ALGORITHMIQUE ET STRUCTURES DE DONNÉES

El Marjou Youssef – Faraj Riham

1 – Introduction :

Dans de nombreux domaines de l'informatique, la recherche de composantes connexes est une tâche d'une importance majeure, notamment dans la reconnaissance d'images, l'analyse de réseaux, la modélisation de systèmes complexes ou encore la gestion de bases de données.

Dans la reconnaissance d'images par exemple, les composantes connexes sont souvent utilisées pour identifier les objets dans une image en reliant les pixels qui appartiennent à un même objet. Dans l'analyse de réseaux, elles sont plutôt utilisées pour identifier les groupes de nœuds connectés dans un réseau, ce qui permet de comprendre la structure et les propriétés du réseau en question.

La recherche de composantes connexes peut être une tâche ardue et coûteuse en temps, spécialement pour les ensembles de données volumineux, il est donc avantageux, voire nécessaire, de trouver des algorithmes rapides de recherche de composantes connexes. Ces algorithmes permettent de trouver les composantes connexes rapidement et efficacement, ce qui facilite leur utilisation dans de nombreux domaines de l'informatique.

Dans ce cadre, le sujet proposé cette année est, à partir d'un fichier contenant des points dans un plan, de trouver les composantes connexes sachant que deux points sont liés si la distance entre eux est inférieure ou égale à la distance seuil.

2 – Enjeux et difficultés :

L'enjeu fondamental de ce défi est la contrainte temporelle ; un bon algorithme de recherche de composantes connexes est celui qui répond à la problématique sans prendre beaucoup de temps, il sera donc indispensable d'optimiser l'algorithme au fur et à mesure pour de meilleures performances. Ainsi, les 4 tests proposés forment une aide qui souligne l'efficacité de notre programme, et poussent naturellement chaque équipe à trouver un meilleur algorithme du fait des high-scores affichés.

Si on suppose avoir trouvé un algorithme qui répond à la problématique, l'enjeu second est de pouvoir traduire cela en un

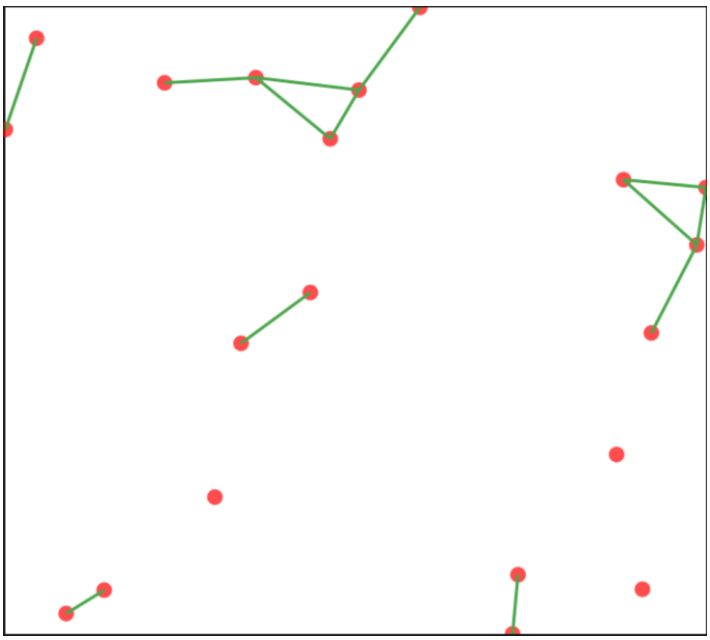
programme en Python, ce qui demande une connaissance des structures de données et leurs caractéristiques, comme par exemple, une table de hachage peut être implémentée directement par un dictionnaire, ou même avec des ensembles « set() ».

