Algorithmique et structures de données : projet

Hamza Bouihi, Mamadou Thiongane

7 mai 2023

Table des matières

On s'intéresse dans ce projet à l'identification de structures dans un nuage de points. On considère en entrée un ensemble de points distincts du plan ainsi qu'une distance seuil δ. Deux points p_i et p_j sont proches si $d(p_i, p_j) \leq \delta$.

À partir de cette information, il est possible d'extraire un graphe un graphe $G = (V, E)$ tel que

- tout point p_i correspond à un sommet $v_i \in V$ du graphe
- tout couple (p_i, p_j) de points proches correspond à une arête $(v_1, v_2) \in E$ du graphe

Dans ce projet, on cherche à calculer la distribution des tailles des composantes connexes de G.

1 Algorithme naïf

1.1 Principe de l'algorithme

L'algorithme se découpe de la manière suivante :

- la fonction `explore_connexe_comp` qui renvoie la taille d'une composante connexe. Pour cela, on crée une variable `pile` qui contient la liste des points à traiter dans la composante étudiée. Tant qu'elle est non vide, on itère sur TOUS les points en vérifiant que le point n'est pas dans la pile ou déjà traité avant de vérifier s'il est suffisamment proche pour le rajouter à la composante connexe.
- la fonction `print_components_sizes` qui se contente d'itérer sur la liste et d'appliquer `explore_connexe_comp` lorsqu'un point n'a pas été affecté à une composante connexe.

1.2 Implémentation

On peut réaliser l'algorithme suivant.

```
1  def explore_connexe_comp(dist, points, visited, first):
2      size, pile = 0, [first]
3      # Tant que la pile des voisins du sommet traité est non vide faire :
4      while pile:
5          indice_pts = pile.pop()
6          # Rend le point courrant traité
7          visited[indice_pts] = True
8          size += 1
9          # Recherche de voisins
10         for i, point in enumerate(points):
11             if not visited[i] and i not in pile:
12                     if points[indice_pts].distance_to(points[i]) <= dist:
13                         pile.append(i)
14     return size
15
16 def print_components_sizes(dist, points):
17     n_pts, components_sizes, grid = len(points), [], {}
18     visited = [False] * n_pts
19     # Tant que les sommets n'ont pas tous été traités, appeler la fonction
20     for k in range(n_pts):
21         if not(visited[k]):
22             size = explore_connexe_comp(dist, points, visited, k)
23             components_sizes += [size]
24     components_sizes.sort(reverse=True)
25     print(components_sizes)
```

1.3 Complexité

Dans toute la suite, on admettra que la complexité moyenne de la méthode `sort` est en $O(n\log(n))$.

On note $C(n)$ la complexité de la fonction `print_components_sizes` et $Q_i(n)$ la complexité de la fonction `explore_connexe_comp`.

$$C(n) = O\left(\sum_{i=1}^{n} Q_i(n)\right)$$

En notant k_i la taille de la i-ème composante connexe (éventuellement nulle),

$$C(n) = O\left(n\sum_{i=1}^{n} k_i\right) = O(n^2)$$

car $\displaystyle\sum_{i=1}^{n} k_i = n$.

2 Algorithme efficace : approche unidimensionnelle

2.1 Principe de l'algorithme

Théorème 1. *Soit $p_i = (x_i, y_i)$ et $p_j = (x_j, y_j)$ deux points. Si $|x_j - x_i| > \delta$, alors $(p_i, p_j) \notin E$.*

Démonstration. Si $(p_i, p_j) \in E$, alors $|x_j - x_i| \leq \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} = d(p_i, p_j) \leq \delta$, d'où le résultat par contraposition. □

On peut alors trier la liste de points dans l'ordre lexicographique afin de réduire le nombre de calculs. En effet, pour chercher les points p_j proches d'un point p_i donné, il suffit de parcourir la liste à partir du premier point p non traité jusqu'à ce que la condition $x_j - x_i \leq \delta$ ne soit plus vérifiée car les points suivants ne peuvent pas appartenir à la composante connexe de p_i d'après le théorème 1. Il faut également réaliser un parcours décroissant entre p et p_i pour vérifier l'existence de points non traités proches de p mais pas de p_i.

