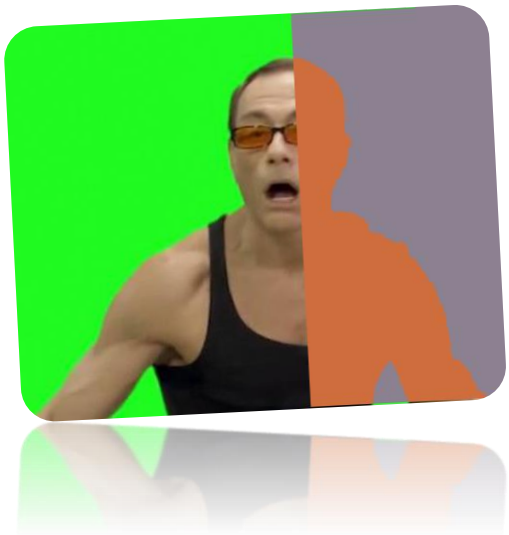
2020/2021

Rapport de Projet

Ligne de partage des eaux



Fait par : EL MOUSTAPHA Mohamed et UDHAYALINGAM Apiram
MATIERE : STRUCTURE DE DONNEES
ENCADRE PAR : M.CHAUSSARD

Table des matières

Introduction

Dans le cadre de notre première année du cycle d'ingénieur informatique à Sup Galilée, nous avons réalisé ce rapport de projet qui concerne le projet final de la matière Structure de Données. L'objectif est de mettre en application nos nouvelles connaissances sur les structures de données qu'on a étudiées telles que les listes, les tableaux à 2 dimensions et les tables de hachage.

Introduction du projet

Le but du projet est de construire un programme pour dessiner la ligne de partage des eaux sur une image. La ligne de partage est un algorithme de segmentation d'une image qui dessine les contours d'un objet.
Pour arriver à construire la ligne, on doit d'abord calculer le gradient de l'image, puis avec l'aide d'une image où seront représentés des points en dehors de l'objet et des points à l'extérieur de l'objet, l'algorithme tentera de trouver la limite la plus précise possible de l'objet en question.

Dans ce rapport, on parlera dans une grande partie de l'implémentation. Puis, on évoquera les résultats avant de s'intéresser à quelques ajouts possibles au programme.

I - Implémentation du projet

Avant de commencer la programmation, on devait définir la structure de manipulation d'une image et de ses fonctions. Pour cela, on va partir d'une bibliothèque de manipulation d'image PNG : lodepng.

1 - Structure de données : myimage

Dans la bibliothèque lodepng, la représentation de l'image sont juste des variables individuelles dont une, qui est un tableau 1D, où est stocké 4 paramètres pour chaque pixel. Les paramètres sont la quantité de rouge, de bleu, de vert et de transparence avec des valeurs variant entre 0 et 255. Etant donné qu'on manipule une image en noir et blanc, la quantité des couleurs rouge, vert, bleu sont les mêmes et on fixe la transparence à 255. Par conséquent, dans notre nouvelle structure on pourra faire un tableau avec un seul paramètre pour représenter l'image. On insérera également dans notre structure un champ hauteur et largeur et nous disposerons de tous les paramètres nécessaires à la manipulation.

Résultat nous obtenons cette structure :

```
typedef struct myimage{
  unsigned hauteur;
  unsigned largeur;
  unsigned char** couleur;
} myimage;
```

- ***hauteur*** : champ pour la hauteur de l'image
- ***largeur*** : champ pour la largeur de l'image
- ***couleur*** : champ qui contiendra un pointeur vers un tableau de pointeur pointant à leur tour vers un autre tableau de unsigned char (type prenant uniquement des valeurs entre 0 et 255 incluses)

2 - Fonction de manipulation du type myimage

Pour manipuler notre structure, nous aurons besoin de ces fonctions :

- une fonction permettant de lire une image en format PNG et de le convertir en une variable manipulable qui est le type myimage.
- une fonction permettant d'écrire une image en type myimage et de le convertir en format PNG.
- une fonction d'allocation mémoire pour le tableau du champ couleur dans le type myimage.
- une fonction de libération d'allocation mémoire pour le tableau du champ couleur dans le type myimage.

- Fonction de lecture d'une image :

```
myimage LireImage(const char* nom_fichier)
```

Cette fonction permet de convertir une image format PNG dans une variable de type myimage.
Pour cela, nous devons d'abord lire une image PNG, on utilise donc la bibliothèque lodepng.

```
error = lodepng_decode32_file(&image, &width, &height, nom_fichier);
if(error)
  printf("error %u: %s\n", error, lodepng_error_text(error));
```

- ***width*** est la variable qui contiendra suite à ces lignes la largeur de l'image
- ***height*** est la variable qui contiendra suite à ces lignes la hauteur de l'image
- ***image*** sera un tableau 1D de taille 4 × width × height où sera stockée successivement la quantité de rouge, vert, bleu et de transparence et ceci pour chaque pixel.