3 – Fichiers utiles :

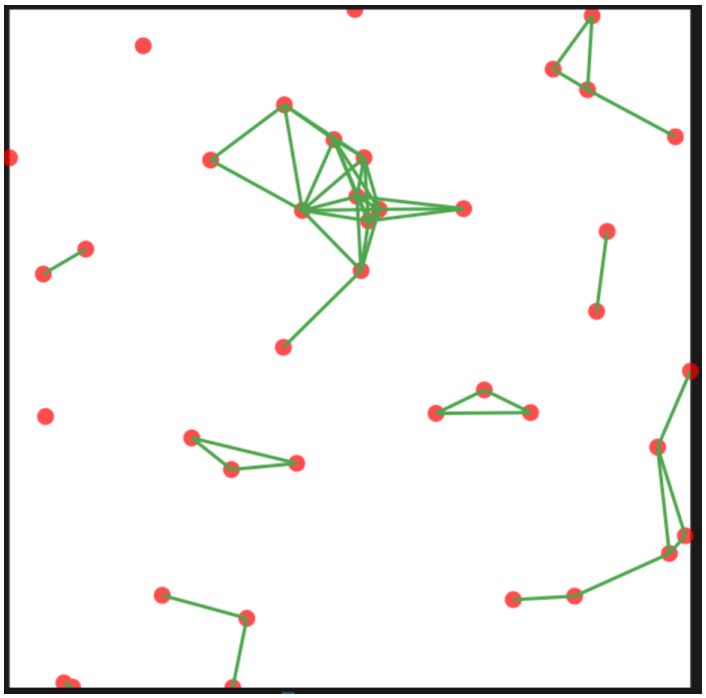
A) afficheur.py :

Tout au long de nos essais, on a jugé essentiel d'avoir en main un programme capable de traduire les fichiers de points en des graphes équivalents, ce qui permet d'avoir, dans un premier temps, une meilleure vision et approche de la problématique, mais aussi lors de la création de tests personnels ; si on suppose vouloir tester si le programme marche bien, il sera indispensable de le tester sur plusieurs fichiers, puisque ceux proposés ne sont pas suffisants et ne traitent pas généralement tous les cas possibles (graphe nul, graphe complet …)

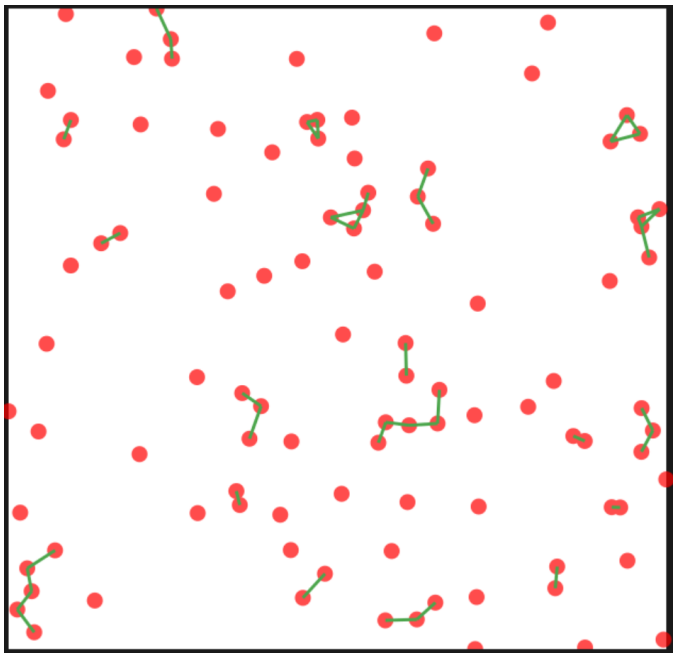
Le fichier 'afficheur.py' est un programme en Python qui permet de dessiner le graphe équivalent au fichier donné en argument et affiche le résultat dans '*terminology*', grâce au module '*tycat*'. Voici le résultat de quelques exemples :



Exemple 1



Exemple 2



Exemple 3

B) generator.py :

L'utilité de ce programme est expliquée dans la prochaine partie. Il prend en argument, dans le terminal, un entier naturel ' *N* ', et génère un fichier contenant *N* points de coordonnées aléatoires, et une distance seuil aussi aléatoire. Le fichier est nommé : '*exemple_*.pts*', où '*' est remplacée par *N*.

