# A propos du calcul de la distribution des tailles des composantes connexes dans un plan

Haroun Al Mounayar

*Ensimag 1A Groupe 2*

Mes remerciements à David Eisenstat, ingénieur logiciel chez google à New York, pour son aide dans ce projet.

# Préface

Dans ce rapport de projet d'algorithmique, je présente l'évolution de mon algorithme et les différentes approches pour résoudre le problème posé. Ainsi, la table de matière que je présente dans la page ci-après n'est autre que les algorithmes que j'ai utilisée et que j'ai amélioré au fur et à mesure.

La solution optimale que j'ai trouvée est donc le dernier point de la table de matière : triangulation de Delaunay. C'est cet algorithme uniquement que je remets sur Teide. Les autres parties sont donnés pour montrer l'évolution de mes approches.

En effet, je ne me contente pas seulement de présenter ma solution optimale, je présente aussi les performances des différents algorithmes, comment j'ai élaboré des tests pour mesurer ces performances, etc. Je donne aussi quelques preuves et théorèmes qui m'ont permis d'améliorer ou de trouver des faiblesses dans certains algorithmes.

La mesure des performances est faite sur un ordinateur HP, intel CORE I7, 8GB RAM et ayant comme seul OS KALI Linux. Les affichages se font soit avec tycat de terminology, gnuplot avec qterminal ou simplement la librairie MatPlotLib de Python. Les codes sont écrits avec Python 3.9.1.

*Merci,*

*21 Fevrier 2021*

*Haroun Al Mounayar*

# Table des matières

# Chapitre 1

# Élaboration de test et un premier algorithme naïf

## 1.1 Élaboration de test

Pour évaluer les performances, il faut écrire un programme CREATE_TEST qui nous génère un fichier de points.

Or, on doit se poser la question : de quoi dépend le temps d'exécution de notre algorithme ? La réponse va constituer les entrées de notre programme CREATE_TEST.

Notre algorithme dépend bien évidemment du nombre de points à étudier. Si on fixe le nombre de composantes connexe, on peut affirmer que notre algorithme ne dépend pas de la distance limite qui précise si deux points sont connectés ou non.

Enfin, il existe une dépendance vis-à-vis de distribution des points dans le plan. En effet même pour un nombre de points et un nombre de composantes connexes fixées, il existe plusieurs distributions de points possibles. Cependant on ne peut pas quantifier cette distribution de points mais on peut preciser dans notre algorithme la forme voulue.

Pour implémenter Create_test, l'idée est de fixer la distance à 1.

Ensuite on repartie aléatoirement le nombre de points (nbre_de_point) sur le nombre de composantes connexes (nbre_de_cc). Par exemple nbre_de_point $= 10$ et nbre_de_cc $= 3$ une répartition possible est $[6, 3, 1]$.

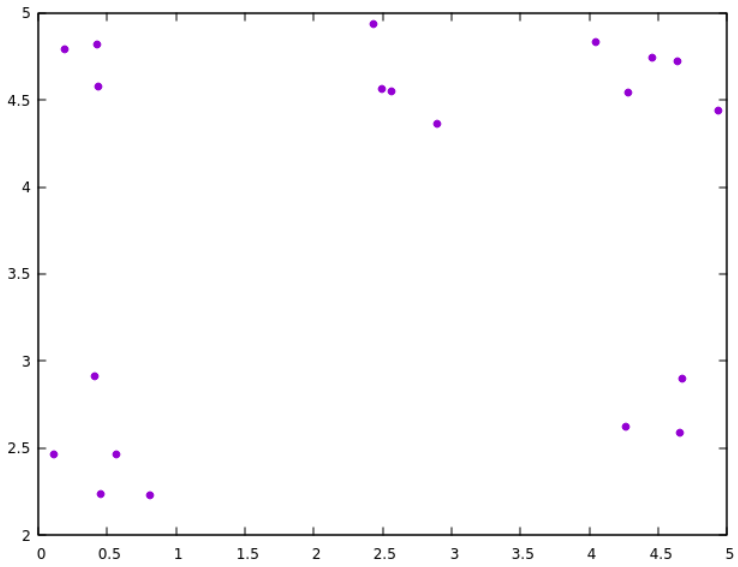
On génère ensuite les points aléatoirement dans des « fenêtres » de taille $1 \times 1$ et on translate ces fenêtres aléatoirement selon $x$ et $y$ d'un nombre pair pour éviter la collision entre fenêtre.

