Compte-rendu SAE Projet IA

Groupe : PEREZ Ioann, RENARD Tanguy (AIL-2)

1. Solution personnel

Solution 1 : Récupérer les couleurs les plus utilisées

- **Idée et principe :**
  L'idée est de recréer l'image en utilisant seulement les *nbCoul* les plus utilisées.

  Pour cela, on récupère les couleurs de chaque pixel que l'on place dans un HashMap ayant pour clé la couleur et pour valeur le nombre de fois qu'elle est utilisée.

  Ensuite, on parcourt cette HashMap *nbCoul* fois, en récupérant la couleur la plus utilisée que l'on place dans un tableau et on supprime cette couleur de la HashMap (pour ne pas la récupérer à chaque itération).

  Pour créer la nouvelle image, on récupère la couleur de chaque pixel et on regarde quelle est la distance la plus petite entre cette couleur et chaque couleur de notre tableau créé précédemment. Une fois cette couleur trouvée, on l'assigne au même pixel de notre nouvelle image.

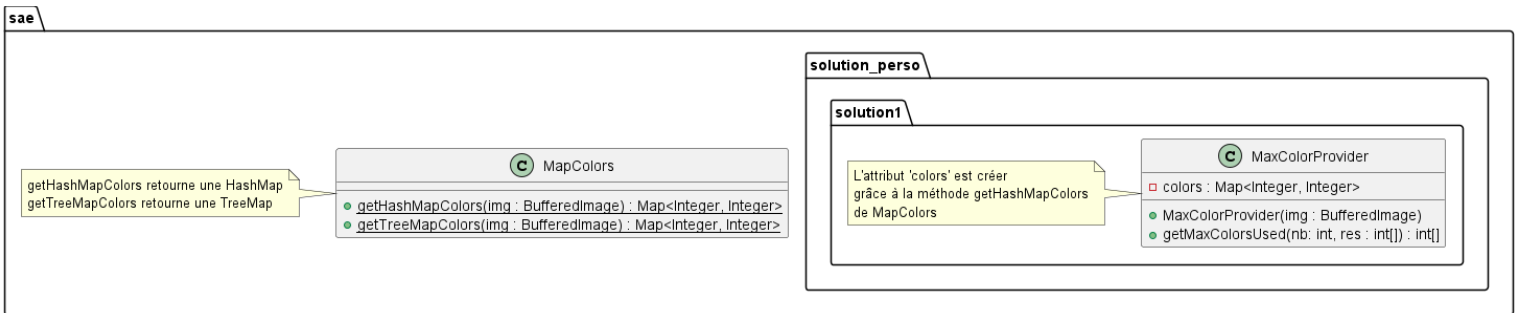
- **Algorithme :**

```
* fonction recupMaxCouleurs
* Entrées : nb > 0 nombre de couleurs à utiliser
* Entrées : TableCouleur la table des couleurs avec leur fréquence
d'apparition
* Entrées : res tableau d'entiers
*
* début
*
* max <-- -1
* kmax <-- 0
* Si nb > 0 alors
* retourner res
* fin si
*
* pour chaque couleur dans dom(TableCouleur) faire
* c <-- couleur.valeur
* si (c > max) alors
* max <-- c
* kmax <-- couleur
* fin si
* res[res.taille - nb] <-- kmax
* TableCOuleur.suppr(couleur)
* res = recupMaxCouleur(nb-1, res)
*
* Fin
*
```

Lexique :

- nb : entier, nombre de couleurs à utiliser pour reconstituer l'image
- TableCouleur : Table[Couleur → Entier] la table des couleurs avec leur fréquence d'apparition
- res : Tableau d'entiers contenant les rgb des couleurs les plus représentative
- max : entier, valeur la plus grande de la fréquence d'une couleur
- kmax : la couleur la plus fréquente

- **Conception :**
  Diagramme de classe :



- **Lancer l'application :**
  La solution se lance via la classe ***MainSAE1*** avec en arguments le chemin de l'image à utiliser et le nombre de couleurs que l'on souhaite dans la nouvelle image

- **Conclusion :**
  Cette méthode n'est pas la plus adaptée car si une couleur est plus utilisé que les autres, alors la nouvelle image ne sera composée essentiellement que de cette couleur.

Solution 2 : Choisir des couleurs aléatoire dans les couleurs de l'image

- **Idée et principe :**
  L'idée est la même que la solution précédente mais en se servant de couleur aléatoire de l'image.

  On récupère toujours les couleurs dans une HashMap mais on récupère *nbCoul* clés aléatoirement de cette HashMap pour les mettre dans notre tableau.

