int32_t tolerance; /*!< Tolérance en pourcentage des différences de couleur */
};
typedef struct Argres Argres;

/**
 *  \brief Processes independently the arguments of the process
 *
 *  Each option and switch will be processed here and will modify appropriately
 *  the parameter `args`
 *
 *  \param[out] t_args Result of the arguments processing
 *  \param[in] t_c Switch or option passed
 */
void get_args(Argres *t_args, const int *const t_c) {
  switch (*t_c) {
  case 0:   break;
  case 'h': help(NOERROR); break;
  case 'i': (*t_args).input = optarg;   break;
  case 'o': (*t_args).output = optarg;  break;
  case 'c': (*t_args).compress = 1;  break;
  case 'u': (*t_args).compress = 0; break;
  case 't':
    (*t_args).tolerance = atoi(optarg);
    if(t_args->tolerance < 0 || t_args->tolerance > 100) {
      help(ARGERROR);
    }
    break;
  case '?':
  default: help(ARGERROR);
  }
}

/**
 *  \brief Traite les arguments passés au processus
 *
 *  Les arguments passés au processus seront traités ici. Les arguments passés
 *  dans cette fonction ne subiront aucune modification. La fonction renvoie une
 *  structure \ref Argres contenant le nom de fichier d'entrée et de sortie
 *  ainsi qu'un booléen indiquant si le fichier d'entrée doit être compressé ou
 *  décompressé.
 *
 *  \param[in] t_argc Nombre d'arguments reçus
 *  \param[in] t_argv Arguments reçus par le processus
 *  \return structure \ref Argres
 */
Argres process_args(const int t_argc, char *t_argv[]) {
  Argres res;
  res.input = NULL;
  res.compress = 1;
  res.output = NULL;
  res.tolerance = 0;
  while (1) {
    int option_index = 0;
    static struct option long_options[] = {
```

```
112           {"help", no_argument, NULL, 'h'},
113           {"input", required_argument, NULL, 'i'},
114           {"output", required_argument, NULL, 'o'},
115           {"tolerance", required_argument, NULL, 't'},
116           {"compress", no_argument, NULL, 'c'},
117           {"uncompress", no_argument, NULL, 'u'},
118           {NULL, 0, NULL, 0}};
119       int c = getopt_long(t_argc, t_argv, "hi:o:t:cu", long_options, &option_index);
120       if (c == -1)
121         break;
122       get_args(&res, &c);
123     }
124     return res;
125   }
126
127   /**
128    *  \brief Fonction `main` lancée avec le processus
129    *
130    *  This function is launched with the process. It will analyze the arguments it
131    *  received, and depending on them will either compress or uncompress the input
132    *  file, or will throw an error and stop in case of incorrect arguments.
133    *
134    *  \param[in] argc Nombre d'arguments reçus par le processus
135    *  \param[in] argv Tableau des arguments reçus par le processus
136    *  \return Code de status du processus
137    */
138   int main(int argc, char **argv) {
139     Argres argresults = process_args(argc, argv);
140     if (NULL == argresults.input) {
141       fprintf(stderr, "ERROR: no input file.");
142       help(ARGERROR);
143     }
144     if (argresults.compress) {
145       compress(argresults.input, argresults.output, argresults.tolerance);
146     } else {
147       uncompress(argresults.input, argresults.output);
148     }
149     return 0;
150   }
```

compress.h

```
1   /**
2    *   \file compress.h
3    *   \brief Déclaration pour la (dé)compression d'images
4    */
5
6   #ifndef SRC_COMPRESS_H_
7   #define SRC_COMPRESS_H_
8
9   #include "ppm.h"
10
11  #define DEFAULT_COMPRESSED_NAME "output.su"
12
13  /// Teste l'éligibilité d'un pixel à une zone
14  int32_t sameColor(Pixel *t_pixel, Zone *t_zone);
15  /// Ajoute un pixel et ses pixels connexes à une zone
16  void addPixelToSelectedZone(Image *t_img, int64_t t_idx, Zone *t_zone);
```

```c
/// Sélectionne la zone correspondant à la couleur d'un pixel
void chooseZoneForPixel(Image *t_img, int64_t t_idx, darray *zones);
/// Créé les zones d'une image
darray *imgToZones(Image *t_img);
/// Écrit tous les \ref Segment d'une zone dans le fichier de sortie
void write_segments(FILE *t_output, darray *t_segments);
/// Écrit les données compressées dans le fichier de sortie
void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones);
/// Compresse l'image d'entrée
void compress(const char *t_input_file, const char *t_output_file,
              int32_t tolerance);

#endif /* SRC_COMPRESS_H_ */
```

compress.c

```c
/**
 *   \file compress.c
 *   \brief Implémentation de la (dé)compression d'images
 */

#include "compress.h"

/**
 *  \var uint32_t tolerance
 *  \brief Color tolerance
 *
 *  Cette variable est la valeur du pourcentage de tolérance couleur lors de la
 *  création de nouvelles zones. Cette variable contient une valeur située entre
 *  0 et 100 inclus.
 */
int32_t tolerance;

/**
 *  Cette fonction permet d'évaluer si le pixel passé en argument est éligible à
 *  la zone passée également en argument. Si la \ref tolerance a pour valeur 0,
 *  alors les couleurs doivent être strictements identiques. Sinon, leur
 *  différence doit être inférieure à la tolérance de couleur.
 *
 *  \param[in] t_pixel Pointeur vers le pixel dont l'éligibilité est testée
 *  \param[in] t_zone Zone à laquelle le pixel est éligible ou non
 *  \return Valeur booléenne, `1` si le pixel est éligible, `0` sinon
 */
int32_t sameColor(Pixel *t_pixel, Zone *t_zone) {
  int diff_red, diff_green, diff_blue;
  if (tolerance == 0) {
    return (t_pixel->red == t_zone->red && t_pixel->green == t_zone->green &&
            t_pixel->blue == t_zone->blue)
               ? 1
               : 0;
  }
  diff_red = (abs((int32_t)t_zone->red - (int32_t)t_pixel->red) * 100) / 255;
  diff_green =
      (abs((int32_t)t_zone->green - (int32_t)t_pixel->green) * 100) / 255;
  diff_blue = (abs((int32_t)t_zone->blue - (int32_t)t_pixel->blue) * 100) / 255;
  return ((diff_red + diff_green + diff_blue) / 3) <= tolerance;
}

```

