Algorithmes et Pensée Computationnelle

Algorithmes spatiaux

Le but de cette séance est de comprendre la structure de données spatiales et d'implémenter quelques algorithmes d'indexation spatiale. Au terme de cette 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 euclidienne 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 : $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Vous pouvez définir un point comme étant un tuple de deux entiers. Exemple : **point1=(4,5); point2=(1,3)**

Solution

```
1  import math #permet d'importer la librarie nécessaire au calcul de la racine carrée
2
3  def calculate_distance(point1,point2):
4      #Implémentez la formule de la distance euclidienne entre 2 points ici
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. L'algorithme fonctionne de la façon suivante :

1. Traversez chaque point.
2. Pour chaque point, calculez sa distance par rapport au point de départ.
3. Retournez les coordonnées du point le plus proche.

Votre fonction devra retourner les coordonnées (x,y) du point le plus proche ainsi que sa distance par rapport au point de départ.

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).

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

Solution

Note : Veillez à ajouter la fonction définie à l'exercice 1 pour que ce bout de code fonctionne.

```
#Question 2
#WARNING : Veillez ajouter le code de la question 1 au code ci-dessous pour que cela fonctionne.
#Librairie qui permettent de visualiser le résultat (pas nécessaire à la résolution du problème)
import matplotlib.pyplot as plt
import numpy as np
import math

def nearest_neighbor(start, point_set):#start correspond au point de départ, point_set correspond à l'ensemble
        des points
    for i in range(len(point_set)):#on parcourt tous les points de l'ensemble

        if i == 0:
            min_distance = calculate_distance(start, point_set[0])
            nearest_nei = point_set[0]
            #La distance minimale n'étant pas définie, on doit l'initialiser à la première itération, c'est ce qu'on fait ici

        distance = calculate_distance(start, point_set[i])

        if distance < min_distance:
            min_distance = calculate_distance(start, point_set[i])
            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 "enregistre" les
         coordonées du point.

    return nearest_nei, min_distance

a = np.array([(1,2), (5,6), (7,8), (2,5), (9,1)])#List de points
b = (3,4)#Point de départ

point, distance = nearest_neighbor(b,a)
#Devrait retourner [2,5]

print("Le voisin le plus proche est ", point, " avec une distance de ", distance)

