Compte Rendu

Ma solution :

1) Idée de départ et principe général

L'idée est de faire plusieurs itérations d'une image en générant aléatoirement les couleurs. à la fin de l'itération, la distance par rapport à l'image originale est calculée, et comparée à la "meilleure" image qui est conservée entre les itérations et créée une seule fois à la fin du code.

Le but de cette solution est d'avoir un algorithme très facile à faire et plus rapide, au coût d'une copie d'image moins fidèle à l'original. Il est donc possible d'obtenir le meilleur résultat possible en une itération, ou ne jamais l'effleurer.

Je trouvais intéressant de comparer une solution basée à 100% sur de l'aléatoire à une solution plus "mathématique" pour pouvoir évaluer la différence en temps de calcul et résultat final, puisque la solution "aléatoire" peut être implémentée de la même manière pour résoudre beaucoup de problèmes.

2) Algorithme

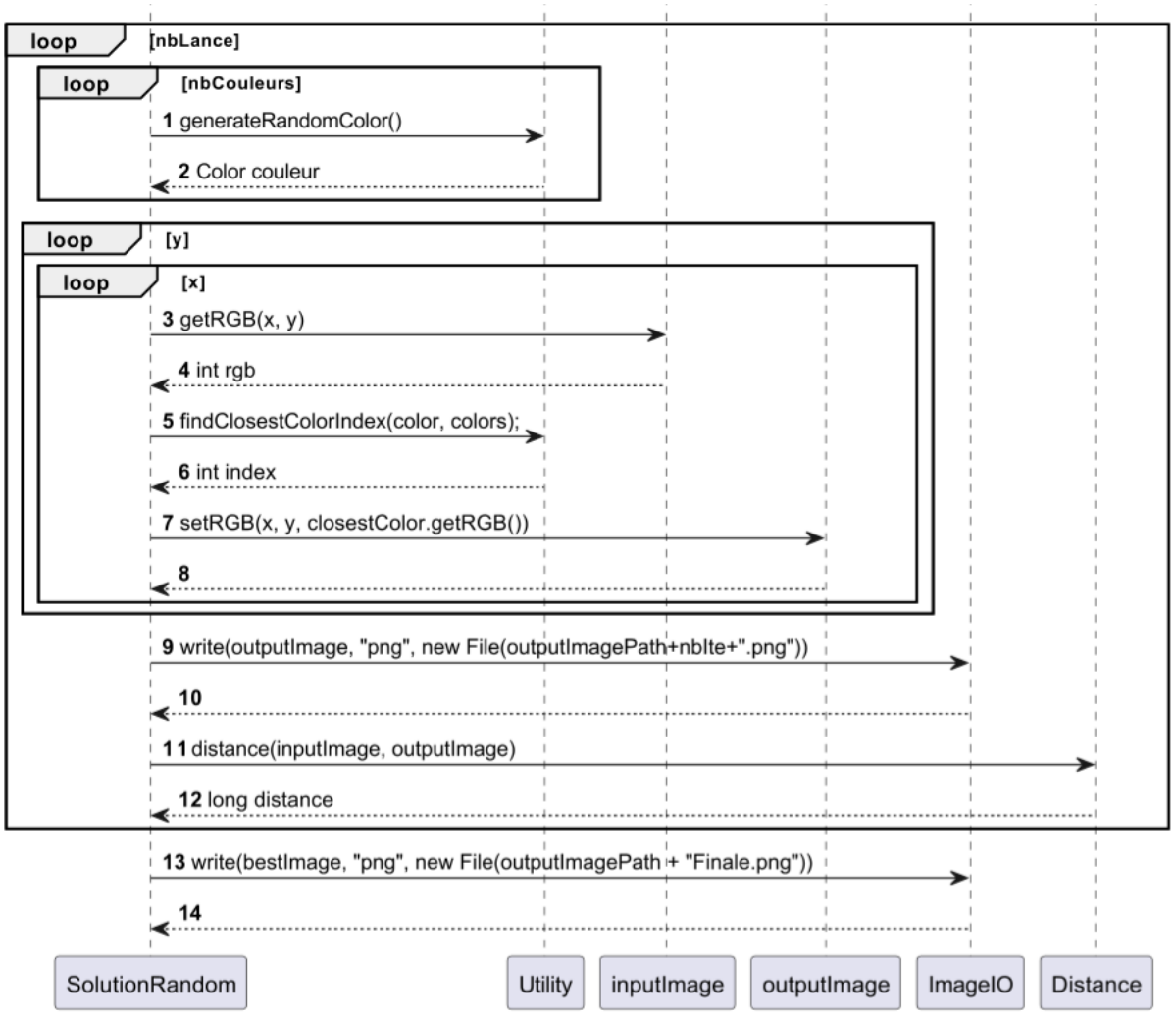
```
Début
Entrées : nbDeCouleurs ≥ 0 nombre de couleurs
pour i de 0 jusqu'à 100 {
        // Prendre des couleurs aléatoires en fonction du nombre de couleurs
          choisies
        pour j de 0 jusqu'à nbDeCouleurs {
            couleurs[j] = Utility.generateRandomColor()
        }

        // Parcourir chaque pixel de l'image
        pour y de 0 jusqu'à hauteurImage {
            pour x de 0 jusqu'à largeurImage {
                couleur = imageEntrée.getRGB(x, y)

                // Trouver la couleur la plus proche parmi les couleurs données
                indiceCouleurProche = Utility.findClosestColorIndex(couleur,
                                                couleurs)
                couleurProche = couleurs[indiceCouleurProche]
                // Appliquer la couleur la plus proche dans l'image de sortie
                imageSortie.setRGB(x, y, couleurProche.getRGB())
            }
        }
```

```
// Calculer la distance de l'image
distanceImage = Distance.distance(imageEntrée, imageSortie)
si meilleureDistanceImage != -1 {
    si distanceImage < meilleureDistanceImage {
        meilleureDistanceImage = distanceImage
        meilleureImage = imageSortie
    }
}
sinon {
    meilleureDistanceImage = distanceImage
    meilleureImage = imageSortie
}
```
Fin

