Algorithmes et Pensée Computationnelle

Algorithmes spatiaux

Le but de cette séance est de comprendre les caractéristiques de données spatiales et les structures de données couramment utilisées pour les manipuler. Lors de cette séance, nous implémenterons des algorithmes de manipulation de structures de données spatiales vus en cours. Au terme de la séance, l'étudiant sera en mesure d'utiliser quelques algorithmes spatiaux pour résoudre des problèmes de base de façon efficiente.

1 Nearest-Neighbor

Dans cette section, nous allons implémenter une recherche du plus proche voisin. Le but de cette méthode est de trouver le voisin le plus proche du point de départ en tenant compte de ce point de "départ" et un ensemble de points. Nous allons implémenter cet algorithme et ensuite l'étendre à un algorithme des "k plus proches voisins" (recherche des k voisins les plus proches plutôt que du seul voisin le plus proche). Nous vous recommandons de traiter les questions dans l'ordre.

Question 1: (*5 minutes*) **La fonction de distance : Python**

Pour implémenter notre recherche, nous avons besoin d'écrire une fonction permettant de calculer la distance entre 2 points. Ecrivez une fonction qui permet de calculer la distance entre 2 points.

Conseil

Soit 2 points en 2 dimensions (x_1, y_1) et (x_2, y_2), la distance euclidienne entre ces 2 points est donnée par : $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. En Python, la fonction permettant de faire une racine carrée est **sqrt** de la librairie **math**. Pour mettre au carré, on utilise **nombre******2**.

Question 2: (*10 minutes*) **Nearest-neighbor search**

Implémentez la recherche du voisin le plus proche. Cette dernière fonctionne de la façon suivante :

1. Traversez chaque point.
2. Pour chaque point, calculez la distance entre ce point et le point de départ.
3. Retournez les coordonnées du point le plus proche.

Conseil

Utilisez la fonction de distance de la question 1 et parcourez les points à l'aide d'une boucle **for**. Si votre input est **[(2, 3), (5, 6), (1, 4), (2, 4), (3, 5)]** et que le point de départ est **[4,4]**, alors l'output devra être (**[3,5] 1.414**). **[3, 5]** étant les coordonnées du point le plus proche et **1.414** étant la distance euclidienne entre notre point de départ et le point le plus proche.

Note : Pour que votre programme fonctionne, écrivez votre algorithme du plus proche voisin dans le même programme que celui de la Question 1.

Question 3: (*15 minutes*) **K-nearest-neighbor search**

Améliorez l'algorithme du voisin le plus proche effectué plus haut afin qu'il puisse retourner des K-plus proches voisins.

Conseil

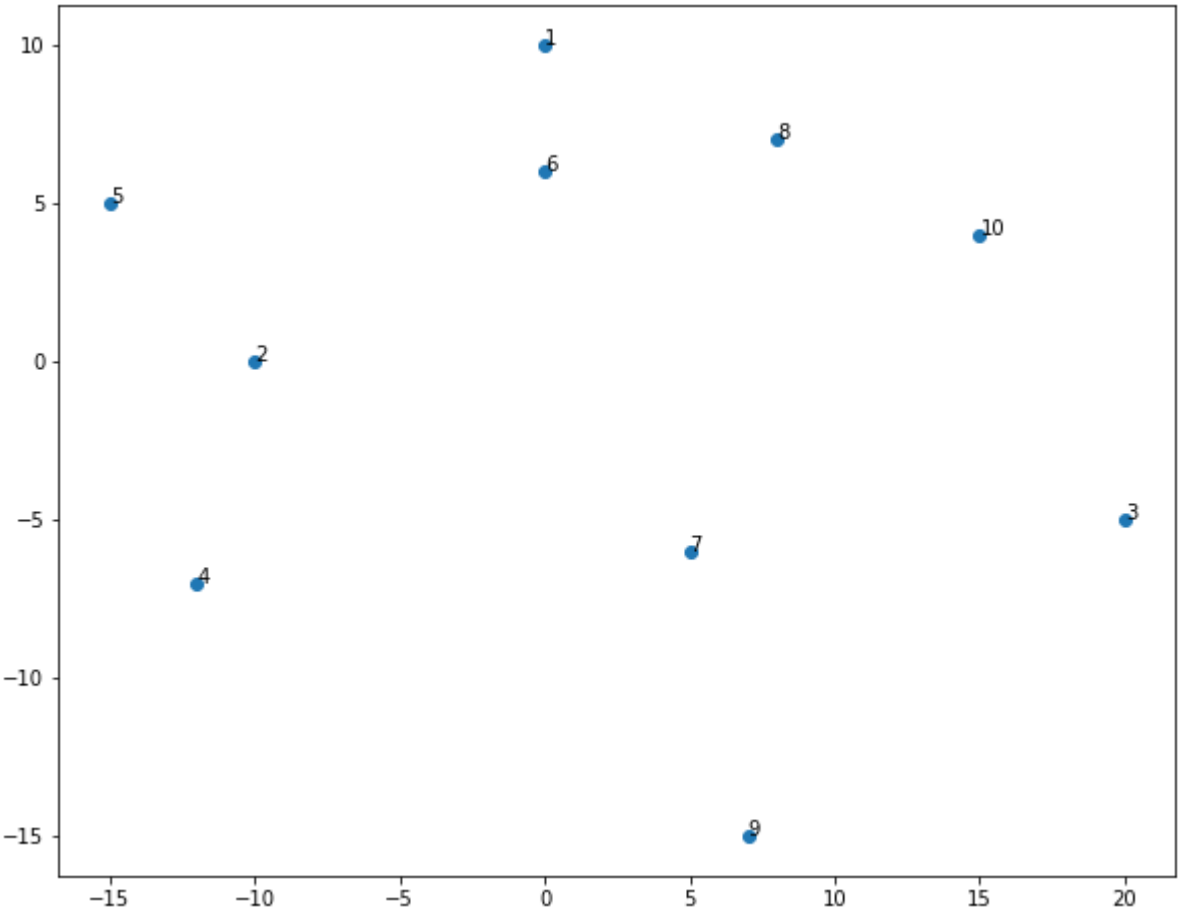
Appliquez l'algorithme du plus proche voisin *K*-fois. À la fin de chaque itération, retirez le voisin le plus proche de l'ensemble des points sur lequel l'algorithme s'applique. De cette façon, vous trouverez le second voisin le plus proche, le troisième, etc...