2.2 Implémentation

On obtient l'algorithme suivant.

```
1  def print_components_sizes(distance, points):
2      n_points = len(points)
3      points.sort()
4      components_sizes = []
5
6      def component_size(i, j):
7          if j_min < j < n_points:
8              if not points[j].processed:
9                  point, other = points[i], points[j]
10                 delta = abs(other.coordinates[0] - point.coordinates[0])
11                 if delta <= distance:
12                     if point.distance_to(other) <= distance:
13                         points[j].processed = True
14                         components_sizes[-1] += 1
15                         component_size(j, j+1)
```

```
                        component_size(j, j-1)
                    if i < j:
                        component_size(i, j+1)
                    else:
                        component_size(i, j-1)
            else:
                if i < j:
                    component_size(i, j+1)
                else:
                    component_size(i, j-1)

    for i in range(n_points):
        if not points[i].processed:
            points[i].processed = True
            components_sizes.append(1)
            j_min = i
            component_size(i, i+1)

    components_sizes.sort(reverse=True)
    print(components_sizes)
```

2.3 Complexité et performances

Dans le meilleur cas, le graphe n'admet qu'une composante connexe et la distribution de points est telle que chaque point n'admet qu'un unique point proche dans sa zone de recherche, auquel cas la complexité de la fonction `connectes_list` est en $O(n \log(n))$ (en raison de de la complexité de la méthode `sort`).

Dans le pire cas, la distance seuil est telle que la zone de recherche regroupe un trop important nombre de points sans qu'ils soient suffisamment proches et la complexité est alors en $O(n^2)$.

Pour mesurer les performances de nos algorithmes, on génère aléatoirement une série d'instances en faisant varier le nombre de points ainsi que la distance seuil puis on exécute nos algorithmes avec les instances générées.

Pour la fonction `connectes_list`, on obtient les performances suivantes en figures 1 et 2.

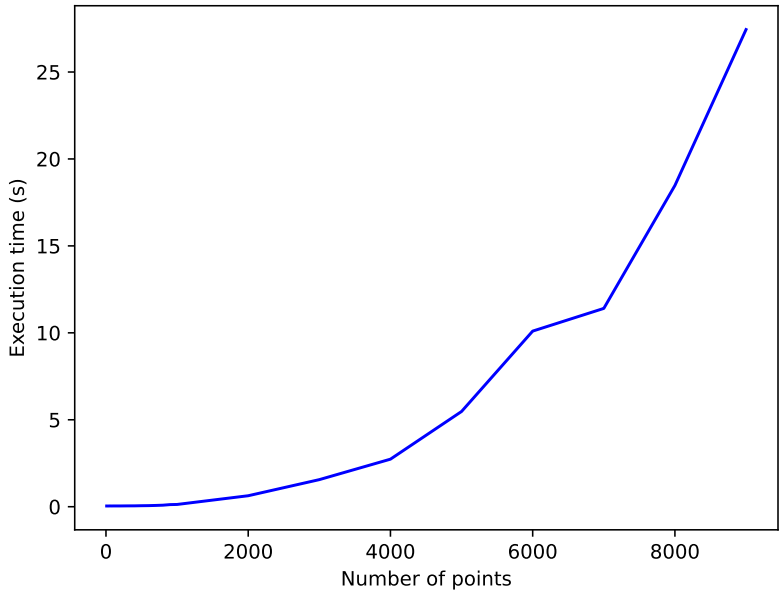

FIGURE 1 – Courbe de performances temporelles de `connectes_list` en fonction du nombre de points (distance = 0.05)