Pour *N = 10,* cela donne le fichier ' *exemple_10.pts* ' suivant :

0.01282383920278367

0.39972790340360753, 0.43311699253403624

0.6991537453724527, 0.022942126652968242

0.8348752489472522, 0.9905265430697814

0.9736913334337036, 0.2738886549714161

0.6189850341177846, 0.9722009001088622

0.9050613375931977, 0.15455935012339972

0.30374372839311503, 0.5168798302715667

0.1935010894029794, 0.2533085314484771

0.5032251892340668, 0.16246895109756043

0.6519802146751859, 0.8677910277676916

C) test.py :

Ce programme est d'une grande importance, car c'est grâce à lui qu'on estimera la complexité temporelle de notre algorithme, mais aussi de comparer les anciennes et nouvelles versions de ce dernier.

Ce programme prend en argument un entier '*N*', et affiche une courbe qui représente le nombre de points dans un fichier traité en fonction du temps de traitement de ce dernier. D'où l'utilité de 'generator.py' puisqu'il permet de créer un grand nombre de fichiers de points aléatoires à tester.

Par exemple, si on choisit de faire './test.py 20' dans l'état courant du fichier (car on peut le modifier selon nos fins : traiter qu'un algorithme, en comparer plusieurs, modifier le pas), cela va afficher

une courbe montrant l'évolution de la durée de test avec l'augmentation du nombre de points. On verra des exemples de tests dans la prochaine partie.

4 – Algorithmes proposées :

A) Depth-First-Search (version naïve):

En utilisant un algorithme de parcours à partir d'un sommet initial, les sommets visités correspondent à la composante connexe à laquelle appartient le sommet initial. Pour obtenir toutes les composantes connexes d'un graphe, il suffit de répéter le parcours à partir d'un sommet non encore visité, tant qu'il en reste. Ainsi, chaque nouveau parcours permet de découvrir une nouvelle composante connexe du graphe. Pour la simple raison qu'il est facile à implémenter, on a fait le choix d'utiliser le DFS comme parcours de graphe.

Notons que le premier problème auquel on est confronté, est de pouvoir traduire le fichier en une structure de donnée qui représente le graphe équivalent ; de manière naïve, on part sur l'utilisation de matrice d'adjacence avec le code suivant :

```
adjacency_matrix = [[0 for _ in range(n)] for _ in range(n)]
for i in range(n):
    for j in range(i+1, n):
        adjacency_matrix[i][j], adjacency_matrix[j][i] = testing(points[i], points[j]), testing(points[i], points[j])
```

La création de la liste '*adjacency_matrix*' en compréhension a pour complexité :

$$O(n^2) + O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

Après la création de la matrice, il suffit d'appliquer le parcours en question pour obtenir les composantes connexes :

```
ALL = []
NOT_SEEN = [i for i in range(len(adjacency_matrix))]
while NOT_SEEN:
    x = NOT_SEEN[0]
    seen = [x]
    pile = [x]
    while pile:
        l = diff(voisins(pile[-1], adjacency_matrix), seen)
        if l:
            seen.append(l[0])
            pile.append(l[0])
        else:
            pile.pop(-1)
    for i in seen:
        if i in NOT_SEEN:
            NOT_SEEN.remove(i)
    ALL.append(seen)
final = [len(x) for x in ALL]
final.sort(reverse=True)
print(f"{final}")
```

Cherchons la complexité dans le pire des cas :

Pour la première boucle, elle est exécutée autant de fois qu'il y a de composantes connexes dans le graphe. Dans le pire des cas, c'est-à-dire le cas où le graphe est constitué d'une seule composante connexe, cette boucle sera exécutée une fois. Dans le cas où le graphe est constitué de k composantes connexes, cette boucle sera évidemment exécutée k fois.

