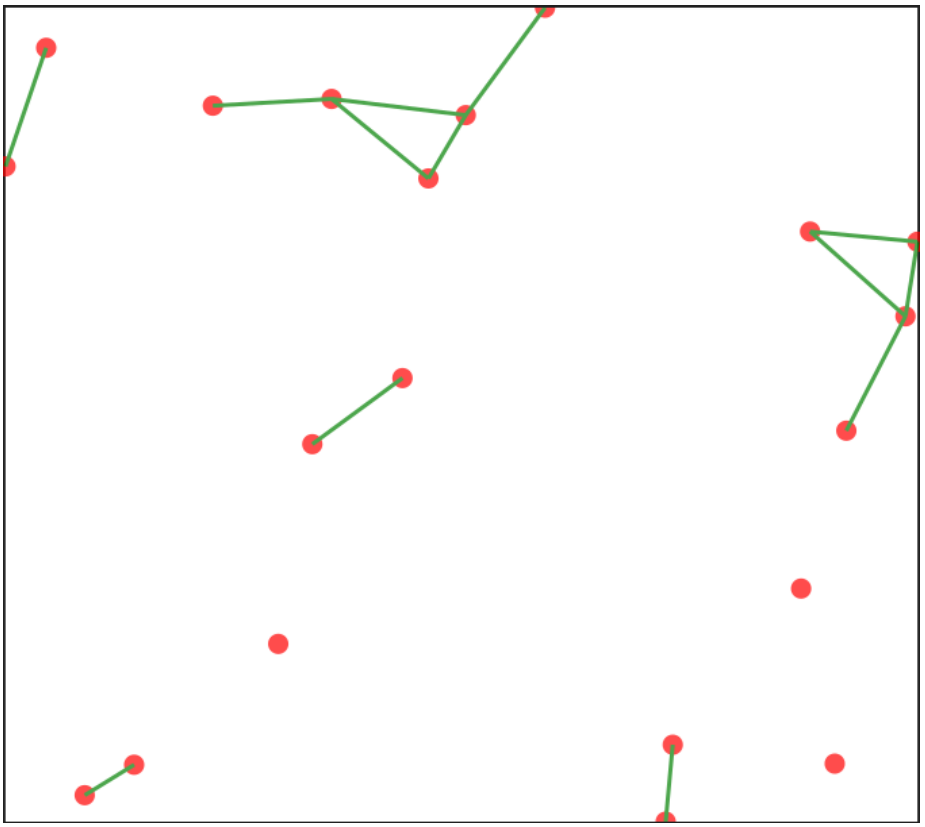
Aymeric BARON
Julian COUX

Rendu projet Algo :



“Implémentation d’un algorithme permettant l’identification de structures dans un nuage de points.”

I. Etude du problème

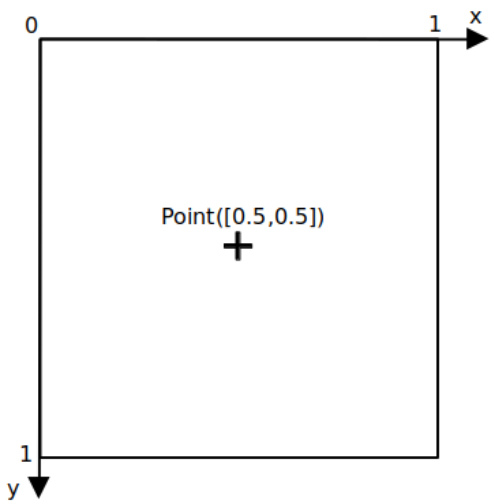
Le travail demandé pour ce projet de fin de première année en algorithmique, consistait à programmer en python un algorithme permettant de reconnaître et d'énumérer des graphes connexes dans un graphique à partir d'un jeu de données de points.
Le but de cet algorithme est assez simple, en entrée de notre algo, on reçoit un jeux de données contenant un nombre n de coordonnées (entre 0 et 1) et une distance minimale. L'objectif est d'analyser ce jeux de données et de relier tous les points de trouvant à une distance inférieure ou égale à celle donnée dans le fichier. Une fois tous les points reliés à leurs voisins, cela va créer des graphes connexes puisque les voisins se relient de proche en proche. Ainsi, l'objectif est de compter le nombre de graphes connexes créés et de compter de combien de points ils sont composés chacuns.