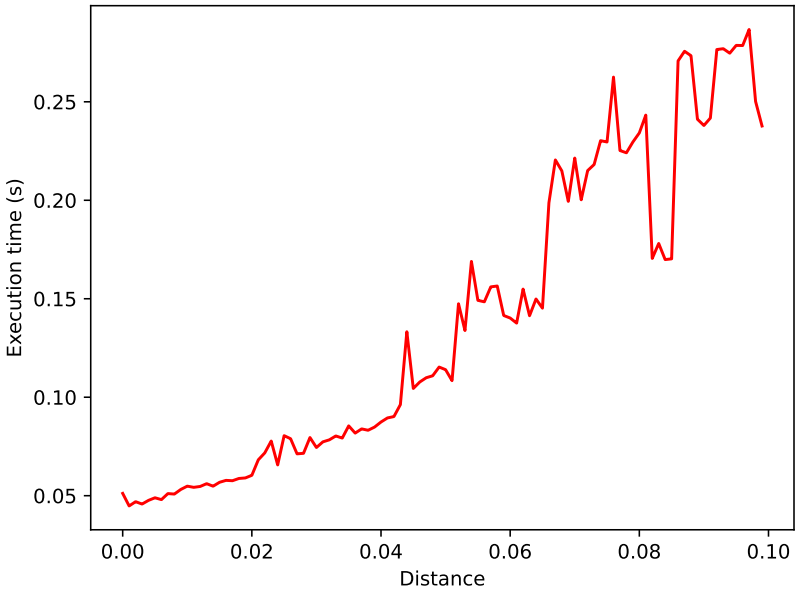

FIGURE 2 – Courbe de performances temporelles de `connectes_list` en fonction de la distance (pour 1000 points)

Sur la figure 2, on observe que le temps d'exécution augmente considérablement avec la distance. Ce résultat était prévisible : en effet, plus la distance est petite, plus la zone de recherche est réduite, ce qui diminue le nombre de calcul.

3 Algorithme efficace : approche bidimensionnelle

3.1 Algorithme utilisant un dictionnaire

3.1.1 Structure de dictionnaire

Un dictionnaire est une collection d'objets non-ordonnée. Chaque élément qu'il possède se compose d'une paire clé-valeur.

Comme les listes, les dictionnaires sont des objets muables et dynamiques. Néanmoins, il possède une propriété particulièrement intéressante en plus : deux éléments différents peuvent avoir la même clé (contrairement aux listes, si on considère que l'indice est une clé).

Cette propriété s'avère être très utile lorsque l'on est amené à regrouper des éléments qui ont une propriété commune : la clé.

3.1.2 Principe de l'algorithme

L'idée principale de l'algorithme est simple : réduire la liste des points qui pourraient appartenir à une composante connexe.

Pour cela, on quadrille le plan afin que le test d'appartenance à la composante connexe porte uniquement sur les carrés voisins du point que l'on est en train de traiter (cf. figure 3).