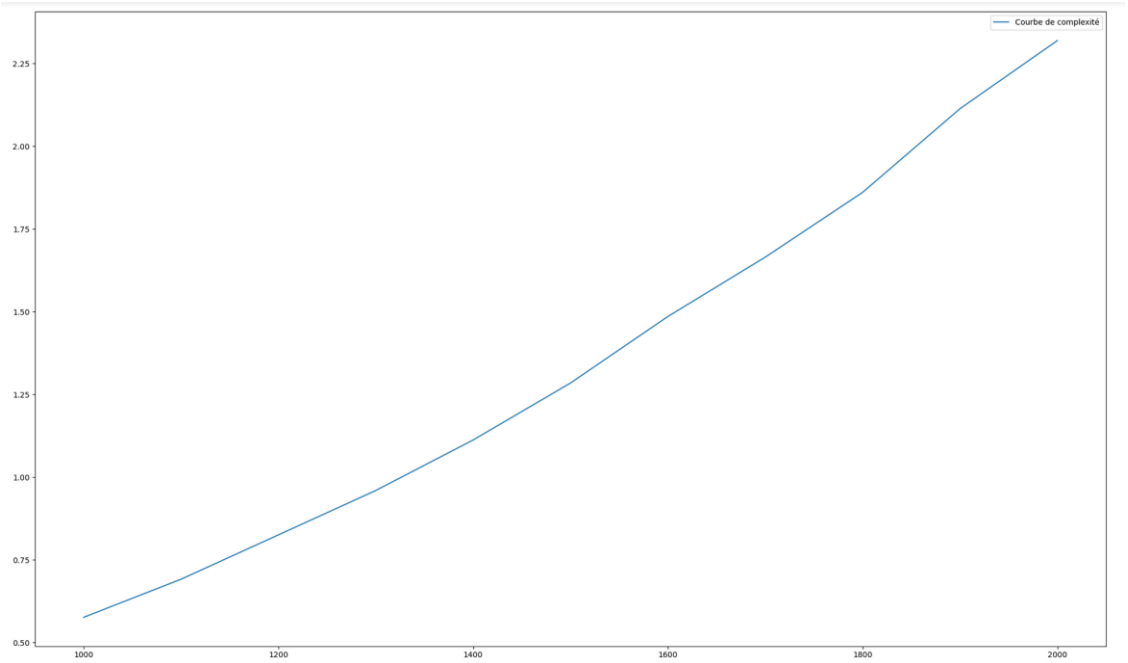
Le corps de cette boucle contient une autre boucle qui parcourt les voisins non encore visités d'un sommet visité, contenus dans la pile. Dans le pire des cas, chaque sommet est connecté à tous les autres sommets du graphe, ce qui signifie que tous les sommets doivent être visités et que cette boucle sera exécutée $O(n^2)$ fois.

La fonction '*diff*' est appelée dans la boucle intérieure pour calculer la liste des voisins non encore visités d'un sommet. Dans le pire des cas, où tous les sommets du graphe sont dans la même composante connexe, cette fonction peut être appelée O(n) fois pour chaque sommet visité.

Ceci donne finalement une complexité temporelle de :

$$n \cdot O\left(n^2\right) = O\left(n^3\right)$$

Voici un graphe montrant l'évolution temporelle de l'exécution de cet algorithme en fonction de la taille du graphe :



B) Union-Find solution :

Ce qui nous a poussé à changer de façon de penser est le format des données d'entrée ; on a jugé obligatoire de traduire le fichier de points en matrice d'adjacence (ou liste) pour pouvoir appliquer un parcours de graphe. Ceci nous a poussé à changer complétement de façon de penser, et de ne plus voir l'ensemble de points comme un graphe de plusieurs composantes connexes, mais plutôt un ensemble d'ensembles de points, où chaque ensemble correspond à une composante connexe.

L'idée est la suivante : à chaque étape, on ajoute un point à la liste contenant les ensembles de points de même partie connexe. Il faut donc faire les changements nécessaires à cette liste à chaque étape pour obtenir à la fin la liste de composantes connexes.

Cet algorithme est connu sous le nom d'Union-Find, et est efficace pour gérer des ensembles de données disjointes, qui correspondent dans ce cas aux composantes connexes. A la différence de notre implémentation, l'algorithme utilise une structure d'arbre, où chaque

racine correspond au représentant d'un ensemble, tandis qu'on utilise dans notre script la structure d'ensemble.