  Toujours sur le même principe, pour chaque pixel de l'image originale, on calcule la distance la plus petite entre sa couleur et les couleurs du tableau. La couleur du tableau ayant la plus petite distance est attribuée au même pixel sur la nouvelle image.

- **Lancer l'application :**
  La solution se lance via la classe ***MainSAE2*** avec en arguments le chemin de l'image à utiliser et le nombre de couleurs que l'on souhaite dans la nouvelle image

- **Conclusion :**
  Cette solution n'est pas concluante à ce stade car elle est très aléatoire. Il faudrait pouvoir réadapter les couleurs en fonction des couleurs de l'image.

Solution 3 : Se servir d'une palette de couleurs

- **Idée et principe :**

  Définir une palette des couleurs les plus utilisées puis ensuite comparer les couleurs de chaque pixel à notre palette de couleur. On garde ensuite la couleur la plus proche et s'il n'est pas déjà "utilisée pour une autre couleur", on l'affecte à notre tableau de couleur.

  Ensuite, on affecte à chaque pixel de la nouvelle image la couleur la plus proche dans notre tableau de couleurs.

- **Lancer l'application :**
  La solution se lance via la classe ***MainSAE3*** avec en arguments le le chemin de l'image à utiliser et le nombre de couleurs que l'on souhaite dans la nouvelle image

- **Conclusion :**
  Cette solution est très peu efficace dû notamment à sa limitation par la palette de couleurs.

Solution 4 : Découper la liste des couleurs et récupérer la plus utilisée de chaque partie

- **Idée et principe :**

  L'idée est de découper les couleurs de l'image en *nbCoul* groupe et de récupérer la plus grande valeur RGB de chaque groupe. Ensuite, on récupère la couleur la plus utilisé dans chaque groupe pour reconstituer l'image avec *nbCoul*

  Pour cela, on récupère les couleurs de l'image dans une TreeMap pour que le RGB des couleurs soit croissant. On parcours ensuite cette chaque partie de cette TreeMap pour trouver la couleur la plus utilisée et l'ajouter à un tableau de couleur.

  Ensuite, pour chaque pixel de l'image originale, on calcule la distance la plus petite entre sa couleur et les couleurs du tableau. La couleur du tableau ayant la plus petite distance est attribuée au même pixel sur la nouvelle image.

- **Algorithme :**

Fonction getMaxColorsFromParts(nb, res, size, nbCouleurs):
Si nb = 0 alors
Retourner res
Fin Si

keys <- ObtenirLesCles(colors)
listKey <- ConvertirEnListe(keys)
indiceCouleur <- longueur(res) - nb
indicePossible <- size / nbCouleurs
max <- -1
kmax <- 0

Pour chaque k dans keys faire:
i <- TrouverIndice(k, listKey)

Si i >= indicePossible * indiceCouleur ET i < indicePossible * indiceCouleur + indicePossible alors
c <- ObtenirValeur(k, colors)

Si c > max alors
max <- c
kmax <- k
Fin Si
Fin Si
Fin Pour

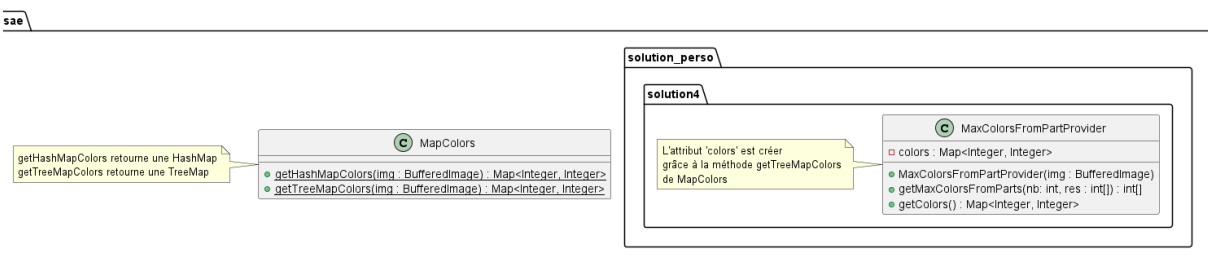
res[indiceCouleur] <- kmax
SupprimerLaCle(kmax, colors)

Retourner getMaxColorsFromParts(nb - 1, res, size, nbCouleurs)

FIN

- **Conception :**

Diagramme de classe :



- **Lancer l'application :**
  La solution se lance via la classe ***MainSAE4*** avec en arguments le chemin de l'image à utiliser et le nombre de couleurs que l'on souhaite dans la nouvelle image

- **Améliorations :** Cette solution est très lente, il faudrait donc optimiser le temps de recherche du max des couleurs et du calcul de la distance entre les couleurs

Solution 5 : Découper la liste et créer des couleurs à partir du RGB moyen de chaque partie

- **Idée et principe :**

  L'idée est de découper les couleurs de l'image en *nbCoul* groupe. Ensuite, on crée nos *nbCoul* couleurs à partir de la moyenne RGB de chaque groupe et ces couleurs serviront à reconstituer l'image.