Pour commencer, nous avons dû étudier et comprendre les attendus. Nous avons tout d'abord réfléchi à comment représenter le graphique de points, une première approche du problème qui nous a aidé à visualiser le problème.

```
segments = []
for i in range(len(points)):
    for j in range(i+1, len(points)):
        distance_pt = points[i].distance_to(points[j])
        if distance_pt <=  distance and i != j:
            segments.append(Segment([points[i], points[j]]))

tycat(points, segments)
```

Avec le fonction tycat, on pouvait afficher les points et les segments les reliant pour tout d'abord réfléchir sur le papier comment un algorithme pourrait faire de même et compter les points des graphes.



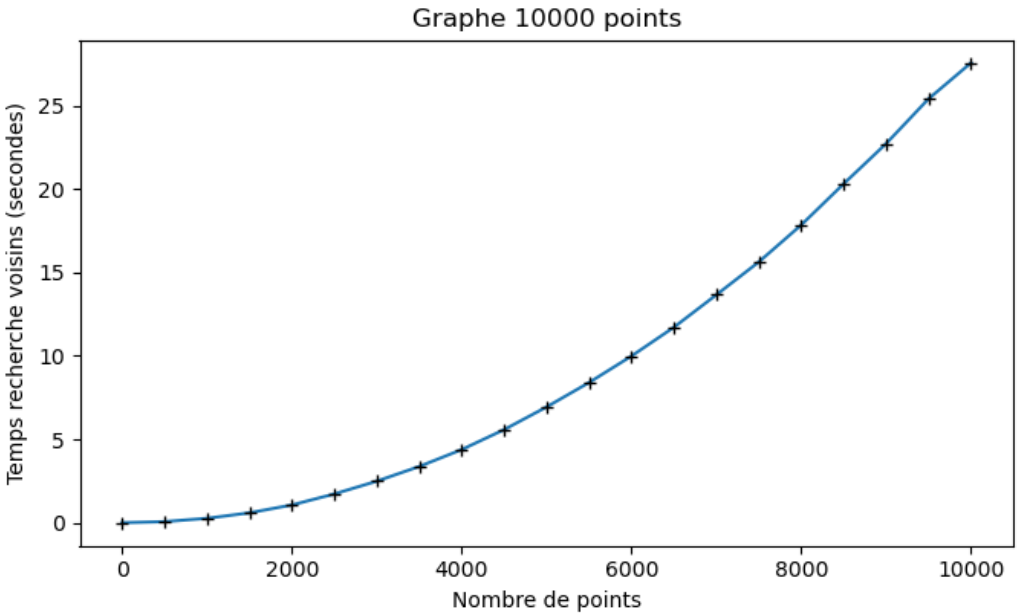
Voici la forme du graphique à étudier. Les points seront de type Point, donc comprenant une coordonnée en x et une en y.
Tout point en dehors de ce graphique ne sera pas considéré.

Maintenant que nous connaissons les attendus et la façon dont se présente le problème, nous avons pu commencer à réfléchir à différents algorithmes et à les coder.

