SAE – Détection de biomes sur des exoplanètes

Lien vers le dépôt GitHub :

https://github.com/CocoDietrich/SAE_detection_de_biomes

Contexte

En 2128, le premier télescope à lentille gravitationnelle solaire (T-LGS) a été lancé grâce à une collaboration entre l'ESA et la NASA. Ce télescope permet de photographier des exoplanètes avec une précision inégalée. Le traitement des photos permet de créer des cartes de surface des planètes en éliminant les variations de luminosité et la présence de nuages.

Objectif de la SAE

Le but de cette SAE est de détecter automatiquement les différents biomes présents sur une exoplanète à partir de ces cartes, puis de définir les écosystèmes appartenant à chaque biome. Un biome est une zone avec des écosystèmes aux caractéristiques similaires (par exemple, des déserts, des forêts tropicales, des lacs).

Travail réalisé :

Prétraitement de l'image

1. **Appliquer un filtre de flou** pour homogénéiser l'image.
    - **Flou par moyenne** : Utilisation d'un filtre où tous les coefficients sont identiques et leur somme est égale à 1.
    - **Flou gaussien** : Utilisation d'un filtre où les coefficients sont plus élevés au centre et décroissent vers les bords, basé sur une distribution gaussienne.

2. **Détection et visualisation des biomes**

    1. **Détection des groupes de pixels de couleur similaire** : Utiliser des algorithmes de clustering pour regrouper les pixels en fonction de la similarité de leur couleur.
    2. **Étiquetage des biomes** : Associer chaque cluster de couleur à un nom de biome en utilisant une correspondance entre couleurs et noms de biomes.
    3. **Affichage des différents biomes** : Préparer une image de fond clair et y afficher les biomes détectés.
    4. **Détection et affichage des écosystèmes pour chaque biome**: Appliquer un algorithme de clustering sur la position des pixels de chaque biome pour identifier les écosystèmes.

Appliquer un filtre de flou :

Flou par moyenne :

Pour effectuer le flou par moyenne, nous avons utilisé la méthode **applyMeanBlur** qui fonctionne ainsi :

On initialise la méthode de la sorte :

```
int width = image.getWidth();
int height = image.getHeight();
BufferedImage blurredImage = new BufferedImage(width, height, image.getType());
```

- On récupère la largeur et la hauteur de l'image.

- Une nouvelle image de même taille est créée pour stocker le résultat.

Ensuite, on créé un filtre de flou :

```
int filterSize = blurLevel * blurLevel;
float[] filter = new float[filterSize];
for (int i = 0; i < filterSize; i++) {
    filter[i] = 1.0f / filterSize;
}
```

- Un filtre de taille blurLevel x blurLevel est créé (blurLevel est un paramètre de la méthode qui définit le niveau du flou). Chaque élément du filtre a une valeur de 1/filterSize.

- Le filtre est une moyenne des pixels dans une zone définie autour de chaque pixel central.

On applique le filtre sur chaque pixel :

```
int offset = blurLevel / 2;
for (int y = offset; y < height - offset; y++) {
    for (int x = offset; x < width - offset; x++) {
        float r = 0, g = 0, b = 0;
        for (int j = -offset; j <= offset; j++) {
            for (int i = -offset; i <= offset; i++) {
                int rgb = image.getRGB( x: x + i,  y: y + j);
                r += ((rgb >> 16) & 0xFF) * filter[(j + offset) * blurLevel + (i + offset)];
                g += ((rgb >> 8) & 0xFF) * filter[(j + offset) * blurLevel + (i + offset)];
                b += (rgb & 0xFF) * filter[(j + offset) * blurLevel + (i + offset)];
            }
        }

        int newR = Math.min(Math.max((int) r, 0), 255);
        int newG = Math.min(Math.max((int) g, 0), 255);
        int newB = Math.min(Math.max((int) b, 0), 255);
```

- Le décalage (offset) est calculé pour centrer le filtre sur chaque pixel.