#Il n'est pas important de comprendre le bout de code suivant, on l'utilise pour visualiser le résultat.
#Le point de départ apparait en orange, le voisin le plus proche en rouge.
plt.scatter(a[:,0],a[:,1])
plt.scatter(b[0], b[1], color = 'orange')
plt.scatter(point[0], point[1], color = 'green')
```

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

Améliorez l'algorithme créé précédemment afin qu'il puisse trouver les 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 : $[[x_1, y_2, distance1], [x_2, y_2, distance2], ...]$. Avec comme entrée [[2,3],[5,6],[1,4],[2,4],[3,5]], comme point de départ [4,4] et un nombre de voisins K=2, votre algorithme retournera : $[[3, 5, 1.4142], [2, 4, 2.0]]$.

Note : Pour que votre programme fonctionne, écrivez votre algorithme du plus proche voisin dans le même programme que celui des questions 1 et 2. Vous pouvez réutilisez le code des questions 1) et 2) pour cet exercice.

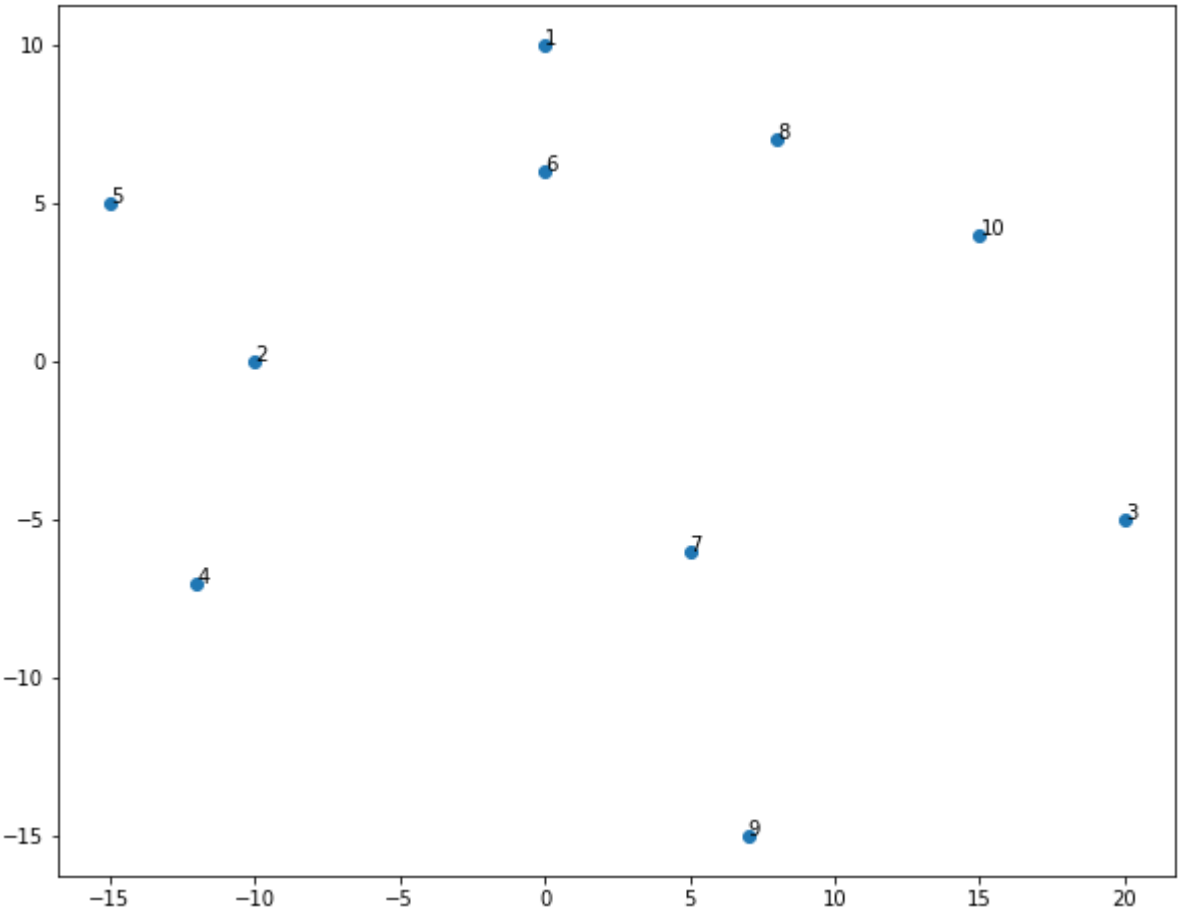
Solution

Note : Veillez ajouter les fonctions des questions 1 et 2 pour que ce bout de code fonctionne.

```
#Question 3
#WARNING :Veillez ajouter les fonctions des questions 1 et 2 pour que le code ci-dessous fonctionne
#Librairie qui permettent de visualiser les données, pas nécessaire pour la résolution de l'exercice
import numpy as np
import matplotlib.pyplot as plt
import math

def K_nearest_neighbor(start,point_set, K):
    temp = point_set #crée une copie de notre ensemble de points
    k_nearest_nei = []

    for j in range(K):#A chaque itération on applique l'algorithme du nearest neighbor mais sur un ensemble de points réduit
        point, distance = nearest_neighbor(start,temp)
        point.append(distance)
        k_nearest_nei.append(point)
        temp.remove(point)#On retire de l'ensemble de points le voisin le plus proche, de cette manière, à chaque itération,
        #le voisin le plus proche sera de plus en plus éloigné.

    return k_nearest_nei

a = [[1,2], [5,8], [7,8], [2,5], [9,1]]#Liste de points
b = (3,4)#Point de départ
K = 2

n = K_nearest_neighbor(b,a,K)
print(n)

#Vous n'avez pas besoin de comprendre ce bout de code, il permet de visualiser votre résultat
a = np.array([[1,2], [5,8], [7,8], [2,5], [9,1]])
plt.scatter(a[:,0],a[:,1])
plt.scatter(b[0], b[1], color = 'Orange')