Ensuite, nous devons convertir ces données dans notre structure ***myimage***.

```
im = AllouerImage(height,width);

for(i = 0; i < height; i++)
  for(j = 0; j< width; j++)
    im.couleur[i][j] = image[4*(i*width + j)];

free(image);
```

On déclare une variable de type ***myimage*** se nommant ***im***. On alloue l'espace nécessaire et les champs hauteur et largeur avec la fonction ***AllouerImage*** (fonction qu'on explique plus tard dans le rapport) pour notre tableau 2D.
Il reste plus qu'à initialiser chaque case avec la valeur du pixel (i,j). C'est ce qu'on fait avec la double boucle for. On parcourt ***image*** de la façon suivante image[4*(i*width+j)] pour sélectionner la valeur du premier paramètre (quantité de rouge) pour le pixel (i,j) et on la stocke dans le champ couleur dans la j-ème case du i-ème tableau du tableau de pointeur.
On a plus qu'à retourner ***im*** sans oublier de libérer la mémoire allouée pour le tableau ***image***.

- Fonction d'écriture d'une image

```
void EcrireImage(myimage im, const char* nom_fichier)
```

Cette fonction permet de convertir une variable de type myimage en une image format PNG en noir et blanc.

Pour cela, on va utiliser une fonction de la bibliothèque de lodepng. Mais pour cela, on doit d'abord passer de notre structure à un tableau ***image*** 1D.

```
uint32_t largeur = im.largeur, hauteur = im.hauteur;
image = malloc(4*largeur*hauteur*sizeof(unsigned char));

for(i = 0; i < hauteur; i++){
  for(j = 0; j < largeur; j++){
    image[4*(i*largeur + j)] = im.couleur[i][j];
    image[4*(i*largeur + j)+1] = im.couleur[i][j];
    image[4*(i*largeur + j)+2] = im.couleur[i][j];
    image[4*(i*largeur + j)+3] = 255;
  }
}
```

On déclare des variables ***largeur*** et ***hauteur*** où on stocke leur valeur respective. On alloue la mémoire nécessaire pour ***image***. Il faut maintenant remplir le tableau ***image***. On sait que chaque pixel à 4 paramètres (quantité de rouge, de vert, de bleu et de transparence), que la quantité de RVB est la même pour une image en noir et blanc et enfin qu'on fixe la transparence à la valeur 255. On remplit le tableau avec un double ***for***.
A présent, on doit utiliser une fonction de lodepng pour convertir le tableau ***image*** en une image en format PNG.

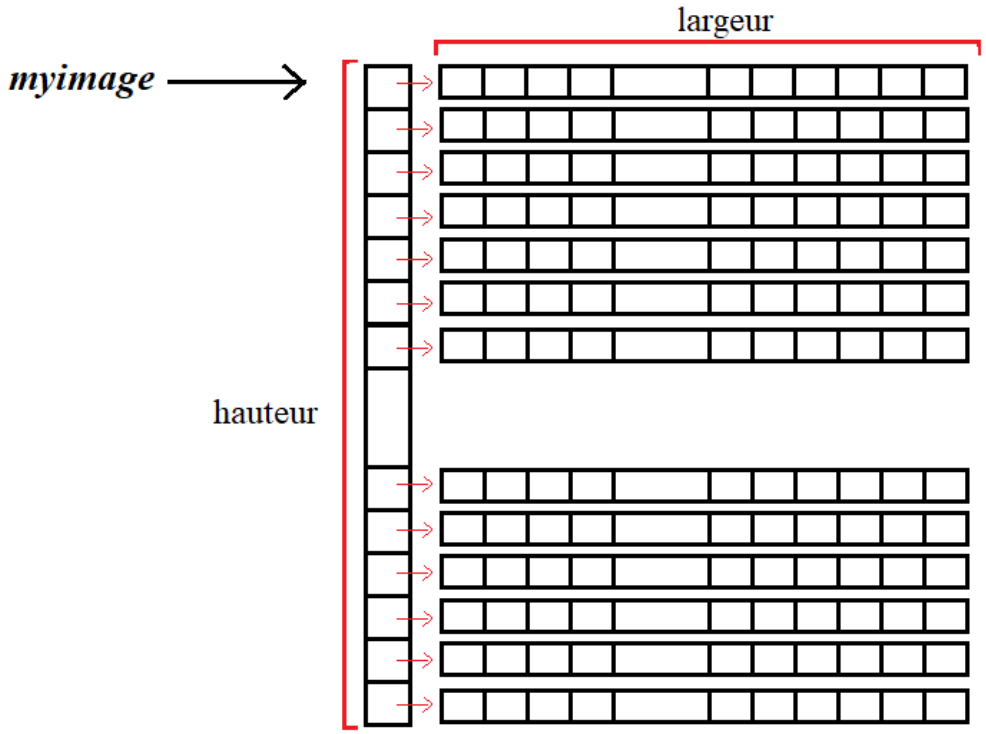
```
unsigned error = lodepng_encode32_file(nom_fichier, image, largeur,
hauteur);
if(error)
  printf("error %u: %s\n", error, lodepng_error_text(error));

free(image);
```

La fonction lodepng_encode32_file permet cette conversion. On n'oublie pas de libérer le tableau ***image*** et l'écriture est terminée.