- Pour chaque pixel de l'image (en excluant les bords), la somme pondérée des valeurs des pixels environnants est calculée pour les canaux rouge, vert et bleu (R, G, B).

- Les nouvelles valeurs de couleur sont calculées en appliquant le filtre et en les limitant à l'intervalle [0, 255].

Finalement on met à jour les pixels de l'image floutée un par un :

```
int newRGB = (newR << 16) | (newG << 8) | newB;

blurredImage.setRGB(x, y, newRGB);
```

- Les nouvelles valeurs de couleur sont combinées en une seule valeur RGB et assignées au pixel correspondant dans l'image floutée.

L'algorithme de flou par moyenne lisse les variations de couleur en moyenne les valeurs des pixels dans une zone définie. Cela réduit les détails fins, ce qui est utile pour des applications telles que la détection de grandes zones de couleur uniforme, comme les biomes sur une carte d'exoplanète. Ce processus est efficace pour éliminer le bruit tout en conservant les transitions entre les zones de couleurs différentes.

Flou gaussien :

Pour effectuer un flou gaussien, on a créé une classe **GaussianBlur**.

Cette classe contient une méthode principale (**applyGaussianBlur**) pour appliquer le flou gaussien sur une image et une méthode auxiliaire (**generateGaussianKernel**) pour générer le noyau gaussien.

Dans un premier temps, on va voir la méthode **applyGaussianBlur**.

L'initialisation est similaire au flou par moyenne.

Ensuite, on génère le filtre gaussien :

```
float[] filter = generateGaussianKernel(blurLevel);
```

- La méthode generateGaussianKernel est appelée pour créer un noyau gaussien de taille définie par blurLevel.

On applique le filtre gaussien. La méthode est similaire au flou par moyenne. La mise à jour des pixels de l'image se fait de manière similaire.

Les différences principales viennent de la méthode **generateGaussianKernel.**

Dans un premier temps, l'initialisation :

```
float[] kernel = new float[size * size];
int center = size / 2;
```

- Un noyau de taille size x size est créé.

- Le centre du noyau est calculé (center), ainsi que l'écart-type (sigma), qui est proportionnel à la taille du noyau.

Ensuite, on calcule les valeurs du noyau :

```
for (int y = 0; y < size; y++) {
    for (int x = 0; x < size; x++) {
        int dx = x - center;
        int dy = y - center;
        float value = (float) Math.exp(-(dx * dx + dy * dy) / (2 * sigma * sigma)) / (2 * (float) Math.PI * sigma * sigma);
        kernel[y * size + x] = value;
        sum += value;
    }
}
```

- Pour chaque élément du noyau, la valeur gaussienne est calculée en utilisant la formule de la distribution gaussienne bidimensionnelle.

- Les valeurs sont stockées dans le noyau et la somme des valeurs est calculée (**sum**).

Il faut ensuite normaliser du noyau :

```
for (int i = 0; i < kernel.length; i++) {
    kernel[i] /= sum;
}
```

- Chaque valeur du noyau est divisée par la somme totale des valeurs (**sum**) pour normaliser le noyau. Cela garantit que l'application du noyau ne modifie pas la luminosité globale de l'image.

L'algorithme de flou gaussien est un moyen efficace de lisser les variations de couleur en tenant compte des distances relatives des pixels voisins à l'aide d'une distribution gaussienne. Cela permet de réduire les détails fins et le bruit tout en conservant les transitions douces entre les différentes régions de couleur de l'image. Ce processus est particulièrement utile pour des applications telles que le prétraitement des images avant l'analyse ou la réduction du bruit dans les images.

Détection et visualisation des biomes :

Détection des groupes de pixels de couleurs différentes :

KMeans :

Cet algorithme implémente le clustering **KMeans** en Java. Le clustering **KMeans** est une méthode de partitionnement qui divise un ensemble de données en **k** clusters. Chaque cluster est représenté par un centroïde, qui est la moyenne des points du cluster. L'objectif est de minimiser la variance intra-cluster, c'est-à-dire, la somme des distances au carré des points au centroïde du cluster.