for i in range(K):
    plt.scatter(n[i][0], n[i][1], color = 'green')
```

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.

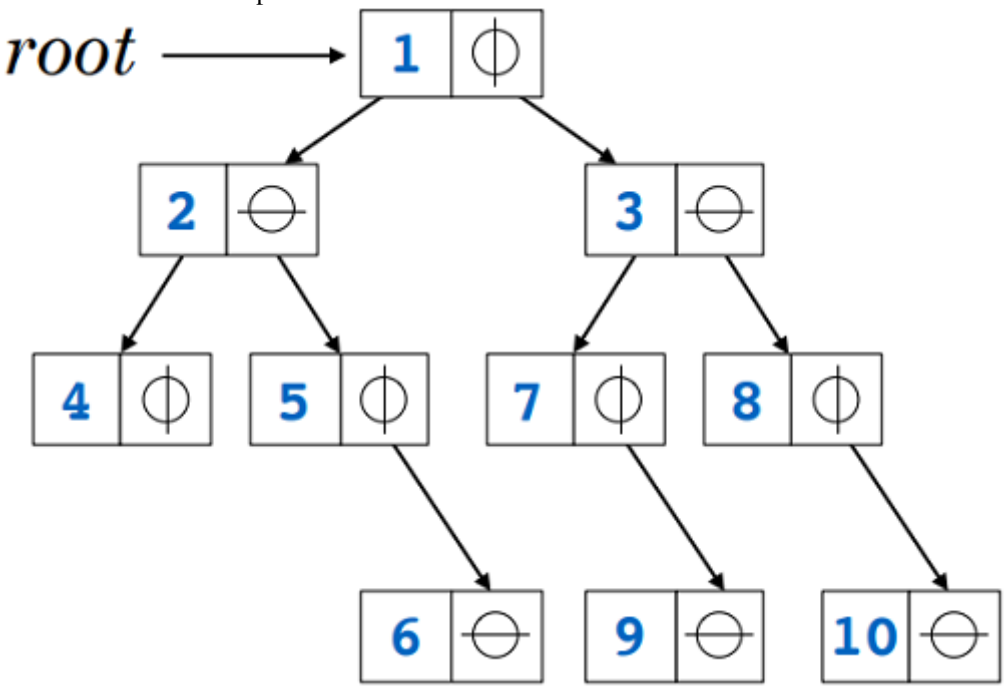


Conseil

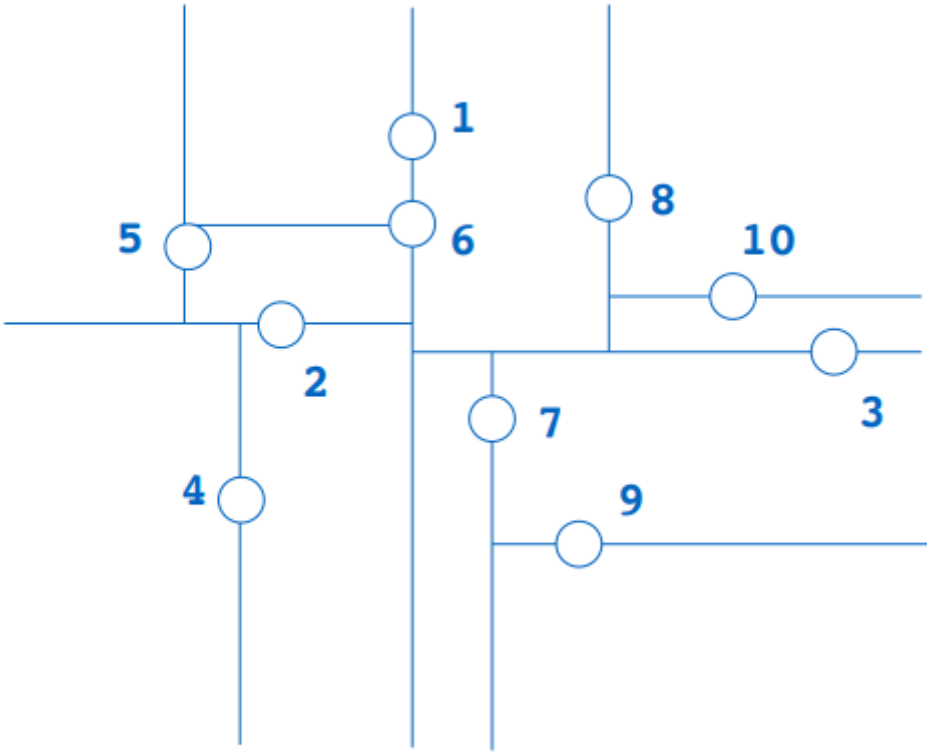
La première division se fait de façon verticale. Assurez-vous de bien insérer les points dans l'ordre (point 1, point 2, etc..). Les nœuds se situant au même niveau devraient diviser l'espace selon le même axe.

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. 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, le code de **Question5.py** devrait afficher : [(0, 10), [(-10, 0), None, None], None].

Solution

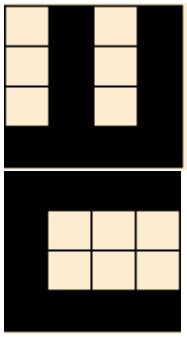
```
1  # Question 5
2
3  root = [(0,10), None, None] #Nous définissons ici juste la racine de l'arbre
4  k = 2 # Ici nous travaillerons en 2 dimensions
5  point = (−10,0) #Point que nous voulons ajouter dans le graphe
6
7  def add_node(node,point,cutaxis = 0):
8
9      if node is None: #Si le noeud n'existe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