Notre programme utilise une fonction auxiliaire qui prend en argument la liste d'ensembles de points, et un point, et met à jour la liste en ajoutant le point passé en paramètre. Précisément, elle fusionne tous les ensembles qui contiennent un point lié au point en question (le point peut être lui-même) :

```python
def adding(liste, point):

    """ fonction intermédiaire qui part d'une liste d'ensembles, où chaque ensemble contient des points de même composante connexe,
        et ajoute le point passé en argument à cette liste en effectuant les changements nécessaires pour garder la caractéristique des
        ensembles en question """

    if not liste:
        liste.append({point})
    else:
        concerned = []
        cache = liste[:]
        for i, ens in enumerate(cache):
            for pt in ens:
                if testing(pt, point):
                    concerned.append(ens)
                    liste.remove(ens)
                    break
        a = set().union(*concerned)
        a.add(point)
        liste.append(a)
```

Le pire des cas correspond à celui où on doit parcourir chaque ensemble jusqu'à sa fin, donc en notant ' n ' le nombre de points dans la liste, la complexité de cette fonction est :

$$O(n)$$

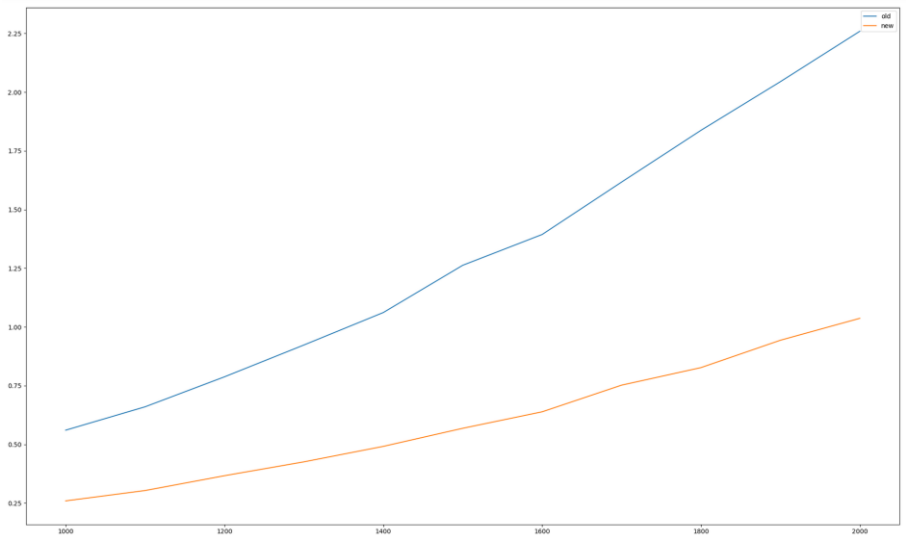
Le programme principal parcourt alors chaque point et met à jour la liste, qui est vide au départ :

```python
l = []
for point in points:
    adding(l, point)
m = [len(x) for x in l]
m.sort(reverse=True)
print(sum(m), f"{m}")
```

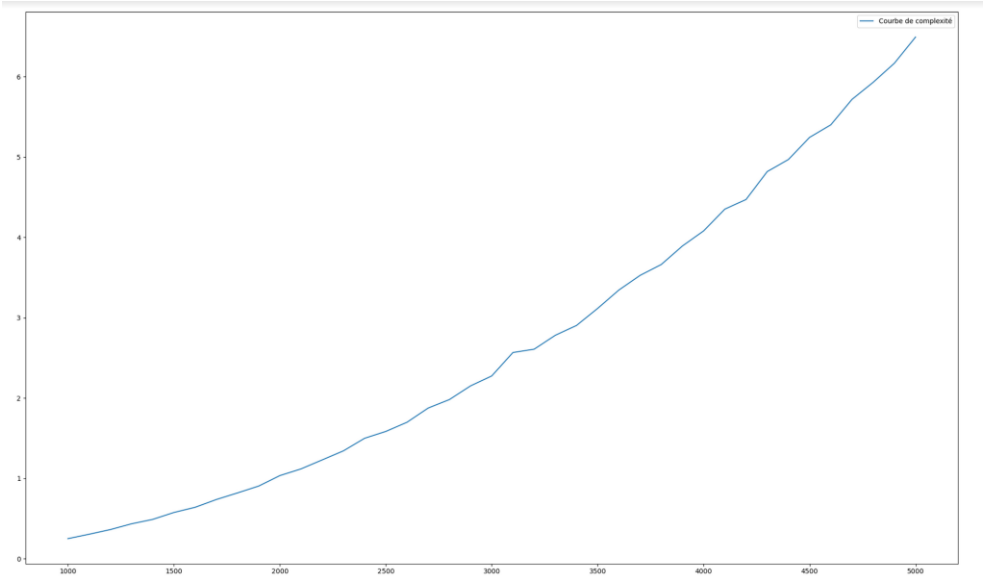
La complexité est, d'après ce qu'on a trouvé pour ' adding ' :

