Tri d'une image

L'objectif de ce TP est de vous faire prendre conscience de la difficulté à trier des tableaux de grande taille. Nous allons pour cela réaliser un tri des pixels d'une image. [1] L'image ne ressemblera plus à grand chose à la fin, mais il sera facile de voir si votre tri a réussi ou non. Un système de correction automatique est mis en place sur Moodle. Néanmoins, nous vous demandons de vérifier d'abord votre code en local, puis, s'il semble fonctionner, le vérifier sur Moodle.

1 Introduction

L'objectif de ce TP est donc de réordonner les pixels d'une image suivant une certaine relation d'ordre que vous aurez préalablement déterminée. Pour cela, nous manipulerons des images au format BMP qui ont l'avantage d'avoir un accès simplifié aux valeurs des pixels.

Détaillons dans un premier temps les éléments qui composent une image. Une couleur peut se représenter par une composition de couleurs dites *primaires*. Au niveau de la lumière, le rouge, le vert et le bleu suffisent pour représenter n'importe quelle couleur de l'arc en ciel. Pour représenter une couleur, on a donc besoin de trois valeurs, appelées *composantes*, chacune représentant une quantité de rouge, de vert ou de bleu. On regroupe alors ces trois composantes, chacune sur un octet, pour former un pixel d'une image. Ainsi, on peut résumer un pixel à un triplet de valeurs (R, V, B). Une image n'est alors qu'un fichier contenant une série de pixels précédés d'une série d'informations permettant, entre autres, de localiser le début et la fin des données à lire.

Un basecode vous est fourni afin de lire et écrire une image au format BMP. Ce code vous permet, via la fonction `readImage()` de récupérer les pixels de l'image dont le chemin est passé en paramètres. La fonction retourne un pointeur sur une structure de type `Image` contenant diverses informations sur la taille des données, le header etc. Le champ `pix` représente le tableau des pixels.

La structure Pixel contient également d'autres représentations d'un pixel. Le format RVB n'est pas le seul existant pour représenter un couleur. On peut aussi utiliser le format YCbCr qui sépare l'image en une composante de luminosité et deux de couleurs, ou alors le format HSV qui représente une couleur selon sa *teinte*, *saturation* et *valeur*. L'appel à la fonction `readImage()` remplit tous les champs de chaque case du tableau de pixels. Vous pouvez l'utiliser sans travail supplémentaire pour réaliser la suite du TP.

1. L'utilité d'un tel TP est clairement discutable. En revanche, son intérêt pédagogique est clairement indiscutable !

2 Création d'une relation d'ordre

Avant même de développer l'algorithme de tri, il est essentiel de savoir quelles sont les données à trier, et comment les ordonner. Nous savons que les données sont stockées dans un tableau dont chaque case est une structure de type `Pixel` contenant un ensemble de valeurs. Il faut donc créer une ou plusieurs fonctions permettant de comparer deux pixels. Nous vous donnons quelques pistes.

- Une première idée est de comparer les pixels suivant un ordre lexicographique, c'est à dire suivant la valeur R, puis en cas d'égalité sur le rouge, selon la valeur G, puis enfin selon la valeur B (ou tout autre ordre des composantes).
- La somme des composantes peut vous indiquer approximativement l'intensité lumineuse d'un pixel. Plus la somme des trois composantes est élevée, plus le pixel apparaît vif.
- Une représentation plus efficace de la luminosité d'un pixel est la représentation YCbCr. Un coup d'œil sur votre moteur de recherche préféré vous expliquera son fonctionnement.
- Une autre transformation nommée HSV vous permettra de trier les pixels selon leur teinte (la valeur H). Encore une fois, nous vous laissons découvrir cette transformation par vos propres moyens.

Rappelons que le principe d'une fonction de comparaison est de prendre en entrée deux variables (ou pointeurs) du même type de donnée et de renvoyer une valeur positive si la première variable est supérieure à la seconde, une valeur négative si elle est inférieure et 0 en cas d'égalité.

> **Todo.** Choisissez au moins une relation d'ordre parmi les 4 ci-dessus (ou inventez-en une nouvelle) et développez-la. Pour les relations impliquant une transformation des couleurs (HSV / YCbCr), il sera nécessaire d'utiliser les champs correspondants de la structure `Image`. **Attention** : votre fonction doit prendre en paramètres deux pointeurs sur `Pixel` et non des `Pixel` directement. Il s'agit bien d'un passage par adresse et non par valeur.

3 Tri des pixels

Il est enfin temps de trier vos pixels ! Pour cela, rien de plus facile puisque vous avez déjà développé un algorithme de tri rapide sur des tableaux d'entiers. Il ne reste plus qu'à utiliser la fonction de comparaison que vous venez de développer et adapter chacune de vos fonctions de tri à des `Pixel` et le tour sera joué.