Votre fonction devra retourner une liste sous la forme : $[(x_1, y_2, distance1), (x_2, y_2, distance2), ...]$. En considérant un input **[(2,3),(5,6),(1,4),(2,4),(3,5)]**, un point de départ **(4,4)** et un nombre de voisins $K = 2$, l'output de votre algorithme devra être : **[(3, 5, 1.4142135623730951), (2, 4, 2.0)]**. Attention au type des variables que vous manipulez. Pour rappel, les tuples sont immuables en Python. Pensez à créer de nouveaux tuples contenant les coordonnées des points et leurs distances.

Note : Pour que votre programme fonctionne, écrivez votre algorithme dans le même programme que celui de la Question 1 et la Question 2.

2 K-dimensional tree

Question 4: (*5 minutes*) **KD-Tree, un échauffement : Papier**
Vous trouverez ci-dessous une liste de points numérotés de 1 à 10. Placez-les dans un KD-Tree et dessinez la séparation de l'espace qui en résulte.



Conseil

La première division se fait de façon verticale. Veillez à bien insérer les points dans l'ordre (point 1, point 2, etc..). Les nœuds se situant au même niveau devraient diviser l'espace selon le même axe.

Question 5: (*15 minutes*) **KD-Tree : Python**

L'objectif de cet exercice est d'écrire une fonction permettant d'ajouter un nœud à un KD-Tree. Les nœuds sont de la forme ((x,y), enfant à gauche, enfant à droite), x et y étant les coordonnées du nœud considéré. Chaque enfant peut être soit un nœud ou une feuille. Complétez le code contenu dans le fichier **Question5.py**.

Conseil

Voici le pseudo-code permettant d'ajouter un nœud à un KD-Tree :

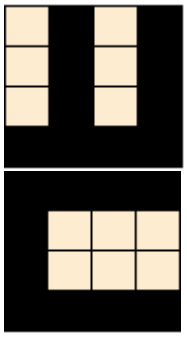
```
ADD(node,point,cutaxis) :
    if node = NIL
        node ← Create-Node
        node.point = point
        return node
    if point[cutaxis] ≤ node.point[cutaxis]
        node.left = ADD(node.left, point, (cutaxis + 1) modulo k
    else
        node.right = ADD(node.right, point, (cutaxis+1) modulo k
    return node
```

Si votre réponse est correct, le code de Question5.py devrait afficher : **[(0, 10), [(−10, 0), None, None], None]**.

3 Quad-Tree

Question 6: (*10 minutes*) **Une mise en train : Papier**

Encodez les images ci-dessous dans un Quad-Tree.