```c
/**
 *  Ajoute un pixel à la zone passé en argument si le pixel à l'index passé en
 *  argument est éligible à la zone. Si un pixel n'a pas encore été visité, cela
 *  veut dire également qu'il ne fait partie d'aucun segment, il sera donc
 *  ajouté à un nouveau segment auquel seront rajoutés tous les pixels connexes
 *  éligibles à la zone. Ensuite, le segment est ajouté à la zone, et la
 *  fonction actuelle est appelée sur tous les pixels supérieurs et inférieurs
 *  aux pixels du segment.
 *
 *  \param[in] t_img Image contenant les pixels explorés
 *  \param[in] t_idx Index du pixel actuel dans l'image `t_img`
 *  \param[out] t_zone Zone à laquelle sera potentiellement ajouté le pixel
 */
void addPixelToSelectedZone(Image *t_img, int64_t t_idx, Zone *t_zone) {
  const size_t img_size = darraySize(t_img->pixels);
  Pixel *current_pixel;
  const uint32_t y = (uint32_t)(t_idx / t_img->sizeX);
  int64_t left_limit, right_limit;
  const int64_t xd_limit = (int64_t)t_img->sizeX * (y + 1);
  if (t_idx >= (int64_t)img_size || t_idx < 0) {
    return;
  }
  current_pixel = darrayGet(t_img->pixels, (size_t)t_idx);
  if (current_pixel->visited || !sameColor(current_pixel, t_zone)) {
    return;
  }
  (*current_pixel).visited = 1;

  for (right_limit = t_idx; right_limit < xd_limit; ++right_limit) {
    current_pixel = darrayGet(t_img->pixels, (size_t)right_limit);
    if (!sameColor(current_pixel, t_zone)) {
      break;
    }
    current_pixel->visited = 1;
  }
  for (left_limit = t_idx; left_limit - (y - 1) * (int64_t)t_img->sizeX >= 0;
       --left_limit) {
    current_pixel = darrayGet(t_img->pixels, (size_t)left_limit);
    if (current_pixel->visited || !sameColor(current_pixel, t_zone)) {
      break;
    }
    (*current_pixel).visited = 1;
  }

  darrayPushBack(t_zone->segments,
                 newSegment((uint32_t)right_limit, (uint32_t)left_limit));
  for (; left_limit <= right_limit; ++left_limit) {
    addPixelToSelectedZone(t_img, t_idx + t_img->sizeX, t_zone);
  }
  for (; left_limit <= right_limit; ++left_limit) {
    addPixelToSelectedZone(t_img, t_idx - t_img->sizeX, t_zone);
  }
}

/**
 *  Sélectionne la zone correspondant à la couleur du pixel. Si aucune zone
 *  existante ne correspond, une nouvelle est créée et est ajoutée à l'image.
```

```
100  *  Chaque pixel est itéré, et ignoré si le pixel a déjà été visité auparavant.
101  *
102  *  \param[out] t_img L'image contenant les pixels à tester
103  *  \param[in] t_idx Index du pixel à tester
104  *  \param[out] t_zones Liste des zones de l'image
105  */
106 void chooseZoneForPixel(Image *t_img, int64_t t_idx, darray *t_zones) {
107   Zone *current_zone;
108   Pixel *pixel;
109   size_t i;
110   pixel = darrayGet(t_img->pixels, (size_t)t_idx);
111   if (pixel->visited) {
112     return;
113   }
114   for (i = 0; i < darraySize(t_zones); ++i) {
115     current_zone = darrayGet(t_zones, i);
116     if (sameColor(pixel, current_zone)) {
117       addPixelToSelectedZone(t_img, t_idx, current_zone);
118       return;
119     }
120   }
121   current_zone = newZone(pixel->red, pixel->green, pixel->blue);
122   darrayPushBack(t_zones, current_zone);
123   addPixelToSelectedZone(t_img, t_idx, current_zone);
124 }
125
126 /**
127  *  Génère les zones de l'image en titérant chaque pixel de l'image.
128  *
129  *  \param t_img Image à convertir en zones
130  *  \return Pointeur vers un \ref darray de structures \ref Zone
131  */
132 darray *imgToZones(Image *t_img) {
133   darray *zones;
134   const size_t nb_pixels = darraySize(t_img->pixels);
135   int64_t i;
136   zones = darrayNew(sizeof(Zone));
137   for (i = 0; i < (int64_t)nb_pixels; ++i) {
138     chooseZoneForPixel(t_img, i, zones);
139   }
140   return zones;
141 }
142
143 /**
144  *  Cette fonction écrit dans \p t_output la taille en `uint64_t` de la zone,
145  *  c'est à dire le nombre de segment qu'elle contient, puis écrit
146  *  individuellement chaque segment dans \p t_output.
147  *
148  *  \param[out] t_output Fichier de sortie
149  *  \param[in] t_segments Segments à écrire dans \p t_output
150  */
151 void write_segments(FILE *t_output, darray *t_segments) {
152   uint64_t nb_segments, j;
153   Segment *segment;
154   nb_segments = darraySize(t_segments);
155   fwrite(&nb_segments, sizeof(nb_segments), 1, t_output);
156   for (j = 0; j < darraySize(t_segments); ++j) {
```