- Fonction d'allocation mémoire du type ***myimage***

```
myimage AllouerImage(unsigned hauteur, unsigned largeur)
```

Cette fonction permet d'initialiser une variable ***myimage*** en allouant de la mémoire d'un tableau de pointeur de longueur "hauteur" et de "hauteur" tableau de type unsigned char de longueur "largeur". On perçoit l'utilisation du tableau de cette façon :



Cette fonction retourne une variable de type ***myimage***.

```
im.couleur = malloc(hauteur*sizeof(unsigned char*));
if(im.couleur == NULL){
  perror("Allocation mémoire");
  assert(0);
}

for(i = 0; i < hauteur; i++)
{
  im.couleur[i] = calloc(largeur,sizeof(unsigned char));
  if(im.couleur == NULL){
    while(i>0){
      free(im.couleur[i-1]);
      i--;
    }
    free(im.couleur);
    perror("Allocation mémoire");
    assert(0);
  }
}
```

On alloue d'abord un tableau de pointeur de ***hauteur*** cases puis on alloue pour chaque case du premier tableau, un tableau de ***largeur*** cases de type ***unsigned char*** qui sont initialisées à 0.
Remarque : Si on n'obtient pas de pointeur qui pointe vers une zone mémoire allouée, on libère la zone mémoire des précédents tableaux.

Il nous reste plus qu'à initialiser les champs ***hauteur*** et ***largeur*** de la variable ***im***.

```
im.hauteur = hauteur;
im.largeur = largeur;
```

Et on retourne ***im***.

- Fonction de libération de la mémoire allouée

```
void LibererImage(myimage im)
```

Cette fonction permet de libérer la mémoire allouée par la variable im. Pour libérer la mémoire allouée, il suffit juste de libérer la mémoire de chaque tableau.

```
for(i = 0; i < im.hauteur; i++)
  free(im.couleur[i]);
free(im.couleur);
```

On libère d'abord les tableaux de ***unsigned char*** puis le tableau de pointeur.

3 - Fonction Gradient

A présent qu'on a pu convertir une image PNG en une structure ***myimage*** qui nous est manipulable. On peut s'attaquer aux fonctions permettant de calculer le gradient d'une image.

- Fonction CalculerGradient

Le gradient se calcule en recherchant la valeur minimale et maximale des pixels appartenant au voisinage d'un pixel (i,j) pour un certain rayon. Puis la valeur du gradient du pixel (i,j) est égale à max-min.

```
myimage CalculerGradient(myimage im, uint32_t r);
```

Cette fonction permet de renvoyer une image représentant le gradient de l'image prise en argument, en fonction du rayon pris en argument.

```
uint32_t i,j;
myimage gradient = AllouerImage(im.hauteur,im.largeur);
for(i = 0; i < im.hauteur; i++)
  for (j = 0; j < im.largeur; j++)
    gradient.couleur[i][j] = CalculerPixel(im,i,j,r);
```

On crée une nouvelle image nommée gradient (de la même dimension que l'image prise en argument). Ensuite, on détermine les caractéristiques de cette image de la manière suivante ;

Pour chaque pixel, on fait appel à la fonction ***CalculerPixel*** pour déterminer la valeur du gradient d'un pixel (i,j). Suite à cet appel, le pixel prend comme valeur la différence du max et du min des valeurs de couleur de ses pixels voisins situés dans un rayon r.

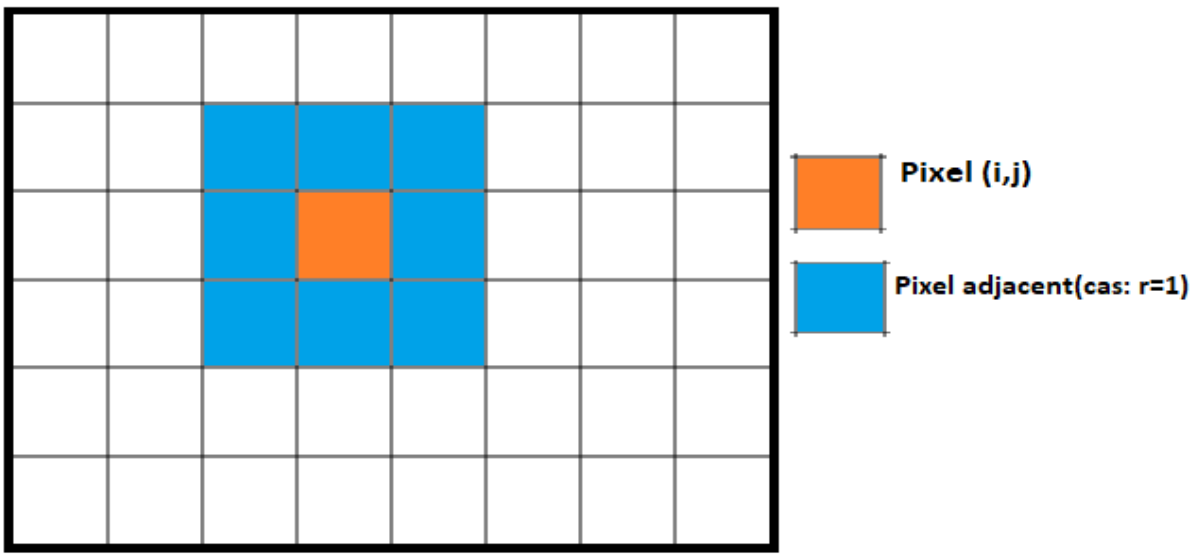
- Fonction CalculerPixel

```
unsigned char CalculerPixel( myimage im, uint32_t i, uint32_t j,
uint32_t r)
```