Voici une explication détaillée du fonctionnement du code :

Classe et Constructeur :

La classe **KMeans** implémente une interface **ClusteringAlgorithm**. Le constructeur initialise le nombre de clusters **k** et le nombre maximal d'itérations **maxIterations**.

Méthode cluster :

Cette méthode contient l'algorithme **KMeans**. Elle prend en entrée les données à clustériser et retourne un tableau d'assignations de clusters.

Dans un premier temps, on initialise les variables :

```
int n = data.length;
int[] clusterAssignments = new int[n];
double[][] centroids = new double[k][data[0].length];
```

- n est le nombre de points de données.
- clusterAssignments garde la trace de l'assignation de chaque point à un cluster.
- centroids stocke les centroïdes des clusters.

Dans un second temps, on initialise les centroïdes :

```
Random rand = new Random();
for (int i = 0; i < k; i++) {
    centroids[i] = data[rand.nextInt(n)];
}
```

La boucle choisit aléatoirement **k** points de données comme centroïdes initiaux.

On s'intéresse, ensuite, aux itérations de l'algorithme **KMeans** :

```
boolean changed = true;
int iterations = 0;

while (changed && iterations < maxIterations) {
    changed = false;
    iterations++;
```

- changed : un indicateur pour savoir si les assignations des clusters ont changé lors de l'itération précédente.

Après cela, on assigne des points aux clusters :

```
for (int i = 0; i < n; i++) {
    int nearestCluster = getNearestCluster(data[i], centroids);
    if (nearestCluster != clusterAssignments[i]) {
        clusterAssignments[i] = nearestCluster;
        changed = true;
    }
}
```

Pour chaque point de données, on trouve le cluster le plus proche, via la méthode getNearestCluster, et on met à jour l'assignation si nécessaire.

La prochaine étape consiste à recalculer les centroïdes :

```
double[][] newCentroids = new double[k][data[0].length];
int[] counts = new int[k];
```

- newCentroids : pour stocker les nouveaux centroïdes.

- counts : pour compter le nombre de points dans chaque cluster.

On accumule les points pour chaque cluster :

```
for (int i = 0; i < n; i++) {
    int cluster = clusterAssignments[i];
    for (int j = 0; j < data[0].length; j++) {
        newCentroids[cluster][j] += data[i][j];
    }
    counts[cluster]++;
}
```

On ajoute les coordonnées des points aux nouveaux centroïdes et on compte le nombre de points par cluster.

On calcule les moyennes :

```
for (int i = 0; i < k; i++) {
    if (counts[i] == 0) continue;
    for (int j = 0; j < data[0].length; j++) {
        newCentroids[i][j] /= counts[i];
    }
}
```

On divise par le nombre de points pour obtenir la moyenne.

Enfin, pour les méthodes auxiliaires, on a :

- getNearestCluster : permet de trouver le cluster dont le centroïde est le plus proche d'un point donné en utilisant la distance euclidienne.

- distance : calcule la distance euclidienne entre deux points.

Finalement, cet algorithme fonctionne en alternant entre deux étapes :

Assigner chaque point au cluster le plus proche.

Recalculer les centroïdes des clusters en prenant la moyenne des points assignés.

Ces étapes sont répétées jusqu'à ce que les assignations ne changent plus ou que le nombre maximal d'itérations soit atteint. L'algorithme peut converger vers des solutions locales optimales et peut nécessiter plusieurs exécutions avec des initialisations différentes pour obtenir un bon résultat.