```
157       segment = darrayGet(t_segments, j);
158       fwrite(&segment->left_limit, sizeof(Segment), 1, t_output);
159     }
160   }
161
162   /**
163    *  Écrit la taille de l'image en abscisse et ordonnées, les deux sous forme de
164    *  `uint64_t` puis le nombre de zones sous forme de `uint64_t`. Puis, pour
165    *  chaque zone son code couleur composé de trois `uint8_t` successifs
166    *  représentant ses couleurs rouge, vert et bleu sont écrit dans le fichier de
167    *  sortie \p t_output. Après chaque écriture de zone, l'ensemble des segments
168    *  de la zone est libéré de la mémoire. Une fois toutes les zones écrites dans
169    *  le fichier de sortie, \p t_zones et libéré de la mémoire.
170    *
171    *  \param[in] t_img \ref Image contenant les dimensions du fichier d'origine
172    *  \param[out] t_output Fichier où sont écrites les données compressées
173    *  \param[in] t_zones Tableau des \ref Zone à écrire puis libérer
174    */
175   void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones) {
176     uint64_t i, nb_zones = darraySize(t_zones);
177     Zone *current_zone;
178     fwrite(&t_img->sizeX, sizeof(t_img->sizeX), 2, t_output);
179     fwrite(&nb_zones, sizeof(nb_zones), 1, t_output);
180     for (i = 0; i < darraySize(t_zones); ++i) {
181       current_zone = darrayGet(t_zones, i);
182       fwrite(&current_zone->red, sizeof(current_zone->red) * 3, 1, t_output);
183       write_segments(t_output, current_zone->segments);
184       darrayDelete(current_zone->segments);
185     }
186     darrayDelete(t_zones);
187   }
188
189   /**
190    *  Convertit une image en zones puis écrit ces zones dans un fichier,
191    *  compressant ainsi l'image passée en argument.
192    *
193    *  \param[in] t_input_file Nom/chemin du fichier `.ppm` d'entrée
194    *  \param[in] t_output_file Nom/chemin du fichier `.su` de sortie
195    *  \param[in] t_tolerance Pourcentage de tolérance de couleur
196    */
197   void compress(const char *t_input_file, const char *t_output_file,
198                 int32_t t_tolerance) {
199     Image *img;
200     darray *zones;
201     FILE *output_file;
202     if (!t_output_file) {
203       t_output_file = DEFAULT_COMPRESSED_NAME;
204     }
205     tolerance = t_tolerance;
206     img = newImage();
207     imageLoadPPM(t_input_file, img);
208     output_file = get_file(t_output_file, "wb");
209     zones = imgToZones(img);
210     write_compressed_file(img, output_file, zones);
211     deleteImage(img);
212     fclose(output_file);
213   }
```

uncompress.h

```c
/**
 *   \file uncompress.h
 *   \brief Décompression de fichiers
 *
 *  Ce fichier contient les déclarations des fonctions nécessaires à
 *  décompresser un fichier en `.su` généré par ce programme vers un fichier
 *  `.ppm` identique à l'original.
 *
 */

#ifndef SRC_UNCOMPRESS_H_
#define SRC_UNCOMPRESS_H_

#include "ppm.h"

#define DEFAULT_UNCOMPRESSED_FILE "output.ppm"

/// Lit les segments compressés dans une zone
void read_compressed_zones(FILE *t_file, darray *t_zones);
/// Lit les zones compressées dans le fichier d'entrée
void read_compressed_file_data(FILE *t_file, darray *zones);
/// Lit les premières données du fichier compressé
void read_compressed_file_meta(FILE *t_file, Image *t_img);
/// Décompresse le fichier d'entrée dans le fichier de sortie
void uncompress(const char *t_input_file, const char *t_output_file);

#endif /* SRC_UNCOMPRESS_H_ */
```

uncompress.c

```c
/**
 *   \file uncompress.c
 *   \brief Décompression de fichiers
 *
 *  Ce fichier contient l'implémentation des fonctions nécessaires à la
 *  décompression d'un fichier en `.su` généré par ce programme vers un fichier
 *  en `.ppm` identique à l'original.
 *
 */

#include "uncompress.h"

/**
 *  Lit le nombre de segments en `uint32_t` contenus dans la zone compressée,
 *  puis pour chaque \ref Segment ajoute ses données décompressées au tableau
 *  dynamique de \p t_zone.
 *
 *  \param[in] t_file Fichier d'entrée contenant les segments à lire
 *  \param[out] t_zone La zone dans laquelle stocker les \ref Segment
 */
void read_segments(FILE *t_file, Zone *t_zone) {
  uint64_t nb_segments, i;
  Segment *segment;
  fread(&nb_segments, sizeof(nb_segments), 1, t_file);
  for(i = 0; i < nb_segments; ++i) {
    segment = newSegment(0, 0);
```

```
27      fread(&segment->left_limit, sizeof(Segment), 1, t_file);
28      darrayPushBack(t_zone->segments, segment);
29    }
30  }
31
32  /**
33   *  Lit les données compressées du fichier d'entrée, tout d'abord le nombre de
34   *  zones dans un `uint42_t`, puis pour chaque zone lit les segments. Une fois
35   *  une zone lue, elle est ajoutée à \p t_zones.
36   *
37   *  \param[in] t_file Fichier d'entrée contenant les données compressées
38   *  \param[out] t_zones Tableau dynamique contenant les zones lues
39   */
40  void read_compressed_zones(FILE *t_file, darray *t_zones) {
41    uint64_t nb_zones, i;
42    Zone *zone;
43    /* read number of zones */
44    fread(&nb_zones, sizeof(nb_zones), 1, t_file);
45    for(i = 0; i < nb_zones; ++i) {
46      zone = newZone(0, 0, 0);
47      /* read RGB into the zone */
48      fread(&zone->red, sizeof(zone->red), 3, t_file);
49      /* read each segments of the zone and add them to their vector */
50      read_segments(t_file, zone);
51      /* add the zone to the zones of the image */
52      darrayPushBack(t_zones, zone);
53    }
54  }
55
56  /**
57   *  Lit les données basiques du fichier compressé, à savoir la taille en
58   *  abscisse et en ordonnée du fichier ppm d'origine et les inscrit dans \p
59   *  t_img.
60   *
61   *  \param[in] t_file Fichier d'entrée à lire
62   *  \param[out] t_img Structure \ref Image stockant les données lues
63   */
64  void read_compressed_file_meta(FILE *t_file, Image *t_img) {
65    /* read sizeX and sizeY at once */
66    fread(&t_img->sizeX, sizeof(t_img->sizeX), 2, t_file);
67  }
68
69  uint8_t *zones_to_data(darray *t_zones, Image *t_img) {
70    uint64_t nb_zones, nb_segments, zoneID, segmentID, k, left_limit, right_limit;
71    uint8_t *data, red, green, blue;
72    Zone *current_zone;
73    Segment *current_segment;
74    data = (uint8_t *)malloc(sizeof(uint8_t) * t_img->sizeX * t_img->sizeY * 3);
75    nb_zones = darraySize(t_zones);
76    for (zoneID = 0; zoneID < nb_zones; ++zoneID) {
77      current_zone = darrayGet(t_zones, zoneID);
78      red = current_zone->red;
79      green = current_zone->green;
80      blue = current_zone->blue;
81      nb_segments = darraySize(current_zone->segments);
82      for (segmentID = 0; segmentID < nb_segments; ++segmentID) {
83        current_segment = darrayGet(current_zone->segments, segmentID);
```