$$O\left(\sum_{i\in[1,n]} i\right) = O\left(n^2\right)$$

Cet algorithme est nettement meilleur que le précédent, et on le remarque bien avec les tests :



Graphe de comparaison



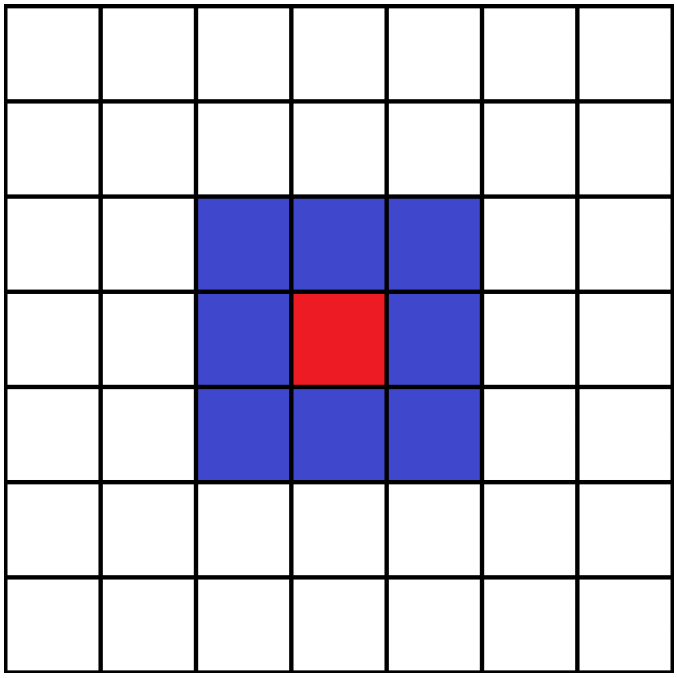
Graphe pour l'algorithme seul

C) Depth-First-Search (Diviser pour regner):

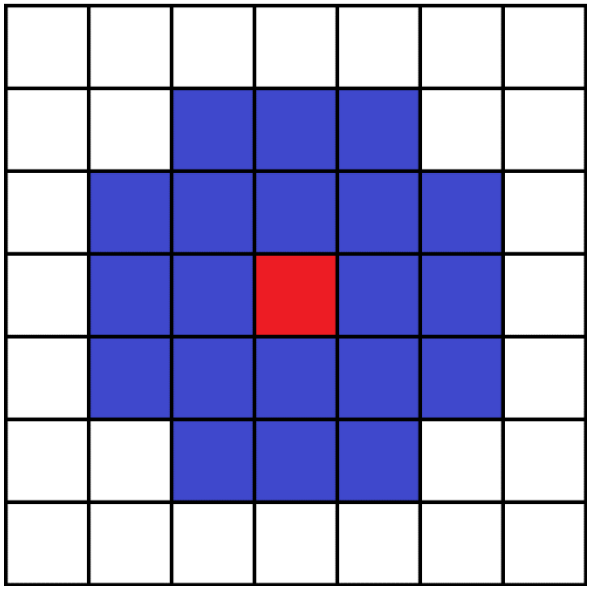
Plutôt que d'appliquer le DFS directement sur l'ensemble de points, qui nous poussera à chaque fois de chercher les voisins d'un point parmi tous les autres points, on partitionne la grille en cellules carrées de même taille, puis on applique le DFS judicieusement sur ces cellules. L'importance de cette approche est de diminuer le nombre de comparaisons lors de la recherche de voisins, il faut donc bien choisir la taille du côté de la cellule.

