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`.

Solution

```
1 import math #permet d'importer la librairie nécessaire au calcul de la
      racine carrée
2
3 def calculate_distance(point1,point2):
4     #Cette fonction retourne la distance euclidienne entre 2 points
5     return math.sqrt((point1[0]-point2[0])**2+(point1[1]-point2[1])**2)
```

Note : Vous auriez pu utiliser `**0.5` en lieu et place de la fonction `math.sqrt()`.

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.

Solution

```python
from Question1_solution import calculate_distance
# Question 2
def nearest_neighbor(start, point_set):  # start correspond au point de départ, point_set correspond
    # à l'ensemble des points
    nearest_nei = None
    min_distance = None
    for i in range(len(point_set)):  # on parcourt tous les points de l'ensemble
        if i == 0:
            # La distance minimale n'étant pas définie, on doit l'initialiser à la première itération, c'est ce qu'on
            # fait ici
            min_distance = calculate_distance(start, point_set[0])
            nearest_nei = point_set[0]

        distance = calculate_distance(start, point_set[i])
        if distance < min_distance:
            min_distance = distance
            nearest_nei = point_set[i]

            # Cette partie du code détermine si le point actuellement considéré, est plus proche du point de départ que les points
        # parcourus jusqu'ici. Si c'est le cas, on redéfinit la distance minimale et on "enregistre" les coordonnées du point

    return nearest_nei, min_distance


if __name__ == '__main__':
    a = [(2, 3), (5, 6), (1, 4), (2, 4), (3, 5)]  # Liste de points
    b = (4, 4)  # Point de départ

    point, distance = nearest_neighbor(b, a)
    print(f'{point}, {distance}')
    # Devrait retourner (3, 5), 1.4142135623730951
```

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.

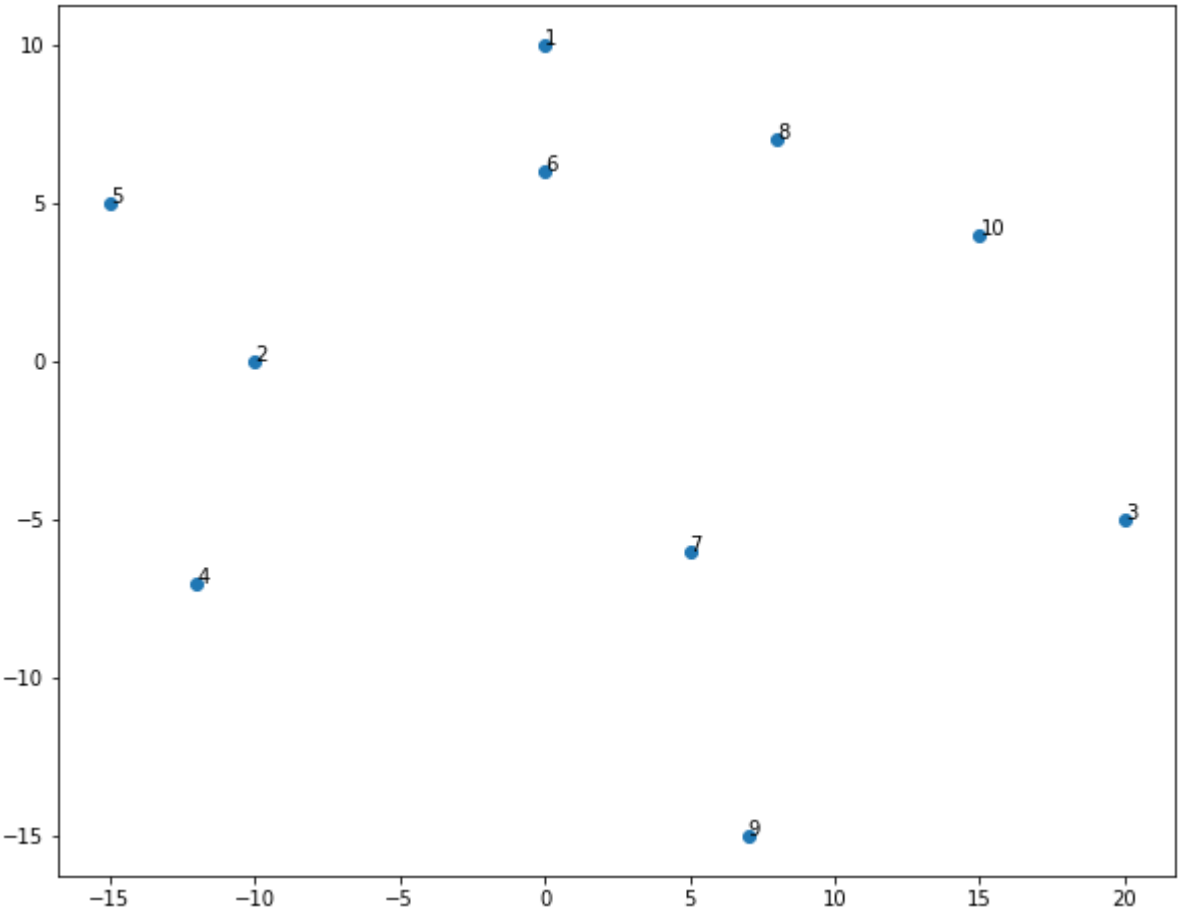
Note : Pour que votre programme fonctionne, écrivez votre algorithme dans le même programme que celui de la Question 1 et la Question 2.

