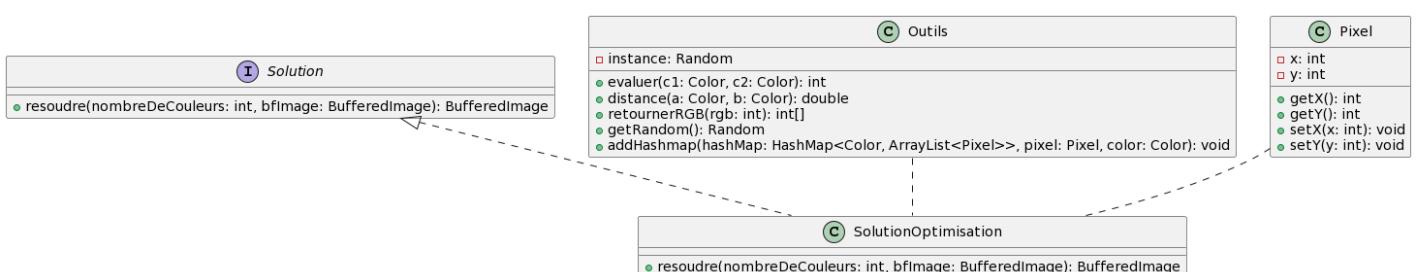


SAE - Manipulation d'images

1- Solution personnelle :

Dans un problème algorithmique comme la diminution du nombre de couleurs sur une image, cibler la difficulté n'est pas une option. Grâce au TP de pré-SAE, nous savons déjà comment réduire une image avec une liste de couleurs connues. La difficulté réside donc dans la recherche des couleurs idéales à la diminution de l'image sans altérer fortement sa qualité. Nous avons donc passé quatre heures à expérimenter différentes solutions. Nous discuterons principalement de quatre d'entre elles : deux qui ont échoué et deux qui ont plus ou moins été réussies.

Pour ce qui est de la conception de notre projet nous avons le diagramme de classe suivant :



La classe "Outils" permet d'utiliser des méthodes utiles telles que le calcul de distance et la génération de nombres aléatoires, etc.

La classe Pixel permet de regrouper les coordonnées x et y d'un pixel.

Pour lancer l'application, vous devez créer une nouvelle instance de Solution ici SolutionOptimisation et utiliser sa méthode "résoudre".

Pour ce faire, vous devez fournir deux paramètres à la méthode : le nombre de couleurs, qui est un entier, et un BufferedImage, qui correspond à l'image que vous souhaitez modifier..

A/ Solution histogramme:

Pour commencer, nous avions envisagé de créer un histogramme des couleurs qui regrouperait les teintes en fonction du nombre de pixels présents dans l'image. Ensuite, après la création de cet histogramme, notre intention était de le trier pour en extraire un certain nombre de couleurs, désigné comme "nbCouleurs".

Cependant, cette méthode s'est avérée inefficace. Les couleurs les plus représentées dans l'image étaient les nuances de vert, ce qui a rendu l'image entièrement verte.

B/ Solution Spectrale :