DBScan :

Dans un premier temps, on initialise les variables :

```
private double eps;
private int minPts;
```

On définit ensuite une méthode **cluster** :

```
public int[] cluster(double[][] data) {
    int n = data.length;
    int[] labels = new int[n];
    int clusterId = 0;

    for (int i = 0; i < n; i++) {
        if (labels[i] != 0) continue; // Déjà visité
        Set<Integer> neighbors = regionQuery(data, i);
        if (neighbors.size() < minPts) {
            labels[i] = -1; // Bruit
        } else {
            clusterId++;
            expandCluster(data, labels, i, neighbors, clusterId);
        }
    }
    return labels;
}
```

- Parcourt les points de données.
- Si un point n'a pas été visité, il cherche ses voisins en utilisant **regionQuery.**
- Si le nombre de voisins est inférieur à **minPts**, le point est marqué comme bruit.
- Sinon, un nouveau cluster est créé et les voisins sont ajoutés au cluster en utilisant **expandCluster :**

```
private void expandCluster(double[][] data, int[] labels, int pointIndex, Set<Integer> neighbors, int clusterId) {
    labels[pointIndex] = clusterId;
    Set<Integer> uniqueNeighbors = new HashSet<>(neighbors);

    while (!neighbors.isEmpty()) {
        int neighborIndex = neighbors.iterator().next();
        neighbors.remove(neighborIndex);

        if (labels[neighborIndex] == -1) {
            labels[neighborIndex] = clusterId;
        }

        if (labels[neighborIndex] == 0) {
            labels[neighborIndex] = clusterId;
            Set<Integer> newNeighbors = regionQuery(data, neighborIndex);
            if (newNeighbors.size() >= minPts) {
                uniqueNeighbors.addAll(newNeighbors);
            }
        }
    }
    neighbors.addAll(uniqueNeighbors);
}
```

- Marque le point initial et ses voisins avec l'identifiant du cluster.
- Elargit le cluster en ajoutant des voisins supplémentaires qui satisfont la condition de densité **minPts.**

La méthode **regionQuery** permet de trouver tous les voisins d'un point donné qui sont dans le rayon **eps**.

```
private Set<Integer> regionQuery(double[][] data, int pointIndex) {
    Set<Integer> neighbors = new HashSet<>();
    for (int i = 0; i < data.length; i++) {
        if (distance(data[pointIndex], data[i]) <= eps) {
            neighbors.add(i);
        }
    }
    return neighbors;
}
```

La méthode **distance** calcule la distance euclidienne entre deux points.

```
private double distance(double[] point1, double[] point2) {
    double sum = 0;
    for (int i = 0; i < point1.length; i++) {
        double diff = point1[i] - point2[i];
        sum += diff * diff;
    }
    return Math.sqrt(sum);
}
```

Etiquetage des biomes :

La classe **Palette** s'occupe de l'étiquetage des biomes. Nous avons une variable BIOME_COLORS contenant chaque couleur demandé associé à leur biome.

Affichage des différents biomes :

Afin de réaliser la visualisation des clusters, ils sont visualisés en coloriant chaque pixel selon son biome.

```
BufferedImage kmeansBiomeImage = visualizeClusters(image, kmeansClusters, k: 10);
ImageIO.write(kmeansBiomeImage, formatName: "png", new File( pathname: "biome_output_kmeans.png"));
```

Détection et affichage des écosystèmes pour chaque biome :

Pour chaque biome détecté, le code applique un nouvel algorithme de clustering pour détecter les écosystèmes et les visualiser.

Pour détecter et afficher les écosystèmes pour chaque biome, on utilise la méthode suivante :

```java
/**
 * Méthode pour visualiser les clusters en coloriant chaque pixel selon son cluster.
 *
 * @param image L'image originale.
 * @param clusters Tableau contenant les numéros de cluster pour chaque pixel.
 * @param k Le nombre de clusters.
 * @return Une nouvelle image avec les clusters colorés.
 */
private static BufferedImage visualizeClusters(BufferedImage image, int[] clusters, int k) {
    int width = image.getWidth();
    int height = image.getHeight();
    BufferedImage clusteredImage = new BufferedImage(width, height, image.getType());

    // Assigner les couleurs des biomes aux clusters
    Color[] clusterColors = new Color[k];
    for (int i = 0; i < k; i++) {
        clusterColors[i] = Palette.BIOME_COLORS[i % Palette.BIOME_COLORS.length];
    }

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int clusterId = clusters[y * width + x];
            clusteredImage.setRGB(x, y, clusterColors[clusterId].getRGB());
        }
    }

    return clusteredImage;
}
```