```
84        left_limit = current_segment->left_limit;
85        right_limit = current_segment->right_limit;
86        for (k = left_limit; k < right_limit; ++k) {
87          data[k * 3] = red;
88          data[k * 3 + 1] = green;
89          data[k * 3 + 2] = blue;
90        }
91      }
92    }
93    return data;
94  }
95
96  /**
97   *  Décompresse le fichier d'entrée et écrit son équivalent décompressé au
98   *  format ppm dans le fichier de sortie.
99   *
100  *  \param[in] t_input_file Nom/chemin du fichier d'entrée
101  *  \param[in] t_output_file Nom/chemin du fichier de sortie
102  */
103 void uncompress(const char *t_input_file, const char *t_output_file) {
104   Image *img;
105   darray *zones = darrayNew(sizeof(Zone));
106   uint8_t *data;
107   FILE *input_file;
108   uint64_t i;
109   if (!t_output_file) {
110     t_output_file = DEFAULT_UNCOMPRESSED_FILE;
111   }
112   img = newImage();
113   input_file = get_file(t_input_file, "rb");
114   read_compressed_file_meta(input_file, img);
115   read_compressed_zones(input_file, zones);
116   data = zones_to_data(zones, img);
117   imageSavePPM(t_output_file, img, data);
118   fclose(input_file);
119   /* free memory */
120   for(i = 0; i < darraySize(zones); ++i) {
121     Zone *zone = darrayGet(zones, i);
122     darrayDelete(zone->segments);
123   }
124   darrayDelete(zones);
125 }
```

ppm.h

```
1  /**
2   *   \file ppm.h
3   *   \brief Fichier d'en-tête pour les fonctions de manipulation d'images ppm
4   *
5   *  En-tête contenant la déclaration de fonctions de lecture et d'écriture de
6   *  fichiers au format ppm. La définition des fonction se trouve dans \ref ppm.c
7   *
8   */
9
10 #ifndef SRC_PPM_H_
11 #define SRC_PPM_H_
12
13 #include "utilities.h"
```

```
14
15  /// \brief Ouvre un fichier avec les autorisations demandées
16  FILE *get_file(const char *t_filename, const char *t_mode);
17  /// \brief Lit le format d'un fichier ppm ouvert
18  void read_file_format(FILE *t_fp, const char *t_filename);
19  /// \brief Vérifie et ignore d'éventuels commentaires du header d'un fichier
20  void check_for_comments(FILE *t_fp);
21  /// \brief Lit les dimensions du fichier ppm ouvert
22  void read_file_size(FILE *t_fp, Image *t_img, const char *t_filename);
23  /// \brief Lit et vérifie le format RGB du fichier ppm
24  void read_rgb(FILE *t_fp, const char *t_filename);
25  /// \brief Lit dans le conteneur les données images du fichier ppm
26  void read_data(FILE *t_fp, uint64_t t_size, uint8_t **t_data,
27                 const char *t_filename);
28  /// \brief Convertit les données brutes de fichier vers des conteneurs de pixels
29  void dataToImage(Image *t_img, uint8_t *t_data, uint64_t t_size);
30  /// \brief Convertit les pixels d'une image en tableau natif OpenGL
31  unsigned char *imageToData(Image *t_img);
32  /// \brief Ouverture et lecture de l'image d'entrée
33  int imageLoadPPM(const char *t_filename, Image *t_img);
34  /// \brief Ouverture et écriture de l'image de sortie
35  void imageSavePPM(const char *t_filename, Image *t_img, uint8_t *data);
36
37  #endif /* SRC_PPM_H_ */
```

ppm.c

```
1   /**
2    *   \file ppm.c
3    *   \brief Fichier de déclaration des fonctions de manipulation d'images ppm
4    *
5    *  Déclaration du corps des fonctions déclarées dans \ref ppm.h
6    */
7
8   #include "ppm.h"
9   #include <assert.h>
10
11  #define RGB_COMPONENT_COLOR 255
12  #define CREATOR "CL"
13
14  /**
15   *  \brief function description
16   *
17   *  Fonction d'ouverture de fichier selon le mode demandé. Si la fonction ne
18   *  peut pas ouvrir le fichier, elle arrête le processus qui renverra la valeur
19   *  `1`. En cas de succès, la fonction renverra un pointeur de fichier vers le
20   *  fichier ouvert.
21   *
22   *  \param[in] t_filename Nom du fichier à ouvrir
23   *  \param[in] t_mode Mode du fichier à ouvrir
24   *  \return Pointeur de fichier
25   */
26  FILE *get_file(const char *t_filename, const char *t_mode) {
27    FILE *fp = fopen(t_filename, t_mode);
28    if (!fp) {
29      fprintf(stderr, "Unable to open file '%s'\n", t_filename);
30      exit(FILE_IO_ERROR);
31    }
```

```c
  return fp;
}

/**
 *  Lit et vérifie le format du fichier passé en argument. Si le format n'est
 *  pas correct, la fonction arrête le processus qui renverra la valeur `1`.
 *
 *  \param[in] t_fp Fichier ppm où lire les données
 *  \param[in] t_filename Nom du fichier ouvert
 */
void read_file_format(FILE *t_fp, const char *t_filename) {
  char buff[16];
  if (!fgets(buff, sizeof(buff), t_fp)) {
    perror(t_filename);
    exit(FILE_IO_ERROR);
  }
  /* check file format */
  if (buff[0] != 'P' || buff[1] != '6') {
    fprintf(stderr, "Invalid image format (must be 'P6')\n");
    exit(FILE_FORMAT_ERROR);
  }
}