Solution

```
1  # Question 3
2  def K_nearest_neighbor(start, point_set, K):
3      k_nearest_nei = []
4
5      for j in range(K):  # A chaque itération, on applique l'algorithme
       du nearest neighbor mais sur un ensemble de points réduit
6          point, distance = nearest_neighbor(start, point_set)
7          k_nearest_nei.append((point[0], point[1], distance))
8          point_set.remove(point)  # On retire de l'ensemble de points
       le voisin le plus proche, de cette manière, à chaque itération,
9          # le voisin le plus proche sera de plus en plus éloigné.
10
11     return k_nearest_nei
```

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.

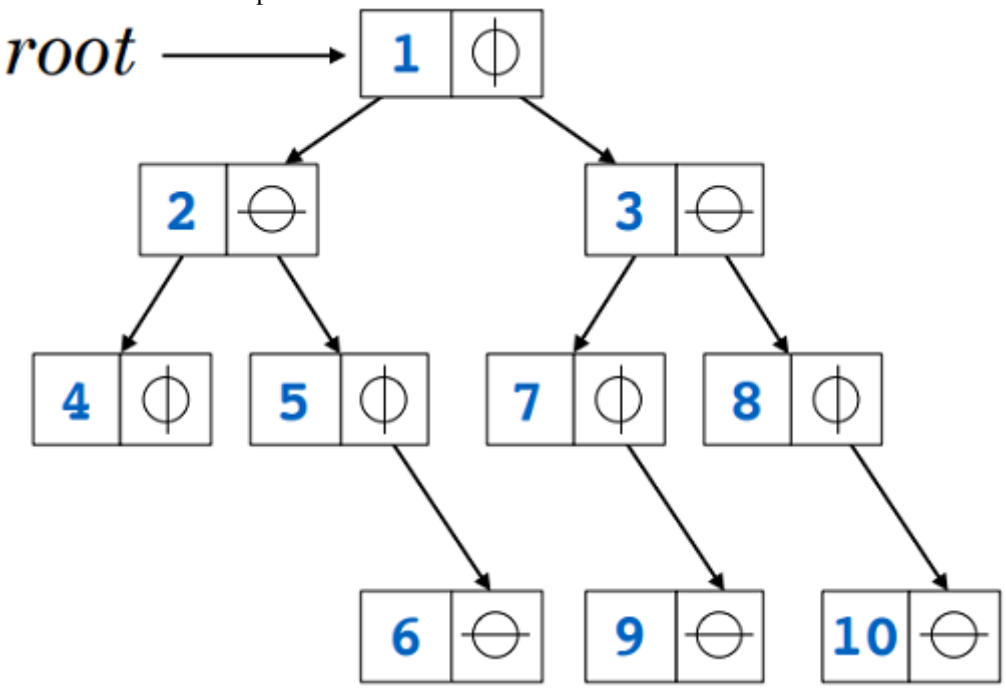


Conseil

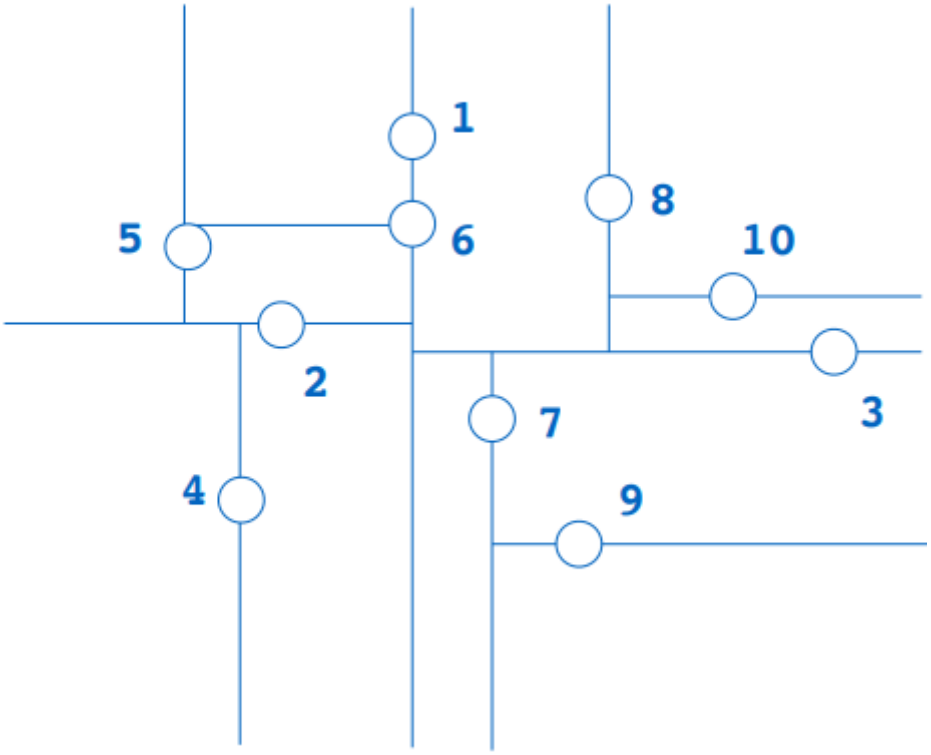
La première division se fait de façon verticale. Veillez à bien insérer les points dans l'ordre (point 1, point 2, etc..). Les nœuds se situant au même niveau devraient diviser l'espace selon le même axe.

Solution

Voici le KD-Tree correspondant :



Et la division de l'espace qui en résulte :



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 correcte, votre programme devrait afficher : `[(0, 10), [(-10, 0), None, None], None]`.
Pour rappel, en Python, `NIL` correspond à `None`. `Create-Node` permet de créer un nouveau nœud, vous pouvez vous inspirer de la structure de `root` définie dans le fichier `Question5.py`.