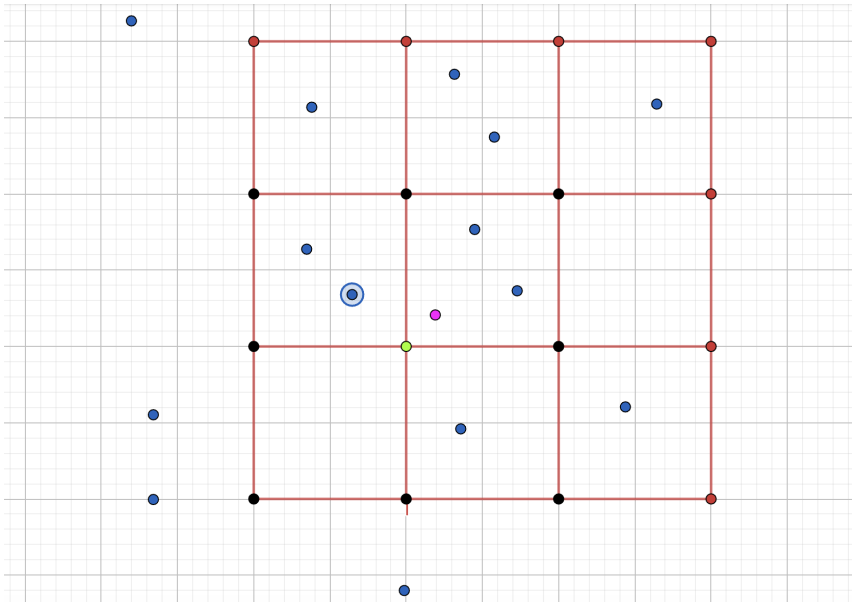

FIGURE 3 – Illustration de l'algorithme

La structure implémentée est un dictionnaire, nommé `grid`, et est implémentée de la manière suivante :

- **clé** : approximation à l'entier près des points de sorte à créer des paquets de points (correspond au point en vert fluo sur la figure 3)
- **valeur** : dictionnaire nommé `square` ayant pour clés les indices des points appartenant au carré et pour valeur les points qui correspondent aux indices (points bleus dans les carrés)

Ainsi, pour énumérer tous les points qui se situent à une distance inférieure au `seuil` du point courant que l'on traite (le violet sur la figure 3), il suffit de tester avec ceux qui appartiennent au même carré que le point courant et aux carrés voisins de celui-ci. Ceci est possible grâce à la variable `square_neighbor`.

3.1.3 Implémentation

On peut implémenter l'algorithme suivant.

```python
def explore_connexe_comp(dist, points, visited, grid, first, square_neighbor):
    size, pile = 0, set([first])

    # Tant que la pile des voisins du sommet traité est non vide faire:
    while pile:
        indice_pts = pile.pop()
        current_pts = points[indice_pts]
        approx_pts = int(current_pts.coordinates[0]/dist),
        int(current_pts.coordinates[1]/dist)

        # Rend le point courant traité
        visited[indice_pts] = True
        grid[approx_pts].pop(indice_pts)
        size += 1

        # Recherche optimisée de possibles voisins
        for coord in square_neighbor:
            key = (approx_pts[0] + coord[0], approx_pts[1] + coord[1])
            for index_possible_neighbor in grid.get(key, {}):
                if not visited[index_possible_neighbor]:
                    if points[indice_pts].distance_to(
                        grid[key][index_possible_neighbor]
                        ) <= dist :
                        pile.add(index_possible_neighbor)

    return size

def print_components_sizes(dist, points):
    n_pts, components_sizes, grid = len(points), [], {}
    visited = [False] * n_pts
    square_neighbor = [(-1, -1), (0, -1), (1, -1), (-1, 0),
                       (0, 0), (1, 0), (-1, 1), (0, 1), (1, 1)]

    # On ordonne les points par paquets qui appartiennent
    # au même carré de coté dist grâce à l'approximation
    # à l'entier le plus proche + structure de dictionnaire
    for k in range(n_pts):
        pts_k = points[k]
        approx_pts = int(pts_k.coordinates[0]/dist),
        int(pts_k.coordinates[1]/dist)
        square = grid.get(approx_pts, {})
        square[k] = pts_k
        grid[approx_pts] = square

    # Tant que les sommets n'ont pas tous été traités, appeler la fonction
    for k in range(n_pts):
        if not visited[k]:
            size = explore_connexe_comp(
                dist, points, visited, grid, k, square_neighbor
                )
            components_sizes.append(size)

    components_sizes.sort(reverse=True)
    print(components_sizes)
```

3.1.4 Complexité et performances

Pour la fonction `connectes_dict`, on obtient les performances suivantes en figures 4 et 5.