/**
 *  Vérifie si le header contient des commentaires et les ignore le cas échéant.
 *
 *  \param[in] t_fp Fichier ppm où lire les données
 */
void check_for_comments(FILE *t_fp) {
  char c;
  c = (char)getc(t_fp);
  while (c == '#') {
    while (getc(t_fp) != '\n') {
    }
    c = (char)getc(t_fp);
  }
  ungetc(c, t_fp);
}

/**
 *  Lit la taille des données image et les écrit dans le conteneur d'images
 *  passé en argument.
 *
 *  \param[in] t_fp Fichier ppm où lire les données
 *  \param[out] t_img Conteneur d'image où écrire les résultats
 *  \param[in] t_filename Nom du fichier ouvert
 */
void read_file_size(FILE *t_fp, Image *t_img, const char *t_filename) {
  if (fscanf(t_fp, "%lu %lu", &t_img->sizeX, &t_img->sizeY) != 2) {
    fprintf(stderr, "Invalid image size (error loading '%s')\n", t_filename);
    exit(FILE_FORMAT_ERROR);
  }
}

/**
 *  Vérifie le format RGB de l'image ppm. Si le format n'est pas correct, la
 *  fonction arrête le processus qui renverra la valeur `1`.
```

```
89   *
90   *  \param[in] t_fp Fichier ppm où lire les données
91   *  \param[in] t_filename Nom du fichier ouvert
92   */
93  void read_rgb(FILE *t_fp, const char *t_filename) {
94    char d;
95    int rgb_comp_color;
96    /* read rgb component */
97    if (fscanf(t_fp, "%d", &rgb_comp_color) != 1) {
98      fprintf(stderr, "Invalid rgb component (error loading '%s')\n", t_filename);
99      exit(FILE_FORMAT_ERROR);
100   }
101   fscanf(t_fp, "%c ", &d);
102   /* check rgb component depth */
103   if (rgb_comp_color != RGB_COMPONENT_COLOR) {
104     fprintf(stderr, "'%s' does not have 8-bits components\n", t_filename);
105     exit(FILE_FORMAT_ERROR);
106   }
107 }
108
109 /**
110  *  \brief function description
111  *
112  *  Lit les données images brutes du fichier ppm ouvert et les stocke dans \p
113  *  t_data.
114  *
115  *  \param[in] t_fp Fichier ppm ouvert source
116  *  \param[in] t_size Taille des données brutes
117  *  \param[out] t_data Pointeur vers le tableau de sortie des données brutes
118  *  \param[in] t_filename Nom du fichier d'entrée
119  *  \return Taille du tableau de données obtenu
120  */
121 void read_data(FILE *t_fp, uint64_t t_size, unsigned char **t_data,
122                const char *t_filename) {
123   *t_data = (unsigned char *)malloc(t_size * sizeof(unsigned char));
124   assert(*t_data);
125   /* read pixel data from file */
126   if (!fread(*t_data, (size_t)1, t_size, t_fp)) {
127     fprintf(stderr, "Error loading image '%s'\n", t_filename);
128     free(*t_data);
129     exit(FILE_IO_ERROR);
130   }
131 }
132
133 /**
134  *  Convertit vers un tableau de `unsigned char` les pixels contenus dans un
135  * conteneur d'image. La taille du tableau de `unsigned char` est la taille du
136  * tableau de pixels multipliée par trois du fait des trois emplacements séparés
137  * par couleur.
138  *
139  *  \param[out] t_img Image dont les pixels doivent être convertis
140  *  \param[in] t_data Données à convertir en structures \ref Pixel
141  *  \param[in] t_size Taille du tableau de `unsigned char`
142  */
143 void dataToImage(Image *t_img, uint8_t *t_data, uint64_t t_size) {
144   uint64_t i;
145   t_img->pixels = darrayNew(sizeof(Pixel));
```

```
146   for (i = 0; i < t_size; i += 3) {
147     darrayPushBack(t_img->pixels,
148                    newPixel(t_data[i], t_data[i + 1], t_data[i + 2]));
149   }
150 }
151
152 /**
153  *  Convertit le vecteur de pixels d'un conteneur d'image en un tableau de
154  *  valeurs de type `uint8_t` afin de permettre l'écriture d'une image dans un
155  *  fichier.
156  *
157  *  \param[in] t_img Conteneur d'image contenant les pixels à convertir
158  *  \return Tableau de pointeurs de `uint8_t`
159  */
160 uint8_t *imageToData(Image *t_img) {
161   Pixel *pixel;
162   uint8_t *data, size;
163   uint64_t i;
164   size = (uint8_t)darraySize(t_img->pixels);
165   data = (uint8_t *)malloc(3 * sizeof(uint8_t) * size);
166   for (i = 0; i < size; i += 3) {
167     pixel = darrayGet(t_img->pixels, i / 3);
168     data[i] = pixel->red;
169     data[i + 1] = pixel->green;
170     data[i + 2] = pixel->blue;
171   }
172   return data;
173 }
174
175 /**
176  *  Ouvre le fichier image avec son nom de fichier passé par le paramètre
177  *  `filename` et charge ses informations et données dans l'objet `img` dans
178  *  lequel les données et l'image seront manipulables. Retourne la valeur 1 en
179  *  cas de succès.
180  *
181  *  \param[in] t_filename Nom du fichier image à ouvrir
182  *  \param[out] t_img Objet \ref Image manipulable
183  *  \return Retourne 1 en cas de succès
184  */
185 int imageLoadPPM(const char *t_filename, Image *t_img) {
186   FILE *fp;
187   uint64_t size;
188   unsigned char *data = NULL;
189   fp = get_file(t_filename, "rb");       /* open PPM file for reading */
190   read_file_format(fp, t_filename);      /* read image format */
191   check_for_comments(fp);                /* check for comments */
192   read_file_size(fp, t_img, t_filename); /* read image size information */
193   read_rgb(fp, t_filename);              /* read rgb component */
194   size = t_img->sizeX * t_img->sizeY * 3;
195   read_data(fp, size, &data, t_filename); /* read data from file */
196   dataToImage(t_img, data, size);
197   free(data);
198   fclose(fp);
199   return 1;
200 }
201
202 /**
```