Il est alors intéressant de pouvoir comparer l'efficacité des algorithmes. Il est évident que chacun d'entre eux est censé renvoyer le même résultat, à savoir un tableau trié de la même façon. En revanche, vous avez vu que trois algorithmes ont une complexité quadratique, alors que le dernier est en $O(N \log N)$.

Remarquons la particularité du prototype des fonctions de tri : toutes contiennent un paramètre `pFunc compFunc` qui est un *pointeur sur fonction*. N'ayez pas peur, leur utilisation est plus simple que celle des pointeurs classique ! Ce pointeur contient l'adresse d'une fonction, ici, une des relations d'ordre. Pour l'utiliser, il suffit d'appeler la fonction `compFunc` sur l'adresse des deux pixels à comparer et de regarder sa valeur de retour.

Exemple 1. Voyons concrètement comment utiliser ce pointeur sur fonction. On suppose que la fonction de tri en question est celle du tri à bulles et que l'on veut comparer les pixels selon la somme de leur composantes. On suppose qu'on dispose d'une image bmp correctement lue et on appelle la fonction de tri de la façon suivante.

```
triBulles (img, compSommeRGB);
```

Le pointeur sur fonction `compFunc` prend alors l'adresse de la fonction `compSommeRGB`. Pour l'utiliser au sein de la fonction de tri, on utilise l'instruction suivante (on suppose que les pointeurs `p1` et `p2` désignent des adresses de pixels valides).

```
int res = compFunc(p1, p2);
```

Les pixels `p1` et `p2` sont alors comparés à l'aide de la fonction `compSommeRGB`.

Todo. Modifiez au moins un des algorithmes quadratique du semestre 1 afin de l'adapter au tri des pixels d'une image. Nous vous conseillons de réaliser vos tests sur des images de faible dimension afin d'obtenir rapidement un résultat. Adaptez ensuite l'algorithme de tri rapide à ce TP et observez la différence en terme de rapidité d'exécution, en particulier sur des images de taille moyenne.

Cette partie clôture la partie minimale que vous avez à rendre. Pour obtenir la note de 10/20, vous devez avoir codé au minimum une relation d'ordre, un tri quadratique et le tri rapide.

4 Pour aller plus loin

Vous pouvez maintenant coder toutes les fonctions de comparaison que vous souhaitez ajouter à votre programme. Citons entre autres le tri selon l'intensité lumineuse, la couleur d'un pixel, la saturation etc.

Il est également possible de modifier l'algorithme de partitionnement du tri rapide. En l'état, l'algorithme vu en cours est loin d'être performant. Le fait de toujours choisir le premier élément du tableau en tant que pivot nuit gravement aux performances de l'algorithme. Vous pouvez choisir un élément aléatoire dans le tableau en tant que pivot, voir choisir un autre algorithme de partitionnement.

Todo. Une idée d'amélioration est donc de modifier l'algorithme de partitionnement. Le temps d'exécution doit diminuer de façon notable si l'algorithme est bien codé. Ajoutez également d'autres fonctions de comparaison.

Maintenant que vous comprenez mieux le fonctionnement du format BMP, vous pouvez adapter votre code à la lecture et au tri d'images en niveaux de gris. Le gris est une couleur particulière qui comporte autant de rouge, de vert et de bleu. Il est donc inutile dans ce cas de stocker les trois composantes, une seule suffit. Dans le header de l'image BMP, le champ *profondeur de couleur* (octets 28 à 29) passe donc de 24 bits par pixels à 8 bits par pixel.

Le header est alors suivi d'une *palette* permettant d'associer une couleur RVB à un code sur 8 bits. Il y a donc 256×3 valeurs qui suivent immédiatement le header et qui composent la palette. Les trois premières valeurs donnent une couleur BVR qui est associée

au code 0, les suivantes au code 1 et ainsi de suite. Le reste de l'image est alors composé d'un code sur 8 bits pour chaque pixel.

On comprend alors que cette technique de stockage ne s'applique pas qu'aux images en niveau de gris. N'importe quelle couleur peut être stockée dans la palette et n'importe quelle image peut se restreindre à un nombre de couleurs particulier.

> **Todo.** Actuellement, votre code prend en charge uniquement les images à 24 bits par pixels. Vous pouvez l'adapter pour prendre en compte les images à 8 bits par pixels. L'idée est de stocker la palette puis pour chaque octet de l'image, stocker la couleur correspondante au code en lisant la palette. Le tri des pixels s'effectue alors normalement, mais vous devez réaliser la conversion couleur → code pour ré-écrire l'image triée.