Cette fonction permet de calculer le gradient d'un pixel en fonction d'un rayon r.

```
k = i-r;
while(k <= i+r && (max != 255 || min != 0))
{
  l = j-r;
  while(l <= j+r && (max != 255 || min != 0))
  {
    if(k >= 0 && k < im.hauteur && l >= 0 && l < im.largeur)
    {
      if(im.couleur[k][l] < min)
        min = im.couleur[k][l];
      if(im.couleur[k][l] > max)
        max = im.couleur[k][l];
    }
    l++;
  }
  k++;
}
```

Image



On parcourt les pixels situés dans un rayon r du pixel (i,j). Lors de ce parcours, on enregistre la valeur maximale et la valeur minimale rencontrée. Ensuite, on renvoie la différence de ces deux valeurs.
Une autre condition d'arrêt qu'on a rajouté par la suite est lorsque max = 255 et min = 0, on arrête la recherche. En effet, il est inutile de continuer à parcourir la zone si on ne peut pas trouver mieux (un pixel a une valeur entre 0 et 255).

4 - Fonction Ligne de Partage des Eaux (LPE)

A présent, qu'on a obtenu le gradient de l'image, on peut calculer la LPE.

Explication de l'algorithme :

Pour appliquer l'algorithme de ligne de partage des eaux et extraire un objet précis de l'image, il faut ouvrir l'image de marqueurs associée à l'image. Dans cette image, les pixels valant 100 représentent les pixels hors de l'objet, les pixels à 200 représentent les pixels dans l'objet, et les pixels à 0 sont ceux dont nous ne savons pas s'ils sont à l'intérieur ou à l'extérieur de l'objet. Le but de l'algorithme de ligne de partage des eaux est d'attribuer la valeur 100 ou 200 à tous les pixels de l'image de marqueurs valant 0 (selon qu'ils soient considérés comme à l'intérieur ou à l'extérieur de l'objet).

```
void CalculerLPE(myimage im_gradiee, myimage im_marqueur);
```

Cette fonction permet de faire une image représentant la ligne de partage des eaux de l'image. Le résultat est retrouvable dans ***im_marqueur***.
Dans cette fonction, nous utiliserons une table de hachage que nous pouvons retrouver dans la bibliothèque hachage (bibliothèque créée par nous-même).
Avant d'expliquer la fonction CalculerLPE, procédons à quelques explications brèves des fonctions de la bibliothèque de hachage.

4.1 - Explication du choix de la table de hachage

On a choisi de traiter le problème avec une table de hachage car cette dernière s'adapte le plus à notre problème. En effet, si on avait utilisé une liste chaînée pour classer toutes les coordonnées de chaque pixel en fonction de leur gradient, il serait beaucoup plus coûteux de la parcourir pour ordonner ces derniers. Puisqu'on doit parcourir toute la liste pour ordonner les pixels. Tandis qu'avec une table de hachage, il devient beaucoup plus pratique et peu coûteux d'ordonner ces éléments de la table. Puisqu'il suffit d'ajouter les éléments, grâce à la valeur du gradient dans la liste correspondante sans parcourir cette liste (pas comme on le ferait avec la liste triée). Il suffit donc de créer une table de hachage dont le tableau pointe vers 256 listes contenant des coordonnées de pixels partageant le même gradient (gradient 0 : tab[0] contient {p1, …}).

4.2 - Fonction de la bibliothèque de la table de hachage

Dans cette partie, nous parlerons uniquement des fonctions qui nous seront utiles pour comprendre le code de notre projet. Nous ne parlerons pas du code pour faire une liste (cf. aux fichiers hachage.c et hachage.h pour en savoir plus).
Pour commencer, nous devons choisir une structure.

```
typedef struct coord{
  uint32_t i;
  uint32_t j;
}coord;
```

Comme nous devons stocker des couples, nous avons opté pour une structure qui puisse stocker des coordonnées, d'où la structure ***coord***.

Une structure que nous utiliserons pour la table de hachage est celle-ci :

```
typedef struct tab_hachage{
 liste **tab;
 uint32_t taille;
} tab_hachage;
```

- ***liste*** : champ qui contiendra un pointeur qui pointe un pointeur vers un tableau de liste
- ***taille*** : champ qui contient le nombre de liste

3 fonctions sont importantes pour notre projet :

- la fonction ***hachage***
- la fonction ***add_tete_th***
- la fonction ***rechercher_min***

- Fonction de hachage

```
  unsigned char hachage(coord d,myimage imagegradiee)
```

Cette fonction permet de retourner l'indice de l'emplacement pour ranger le couple d dans la table de hachage avec l'aide de l'image gradient.

```
return  imagegradiee.couleur[d.i][d.j];
```

On se sert des valeurs de l'image gradient pour trier notre couple dans la table de hachage (explication de choix de trie dans la 2ème partie : programme en général). On sélectionne en indice pour le choix de la liste, la valeur du pixel gradient à l'emplacement (d.i,d.j).

- Fonction d'ajout d'un élément dans la table de hachage

```
  void add_tete_th( tab_hachage *th, coord d, myimage imagegradiee)
```

Cette fonction permet d'ajouter un couple de coordonnées dans une des listes en fonction de la valeur du gradient du pixel de ces coordonnées.

```
liste_ajoutTete(th->tab[hachage(d,imagegradiee)],d);
```

Pour cela, on appelle la fonction ***hachage***, qui retourne la liste dans lequel on placera nos coordonnées. Et enfin, on appelle la fonction ***liste_ajoutTete***, qui ajoute à la tête de la liste, dont l'indice a été retourné par la fonction ***hachage***, les coordonnées.