Commentaire #1 : 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'un 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 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, SE, SW**.

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

Solution

Image 1 :

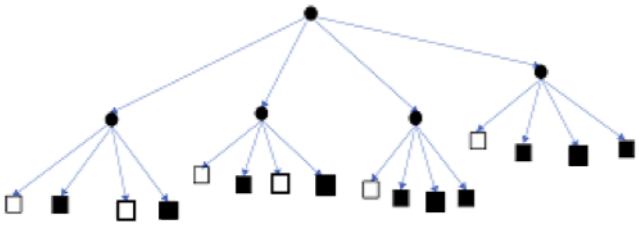
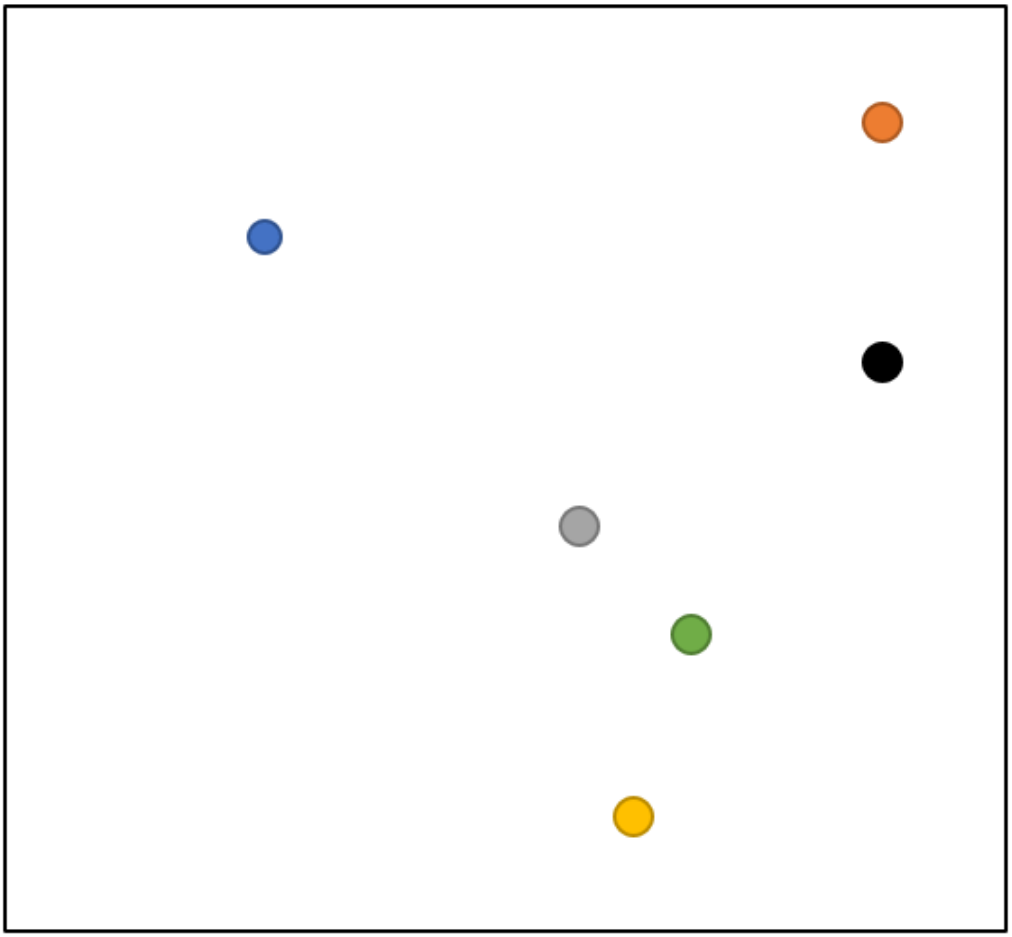


Image 2 :