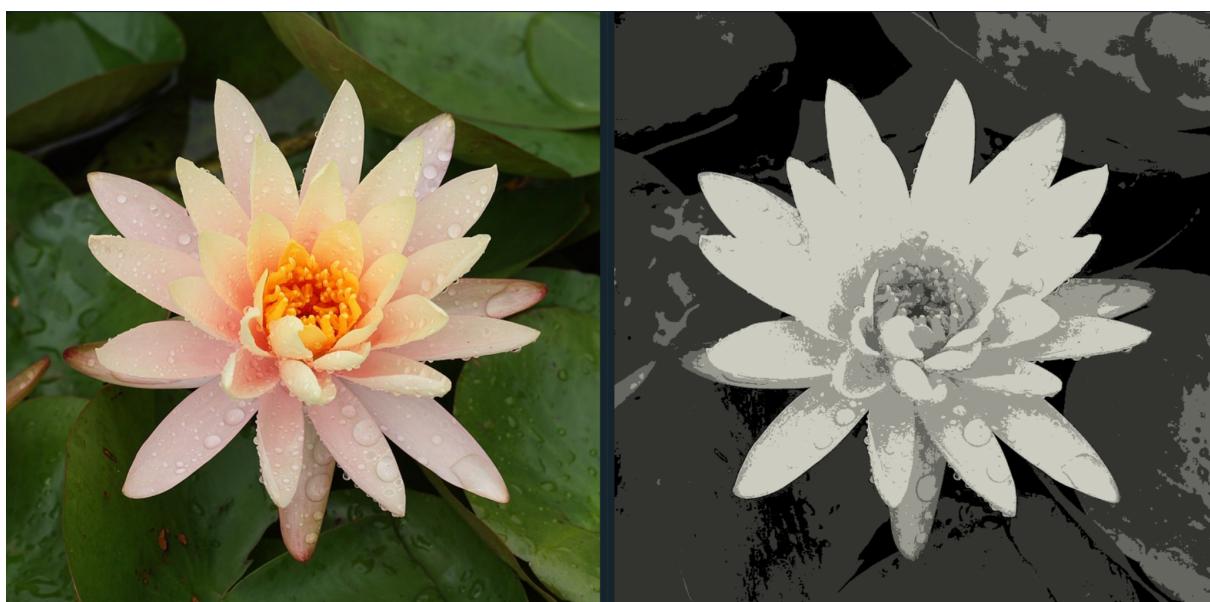
Une autre version se situe dans la continuité de la précédente. L'idée principale est de visualiser l'image comme une bande de couleur semblable au spectre du visible. Dans cette logique, la première étape est d'en définir les extrémités. Pour ce faire, nous parcourons intégralement l'image en stockant les valeurs R,G et B minimal et maximal.

Ensuite nous reconstruirons les deux couleurs min et max qui représenteront les extrémités. Le reste de l'algorithme consiste à diviser

notre spectre par le nombre de couleurs que l'on souhaite avoir. De cette façon, chaque coupure correspondrait à une couleur visible sur le rendu final.

Seulement plusieurs points négatifs sont évocables. Les couleurs finales ne sont en théorie pas obligatoirement présentes sur l'image originelle. D'un autre côté, lors de notre "découpage du spectre", nous partons d'une extrémité afin de soustraire ou augmenter les attributs RGB simultanément. Le problème est qu'augmenter autant le rouge, vert et bleu retournera toujours une nuance de gris. Cette solution n'est donc pas acceptable.

exemple pour 5 couleurs :



C/ Solution Optimisation:

Après avoir testé ces deux versions, nous avons tenté de sélectionner aléatoirement différentes couleurs de l'image de base tout en veillant à ce qu'elles soient différentes (afin d'éviter les répétitions de couleurs).

Ensuite, nous avons comparé toutes les couleurs de chaque pixel par rapport à la couleur précédemment sélectionnée. Nous avons stocké ces couleurs dans un hashmap, en utilisant comme clé les couleurs d'initialisation et comme valeur une liste de Pixel.
(Un pixel est une classe qui possède deux attributs, x et y.).

Ensuite, nous avons cherché à optimiser les couleurs que nous avions obtenues. Pour cela, nous avons parcouru notre liste de pixels contenue dans le hashmap afin de reconstruire les couleurs et les comparer à la couleur de base (celle que nous avons choisie au départ). Nous avons ensuite sélectionné la couleur ayant la distance la plus petite et l'avons remplacée par la clé correspondante.

L'algorithme:

```
// On récupère l'image  
//On crée une copie de l'image  
//On sélectionne au hasard des couleurs aléatoire différentes  
//Pour chaque pixel de l'image on trouve la couleur qui a la plus petite distance  
et on l'ajoute dans un map  
  
//Pour chaque couleur du map( couleur d'initialisation ), on trouve le pixel qui a  
la plus petite distance et on les échanges.  
  
//Avec la copie de l'image on change les couleurs avec les nouvelles
```

Avec cette solution nous avons alors les images suivantes:

Pour 5 couleurs :



Pour 10 couleurs :



Pour améliorer cette solution, je pense qu'il faudrait revoir la manière d'optimiser les pixels correspondant aux couleurs choisies initialement. Il faudrait les optimiser en tenant compte de l'ensemble des couleurs de l'image et s'arrêter quand on a la meilleure des versions

D/ Solution Histogramme Améliorée :

Dans la dernière version de la solution Histogramme, nous nous sommes rendu compte que la majeure partie des pixels étaient de couleur verte. Le résultat n'est donc pas ressemblant à l'image de base.