II. Première approche

La première approche consiste à traiter le problème de manière récursive. L'objectif est d'abord de créer un dictionnaire dont la clé est un point et la valeur est une liste de tous ses voisins. L'objectif de ce dictionnaire est que pour un point donné, on connait tous ses proches voisins. Le coût de la création de ce dictionnaire est de $O(n^2)$ car on a deux *for* imbriquées qui parcourent chacun tous les points.
Maintenant, l'objectif est de, à partir d'un point aléatoire, parcourir tous les chemins possibles depuis celui-ci en passant par ses plus proches voisins. En parcourant tous ses voisins et voisins de voisins de proche en proche, on aura fini par trouver un graphe connexe.
Pour ce faire, j'ai d'abord créé un dictionnaire, initialement vide, destiné à prendre comme clé tous les points que nous avons déjà visités et comme valeur, True si ils sont visités.J'ai choisi un dictionnaire pour que la condition *if point[0] not in dictionnaire :* soit en temps constant (O(1), contre O(n) pour les listes). Ainsi, si un graphe a un cycle, on ne tournera pas en rond indéfiniment. Cela nous permet de mémoriser quels points nous avons déjà vu.
Ensuite, nous faisons un appel récursif sur chaque voisin de ce point. Puis sur chaque voisin de ce voisin… et ainsi de suite tant que il existe des voisins que nous n'avons pas encore visité. Une fois la récursion terminée, cela signifie que nous avons parcouru un graphe connexe, dont nous connaissons la taille puisque à chaque nouvel appel récursif on incrémente une variable *nb_points*.

Après de nombreux essais et modifications, nous avons réussi à rendre cet algorithme fonctionnel. Avant de vouloir améliorer notre temps avec les tests en ligne, nous devions tester nous même notre algorithme pour analyser ses performances.
Pour cela, nous avons créé un programme de test qui génère un nombre n de points répartis aléatoirement sur la graphique. Et pour chaque jeux de données, nous mesurons le temps d'exécution de l'algorithme. Ce qui nous permet, après de nombreuses mesures, d'établir un graphique du temps d'exécution en fonction du nombre de points. Ce qui nous donne une première idée de la complexité de l'algorithme.
Voici un exemple de ce que donne ce programme de test pour un nombre de points allant de 1 à 10000, avec un pas de 500.



Avec l'étude de la création du dictionnaire, on sait déjà qu'on est au moins en $O(n^2)$, ce qui se confirme par le premier graphique qui a une pente plutôt exponentielle.

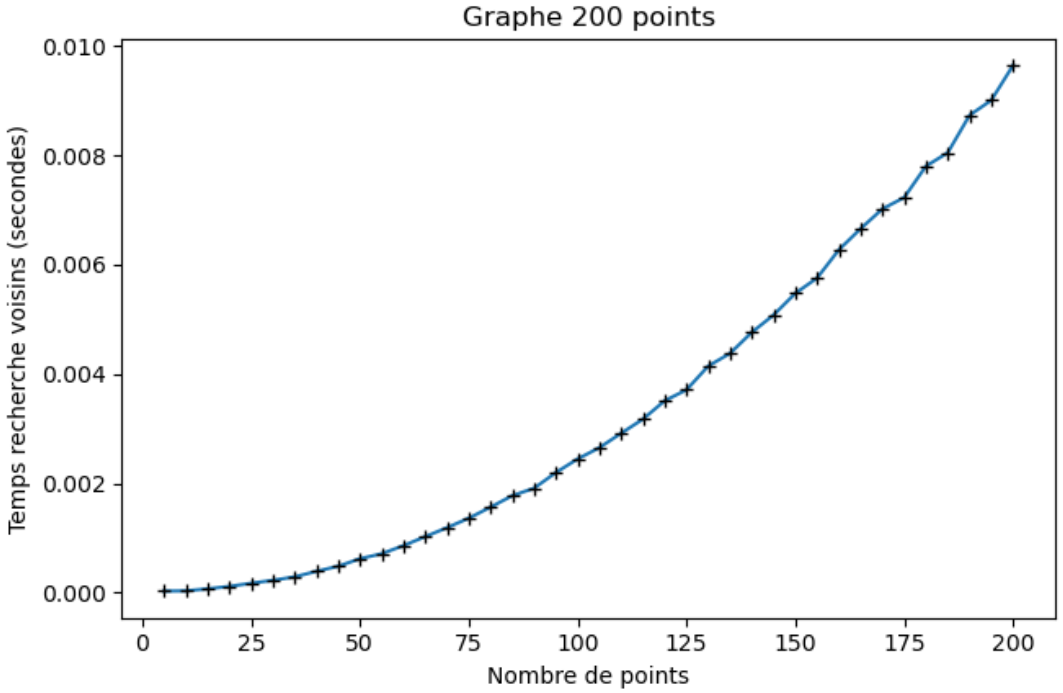
Calcul théorique de la complexité :

On a vu que la création du dictionnaire coûte en moyenne $O(n^2)$.