Solution

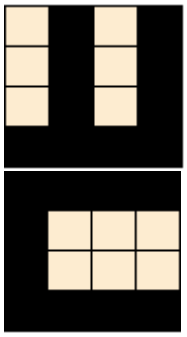
```
1  # Question 5
2
3  def add_node(node,point,k,cutaxis = 0):
4
5      if node is None: #Si le noeud n'existe pas, nous sommes donc dans
       une feuille, et il faut créer le noeud
6          node = [point,None,None]
7          return node
8
9      if point[cutaxis] <= node[0][cutaxis]:
10         node[1] = add_node(node[1], point, k, cutaxis + 1 % k)
11
12     else:
13         node[2] = add_node(node[2], point, k, cutaxis + 1 % k)
14
15     return node
16
17
18 root = [(0,10), None, None] #Nous définissons ici juste la racine de
       l'arbre
19 k = 2 # Ici nous travaillerons en 2 dimensions
20 point = (-10,0) #Point que nous voulons ajouter dans le graphe
21 add_node(root,point,k)
22 print(root)
```

Si la coordonnée du point à ajouter est inférieure à celle du nœud selon l'axe de découpe en considération, alors le point doit se trouver dans le sous-arbre de gauche. Par convention, le nœud de gauche correspond dans la liste [(x,y), nœud de gauche, nœud de droite] à l'indice 1, par conséquent, on appelle la fonction de façon récursive pour ajouter le point, mais cette fois-ci en partant d'un cran plus bas dans l'arbre. Cela se répète jusqu'à ce qu'on ait atteint les feuilles et qu'un nouveau nœud doive être créé.

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.