  Ensuite, on compare la couleur de chaque pixel de l'image avec notre tableau de couleur et on assigne à la nouvelle image la couleur la plus proche de l'image originale.

- **Algorithme :**

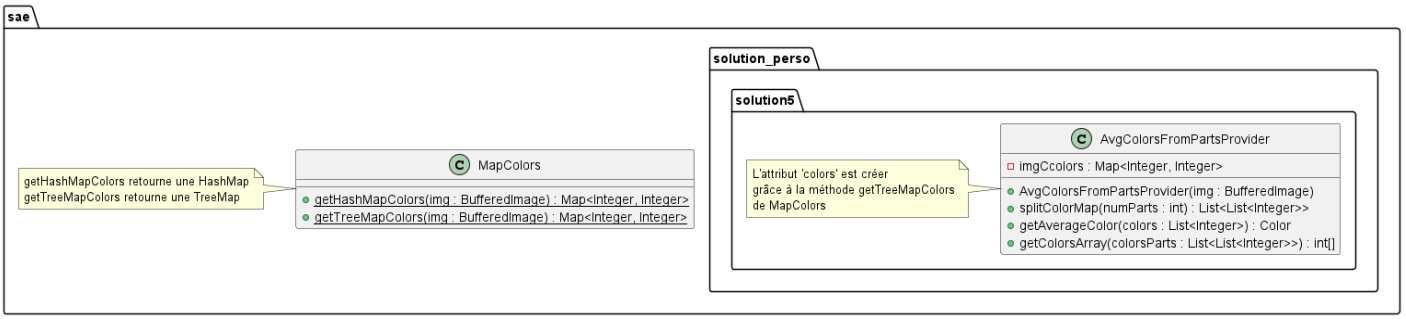
```
fonction ColeurParPartie (Liste(Liste(entier) TableDécoupée) :
Tableau[entier]
*
* début
*
* color <-- tab[0 .. TableDécoupée.taille]
* pour i de 0 à TableDécoupée.taille faire
* color[i] <-- calculerCouleurMoyenne(RGB(val(TableDécoupée, i))
* Fin pour
*
* retourner color
*/
```

Lexique :

- TableDécoupée : Liste(Liste(entier)), liste contenant chaque partie de la Table de couleurs des images
- color : Tableau d'entier, tableau contenant les couleurs les plus représentative

- **Conception :**

  Diagramme de classe :



- **Lancer l'application :**

  Cette solution se lance à l'aide de la classe MainSAE5 en donnant comme argument le chemin de l'image que l'on veut reconstituer et le nombre de couleurs que l'on veut utiliser.

- **Améliorations :** Cette solution est assez efficace mais le fait de récupérer les RGB moyens a tendance à appliquer un filtre sur l'image. Cet algorithme est plus efficace à partir de 10 couleurs.

  Pour l'améliorer, il faudrait pour que la couleur avec la moyenne des RGB d'une partie soit un point de départ, et ensuite adapter chaque groupe pour adapter la moyenne RGB.

2. Solution proposée par la SAE

- Idée et principe :

  L'idée est d'initialiser des centroïdes qui serviront des bases à des groupes de couleurs. Ensuite, on affecte chaque couleur au groupe ayant le centroïde avec la distance la plus courte de la couleur.

  Après que chaque couleur ait été affectée à un groupe, on calcule la moyenne RGB de chaque groupe et on affecte cette valeur à chaque centroïde. On répète l'opération jusqu'à ce que les centroïdes ne varient plus ou quasiment pas.

  Ces centroïdes seront alors le RGB de la couleur la plus représentative de chaque groupe.