Ensuite, la boucle externe de l'algorithme, itère sur tous les points lancer l'appel récursif sur tous ses voisins, donc un $O(n)$. Puis, dans la fonction récursive, la boucle interne de la fonction 'recherche_voisins' explore les ensembles de proche en proche jusqu'à ce qu'il n'y ait plus de voisins à ajouter. Cela peut se produire au maximum lorsque tous les points sont connectés entre eux, ce qui peut prendre $O(n)$ itérations.

Dans l'ensemble, la complexité de cet algorithme est donc de $O(n^2+n*n)=O(2n^2)=O(n^2)$.

Nous avons également fait un test plus précis, sur un petit nombre de points pour voir comment se comporte l'algorithme lorsque, visuellement, le temps ne change pas trop (< 50ms).



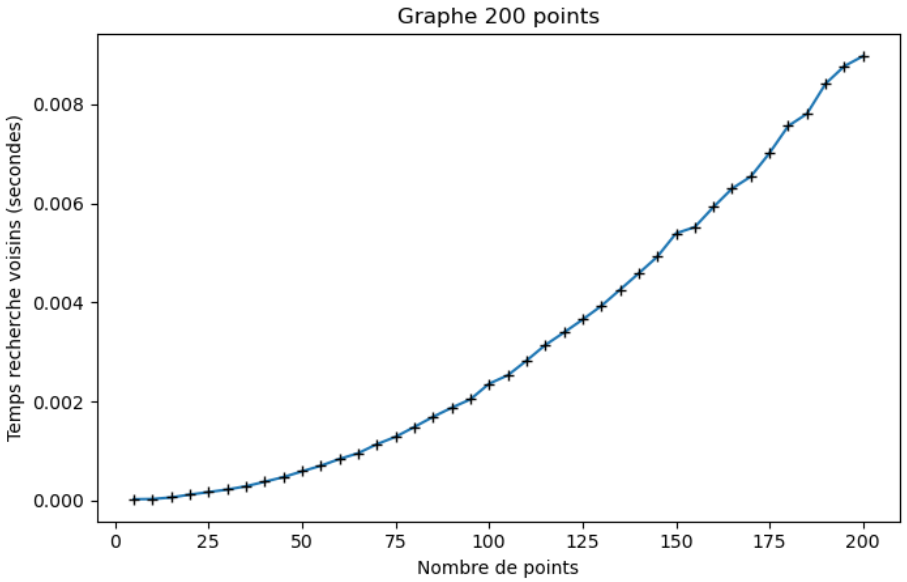
On peut voir que, sur un petit nombre de points ~200, cet algorithme est très rapide. Pour un nombre de points < 200, le résultat est quasiment immédiat. Cependant, on peut voir que le temps augmente très rapidement. Ce qui explique des temps aussi élevés pour des nombres de points > 1000. Et qui confirme encore une fois que la complexité est de l'ordre d'un $O(n^2)$.