- Fonction de recherche du couple (i,j) ayant une plus faible valeur de gradient

```
   coord rechercher_min( tab_hachage *th)
```

Cette fonction sert à renvoyer et retirer de la liste, le couple (i,j) ayant la valeur du pixel gradient la plus faible.
Pour cela, comme le tableau de liste est trié (explication dans la seconde partie), on fouille d'abord les premières listes qui correspondent aux valeurs des gradients : 0, 1, etc. Puis, lorsqu'on trouve le couple dans une des listes, on le retire de la liste et on le renvoie.

```
uint32_t i = 0;
while(est_vide(th->tab[i]))
  i++;
return liste_retirerTete(th->tab[i]);
```

4.3 - Fonction CalculerLPE

A présent, passons à l'explication de notre programme ***CalculerLPE***.
Le programme va d'abord ajouter, dans une liste triée L, les pixels (i,j) qui appartiennent aux pixels marqueurs dans l'image marqueur. Ensuite, tant qu'il existe un pixel de l'image marqueur dont ses voisins ne sont pas traités, on appliquera ceci :
On regarde ses voisins. Si le voisin n'est pas un pixel de l'image marqueur, alors ce voisin devient un pixel marqueur et prend la même valeur que (i,j) et on l'ajoute dans la liste à traiter.
Une fois que tous les pixels ont été traités, on retrouve une image avec des zones qui décrivent précisément l'objet qu'on voulait obtenir.

```
coord d,aux;
uint32_t k,l;
tab_hachage* L = new_tab_hachage(256);
for(d.i = 0; d.i < im_marqueur.hauteur; d.i++){
  for(d.j = 0; d.j < im_marqueur.largeur; d.j++){
    if (im_marqueur.couleur[d.i][d.j] != 0)
      add_tete_th(L,d,im_gradiee);
  }
}
```

D'abord, nous déclarons et initialisons une table de hachage avec 256 listes dans cette table. On fait 256 car pour chaque liste, il y aura un ensemble de couples (i,j) qui correspondent au pixel (i,j) dont la valeur de son gradient est de la k-ème liste. Or la valeur d'un pixel varie entre 0 et 255, soit 256 valeurs de pixel.
On ajoute dans la table de hachage, tous les pixels qui appartiennent à l'ensemble des pixels de l'image marqueur en fonction de la valeur du gradient de chaque pixel.

```
while (!th_est_vide(L))
{
  d = rechercher_min(L);
  for(k = d.i-1; k <= d.i+1; k++){
    for(l = d.j-1; l <= d.j+1; l++){
      if(k >= 0 && k < im_marqueur.hauteur && l >= 0 && l <
im_marqueur.largeur)
      {
        if(im_marqueur.couleur[k][l] == 0){
          aux.i = k;
          aux.j = l;
          im_marqueur.couleur[k][l] = im_marqueur.couleur[d.i][d.j];
          add_tete_th(L,aux,im_gradiee);
        }
      }
    }
  }
}
```

Tant que la table de hachage n'est pas vide, on fait ça :
On retire un pixel (i,j) tel qu'il respecte gradient(i,j) ≤ gradient(x,y), ∀(x,y) ∈ L. On regarde ensuite les voisins autour. On vérifie bien que le voisin (k,l) existe et s'il est de valeur 0 dans l'image marqueur alors on donne la valeur du marqueur du pixel (i,j) et enfin, on ajoute le pixel (k,l) dans la table de hachage.
Lorsque la table de hachage est vide. Cela signifie que tous les pixels de l'image ont été traités. On peut retrouver le résultat dans l'image marqueur qui aura été modifié au fil de l'algorithme.
On n'oubliera pas de libérer la table de hachage.

```
free_hachage(L);
```

5 - Fonction Main

Voici les instructions présentent dans le main :

```
myimage image;
myimage im_gradiee;
myimage im_marqueur;

image = LireImage(argv[1]);
im_marqueur = LireImage(argv[2]);
im_gradiee = CalculerGradient(image,1);
CalculerLPE(im_gradiee,im_marqueur);
EcrireImage(im_marqueur,argv[3]);

LibererImage(image);
LibererImage(im_gradiee);
LibererImage(im_marqueur);
```

argv[1] correspond au nom de l'image à traiter, argv[2] correspond au nom de l'image marqueur qui lui correspond et argv[3] est l'image résultante.

On lit l'image à traiter et son image marqueur, avant de calculer le gradient de cette première. Ensuite, on calcule la ligne de partage des eaux de l'image intermédiaire. La fonction ***CalculerLPE*** prend alors l'image gradient et l'image marqueur comme argument et nous renvoie l'image traitée. On écrit cette dernière grâce à la fonction ***ÉcrireImage***. Enfin, on libère les espaces de mémoire alloués.

II - Résultat et ajout intéressant

1 - Résultat de notre programme

Nous avons essayé sur plusieurs images et nous obtenons bien ce qu'on voulait.