Avec python on a le programme suivant :

```
import random


def create_test(nbre_de_point, nbre_cc):
    """
    Create a file of points.
    """

    liste_taille_cc = [random.random() for _ in range(nbre_cc)]
    somme = sum(liste_taille_cc)

    for i in range(len(liste_taille_cc)):
        liste_taille_cc[i] = int(liste_taille_cc[i]*nbre_de_point/somme)

    liste_taille_cc[0] += nbre_de_point-sum(liste_taille_cc)

    fichier = open("test_aux.pts", "w")
    print("1", file=fichier)
```

```
    liste_fenetre = []
    for i in range(0, nbre_cc+1, 2):
        for j in range(0, nbre_cc+1, 2):
            liste_fenetre.append((i, j))

    for taille in liste_taille_cc:
        translation_x, translation_y = random.choice(liste_fenetre)
        liste_fenetre.remove((translation_x, translation_y))
        for _ in range(taille):
            # Ces deux lignes permettent de choisir la distribution des points
            # dans le plan. Ici on a pris une distribution aleatoire.
            # Celle ci va etre modifier dans la suite selon les cas.
            x = random.random() + translation_x
            y = random.random() + translation_y
            print(x, ", ", y, file=fichier)


create_test(20, 5)
```

Avec gnuplot le lancement du programme donne le fichier suivant :




## 1.2 Un premier algorithme naïf

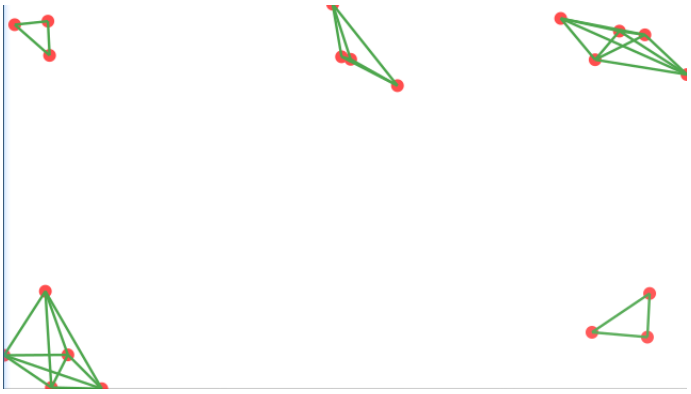
Dans cette section, on va créer un premier algorithme naïf pour se donner une idée et visualiser. Une façon de faire et de calculer toutes les distances entre les points. Si c'est plus petit que la distance limite, on relie les deux points ensembles.

La fonction en python se traduit donc comme se suit.

```
"""
Algorithme naive pour visualiser.
"""

from geo.segment import Segment

def segment_connexe_impose(distance, points):
    """
    Return a list of segment connected
```

```
    """
    result = []
    longueur = len(points)
    for i in range(longueur):
        for j in range(i+1, longueur):
            if points[i].distance_to(points[j]) < distance:
                seg = Segment([points[i], points[j]])
                result.append(seg)
    return result
```

Le lancement du programme dans terminology avec tycat permet de visualiser les connexions pour le même fichier crée par notre CREATE_TEST dans la section 1.1



Une première remarque qu'on peut se faire, c'est qu'on calcule beaucoup de distance : il y a des distances inutiles. En effet, pour calculer la taille des composantes connexe, on n'a pas besoin de les calculer en 'détails'. Cela est justifié par le faîte que s'il existe juste une liaison entre un point A et un sommet d'une composante connexe alors A appartient à cette composante connexe ; il est inutile de calculer la distance entre A et les sommets restants.

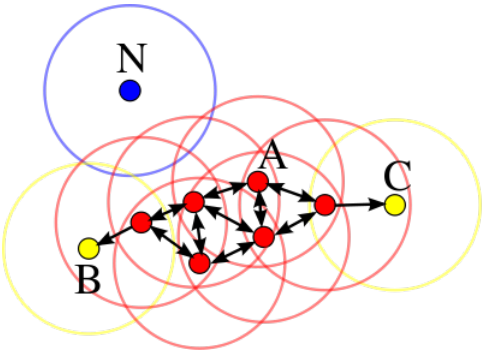
Pas besoin de faire une mesure de performance pour savoir que notre algorithme n'est pas efficace (surtout que notre algorithme ne fait que dessiner des segments et ne calcule pas la taille des composantes connexex)

# Chapitre 2

# Méthode DBSCAN naïve

## 2.1 Principe

DBSCAN veut dire Density-Based Spatial Clustering of Applications with Noise. Étant donné un nombre de points, une distance et un seuil $M$ l'algorithme regroupe les points dans des Clusters s'il existe un chemin de chaque point à l'autre et si le nombre de points dans le Cluster est supérieur à $M$.