Solution

Image 1 :

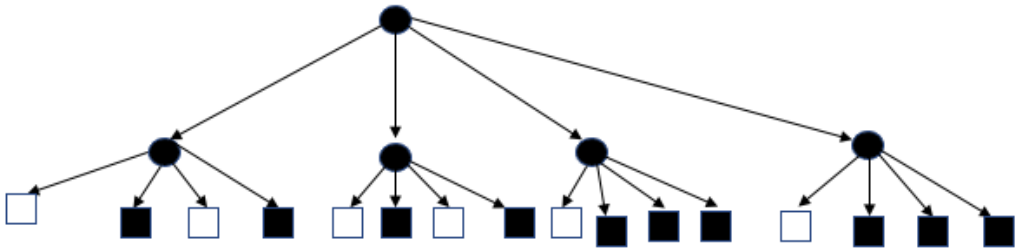
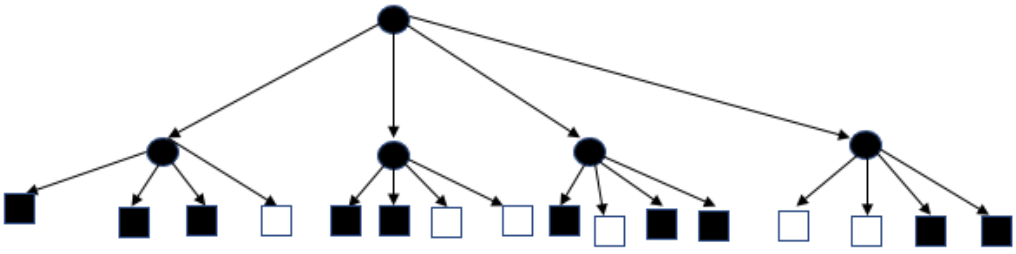
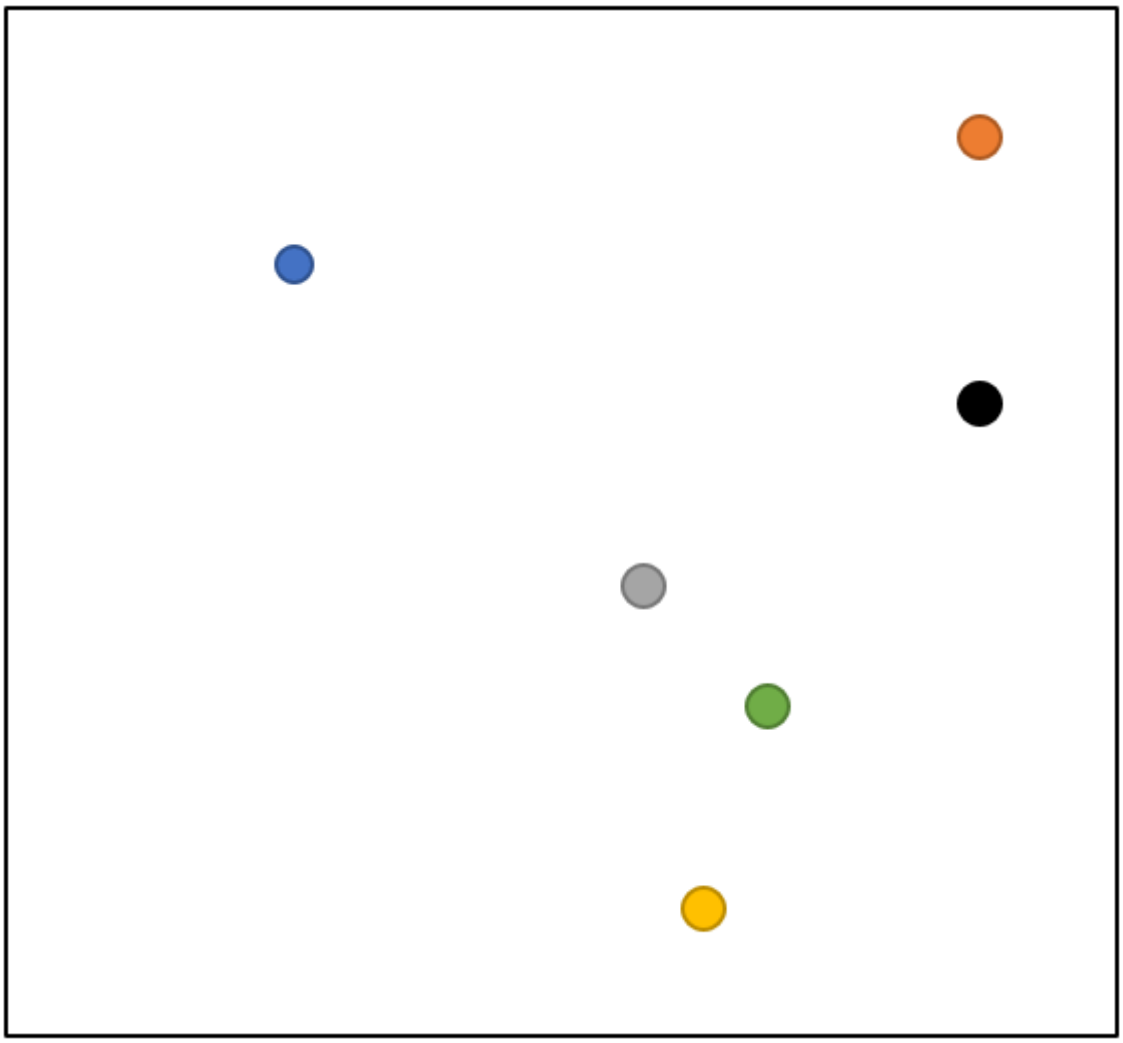