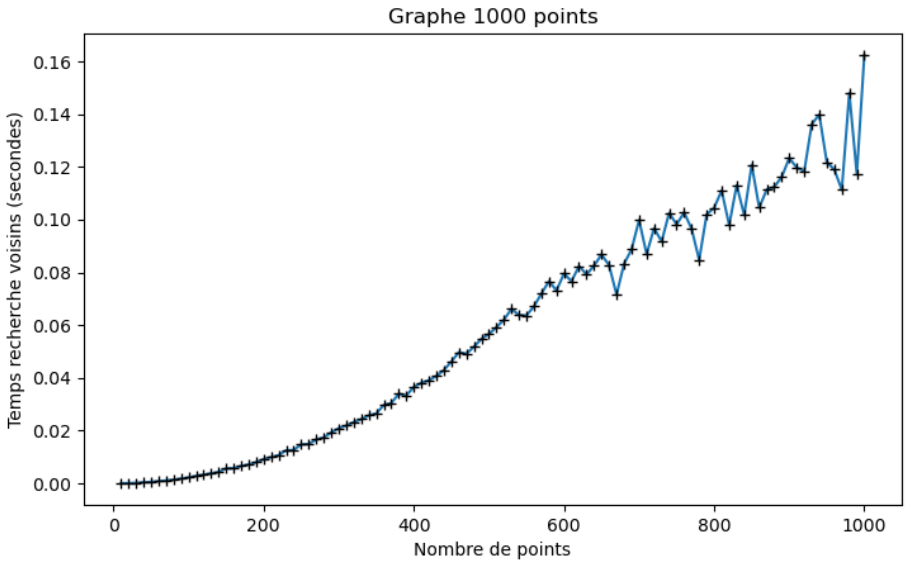
Un des problèmes rencontrés était qu'avec un algorithme récursif comme celui-ci, si tous les points forment un même graphe alors il va y avoir n appels récursifs. Or, en python, par défaut, le nombre d'appels récursifs maximal est de 1000. Cependant, lors de nos tests nous avons rapidement dépassé les 1000 points, ce qui pose problème puisque lorsqu'on avait un sous-graphe composé de 1000 points ou plus, le programme plantait puisqu'il y avait trop d'appels récursifs. Un moyen de résoudre ce problème sans modifier l'algorithme est simplement de modifier ce paramètre par défaut avec la commande : `setrecursionlimit(20000)`. Et de mettre une valeur arbitrairement grande pour être sûr de ne jamais la dépasser.