```
 *  Ouvre le fichier image avec son nom de fichier passé par le paramètre
 *  `filename` et y écrit les informations trouvées dans l'objet `img`.
 *
 *  \param[in] t_filename Nom du fichier image à ouvrir
 *  \param[in] t_img Objet \ref Image à écrire
 *  \param[in] t_data Données décompressées de l'image au format natif ppm
 */
void imageSavePPM(const char *t_filename, Image *t_img, uint8_t *t_data) {
  FILE *fp;
  fp = get_file(t_filename, "wb"); /* open file for output */
  /* write the header file */
  fprintf(fp, "P6\n");                                    /* image format */
  fprintf(fp, "# Created by %s\n", CREATOR);              /* comments */
  fprintf(fp, "%lu %lu\n", t_img->sizeX, t_img->sizeY); /* image size */
  fprintf(fp, "%d\n", RGB_COMPONENT_COLOR); /* rgb component depth */
  fwrite(t_data, (size_t)1, (size_t)(3 * t_img->sizeX * t_img->sizeY), fp);
  free(t_data);
  fclose(fp);
}
```

utilities.h

```
/**
 *   \file utilities.h
 *   \brief Déclaraction des structures de données et fonctions utilitaires.
 *
 *  Dans ce fichier sont déclarées et implémentées les structures qui serviront
 *  de conteneurs aux données manipulées. Sont également déclarées les fonctions
 *  utilitaires pour la manipulation de ces structures.
 */

#ifndef SRC_UTILITIES_H_
#define SRC_UTILITIES_H_

#include "darray.h"
#include <stdio.h>

/*****************************************************************************/
/*                              DEFINE DIRECTIVES                            */
/*****************************************************************************/

#ifdef Debug
#define DEBUG if (1)
#else
#define DEBUG if (0)
#endif

/*****************************************************************************/
/*                              STRUCT DECLARATION                           */
/*****************************************************************************/

struct Image;
typedef struct Image Image;
struct Pixel;
typedef struct Pixel Pixel;
struct Zone;
typedef struct Zone Zone;
struct Segment;
```

```c
typedef struct Segment Segment;

/*****************************************************************************/
/*                              STRUCT IMPLEMENTATION                        */
/*****************************************************************************/

/**
 *  \brief Conteneur d'une image
 *
 *  Une image est une structure définie par ses dimensions verticales et
 *  horizontales x et y, et contenant pour chacune des coordonnées possibles
 *  selon ses dimensions un pixel de type \ref Pixel. Ces pixels sont stockés
 *  dans un tableau dynamique \ref darray.
 */
struct Image {
  uint64_t sizeX; /*!< Largeur de l'image */
  uint64_t sizeY; /*!< Hauteur de l'image */
  darray *pixels; /*!< Vecteur à une dimention de \ref Pixel */
};

/**
 *  \brief Conteneur d'un pixel
 *
 *  Un pixel est défini par sa couleur représenté en RGB (rouge, vert, bleu).
 *  Il contient également une valeur booléenne afin de savoir si le Pixel fut
 *  visité précédemment par l'algorithme de compression.
 */
struct Pixel {
  uint8_t red;     /*!< Couleur rouge du pixel */
  uint8_t green;   /*!< Couleur verte du pixel */
  uint8_t blue;    /*!< Couleur bleue du pixel */
  uint8_t visited; /*!< Le pixel a-t-il été visité avant */
};

/**
 *  \brief Conteneur de zone de couleur unie
 *
 *  Une zone est un ensemble de pixels de même couleur ou de couleur similaire
 *  dont on conserve uniquement les marges dans le tableau dynamique.
 */
struct Zone {
  uint8_t red;      /*!< Couleur rouge de la zone */
  uint8_t green;    /*!< Couleur verte de la zone */
  uint8_t blue;     /*!< Couleur bleue de la zone */
  darray *segments; /*!< Vecteur de \ref Segment */
};

/**
 *  \brief Conteneur de segment de couleur unie
 *
 *  Un segment est un ensemble de pixels de même ordonnée et de couleur unie ou
 *  similaire. Il se définit par son ordonnée y et de ses deux pixels de bordure
 *  à son extrême droite et à son extrême gauche.
 */
struct Segment {
  uint32_t left_limit;  /*!< extrême gauche du segment  */
  uint32_t right_limit; /*!< extrême droit du segment */
```

```
};

/*****************************************************************************/
/*                          Utility functions declaration                    */
/*****************************************************************************/

/// \brief Création d'un nouveau pixel
Pixel *newPixel(uint8_t t_r, uint8_t t_g, uint8_t t_b);
/// \brief Création d'une nouvelle image
Image *newImage();
/// \brief Destructeur d'une image
void deleteImage(Image *t_self);
/// \brief Constructeur d'un segment de couleur unie
Segment *newSegment(uint32_t t_right_limit, uint32_t t_left_limit);
/// \brief Constructeur de conteneur de zone
Zone *newZone(uint8_t t_r, uint8_t t_g, uint8_t t_b);

#endif /* SRC_UTILITIES_H_ */
```

utilities.c

```
/**
 *   \file utilities.c
 *   \brief Implémentation des fonctions utilitaires
 *
 *  Dans ce fichier sont implémentées les fonctions utilitaires pour la
 *  manipulation des structures de données déclarées dans le fichier header
 *  correspondant.
 */

#include "utilities.h"

/**
 *  Créé un nouveau pixel initialisé avec les arguments `r`, `g` et `b` et
 *  renvoie un pointeur vers ce pixel créé.
 *
 *  \param[in] t_r Valeur rouge du pixel
 *  \param[in] t_g Valeur verte du pixel
 *  \param[in] t_b Valeur bleue du pixel
 *  \return Pointeur sur une structure de type \ref Pixel
 */
Pixel *newPixel(uint8_t t_r, uint8_t t_g, uint8_t t_b) {
  Pixel *res;
  res = (Pixel *)malloc(sizeof(Pixel));
  res->red = t_r;
  res->green = t_g;
  res->blue = t_b;
  res->visited = 0;
  return res;
}