Image 2 :



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.



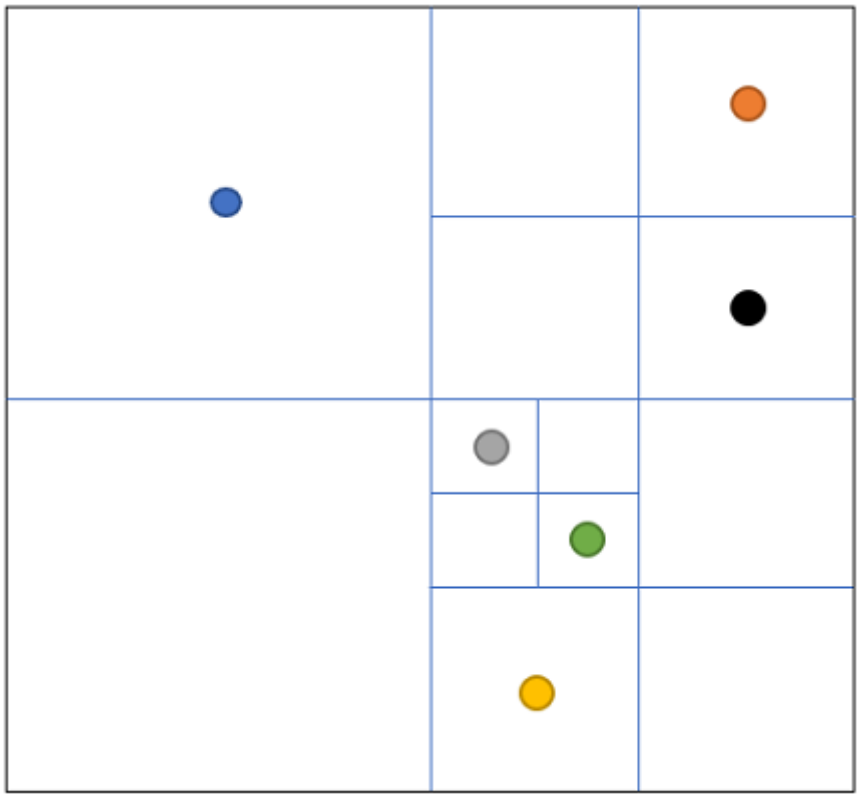
Conseil

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

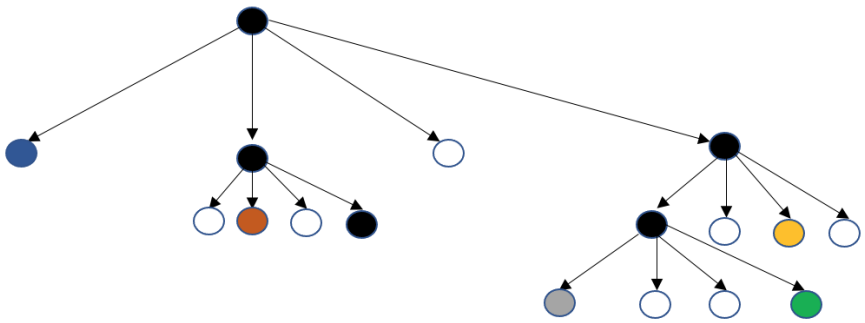
Remarque : Les différentes branches de l'arbre n'auront pas toutes la même profondeur.

Solution

Voici la division de la carte qui permet de construire le Quad-Tree :



Le Quad-Tree qui en résulte :



Note : Un rond blanc correspond à un quadrant vide.