Exemple : On sépare la route de la forêt

Image de la route initiale

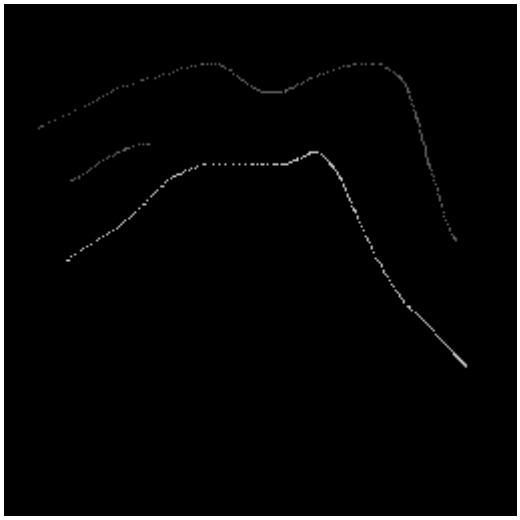


Image marqueur de la route

Résultat de la ligne de partage

Nous obtenons en gris foncé, la zone correspondant à la forêt et en gris clair la route.

2 - Détection de plusieurs objets

Notre programme fonctionne également pour la détection de plusieurs objets. Il suffit juste de mettre une couleur pour chaque objet.

Exemple : On détecte un chien et un chat avec 2 couleurs différentes

Image du chien et du chat initiale

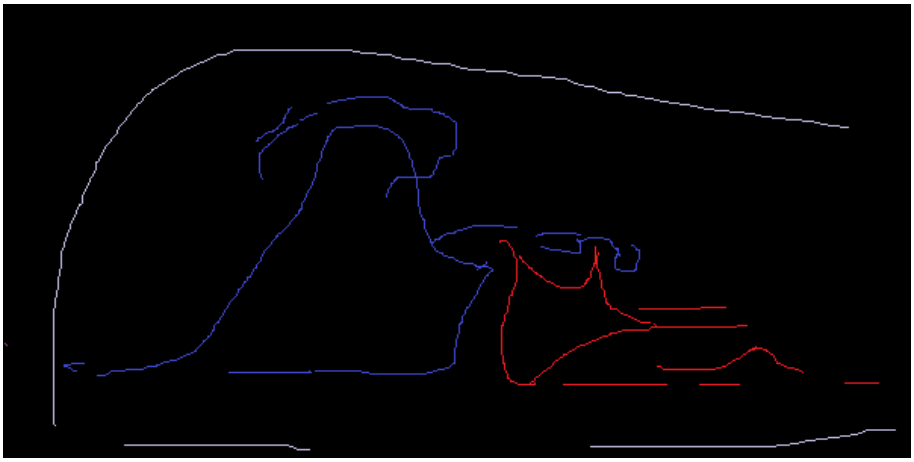


Image marqueur du chien et du chat

Image de la ligne de partage du chat et du chien

La détection fonctionne, cependant, elle n'est pas aussi précise qu'on voudrait. Le problème pourrait probablement venir de l'image marqueur à cause de pixels qui ne devrait pas apparaître à cet emplacement. On pourrait faire un algorithme qui détecte automatiquement où placer les pixels intérieurs et extérieurs pour régler ce souci. Mais globalement, on peut reconnaître le chien et le chat.

Voici un autre exemple de résultat :

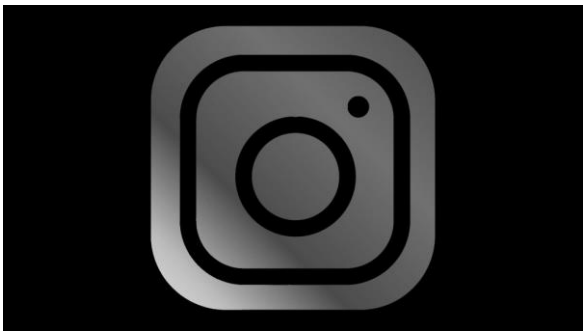


Image logo Instagram

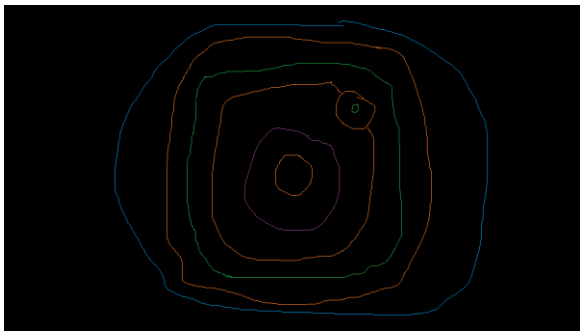


Image marqueur du logo Instagram

Image logo Instagram après la détection de contour

Ici, le programme a bien fonctionné.

3 - Présenter les résultats en couleur

Pour créer des couleurs aléatoires, il faut créer une fonction identique à la fonction ***EcrireImage*** mais avec certaines modifications.

```
void EcrireImageCouleur(myimage im, const char* nom_fichier)
```

Cette fonction permet de convertir une variable de type ***myimage*** en une image format PNG en couleur.

```
_Bool valeur[256] = {0};
unsigned char r[256],v[256],b[256];

for(i = 0; i < hauteur; i++){
  for(j = 0; j < largeur; j++){
    if(valeur[im.couleur[i][j]] == 0){
      valeur[im.couleur[i][j]] = 1;
    }
  }
}
for(i = 0; i < 256; i++){
  if(valeur[i]){
    r[i] = rand()%256;
    v[i] = rand()%256;
    b[i] = rand()%256;
  }
}
```

On va d'abord repérer les valeurs disponibles sur l'image avec un tableau booléen où à l'i-ème valeur 1 signifie qu'on a cette valeur sinon 0. Puis, on doit attribuer pour chaque valeur de pixel, 3 valeurs qui seront respectivement la quantité de rouge, de vert et de bleu et ceci de manière aléatoire. La probabilité d'obtenir la même couleur étant faible, nous ne vérifions pas que nous avons 2 pixels de couleur identique.