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

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

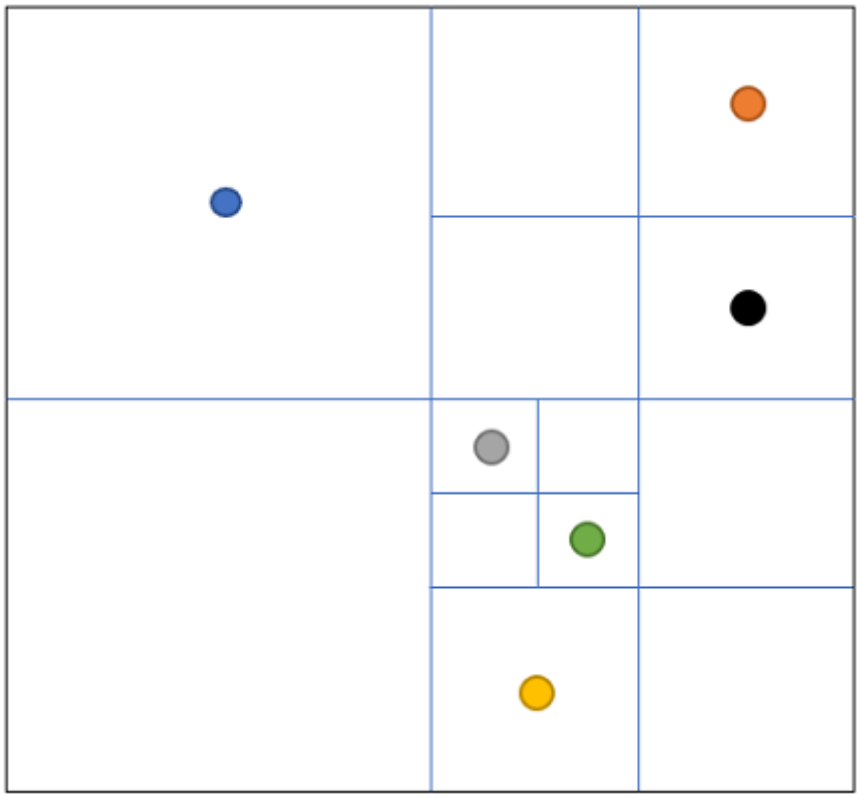


Commencez par diviser la carte de la ville de la façon adéquate puis construisez le graphe.

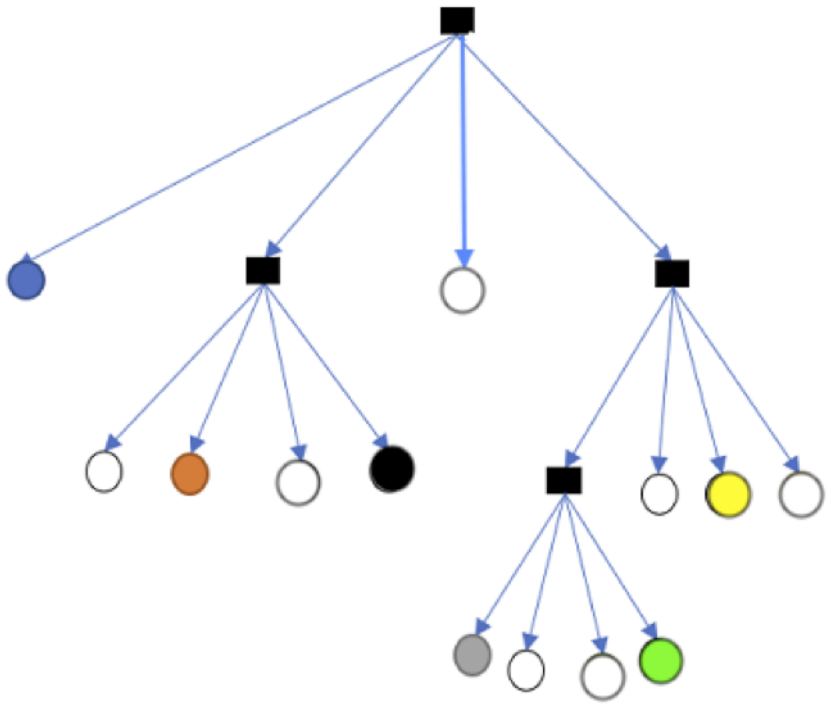
Remarque : Les différentes branches de l'arbre n'auront pas toutes la même profondeur.

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.