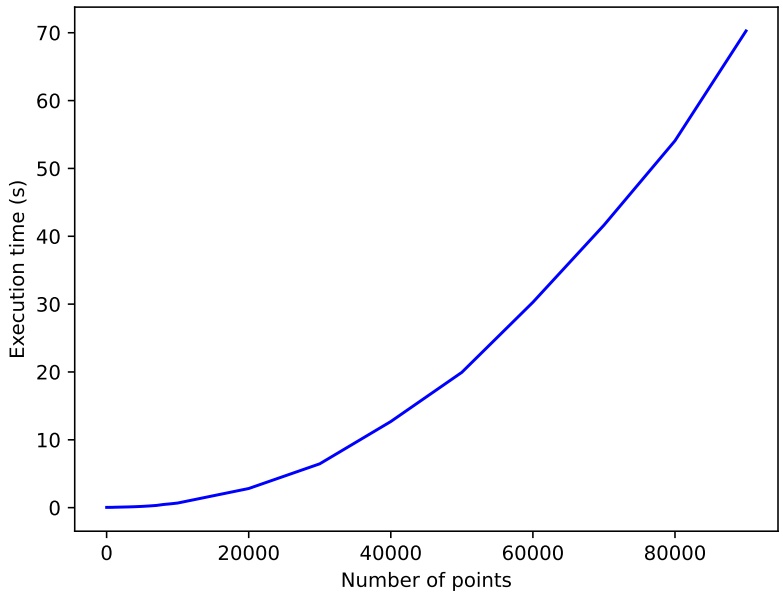

FIGURE 4 – Courbe de performances temporelles de `connectes_dict` en fonction du nombre de points (distance = 0.05)

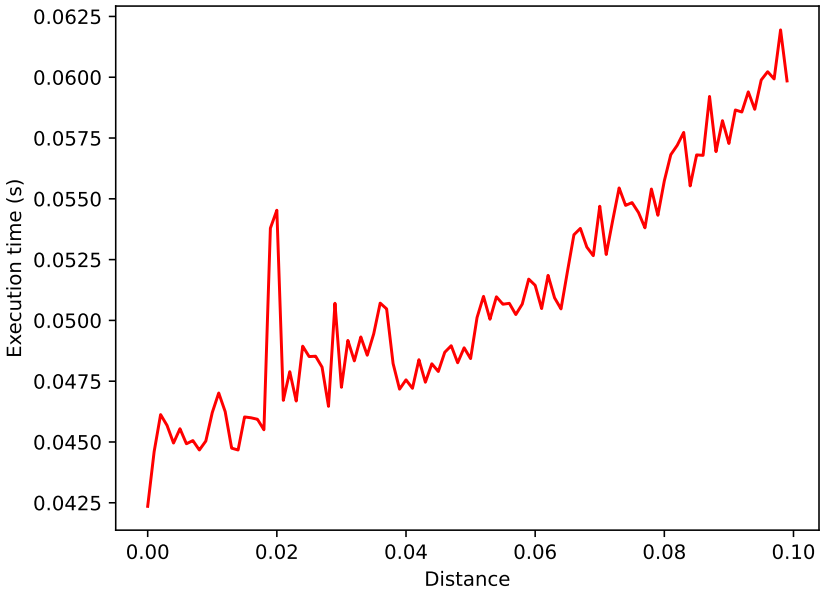

FIGURE 5 – Courbe de performances temporelles de `connectes_dict` en fonction de la distance (pour 10000 points)

3.2 Algorithme utilisant un arbre binaire de recherche

3.2.1 Structure d'arbre bidimensionnel

Un arbre bidimensionnel est une structure de données de partition du plan permettant de stocker des points.