Ensuite, on extrait les positions des pixels appartenant à un certain biome sont extraites et stockées dans un tableau.

```java
for (int i = 0; i < positions.length; i++) {
    int x = (int) positions[i][0];
    int y = (int) positions[i][1];
    ecosystemImage.setRGB(x, y, biomeColor.getRGB());
}
```

On applique ensuite l'algorithme de clustering (KMeans ou DBScan). Ici KMeans :

```java
// Appliquer un nouvel algorithme de clustering pour détecter les écosystèmes
ClusteringAlgorithm ecosystemAlgorithm = new KMeans( k: 3,  maxIterations: 100); // 3 clusters pour les écosystèmes
EcosystemDetection ecosystemDetection = new EcosystemDetection(ecosystemAlgorithm);
int[] ecosystemClusters = ecosystemDetection.detectEcosystems(biomePositions);
```

Enfin, pour visualiser les environnements, on utilise la couleur du biome et l'image résultante :

```
// Visualiser les écosystèmes en utilisant la couleur du biome
BufferedImage ecosystemImage = visualizeEcosystemClusters(image, biomePositions, ecosystemClusters, Palette.BIOME_COLORS[biome]);
try {
    ImageIO.write(ecosystemImage, formatName: "png", new File( pathname: "ecosystem_output_" + algorithmType + "_biome_" + biome + ".png"));
} catch (IOException e) {
    e.printStackTrace();
}
```

Conclusion

La SAE de détection de biomes sur des exoplanètes vise à automatiser l'identification et la cartographie des différents biomes et écosystèmes à partir d'images haute résolution obtenues par un télescope à lentille gravitationnelle solaire. Le processus se divise en plusieurs étapes, allant du prétraitement des images à la détection et à la visualisation des biomes et des écosystèmes.

1. Prétraitement de l'image :

- Flou par moyenne : Cette méthode homogénéise l'image en lissant les variations de couleur, ce qui aide à éliminer le bruit tout en conservant les transitions entre différentes zones de couleur.

- Flou gaussien : Utilise une distribution gaussienne pour lisser l'image, réduisant ainsi les détails fins et le bruit tout en maintenant des transitions douces.

2. Détection et visualisation des biomes :

- Clustering avec KMeans : Cet algorithme partitionne les pixels en groupes de couleur similaire, représentant différents biomes. Les clusters sont ensuite étiquetés avec des noms de biomes en fonction de leur couleur.

- Clustering avec DBScan : Cet algorithme de clustering basé sur la densité identifie les clusters en fonction de la densité de points, permettant une détection plus flexible des biomes, notamment en présence de bruit et de variations de densité.

3. Étiquetage et visualisation des biomes :

- Les biomes détectés sont étiquetés et visualisés en coloriant chaque pixel selon son biome, permettant une visualisation claire des différentes zones de la planète.

4. Détection et affichage des écosystèmes pour chaque biome :

- Après avoir détecté les biomes, un nouvel algorithme de clustering est appliqué aux pixels de chaque biome pour identifier les écosystèmes spécifiques. Ces écosystèmes sont ensuite visualisés avec des couleurs représentant les biomes, permettant une analyse détaillée de la structure écologique de chaque biome.

Contributions et Implications

Ce projet démontre comment des techniques avancées de traitement d'image et de clustering peuvent être utilisées pour analyser des données complexes obtenues à partir d'exoplanètes. La capacité à détecter automatiquement les biomes et les écosystèmes sur des planètes lointaines offre des perspectives intéressantes pour la recherche en astrobiologie et la compréhension des environnements extraterrestres. En automatisant ce processus, la méthode proposée permet d'analyser efficacement de grandes quantités de données d'images, facilitant ainsi la découverte de nouveaux biomes et écosystèmes.

Les techniques présentées peuvent également être appliquées à d'autres domaines nécessitant la détection de motifs dans des données d'image, ouvrant la voie à de nombreuses applications potentielles dans l'exploration spatiale et l'analyse environnementale.