/**
 *  Constructeur d'un conteneur d'image. Les dimensions sont initialisées à zéro
 *  (0) et son tableau de pixels a été créé et initialisé en tableau vide. Le
 *  constructeur renvoie un pointeur vers la nouvelle structure \ref Image.
 *
 *  \return Pointeur vers une structure \ref Image initialisée.
 */
```

```
38  Image *newImage() {
39    Image *res;
40    res = (Image *)malloc(sizeof(Image));
41    res->sizeX = 0;
42    res->sizeY = 0;
43    res->pixels = darrayNew(sizeof(Pixel));
44    return res;
45  }
46
47  /**
48   *  Destructeur d'un conteneur d'image. Le destructeur appellera le destructeur
49   *  du vecteur de pixels qui sera libéré de la mémoire, puis ultimement le
50   *  destructeur libérera la structure \ref Image pointée par le pointeur passé
51   *  en argument.
52   *
53   *  \param[in] t_self Conteneur d'image à détruire
54   */
55  void deleteImage(Image *t_self) {
56    darrayDelete(t_self->pixels);
57    free(t_self);
58  }
59
60  /**
61   *  Constructeur d'un conteneur de segment. Le constructeur va initialiser les
62   *  valeurs qu'il contiendra avec les arguments qui lui seront passés lors de
63   *  l'appel de la fonction.
64   *
65   *  \param[in] t_right_limit Abscisse extrême droite du segment
66   *  \param[in] t_left_limit Abscisse extrême gauche du segment
67   *  \return Pointeur sur un conteneur de segment
68   */
69  Segment *newSegment(uint32_t t_right_limit, uint32_t t_left_limit) {
70    Segment *res = (Segment *)malloc(sizeof(Segment));
71    res->left_limit = t_left_limit;
72    res->right_limit = t_right_limit;
73    return res;
74  }
75
76  /**
77   *  \brief function description
78   *
79   *  Constructeur de conteneur de zone, initialise grâce aux arguments la couleur
80   *  de la zone et initialise un tableau dynamique vide de \ref Segment. Renvoie
81   *  un pointeur vers la structure nouvellement créée.
82   *
83   *  \param[in] t_r Valeur rouge de la couleur de la zone
84   *  \param[in] t_g Valeur verte de la couleur de la zone
85   *  \param[in] t_b Valeur bleue de la couleur de la zone
86   *  \return Pointeur vers la structure créée
87   */
88  Zone *newZone(uint8_t t_r, uint8_t t_g, uint8_t t_b) {
89    Zone *res = (Zone *)malloc(sizeof(Zone));
90    res->red = t_r;
91    res->green = t_g;
92    res->blue = t_b;
93    res->segments = darrayNew(sizeof(Segment));
94    return res;
```

```
}
```

darray.h

```
/**
 * \file darray.h
 * \brief Implémentation de \ref darray et déclaration des fonctions pour ce
 * type
 */

#ifndef SRC_DARRAY_H_
#define SRC_DARRAY_H_

#include "errorcodes.h"
#include <stdint.h>
#include <stdlib.h>

/**
 * \struct darray
 * \brief Tableau dynamique
 *
 * Les objets `darray` offrent la possibilité d'avoir des tableaux à taille
 * variable en C, similairement aux objets `vector` en C++.
 */
typedef struct {
  void *begin; /*!< Pointeur sur le premier élément du tableau */
  void *end; /*!< Pointeur sur l'élément situé immédiatement après le dernier
                élément du tableau */
  uint64_t element_size; /*!< Taille des éléments stockés dans le tableau */
  uint64_t capacity;     /*!< Capacité maximale du tableau actuel */
} darray;

/// \brief Créé un nouvel objet \ref darray vide
darray *darrayNew(uint64_t element_size);
/// \brief Augmente la capacité d'un \ref darray
void darrayExtend(darray *self);
/// \brief Insère un élément à l'endroit pointé dans un \ref darray
void darrayInsert(darray *self, void *pos, void *elem);
/// \brief Supprime l'élément pointé dans l'objet \ref darray
void darrayErase(darray *self, void *pos);
/// \brief Retourne l'élément du \ref darray au idx-ème index
void *darrayGet(darray *self, uint64_t idx);
/// \brief Insère un élément à la fin de l'élément \ref darray
void darrayPushBack(darray *self, void *elem);
/// \brief Supprime le dernier élément de l'élément \ref darray
void darrayPopBack(darray *self);
/// \brief Détruit l'élément \ref darray
void darrayDelete(darray *self);
/// \brief Renvoie la taille de l'élément \ref darray
uint64_t darraySize(darray *self);
/// \brief Renvoie la taille de l'élément \ref darray
uint64_t darrayElemSize(darray *self);

