Mini-Projet C++ **RIBER Rafael**
ColoReduce **EL-BA1**
ICC Pratique - CS-119
Novembre - Décembre 2018

Phase d'analyse

Le programme est divisé en 5 fonctions de lecture regroupées sous la fonction `fileRead`, 4 fonctions de transformation (`normalize`, `filter`, `getPixelValue` et `blackEdge`), et 2 fonctions de rendu (`render` et `printRGB`):

Lecture: Au sein de la fonction `fileRead`, executée dans `main`, on commence par déclarer une structure `"input"` de type `InputImg`. On execute ensuite les fonctions `inputReduced, inputThresholds, inputFilters, inputDimensions` et `inputPixels` qui remplissent chacune un champ de cette structure qui leur est passée par référence, respectivement le nombre `nbR` et les valeurs des couleurs réduites, les valeurs des seuils, le nombre de filtres `nbF`, les dimensions `nbL` et `nbC`, et les différentes couleurs des pixels de l'image à traiter, tout en vérifiant la validité de ces données et en renvoyant des erreurs en cas d'une mauvaise valeur. Le résultat de cette lecture est stocké dans la structure de type `InputImg` appelée `"image"` au sein de la fonction `main`. La fonction `colorRead`, qui lit 3 valeurs d'intensité et les stocke sous forme d'une structure `Color` est utilisée tout au long de la partie lecture pour lire les différentes valeurs RVB en entrée.

Seuillage: Cette structure est ensuite passée par référence à la première fonction de transformation executée dans `main` appelée `normalize`, qui crée un tableau de type `NormImg` appelé `"normOut"`, calcule pour chaque pixel son intensité normalisée I_N à partir des intensités RVB de ce dernier et détermine ensuite le code couleur de chaque pixel selon la valeur de cette intensité, ainsi que les seuils et la liste de couleurs réduites donnés en entrée. Le résultat de cette opération est stocké au sein de `main` dans un tableau `"norm"` de type `NormImg`.

Filtrage: On passe par référence à la fonction `filter` le tableau `"norm"`, ainsi que les dimensions de l'image d'entrée, le nombre `nbF`, et le nombre `nbR`. Pendant l'opération de filtrage (dans la fonction `filter`), la fonction `getPixelValue` calcule la nouvelle valeur de chaque pixel selon la valeur des pixels voisins. Le tableau est modifié par référence et l'image filtrée se trouve ainsi dans le tableau `"norm"` à la fin des `nbF` filtrages. La fonction `blackEdge`, executée au sein de `filter` après les filtrages, applique un bord noir d'un pixel de largeur tout autour de l'image filtrée, dans le cas où `nbF` > 0.

Rendu et fin: La fonction `render` génère un tableau `RGBImg` de structures de type `Color` contenant les valeurs RVB de chaque pixel à partir de l'indice entier calculé durant le seuillage et le filtrage ainsi que la couleur à l'indice correspondant dans la liste de couleurs réduites donnée en entrée. Le résultat est stocké dans le tableau `"rendered"` de type `RBGImg`.

La fonction `printRGB` affiche dans le terminal le tableau `rendered`, en ajoutant l'en-tête "P3" et en imprimant des espaces entre les valeurs RVB successives, ainsi que des retours à la ligne après chaque ligne de l'image.

Algorithme de filtrage

Ce pseudocode décrit un seul filtrage.

Algorithme 1 : FILTRAGE

Entrées : Tableau *source* de taille $C \times L$, nombre de couleurs réduites r.
Résultat : Tableau *destination*
Remarque: Les listes et tableaux sont indexés à 0.

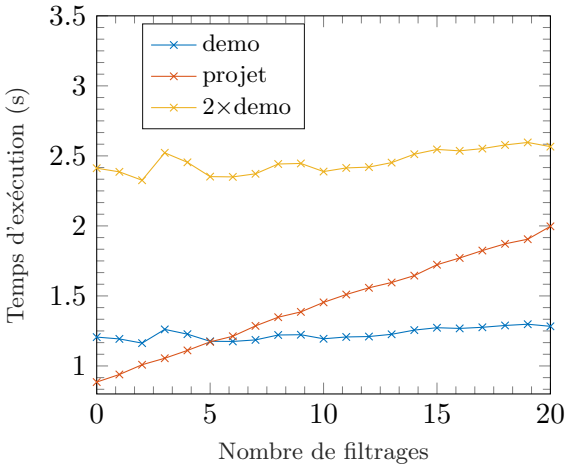
```
1  destination ← source
2  val ← 0
3  count est une liste de longueur fixe r.
4  Pour x de 1 à L − 1
5      Pour y de 1 à C − 1
6          current ← 0
7          Pour i de −1 à 1
8              Pour j de −1 à 1
9                  Si i ≠ 0 ou j ≠ 0
10                     current ← source[x + i][y + j]
11                     Pour c de 0 à r
12                         Si c = current
13                             count[c] = count[c]+1
14                             Si count[c] ≥ 6
15                                 val ← c
16                                 Continuer à la ligne 20.
17                             Sinon
18                                 val ← 0
19         destination[x][y] ← val
20 Sortir destination
```

Analyse de complexité

Dans le pire des cas, on parcourt tous les pixels n'étant pas en bordure de l'image f fois, en parcourant une liste de courleurs de longueur maximale r_{max} pour chacun des 8 voisins d'un pixel, puis les pixels de bordure une seule fois.
On a donc $N_i = (nbC - 2)(nbL - 2)$ pixels à filtrer f fois, et $N_b = (2nbC + 2(nbL - 2))$ pixels en bordure, le tout multiplié par le nombre maximum de couleurs réduites $r_{max} = 255$ que l'on parcourt 8 fois pour les pixels voisins, ce qui donne:

$$\begin{aligned} N_{pixels} &= 8 \cdot r_{max} \cdot (f \cdot N_i + N_b) \\ &= 8 \cdot r_{max}(f \cdot (nbC - 2)(nbL - 2) + (2nbC + 2(nbL - 2))) \\ &\leq 8 \cdot 255 \cdot f \cdot nbC \cdot nbL \end{aligned}$$

Le terme $r_{max} \cdot 8$ étant constant, on a ainsi une complexité $\mathcal{O}(f \cdot nbC \cdot nbL)$.



Sur ce graphique, on voit le temps d'exécution de mon programme `projet` sur le fichier test `tree.txt` selon le nombre de filtrage appliqué, en comparaison avec le programme `demo`.