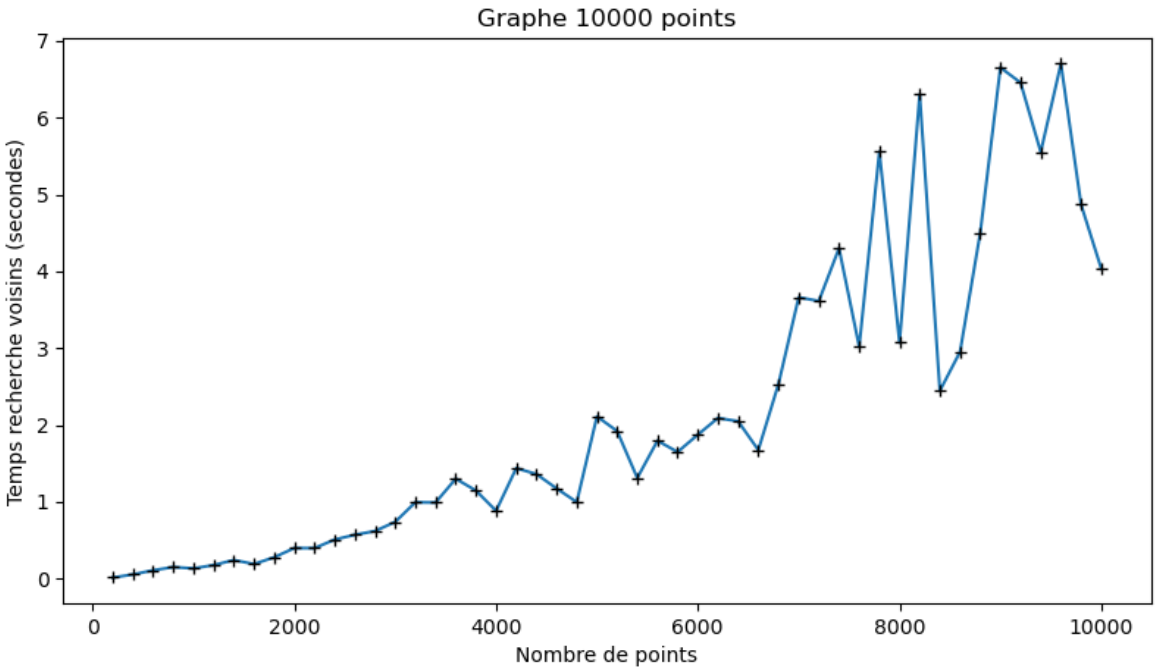
III. Seconde approche

Cette seconde approche consiste à partir toujours du premier point de la liste *points* et de parcourir tous les éléments restants dans cette liste pour voir s'ils sont proches du premier point ou non. Dès qu'un point proche est trouvé, on le supprime de *points* et on fait les mêmes tests sur celui-ci. De cette manière, tout point proche d'un point appartenant à la composante connexe du premier point sera bien compté une et une seule fois car il est tout de suite supprimé.

De plus, la suppression du point permet de ne pas parcourir à nouveau des points appartenant déjà à une composante et donc de gagner en complexité. Néanmoins, nous avons dû créer une liste provisoire gardant les points proches pour effectuer des tests plus tard sur ceux-ci car ils sont directement supprimés. Nous avons aussi ajouté une variable additive à l'indice dans la liste parcouru dans le but de ne pas passer plusieurs fois sur un même point et surtout ne pas dépasser la taille de la liste.







La première chose frappante dans les graphes que nous avons effectué sur cet algorithme est la non régularité du temps utilisé dont l'écart augmente avec l'augmentation du nombre de points. Cela est sûrement dû au lien entre la place des points dans l'espace et le temps de calcul. En effet, si tous les points sont proches du premier point, ils seront tous supprimés dès la première itération. Au contraire, s' ils appartiennent tous à des composantes connexes différentes, on aura besoin de n itérations où la liste réduira de taille seulement de 1 à chaque itérations.

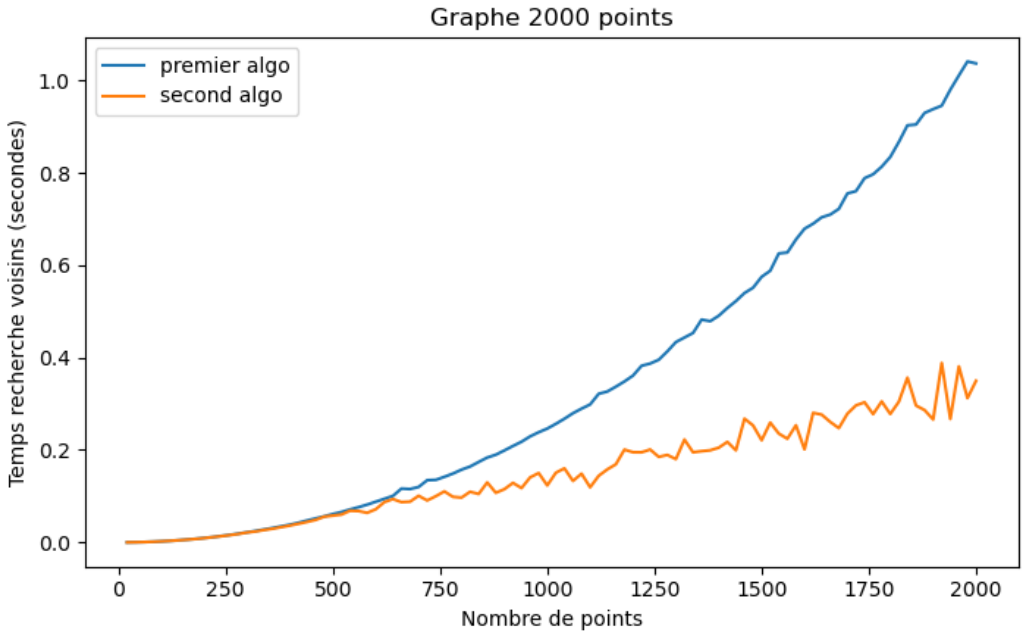
Calcul théorique de la complexité :

La complexité de cet algorithme dans le pire cas est en $O(n^2)$ par rapport au nombre de points dans la liste *points*. En effet, la fonction "nbr_points_proche" parcourt tous les points à chaque itération et supprime les points qui ont déjà été traités. Le nombre total de points traités est donc au plus égal à la somme des entiers de 1 à n, c'est-à-dire $O(n^2)$. La fonction "liste_tailles" appelle la fonction "nbr_points_proche" jusqu'à ce que tous les points aient été traités, ce qui conduit également à une complexité en $O(n^2)$. Ainsi, la complexité totale de l'algorithme est en $O(n^2)$.