Notre première idée est de considérer comme taille du coté la distance seuil, ce qui va nous permettre de nous restreindre aux 8 cellules voisines lors de la recherche du voisin pour le DFS, puisque deux cellules espacées par une cellule ou plus ne peuvent contenir chacun un point lié à l'autre. Le problème avec ce choix de distance est qu'on ne peut considérer deux points d'une même cellule comme liés, ce qui va compliquer beaucoup l'implémentation du programme.

Une meilleure longueur est de prendre la distance seuil divisée par la racine de 2, ce qui impose la liaison de tous les points dans une cellule, et considérer deux cellules liés équivalente à deux points liés où chaque point est dans une cellule. Le seul problème qu'il faut remédier est celui du voisinage, puisque qu'il se peut que deux cellules soient liées alors qu'une cellule est entre les deux. Dans ce choix, il suffit de considérer encore 12 autres cellules pour le choix du voisin :



1 essai : distance seuil



Second essai : division par sqrt(2)

Cherchons à présent la complexité de cet algorithme :

```python
def testing(point1, point2):

    """ Cette fonction retourne 1 si la distance entre les deux points
    est inférieure à la distance donnée en argument dans composants() """

    test = Point.distance_to(point1, point2) <= distance

    return test


def get_cell(pt, d):

    """ gets the cell"""

    return (int(pt.coordinates[0]*(2**(1/2))/d), int(pt.coordinates[1]*(2**(1/2))/d))


def is_in(cell, dic):

    """ checks whether the cell is in the dic or not """
    try:
        dic[cell]
        return True
    except KeyError:
        return False
```

```python
def neighbor(cell, dic):
    """ gets list of neighbors """

    def adjacency1(cell1, cell2, dic):

        """ checks adjacency for two cells """

        pts1 = dic[cell1]
        pts2 = dic[cell2]

        for pt1 in pts1:
            for pt2 in pts2:
                if testing(pt1, pt2):
                    return True
        return False

    a = [(cell[0]+i, cell[1]+j) for i in range(-2, 3) for j in range(-2, 3) if i!= 2 or j !=2]

    return [x for x in a if (is_in(x, dic) and adjacency1(cell, x, dic))]
```

Ces quatre fonctions ont toutes une complexité constante, c'est-à-dire *O(1)*, puisqu'elles effectuent toutes des opérations de base sur les points et les cellules, sans boucle.

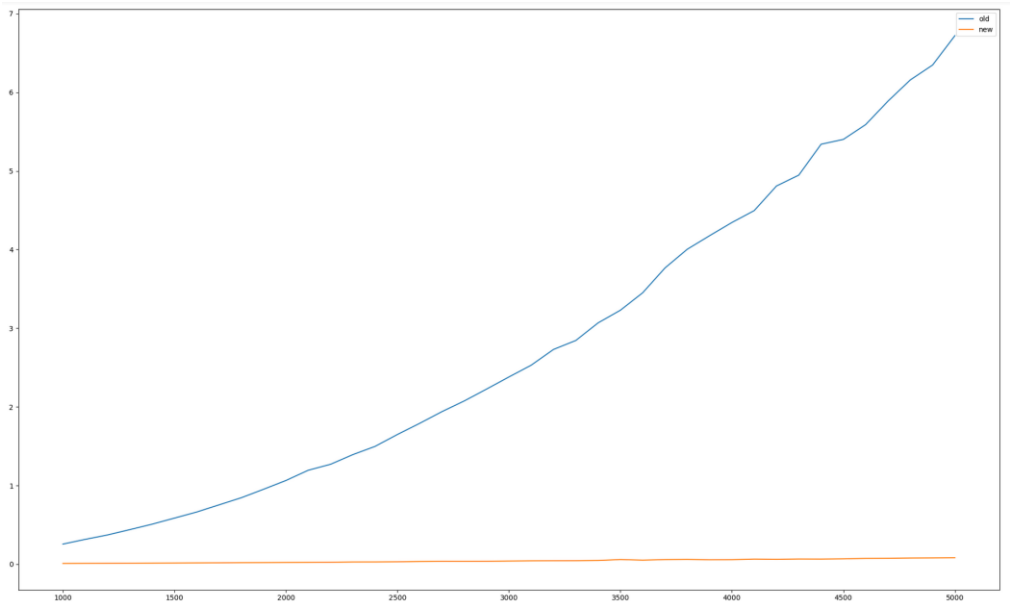
```python
cells = {}

for pt in points:
    a = get_cell(pt, distance)
    try:
        cells[a].append(pt)
    except KeyError:
        cells[a] = [pt]



dic_cell_to_index = {}
dic_index_to_cell = {}
for i, a in enumerate(cells.keys()):
    dic_cell_to_index[a] = i
    dic_index_to_cell[i] = a
```