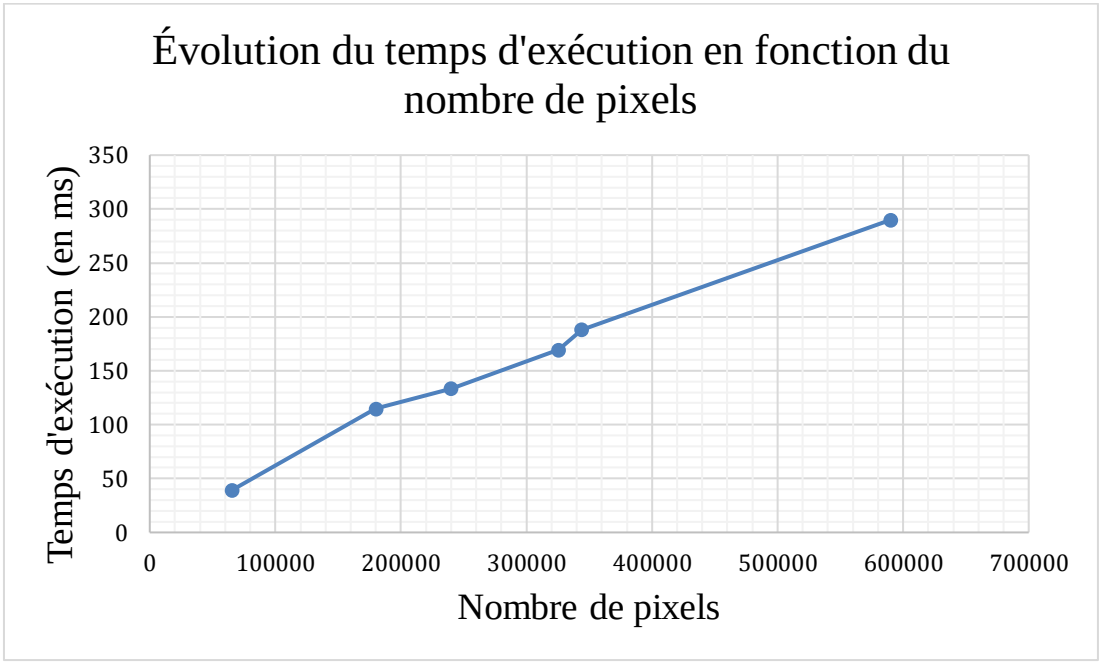
```
for(i = 0; i < hauteur; i++){
  for(j = 0; j < largeur; j++){
    image[4*(i*largeur + j)] = r[im.couleur[i][j]];
    image[4*(i*largeur + j)+1] = v[im.couleur[i][j]];
    image[4*(i*largeur + j)+2] = b[im.couleur[i][j]];
    image[4*(i*largeur + j)+3] = 255;
  }
}
```

On attribue les pixels colorés dans l'image finale.

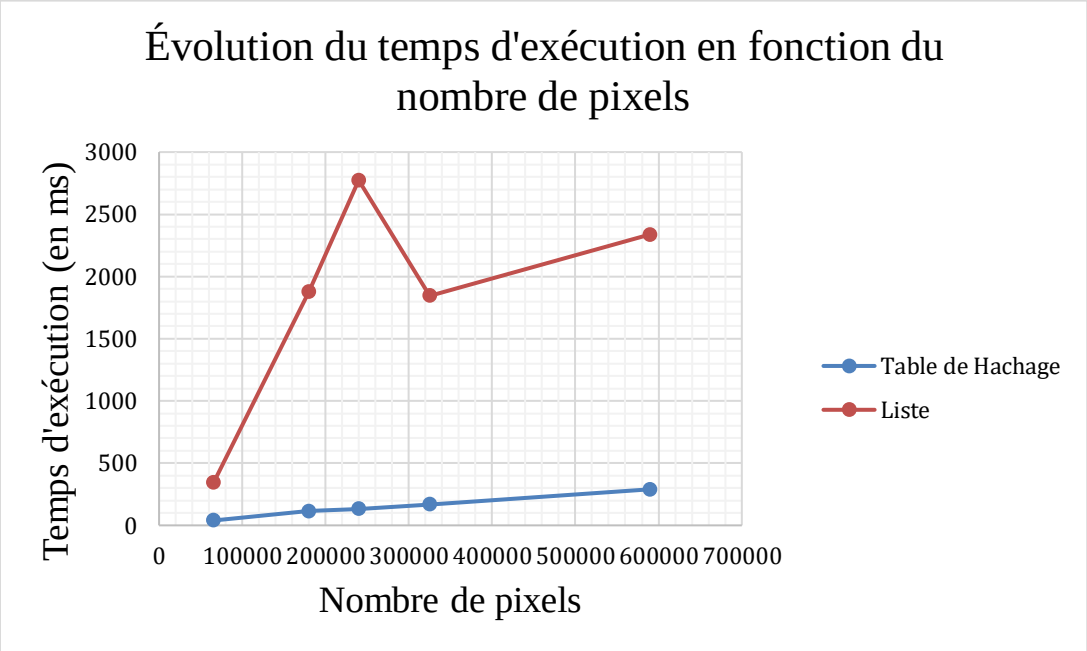
Exemples : image de la route et chien/chat avec les zones colorées aléatoirement

4 - Performance du programme

Nous avons étudié le temps d'exécution en fonction du nombre de pixels. Voici une courbe représentant le temps d'exécution d'une image en fonction du nombre de pixels de cette image.