3) Conception



Utility et Distance sont des classes statiques. Un diagramme de classe ne serait donc pas très utile.

4) Manière de lancer l'application

Pour lancer cette application, il faut indiquer un nombre de couleurs en premier argument, ainsi que le nombre d'itérations souhaité en deuxième argument. Il est aussi possible de modifier l'image d'entrée en modifiant la variable "inputImagePath".

5) Pistes pour améliorer la solution

J'ai essayé de faire une deuxième version (SolutionRandomV2) en essayant de gérer l'évaluation par couleur au lieu de par image. Le but était de réduire au maximum l'impact de l'aléatoire en créant un tableau composé des couleurs ayant obtenu le meilleur score à l'itération précédente. Cependant, je n'ai pas eu le temps de la finir. Une amélioration pourrait être de n'utiliser qu'un seul tableau pour éviter d'avoir plusieurs fois une couleur similaire à cause de son score. Mais utiliser l'aléatoire plusieurs fois pour trouver les couleurs optimales rend le programme très long car il est nécessaire de faire de nombreuses fois les mêmes calculs à cause des multiples itérations que requiert l'aléatoire. Il faudrait donc utiliser des formules mathématiques pour réduire l'utilisation de l'aléatoire, ainsi que le nombre d'itérations.

La structure du code pourrait aussi être améliorée. J'aurais pu réaliser une ou plusieurs autres classes contenant des parties du code sous forme de méthodes, et créer un main appelant une méthode qui lance toute l'application, ou le code allégé grâce aux fonctions. Le code serait donc plus compréhensible et mieux structuré. Cependant, je souhaité pouvoir gagner du temps en me concentrant sur un code qui fonctionne, sans avoir à penser à une structure au préalable comme je suis tout seul pour cette SAE.

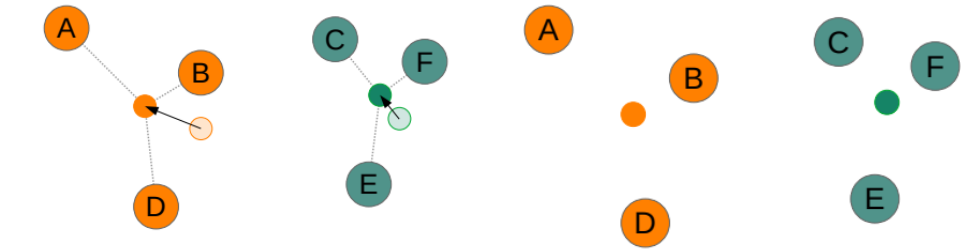
Il serait aussi possible d'utiliser des threads avec de la programmation répartie pour envoyer une itération sur plusieurs machines. Ainsi, les temps de calculs pourraient être divisés par le nombre de machines disponibles, ce qui rendrait l'algorithme très rapide.

Solution K-means :

1) Principe de la solution

L'algorithme commence par initialiser les centroïdes avec des valeurs aléatoires. Ensuite, plusieurs itérations vont être effectuées en stockant, une par une, les couleurs de l'image les plus proches d'un centroïde, dans le groupe de ce centroïde. Les centroïdes vont ensuite prendre la valeur de la moyenne des couleurs contenu dans leur groupe, avant de recommencer une nouvelle itération ou les groupes seront réinitialisés.