Un arbre bidimensionnel est un arbre binaire dont chaque nœud contient un point. Si $p_i = (x_i, y_i)$ est un point stocké dans un nœud à profondeur paire (resp. impaire), alors, pour tout point $p_j = (x_j, y_j)$:

- si p_j appartient au sous-arbre gauche du nœud, $x_j < x_i$ (resp. $y_j < y_i$)
- si p_j appartient au sous-arbre droit du nœud, $x_j > x_i$ (resp. $y_j > y_i$)

On implémente une classe `Tree` correspondant à cette structure de données.

```python
class Node:
    def __init__(self, point, axis, left=None, right=None):
        self.point = point
        self.axis = axis
        self.left = left
        self.right = right


class Tree:
    def __init__(self, points):
        self.root = self.construct(points)

    def construct(self, points, depth=0):
        if not points:
            return None

        axis = depth % 2
        median = mediane(points, axis)
        m = len(points) // 2

        return Node(
            median,
            axis,
            self.construct(points[:m], depth+1),
            self.construct(points[m+1:], depth+1)
        )
```

Cette classe utilise une fonction `mediane` qui reprend le principe du tri rapide afin d'obtenir une complexité minimale en $O(n)$.

```python
def partition(points, axis, g, d):
    pivot = points[g]
    m = g
    for i in range(g+1, d):
        if points[i].coordinates[axis] < pivot.coordinates[axis]:
            m += 1
            if i > m:
                points[i], points[m] = points[m], points[i]
    if m > g:
        points[m], points[g] = points[g], points[m]
    return m


```

```
def mediane(points, axis):
    k = len(points) // 2

    def aux(g, d):
        m = partition(points, axis, g, d)
        if m == k:
            return points[m]
        elif m < k:
            return aux(m+1, d)
        else:
            return aux(g, m)

    return aux(0, len(points))
```

La construction de l'arbre est alors un algorithme "diviser pour régner" dont la complexité vérifie $C(n) = 2C(\frac{n}{2}) + O(n)$ soit $C(n) = O(n\log(n))$, d'après le master theorem.

3.2.2 Principe de l'algorithme

Pour chaque point p_i de la liste de points, on réalise un parcours en profondeur depuis la racine de l'arbre en utilisant les propriétés de l'arbre bidimensionnel pour réduire le nombre de calculs :

- on considère le point p_j stocké à la racine et on calcule la différence $d_x = x_j - x_i$
- si $|d_x| \leq \delta$, alors on vérifie que p_j n'a pas été traité puis on vérifie si $d(p_i, p_j) \leq \delta$. Dans ce cas, on marque p_j comme étant traité puis on parcours de nouveau l'arbre depuis la racine avec p_j. On continue ensuite la recherche en profondeur avec p_i dans les sous-arbres enracinés en p_j selon la coordonnée y
- sinon, on a $d_x > \delta$ (resp. $d_x < -\delta$) et il suffit de continuer le parcours en profondeur uniquement dans le sous-arbre enraciné gauche (resp. droit) en p_j

3.2.3 Implémentation

L'algorithme correspondant est le suivant.

```
def print_components_sizes(distance, points):
    tree = Tree(points)
    components_sizes = []

    def search(node, point):
        if node is not None:
            other = node.point
            axis = node.axis
            delta = other.coordinates[axis] - point.coordinates[axis]
            if abs(delta) <= distance:
                if not other.processed and point.distance_to(other) <= distance:
                    other.processed = True
                    components_sizes[-1] += 1
                    search(tree.root, other)
                search(node.left, point)
                search(node.right, point)
            elif delta > 0:
                search(node.left, point)
            else:
                search(node.right, point)

    for point in points:
        if not point.processed:
```

```
            point.processed = True
            components_sizes.append(1)
            search(tree.root, point)

    components_sizes.sort(reverse=True)
    print(components_sizes)
```

3.2.4 Complexité et performances

La fonction de construction de l'arbre bidimensionel est de complexité en $O(n \log(n))$.

Dans le meilleur cas, le parcours en profondeur dans l'arbre ne se réalise que dans un sous-arbre à chaque appel et la complexité est en $O(\log(n))$. Dans le pire cas, le parcours en profondeur a une complexité en $O(n)$.