Nous voyons sur cette courbe que notre programme est d'une complexité temporelle linéaire.
Nous avons essayé une version avec les listes chaînées triées.



On remarque que la structure de la liste est bien moins performante que celle de la table de hachage. De plus, sa complexité temporelle n'est pas linéaire

5 - Tracer le contour des objets

Il n'est pas compliqué de ressortir le contour des objets puisqu'il suffit juste de tracer d'une couleur spécifique les pixels dont les voisins aux alentours n'ont pas tous la même couleur.

```
void TracerLPE(myimage im, myimage im_LPE,const char* nom_fichier)
```

Cette fonction permet de tracer dans l'image ***im***, le contour des objets.
La fonction fonctionne comme ***EcrireImage*** mais avec certaines modifications.

```
for(i = 0; i < hauteur; i++){
  for(j = 0; j < largeur; j++){
    if(est_LPE(im_LPE,i,j)){
      image[4*(i*largeur + j)] = 255;
      image[4*(i*largeur + j)+1] = 0;
      image[4*(i*largeur + j)+2] = 0;
    }
    else{
      image[4*(i*largeur + j)] = im.couleur[i][j];
      image[4*(i*largeur + j)+1] = im.couleur[i][j];
      image[4*(i*largeur + j)+2] = im.couleur[i][j];
    }
    image[4*(i*largeur + j)+3] = 255;
  }
}
```

Pour chaque pixel de l'image de la Ligne de Partage, on regardera avec la fonction ***est_LPE*** si le pixel est dans le contour ou pas. Si c'est le cas, on colore le pixel en rouge, sinon on copie la couleur du pixel dans l'image initiale.

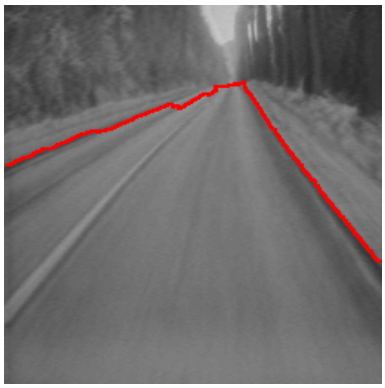
```
_Bool est_LPE(myimage im_LPE, uint32_t i, uint32_t j)
```

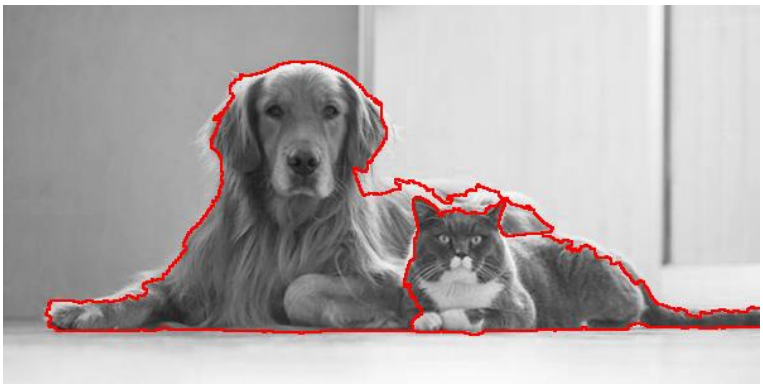
Cette fonction permet de renvoyer 1 si le pixel (i,j) a des voisins qui n'ont toutes pas la même valeur entre eux. Si ce n'est pas le cas alors elle renvoie 0.

```
uint32_t k,l;
for(k = i-1; k <= i+1; k++)
  for(l = j-1; l <= j+1; l++)
    if(k >= 0 && k < im_LPE.hauteur && l >= 0 && l < im_LPE.largeur)
      if(im_LPE.couleur[k][l] != im_LPE.couleur[i][j])
        return 1;
return 0;
```

On parcourt les voisins autour, on vérifie bien que le pixel (k,l) existe. Si la valeur du pixel (k,l) est différente du pixel (i,j), on renvoie 1 pour dire que le pixel (i,j) est sur le contour. Si on a exploré tous les voisins et qu'ils ont tous la même valeur, on en revoit 0 pour dire qu'on n'est pas sur le contour.

Exemples : image de la route et chien/chat avec les contours





Le tracé du contour a très bien fonctionné.

Conclusion

Pour conclure, ce projet nous aura apporté beaucoup d'expérience sur la mise en pratique de nos connaissances en structure de données. Il nous a éclairés sur l'importance du choix sur les structures de données à utiliser face à un problème. On peut le voir ici, via notre choix sur la table de hachage plutôt que sur la liste. Ce projet nous a également fait apprendre comment rédiger un rapport de projet d'informatique.

Notre programme fonctionne globalement bien. Cependant, des limites de notre programme pourraient être la création d'une image marqueur qui peut être difficile pour certaines images car elle influe sur la précision de la détection du contour. Une amélioration possible du projet serait l'implémentation d'une fonction qui génère les images marqueurs.