Pour pallier ce problème, le meilleur moyen est de définir une distance minimale "min" entre deux couleurs avant qu'elles soient considérées comme différentes. En suivant cette piste, nous remarquons que ce min est inversement proportionnel au nombre de couleurs voulu. C'est à dire que si l'on souhaite avoir 3 couleurs par exemple vert, bleu et rouge, la couleurs représentative de cette dernière catégorie devra avoir un min suffisant afin de pouvoir englober l'ensemble des nuances de rouge.

Notre choix a été de définir un entier optimal pour équilibrer la distance minimale nécessaire entre deux couleurs et le nombre de couleurs demandé. Nous choisissons l'entier 3 comme référence.

De cette manière le min de 2 couleurs correspond à l'écart total entre les deux couleurs (dont nous avons déjà parlé pour la solution spectrale) divisé par le nombre de couleurs fois 3 :

$$\text{min} = \text{ecart} / (3 * \text{nombreDeCouleurs});$$

De cette façon, nous obtenons l'algorithme suivant :

```
// récupération de l'image
// parcours une première fois l'ensemble des pixels
    // on classe par couleurs
        // on trouve un min et un max de la méthode getRGB
// calcul de l'écart colorimétrique de l'image
// calcul du min
// parcours de la liste de couleurs rencontrés
    // si distance avec les couleurs déjà sélectionnées pour le rendu final
        supérieure au min alors on ajoute
```

// parcours une deuxième fois l'ensemble des pixels

// on compare la couleur du pixel avec les couleurs sélectionnées et
on écrit sur la nouvelle image

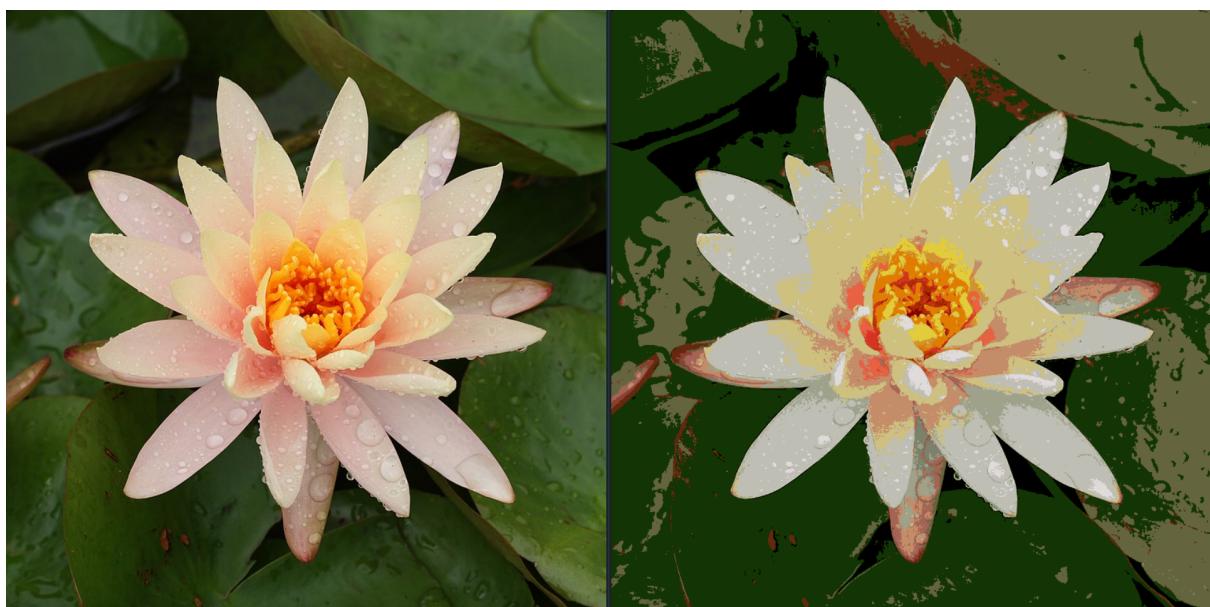
// écrire l'image

Nous obtenons alors les exemple suivants :