La première partie de ce code a une complexité en *O(n)*, où *n* est le nombre de points. Puisque le nombre de cellules est inférieur ou égal à celui des points, la deuxième partie du code est en *O(n)* aussi, ce qui donne en total une complexité en *O(n)*.

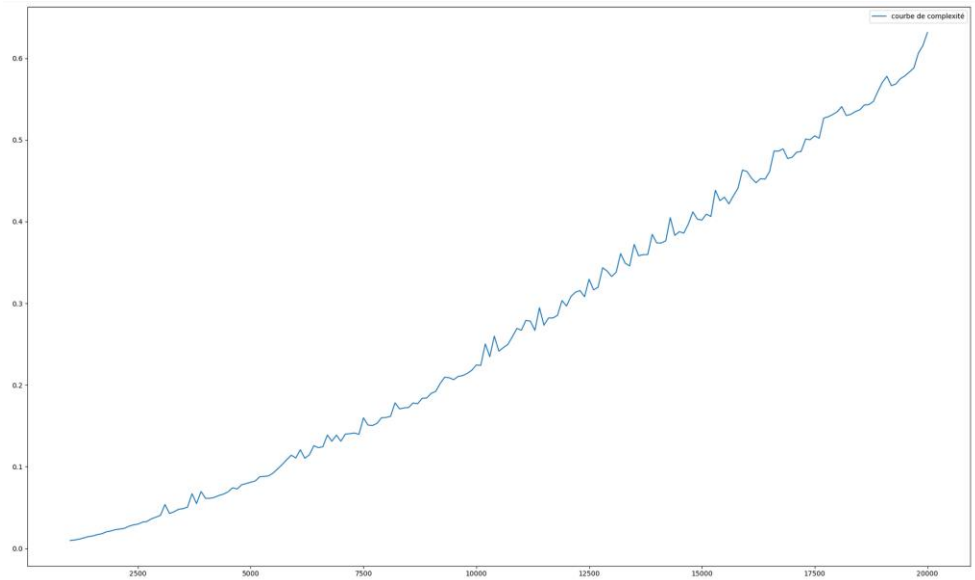
Le reste du programme, qu'on peut retrouver dans le fichier ' connectes_final.py ' correspond à l'application du DFS sur les cellules, si on note donc ' e ' le nombre d'arêtes entre les cellules, on obtient comme complexité finale :

$$O(n+e)+O(n)=O(n+e)$$

Cet algorithme est plus performant que le précédent, ce qui peut être bel est bien déduit des courbes suivantes :



Graphe de comparaison



Graphe de l'algorithme seul

5 – Conclusion :

Ce qu'il faut noter, c'est que plusieurs paramètres entrent en jeu lorsqu'on veut modifier la complexité temporelle de l'algorithme. Entre autres, il se peut que le BFS soit meilleur pour le dernier algorithme proposé, puisque ce dernier parcourt les voisins les plus proches dans un premier temps, et ce qui peut être bénéfique dans ce cas, car pour une cellule, on a peut déjà cerner les voisins les plus proches dans les 20 cellules (voir schéma). Aussi, la distance entre les points fait varier la connexion globale du graphe, c'est-à-dire pour une distance seuil qui tend vers 1, on est sûr que tous les points sont liés, et donc l'application de l'Union-Find est meilleur dans ce cas, qui sera ici de complexité $O(n)$, tandis que les parcours de graphe prendront plus de temps.