- Algorithme :

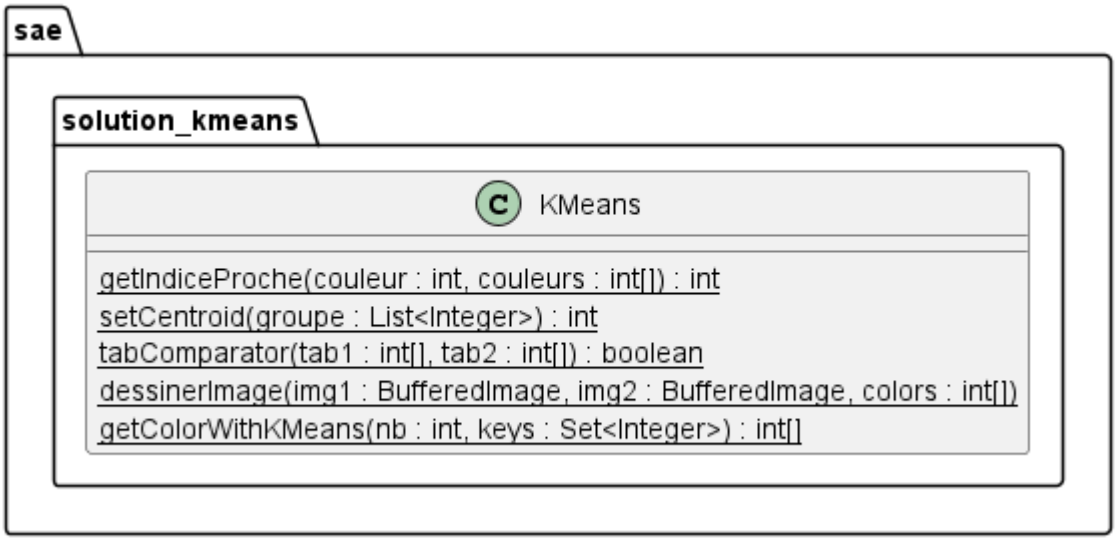
Entrées : $n_g \geq 0$ nombre de groupes
Entrées : $D = \{d_i\}_i$ données
Résultat : Centroïdes mis à jour

```
   /* 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)
```

-

- Conception :

  Diagramme de classe :



- Lancer l'application : Cette partie de l'application se lance avec la classe MainKMeans en lui donnant comme arguments le chemin de l'image et le nombre de couleurs que l'on veut utiliser

- Améliorations : Aucune

3. Tests

Pour les tests nous avons décidé d'utiliser la peinture "Klimt_small.png" car c'est pour nous une image plutôt clair où les couleurs sont visibles facilement, et également l'image possède le plus de couleurs de teintes différentes

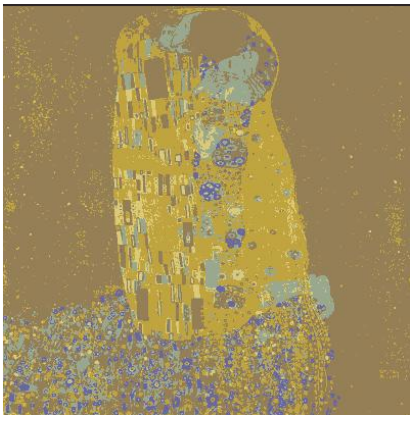
De plus nous avons fait des tests en prenant un maximum de 5 couleurs puis 100 couleurs pour chaque image. Cela nous permettra d'effectuer une analyse sur peu des couleurs, puis par la suite sur un nombre plus importants.

Résultats obtenus :

- **Solution 1** : n couleurs max
    - 5 couleurs → 56 ms :

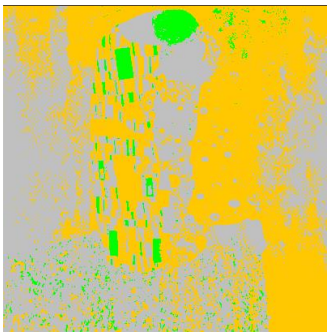
    - 100 couleurs → 190 ms :

- **Solution 2** : couleurs aléatoires
    - 5 couleurs → 54 ms :



- 100 couleurs → 197 ms :

- **Solution 3** : palette couleur prédéfini
  - jaune, vert, magenta, orange, gris clair → 54 ms :



- Solution 4 : max des morceaux de la liste des couleurs découpée n fois

*

  - 5 couleurs → 20182 ms :

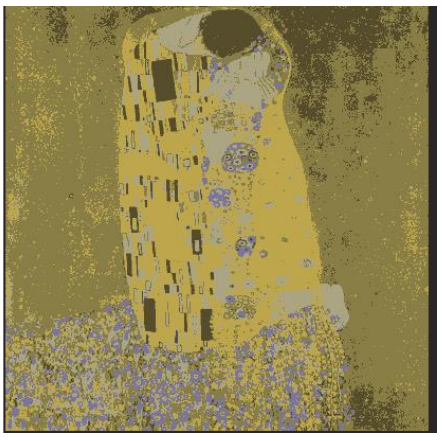
  - 100 couleurs → trop long

  - 50 couleurs → 93151 ms :

