R_tree_solutions

November 27, 2019

1 Algorithmique et Pensée Computationnelle

2 R-Tree

Un R-tree est un arbre qui représente une surface et dont chaque branche coupe cette surface en une sous-surface. De cette manière, chaque branche garantit que tous ses enfants seront compris dans la sous-surface correspondant à cette branche.

La donnée est finale est enregistrée dans les feuilles de l'arbre

*Dans le diagramme suivant, on peut voir comment les branches se reflètent sur la surface:

2.1 Exercice

Dans cet exercice, nous utilisons des dictionnaires pour créer des R-tree.

La surface totale est de `400x400`, chaque sous-surface est donnée par un `point (x, y)` qui correspond à l'angle supérieur droit de la surface, `width` correspond à la longueur de la surface et `height` à sa hauteur. Les enfants de chaque branche sont enregistrées dans `children`. Si la valeur de `children` est laissée à `None`, cet élément est une feuille.

[ ]:
```
point = "point"
width = "width"
height = "height"
children = "children"

r_tree = [
    {
        point: (0, 0),
        width: 400,
        height: 400,
        children: [
                {
                    point: (0, 0),
                    width: 200,
                    height: 400,
                    children: [
                        {
```

```
                    point: (0, 0),
                    width: 200,
                    height: 200,
                    children: [
                        {
                            point: (20, 40),
                            width: 20,
                            height: 40,
                            children: None
                        },
                        {
                            point: (100, 10),
                            width: 45,
                            height: 100,
                            children: None
                        },
                        {
                            point: (120, 140),
                            width: 60,
                            height: 10,
                            children: None
                        },
                    ]
                },
                {
                    point: (0, 200),
                    width: 200,
                    height: 200,
                    children: None
                }
            ]
        },
        {
            point: (200, 0),
            width: 200,
            height: 400,
            children: [
                {
                    point: (220, 40),
                    width: 20,
                    height: 40,
                    children: None
                },
                {
                    point: (300, 10),
                    width: 45,
                    height: 100,
```

```
                            children: None
                        },
                        {
                            point: (320, 140),
                            width: 60,
                            height: 10,
                            children: None
                        }
                    ]
                },
        ]
    }
]
```

2.2 Exercice 1

Dessinez cet arbre au papier, ainsi que la surface et les sous-surfaces.

2.3 Il y avait une erreur dans l'énoncé, le `point` signifie l'angle en haut à `GAUCHE`

2.4 Exercice 2

Ecrivez un algorithme qui trouve toutes les feuilles comprises dans une sous-surface

[ ]:
```python
def overlap(point1, point2):
    x1, y1 = point1[point]
    x2, y2 = point2[point]
    width1, height1 = (point1[width], point1[height])
    width2, height2 = (point2[width], point2[height])
    # On vérifie si les deux surfaces ont au moins un point en commun sur l'axe␣
 ↪des X
    if x1 > x2 + width2 or x2 > x1 + width1:
        return False
    # La même avec y, s'ils ont un point en commun à la fois en x et en y, ils␣
 ↪se touchent
    return not (y1 + height1 < y2 or y2 + height2 < y1)

def in_interval(node, surface):
    if not overlap(node, surface):
        return []
    if node[children]:
        elements = []
        for i in node[children]:
            elements += in_interval(i, surface)
        return elements
```

```
    return [node]

in_interval(r_tree[0], {point: (50, 30), width: 200, height: 150})
```

[ ]:

quad_tree_solutions

November 27, 2019

1 Algorithmique et Pensée Computationnelle

1.0.1 Week 9 - Corrections Quad-tree

-> level 3

[ ]:

NN_exercice_solutions

November 27, 2019

1 Algorithmique et Pensée Computationnelle

1.1 Nearest neighbor search

On l'appele **Recherche des plus proches voisins** en français.

La recherche des plus proches voisins, ou des k plus proches voisins, est un problème algorithmique classique. De façon informelle le problème consiste, étant donné un point à trouver dans un ensemble d'autres points, quels sont les k plus proches.

Cliquez ici pour plus d'informations.

Calculer la distance entre deux points en utilisant la formule de distance euclidienne. Voici la formule :

point1 = (x1, y1)
point2 = (x2, y2)

distance = Racine Carrée de ((x1 - x2)^2 + (y1 - y2)^2)

Vous pouvez utiliser **math.sqrt(...)** pour calculer la racine carrée d'un nombre.

[ ]:
```
import math

def calculate_distance(point1, point2):
    # votre code ici
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
```

On vous donne une liste de points et un point pour trouver son voisin le plus proche.
Suivez les étapes suivantes pour implémenter l'algorithme du voisin le plus proche.

1. Déplacez tous les points.
2. Pour chaque point, calculez la distance entre le nœud actuel et le nœud donné.
3. Retournez le nœud le plus proche et sa distance au nœud donné.

[ ]:
```
def nearest_neighbor(thePoint, pointList):
    #votre code ici
    closest_point = None
    min_value = 999999
    for p in pointList:
        distance = calculate_distance(p, thePoint)
```

```
        if distance < min_value:
            min_value = distance
            closest_point = p
    return (closest_point, min_value)
```

[ ]:
```
pointList = [[2,3],[5,6],[1,4],[2,4],[3,5]]
point = [1,2]
result = nearestNeighbor(point, pointList)
print("Point: {0} Distance: {1}".format(result[0], result[1]))
```

[ ]:
```

```

Kd-Trees-solutions

November 27, 2019

1 Algorithmique et Pensée Computationnelle

2 Kd-trees

2.1 Exercice papier

Insérez les données suivantes dans un Kd-tree où k = 2 avec une représentation sous forme d'arbre et une sous forme d'un plan 2d (x,y) :

(51,75),(70,70),(25,40),(35,90),(60,80),(10,30),(1,10),(55,1),(50,50)

À l'aide de votre arbre, trouvez les deux points qui ont les valeurs minimales pour x et y

Les points sont: (1,10) et (55,1)

2.2 Exercice Python

Complètez la fonction ci-dessus pour ajouter un point dans un arbre, sa racine est sous la forme:

root = [(tuple de valeur), enfant à gauche(node), enfant à droite(node)]

NB: Les noeuds sont tous sous la forme ci-dessus ((valeur), noeud de gauche, noeud de droite).

par exemple :

root = [(51,75), None, None]

[9]:
```
def add_node(node,point,cutaxis = 0):
    value = 0
    left = 1
    right = 2
    #les variables ci-dessus représentent les positions dans la liste qui␣
 ↪represente un noeud.

    if node is None:
        node = [point, None, None]
        return node

    elif point == node[value]:
```

```
        raise Exception("Duplicate")

    elif point[cutaxis] < node[value][cutaxis]:
        node[left] = add_node(node[left], point, (cutaxis + 1 ) % k)

    else:
        node[right] = add_node(node[right],point, (cutaxis + 1 ) % k)
```

[10]:
```
root = [(51,75), None, None]
k = 2 #le nombre de dimension
point = (61,77)
add_node(root,point,0)
root
```

[10]:
```
[(51, 75), None, [(61, 77), None, None]]
```

[ ]:
```

```