Ainsi, la fonction `connectes_tree` a une complexité quasi-linéaire.

Pour la fonction `connectes_tree`, on obtient les performances suivantes en figures 6 et 7.

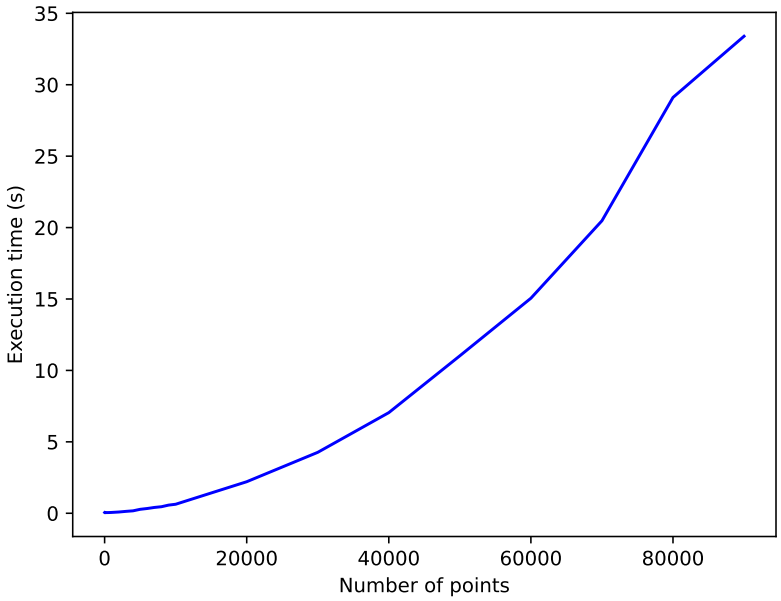

FIGURE 6 – Courbe de performances temporelles de `connectes_tree` en fonction du nombre de points (distance = 0.05)

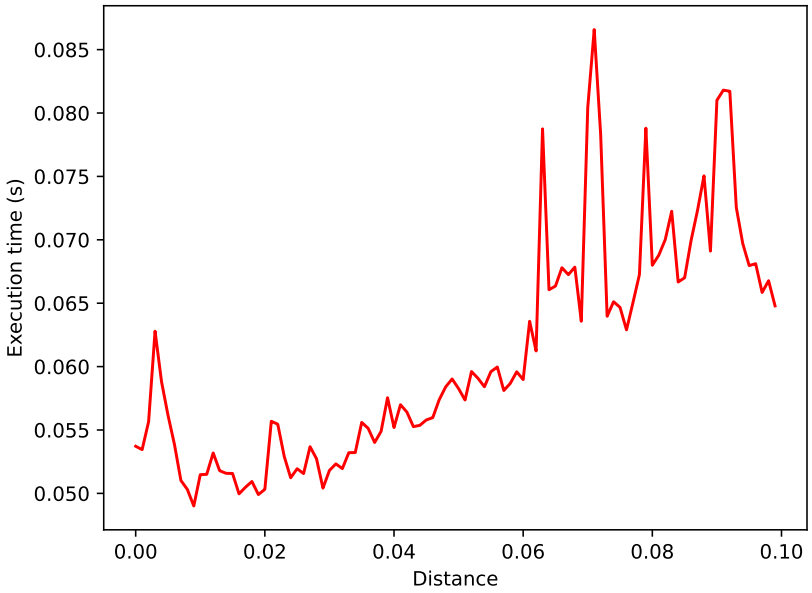

FIGURE 7 – Courbe de performances temporelles de `connectes_tree` en fonction de la distance (pour 10000 points)

Sur la figure 7, on remarque que `connectes_tree` est plus performant lorsque le nombre de composantes connexes est important.