- pour 5 couleurs :



- pour 20 couleurs :



Cette solution donne des résultats pas très concluants avec peu de couleur car la couleurs que je définis comme "max" se rapprochera fortement du rouge, pouvant alors dégrader l'image.

2- Solution Obligatoire:

Le principe de cette solution est simple: elle repose sur la méthode K-means, qui utilise le principe de clustering pour regrouper les données en différents groupes.

Cette approche ressemble grandement à la SolutionOptimisation mentionnée précédemment mais elle présente quelques différences.

Dans cette méthode, nous commençons par créer des groupes en associant aléatoirement des couleurs de l'image, c'est ce qu'on appelle des centroïdes.

Ces centroïdes servent de points de référence pour la construction des groupes.

À la différence de notre idée initiale, nous répétons cette étape jusqu'à ce que l'algorithme ait terminé d'optimiser les groupes en ajustant les positions des centroïdes en fonction de ces groupes.

La mise à jour des centroïdes se fait en calculant le barycentre des couleurs présentes dans chaque groupe, ce qui permet d'obtenir une estimation plus précise de la couleur du groupe.

L'algorithme de k-means:

// Pour tous les nombres de couleurs on initialise les centroïdes.

//Tant que pas fini on:

//Pour toutes les couleurs on initialise les groupes

//Pour tous les pixels de l'image on remplit les groupes par rapport à la distance des pixels et des centroïdes.

//Pour tous les groupes on mets à jours les centroïdes en calculant le barycentre.

Nous obtenons alors les exemple suivants :

- pour 5 couleurs :



- pour 20 couleurs :



3- Statistiques et comparaison :

Dans notre projet, deux fichiers permettent de tester nos solutions. Le fichier MainSolution permet de choisir un algorithme et d'écrire le résultat dans une image "resultatMaSolution.png". Ce fichier est simple d'utilisation. Le second fichier Tests réalise des statistiques permettant de déterminer les algorithmes les plus performants. Les performances d'un algorithme se jugent sur son temps d'exécution et la qualité du résultat rendu. Ces statistiques sont retournées dans le terminal sous la forme de chaînes de caractères comme suit :

```
/*      Solution spectrale :      */
Moyenne temps : 145 millisec
Moyenne qualite : 1,379,191,143

/*      Solution Histogramme :      */
Moyenne temps : 207 millisec
Moyenne qualite : 11,226,180,454

/*      Solution Histogramme Plus :      */
Moyenne temps : 1499 millisec
Moyenne qualite : 2,393,168,520

/*      Solution Optimisation :      */
Moyenne temps : 61 millisec
Moyenne qualite : 836,985,004

/*      Solution KMeans :      */
Moyenne temps : 259 millisec
Moyenne qualite : 618,006,887
```

Pour une bonne analyse, les algorithmes nécessitant une sélection aléatoire des pixels de départ ont été répétés 20 fois, dont la moyenne a été calculée. C'est le cas pour KMeans et Optimisation. Les autres programmes ont été exécutés une seule fois, car leur solution ne varie pas en fonction de nombres aléatoires.

On remarque donc évidemment que l'algorithme KMeans est le plus performant en termes de rapport temps/qualité. Néanmoins, on distingue que notre algorithme d'optimisation se démarque réellement des autres, proposant une exécution rapide et efficace. Cela est dû à sa proximité avec KMeans, proposant alors une qualité moyenne très peu différente.

Pour finir, si l'on souhaite améliorer ces algorithmes, il serait judicieux de choisir délibérément les couleurs de départ à partir de l'image plutôt que de les prendre de manière aléatoire.