- Solution 5 : moyenne des morceaux de la liste des couleurs découpée n fois
    - 5 couleurs → 63 ms :

    - 100 couleurs → 209ms :

- Solution KMeans :
    - 5 couleurs → 294 ms :



    - 100 couleurs → 1626 ms :

Pour les tests de la première solution, on peut constater que pour 5 ou 100 couleurs, la vitesse des résultats est très rapide, en effet on récupère seulement les n couleurs les plus utilisées dans la liste puis on parcours les pixels pour calculer la distance. Néanmoins comme on peut le constater avec peu de couleur, l'image n'est presque pas visible. Cela peut s'expliquer que les couleurs les plus utilisées appartiennent toutes à un même teint. Cette solution n'est donc pas optimale malgré sa rapidité.

Pour la deuxième solution où l'on récupère des couleurs tirées aléatoirement dans la liste, on peut constater qu'elle est également très rapide, de plus le résultat est plutôt satisfaisant, cela est du au fais que l'image possède beaucoup de couleur donc on a potentiellement plus de chance de tiré des couleurs similaires. Néanmoins cette pratique reste très aléatoire, on peut retourner un bon résultat comme un total différent. Il faudrait avoir un nombre de couleur extremement grand pour augmenté les chance de réussite.
Malgré le résultat, cette solution n'est donc pas optimale.

Pour la troisième solution où l'on effectue les comparaisons avec des couleurs prédéfinies en amont. Cette solution est également rapide mais comme on peut le constater le résultat n'est pas du tout celui attendu. Cela peut s'expliquer car l'image possède beaucoup de couleur assez éloigné des couleurs primaires les plus utilisés. De plus si l'on souhaite ajouter beaucoup de couleur, il faudrait toute les ajoutés à la main et cela variera en fonction de l'image.
Donc cette solution n'est pas optimale.

Pour la quatrième solution où l'on découpe notre liste des couleurs par n puis pour chaque partie on calcule la couleur la plus utilisée, le résultat obtenu est le plus satisfaisant, en effet on obtient déjà un très bon résultat avec 5 couleurs. Mais en contrepartie celui ci demande énormément de mémoire et qui ne cesse d'augmenter considérablement à chaque ajout de couleur. Par exemple, pour obtenir seulement 100 couleurs, cela serait beaucoup trop long pour la tâche demandée.
Cette solution est donc optimale pour le résultat mais sa limite est la mémoire qu'elle demande.

Pour la cinquième solution, où on découpe comme pour la précédente mais cette fois-ci on calcule la moyenne des morceaux pour construire les couleurs, on peut nettement voir une amélioration niveau temps. En effet, calculer simplement la moyenne est beaucoup moins coûteux. Mais malgré que le résultat reste satisfaisant malgré que celui-ci le soit moins que le précédent. Mais celui-ci est plus rentable quand on regarde le temps, donc c'est une bonne alternative s'il y a beaucoup de couleurs.
Celui- ci est une bonne solution. malgré que le résultat ne soit pas le plus satisfaisant.

Puis pour la dernière solution on effectue l'algorithme K Means afin de récupérer les couleurs centroïdes optimales. Comme on peut le voir, niveau temps celle-ci n'est pas la

plus rapide, néanmoins celle-ci reste correcte, de plus le résultat est clairement supérieurs à ceux des autres solutions.

Pour lancer les tests, il suffit de se rendre dans la classe du main souhaité, de définir la valeur de l'attribut "nbCouleurs". On peut par la suite lancer le programme ce qui générera le résultat dans le répertoire "test_image" du projet.

Pour conclure la solution la plus optimale est clairement la dernière avec l'algorithme K Means qui, par le déplacement des centroids à chaque itération, nous permet d'obtenir un résultat optimal, mais cette solution peut également être améliorer notamment pour l'initialisation des couleurs qui dans notre programme sont choisi de manière aléatoires.
On pourrait donc trouver un moyen de les initialiser de façon à ce qu'il soit plutôt proche de leurs résultats attendus et cela sans utiliser beaucoup de mémoire. De cette façon on gagnera quelques itérations.
On pourrait par exemple pour cela s'inspirer de nos programmes où l'on découpe la liste des couleurs de manière égale afin de choisir une couleur pour chacune d'elles (aléatoire ou moyenne par exemple).