4 Conclusion

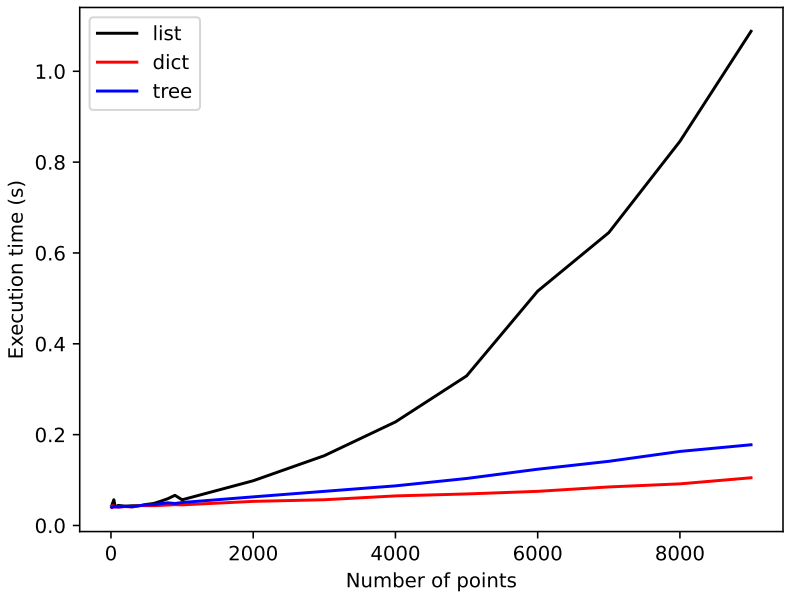

FIGURE 8 – Courbe de performances temporelles des différents algorithmes en fonction du nombre de points (distance = 0.01)

D'après la figure 8, la fonction `connectes_list` à un coût plus élevé que les deux autres fonctions, notamment lorsque que le nombre de points devient important. On peut conclure qu'une approche bidimensionnelle est préférable à une approche unidimensionnelle.

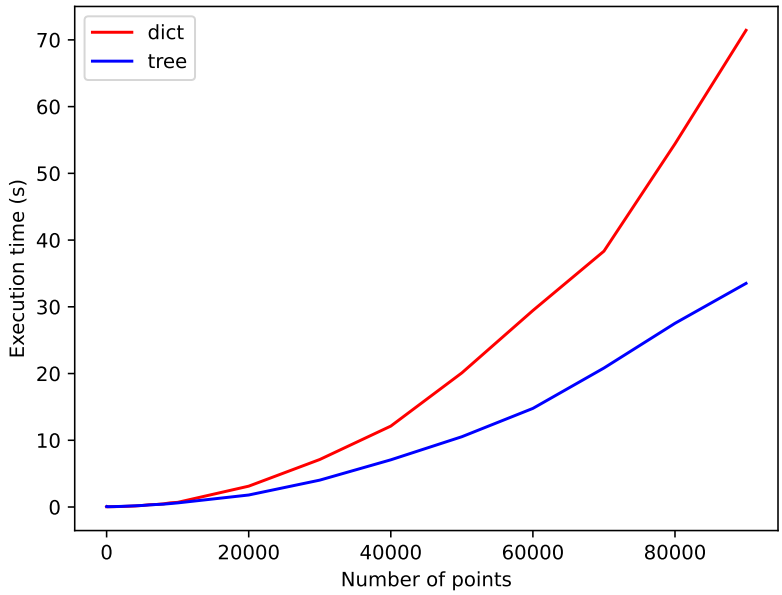

FIGURE 9 – Courbe de performances temporelles des différents algorithmes en fonction du nombre de points (distance = 0.05)

Malgré son efficacité pour un très grand nombre de points par rapport à `connect_dict`, un inconvénient de `connect_tree` est le nombre important d'appels récursifs qu'il nécessite, ce qui peut empêcher l'exécution du programme si ce nombre dépasse la limite autorisée. Si nous avions eu plus de temps, nous aurions tenté réaliser une version itérative de ce programme.