Cela signifie donc que nous allons avoir des centroïdes qui vont bouger à chaque itération, être reliés à des groupes contenant les couleurs les plus proches, pour enfin converger vers une seule couleur.



2) Algorithme correspondant à cette solution

Algorithme 1 : K-Means

Entrées : $n_g \geq 0$ nombre de groupes
Entrées : $D = \{d_i\}_i$ données
Résultat : Centroïdes mis à jour

```
   /* Initialisation centroïdes                                   */
1  pour i ∈ [0, n_g] faire
2      c_i ← random(D)
   /* Boucle principale                                           */
3  tant que (non(fini)) faire
       /* Initialisation Groupes                                  */
4      pour i ∈ [0, n_g] faire
5          G_i ← ∅
       /* Construction des Groupes                                */
6      pour d ∈ D faire
7          k ← indiceCentroidePlusProche(d, {c_i}_i)
8          G_k ← G_k ∪ d
       /* Mise à jour des centroïdes                              */
9      pour i ∈ [0, n_g] faire
10         c_i ← barycentre(G_i)
```

3) Manière de lancer l'application

Pour lancer cette application, il faut renseigner le premier argument représente le nombre de couleurs souhaité. Le deuxième argument n'est pas obligatoire et représente la graine utilisée pour générer de l'aléatoire.

4) Pistes possibles pour améliorer la solution

Le principal défaut de cet algorithme est le fait de créer plusieurs images plutôt qu'une seule à la fin. Mais c'est nécessaire pour pouvoir étudier et analyser le résultat des tests.

Tests :

1) Base de test utilisée

J’ai commencé par utiliser l’image “originale.jpg” pour comparer la distance des images générées par les algorithmes, ainsi que leurs temps de calculs respectifs. Cela est une bonne base car nous testons les algorithmes pour effectuer une tâche pour laquelle il est prévu, c'est-à-dire réduire le nombre de couleurs dans une image.

J’ai ensuite tester un cas pour lequel les algorithmes ne sont pas prévus, qui est d’avoir plus de couleur que l’image, avec comme donnée l’image “coul_10.png”. Ces tests sont intéressants pour trouver les éventuels défauts ou bugs des algorithmes.

2) Résultats moyens obtenus

Tests Image “originale.jpg” pour 10 couleur:
Distance SolutionRandom : 1489076051
Temps de calcul SolutionRandom : 4143 millisecondes
Nombre d'itérations : 20

Distance SolutionRandom : 1270626323
Temps de calcul SolutionRandom : 20561 millisecondes
Nombre d'itérations : 100

Distance SolutionKMeans: 263917027
Temps de calcul SolutionKMeans : 5004 millisecondes
Nombre d’itérations : 20
Seed : 68

<u>Tests Image “coul_10.png” pour 20 couleurs:</u>
Distance SolutionRandom : 590641872
Temps de calcul SolutionRandom : 6360 millisecondes
Nombre d'itérations : 25

la SolutionKMeans renvoie une erreur “centroide isolé” car il y a plus de centroïdes que de couleurs dans l’image.

3) Analyse et vos conclusions

L’algorithme “SolutionRandom” devrait renvoyer un résultat, en moyenne, plus proche de l’image originale plus le nombre d’itérations augmentent.

L’algorithme “SolutionKMeans” devrait renvoyer le meilleur résultat, mais ne pourra pas créer une image avec plus de couleurs que l’originale, puisqu’un centroïde ne sera le plus proche d’aucune couleur, ce qui renverra l’erreur “centroide isolé”.

Après avoir effectué les tests, nous pouvons voir que, en moyenne, l’algorithme aléatoire est plus rapide que celui de K-means, mais il est plus éloigné des couleurs de l’image originale. Cependant, il permet de retourner une image avec plus de couleurs que l’originale, là où l’algorithme de K-means renvoie une erreur.

Ces résultats sont donc en conformités avec mes attentes, même si je n’avais pas prévu que l’algorithme aléatoire soit plus rapide que celui de K-means.

Les écarts entre les différents résultats d’un même algorithme est principalement dû à l’aléatoire, mais aussi à la puissance de la machine. Si je devais effectuer ces mêmes tests sur une machine plus puissante, les temps de calculs seraient sûrement plus rapides.

4) Manière de lancer les tests

Les tests se lancent directement avec la fonction main de chaque algorithme. Les résultats sont écrits à la fin de la console.

5) Piste pour améliorer les tests

Il aurait été intéressant de créer une classe test à part pour ne pas surcharger le main des algorithmes. Il aurait aussi été intéressant d’avoir des diagrammes générés automatiquement traçant une courbe en fonction de la distance de chaque image, même si on peut deviner à quoi la courbe va ressembler.