#endif /* SRC_DARRAY_H_ */
```

darray.c

```
/**
 * \file darray.c
```

```
3   * \brief Implémentation des fonctions pour le type \ref darray
4   */
5
6   #include "darray.h"
7   #include <stdio.h>
8   #include <string.h>
9
10  /**
11   * `darrayNew` permet de créer un nouvel objet de type \ref darray ne
12   * contenant aucun élément. Le seul paramètre, `element_size`, est utilisé afin
13   * de connaître l'espace mémoire à allouer à chacun des éléments dans le
14   * tableau. Cela implique qu'un objet \ref darray ne peut contenir que des
15   * éléments du même type.
16   *
17   * \param[in] t_element_size Taille des objets stockés
18   * \return Pointeur sur le nouvel objet \ref darray
19   */
20  darray *darrayNew(uint64_t t_element_size) {
21    darray *ret;
22    ret = (darray *)malloc(sizeof(darray));
23    ret->begin = NULL;
24    ret->end = ret->begin;
25    ret->element_size = t_element_size;
26    ret->capacity = 0;
27    return ret;
28  }
29
30  /**
31   * `darrayInsert` insère l'élément `elem` avant l'élément pointé par `pos` dans
32   * l'objet \ref darray. Cela décalera tous les éléments stockés dans \a `self`
33   * pen d'un cran vers la fin du tableau et insérera à l'endroit pointé le nouvel
34   * élément. Cette fonction modifie les membres `begin` et `end` et
35   * potentiellement `capacity` de `self`.
36   *
37   * \param[in] t_self Objet \ref darray dans lequel on souhaite insérer un nouvel
38   * élément
39   * \param[in] t_pos Position à laquelle on souhaite insérer un nouvel élément
40   * \param[in] t_elem Élément que l'on souhaite insérer
41   */
42  void darrayInsert(darray *t_self, void *t_pos, void *t_elem) {
43    char *itr;
44    int64_t pos_aux;
45    pos_aux = (char *)t_pos - (char *)t_self->begin;
46    if (darraySize(t_self) >= t_self->capacity) {
47      darrayExtend(t_self);
48    }
49    itr = (char *)t_self->begin + pos_aux;
50    memmove(itr + t_self->element_size, itr, ((char *)t_self->end - itr));
51    memcpy(itr, t_elem, t_self->element_size);
52    (*t_self).end = (char *)t_self->end + t_self->element_size;
53  }
54
55  /**
56   *  Étend la capacité d'un \ref darray en réallouant sa mémoire, multipliant
57   *  sa capacité par deux. Si la réallocation mémoire ne réussit pas, le
58   *  programme s'arrête immédiatement, renvoyant la valeur de \ref PTR_ERROR
59   *
```

```
 *  \param[in] t_self L'objet darray à étendre
 */
void darrayExtend(darray *t_self) {
  void *new_array;
  uint64_t size;
  size = darraySize(t_self);
  new_array =
      realloc(t_self->begin, (darraySize(t_self) + 1) * t_self->element_size);
  if (!new_array) {
    fprintf(stderr, "Failed memory reallocation at %s:%d\nAborting...",
            __FILE__, __LINE__ - 2);
    exit(PTR_ERROR);
  }
  (*t_self).begin = new_array;
  ++(*t_self).capacity;
  (*t_self).end = (char *)t_self->begin + size * t_self->element_size;
}

/**
 * `darrayErase` supprime l'élément de objet \ref darray `self` pointé par
 * `pos`. Cela décalera tous les éléments suivants dans le tableau d'un cran
 * vers le début du tableau de manière à ce qu'il n'y aie pas d'élément vide
 * entre les membres `begin` et `end` de `self`. Par ailleurs, le membre `end`
 * de `self` se retrouve modifié par la fonction.
 *
 * \param[out] t_self Objet \ref darray dont on souhaite supprimer un élément
 * \param[in] t_pos Élément de `self` que l'on souhaite supprimer
 */
void darrayErase(darray *t_self, void *t_pos) {
  memmove(t_pos, (char *)t_pos + t_self->element_size,
          (((char *)t_self->end - t_self->element_size) - (char *)t_pos));
  (*t_self).end = (char *)t_self->end - t_self->element_size;
}

/**
 * `darrayPushBack` ajoute un nouvel élément `elem` à l'objet `self` à la fin du
 * tableau de ce dernier. Cette fonction modifie le membre `end` de `self`.
 *
 * \param[out] t_self Objet \ref darray à la fin duquel on souhaite ajouter un
 * nouvel élément
 * \param[in] t_elem Élément que l'on souhaite ajouter à la fin de `self`
 */
void darrayPushBack(darray *t_self, void *t_elem) {
  darrayInsert(t_self, t_self->end, t_elem);
}

/**
 * `darrayPopBack` permet de supprimer le dernier élément de l'objet \ref
 * darray passé en argument. Cette fonction modifie le membre `end` de ce
 * dernier objet.
 *
 * \param[out] t_self Objet dont on souhaite supprimer le dernier élément
 */
void darrayPopBack(darray *t_self) {
  darrayErase(t_self, (char *)t_self->end - t_self->element_size);
}

```

```c
/**
 * `darrayDelete` supprime tous les éléments contenus par l'objet \ref darray
 * passé en argument avant de libérer la mémoire occupée par l'objet lui-même.
 * L'objet passé en argument ne sera plus utilisable après utilisation de cette
 * fonction.
 *
 * \param[out] t_self Objet \ref darray à supprimer
 */
void darrayDelete(darray *t_self) {
  free(t_self->begin);
  free(t_self);
}

/**
 * `darraySize` renvoie le nombre d'éléments contenu dans le \ref darray
 * `self` passé en arguments. Cette fonction ne modifie pas l'élément passé en
 * argument.
 *
 * \param[out] t_self Objet \ref darray dont on souhaite connaître le nombre
 * d'éléments
 * \return Nombre d'éléments contenus dans `self`
 */
uint64_t darraySize(darray *t_self) {
  return (uint64_t)((char *)t_self->end - (char *)t_self->begin) /
         t_self->element_size;
}

/**
 * `darrayGet` permet de récupérer un élément d'un objet \ref darray grâce à
 * son index dans le tableau de l'objet `self`. Si l'index est trop grand, alors
 * le pointeur `NULL` sera renvoyé, sinon un pointeur de type `void*` pointant
 * sur l'élément correspondant sera renvoyé. Cette fonction ne modifie pas
 * l'objet `self`.
 * \param[out] t_self Objet \ref darray duquel on souhaite obtenir un pointeur
 * sur l'élément à l'index `idx`
 * \param[in] t_idx Index de l'élément que l'on souhaite récupérer
 * \return Pointeur de type `void*` pointant sur l'élément si l'index est
 * valide, sur NULL sinon.
 */
void *darrayGet(darray *t_self, uint64_t t_idx) {
  if (t_idx >= darraySize(t_self)) {
    fprintf(stderr, "Error in `darrayGet`, out of bound idx: %zu (max: %zu)\n",
            t_idx, darraySize(t_self));
    exit(PTR_ERROR);
  }
  void *itr;
  itr = (char *)t_self->begin + t_idx * t_self->element_size;
  return itr;
}
```