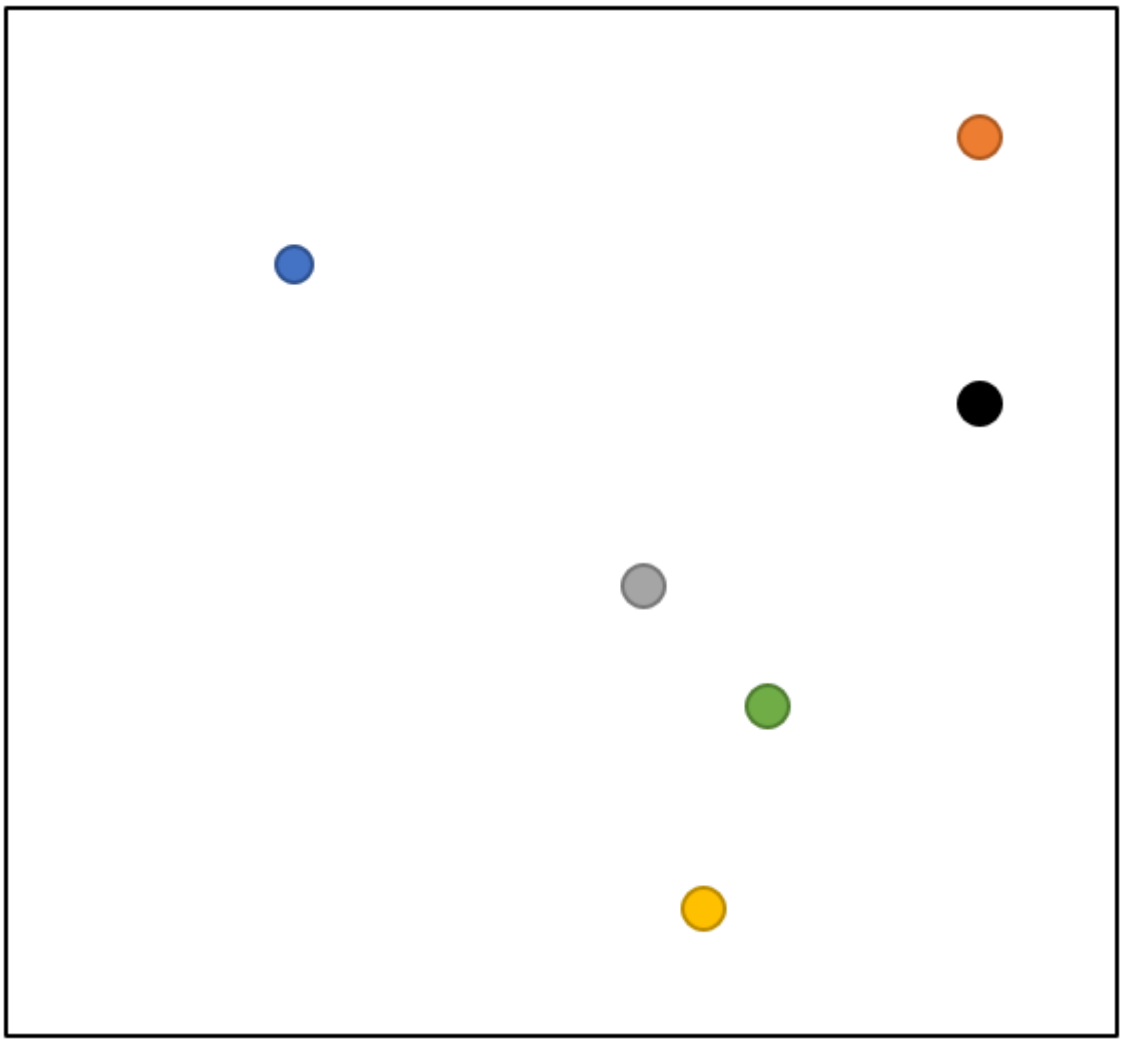
Conseil

Pour réussir cet exercice vous devez diviser chaque nœud en 4 sous-espaces (le plus petit sous-espace étant un carré) de taille égale, et ce autant de fois que nécessaire. La branche la plus à gauche correspond au quadrant **NW** puis en allant de gauche à droite : **NE, SW, SE**.

Indice : Votre arbre devrait avoir une profondeur de 2 et disposer de 16 feuilles.

Question 7: (*10 minutes*) **Une mission capitale : Papier** Optionnel

Récemment embauché par la CIA, vous êtes à la recherche d'un individu se cachant dans une des villes suivantes : Bleu, Orange, Noir, Gris, Vert et Jaune. Votre mission, si vous l'acceptez, est de créer un Quad-Tree qui vous permettra de géolocaliser le criminel de façon efficace. Vous trouverez ci-dessous une carte de villes. Créez le Quad-Tree et rétablissez la justice.



Conseil

Commencez par diviser la carte de la ville de façon adéquate puis construisez le graphe.

Remarque : Les différentes branches de l'arbre n'auront pas toutes la même profondeur.