La complexité de cet algorithme dans le meilleur cas est en $O(n)$ par rapport au nombre de points dans la liste *points*. En effet, comme vu au paragraphe précédent, si tous les points sont proches du premier point, alors la liste est parcourue une seule fois et les points sont supprimés au fur et à mesure ce qui correspond à une complexité de $O(n)$.

IV. Points à améliorer et autres idées

Comparaison des deux algorithmes :



Sur un nombre de points < 500, on peut voir que les algorithmes sont aussi rapides. Cependant, lorsque le nombre de points augmente, on peut voir que le second algorithme est largement plus efficace, de fait, pour 2000 points, il y a une différence de ~700 ms. Ce qui s'explique par sa complexité qui est largement meilleure que celle du premier algorithme, plus le nombre de points augmente, plus l'écart entre les deux sera grand.

Bien que théoriquement, la complexité de ces deux algorithmes soit la même, les temps d'exécution pour des nombres de points importants sont très différents. Ce qui peut s'expliquer par la différence sur la constante, en effet pour le premier algorithme, la complexité est exactement de $O(2n^2)$, contre $O(n^2)$ pour le second.

Suite à ces résultats théoriques, nous ne comprenons pas pourquoi la seconde approche ne passe pas plus de tests que la première dans les tests automatiques sur Chamilo. Néanmoins on remarque que pour un nombre élevé de points, le graphe de la seconde approche est assez irrégulier, dépendant sûrement du placement des points dans l'espace donné. Cela joue peut-être dans le dépassement du timeout pour les tests automatiques.

Nous avons également pensé à de nombreuses autres méthodes pour répondre au problème de base mais que nous n'avons pas réussi à implémenter ou à faire fonctionner. Notamment :

1. Utiliser la méthode de diviser-pour-régner:

L'objectif serait de diviser notre graphique en plusieurs sous-graphiques de taille suffisamment petite pour que tous les points qui appartiennent à ce graphique sont voisins et forment un graphe connexe. La démarche serait la suivante :

-Diviser le problème en plusieurs sous problèmes. On trie les points d'abord par ordre croissant selon les x, puis selon les y dans deux listes différentes. Ensuite, on crée des sous-graphiques de taille : $\frac{dmin}{\sqrt{2}}$. De cette manière, tous les points qui appartiennent à un sous-graphique sont voisins car *dmin* sera la taille de la diagonale du carré de côté $\frac{dmin}{\sqrt{2}}$. Comme nous avons trié les listes de points selon l'axe x et l'axe y, pour chaque sous-graphique, il est rapide de trouver quels points sont dedans et donc lesquels sont voisins.

-Ensuite, l'objectif est que pour chaque sous-graphe, on regardera par récursivité si ses points sont voisins avec les points des sous-graphes adjacents. De cette manière, si un point du sous-graphe donné est voisin avec un point d'un sous-graphe adjacent, alors tous les points du premier et tous les points du deuxième forment un graphe connexe. Par des appels récursifs sur les sous-graphes, grâce à cet algorithme, on serait capable de résoudre le problème initial.

V. Conclusion

Pour conclure, ce projet était très intéressant. Autant d'un point de vue technique dans le réalisation d'un algorithme que d'un point de vue étude avec les tests de performances de nos algorithmes. Nous avons tout particulièrement apprécié la recherche de performance dans ce projet. En effet, cela permet de ne pas seulement chercher à créer un algorithme fonctionnel mais d'aller plus loin dans la recherche d'une complexité qui soit la plus faible possible en explorant différentes solutions.