Dans le schéma les points en rouge A, c'est le Cluster. Les points en jaune C, B sont en train d'être découverts par le cluster. Le point bleu N est du bruit, car le nombre dans le Cluster contenant N est inférieur a M. Par la suite et pour répondre au problème posé, on fixe $\boxed{M = 1}$.

## 2.2 Algorithme

Les etapes de l'algorithme avec $M = 1$ sont les suivantes :

— Entrée : POINTS ,une liste de points dans le plan, DISTANCE, un nombre positif

— Execution :

1. On prend un point A au hasard de POINTS
2. Si le point A n'a pas été visité :
   — Marquer A comme visité
   — Initialiser un cluster $\{A\}$
   — Calculer la liste VOISIN du voisinage de A en utilisant DISTANCE
   — Pour chaque point $V$ de VOISIN :
      — Trouver les cluster en partant de $V$ en marquant les points comme visités
      — Unir les cluster avec ce lui de $\{A\}$
3. Retourner la liste de Cluster obtenue

— Fin

## 2.3 Implementation dans Python

On a décidé d'implémenter l'algorithme de façon itérative et en l'adaptant pour qu'il renvoie directement la taille des clusters.

Les fonctions sont donc :

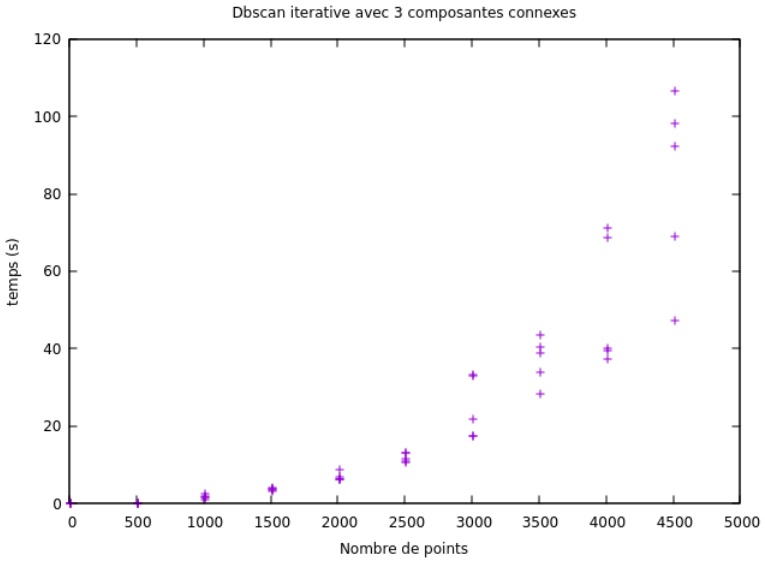
```
"""
Methode dbscan iterative
"""


def recherche_voisin(distance, un_point, points):
    """
    Recherche les voisins de un_point dans la liste points.
    """
    voisinage = []
    for des_points in points:
        if 0 < un_point.distance_to(des_points) <= distance:
            voisinage.append(des_points)
    return voisinage


def etendre_cluster(cluster, distance, un_point, points):
    """
    Etend le cluster.
    """
    voisin = recherche_voisin(distance, un_point, points)
    i = 0
    while i < len(voisin):
        if voisin[i] not in Visited:
            Visited.add(voisin[i])
            voisin_prime = recherche_voisin(distance, voisin[i], points)
            voisin = voisin + voisin_prime
        cluster.add(voisin[i])
        i += 1


def DBScan(distance, points):
    """
    Implemente la methode DBscan.
    """
    global Visited
    Visited = set()
    global liste_cluster
    liste_cluster = []
    for i in range(len(points)):
        if points[i] not in Visited:
            Visited.add(points[i])
            cluster = {points[i]}
            etendre_cluster(cluster, distance, points[i], points)
            liste_cluster.append(cluster)
    return liste_cluster
```

## 2.4 Performances

L'algorithme présenté dans 2.3 possède plusieurs problèmes. Cependant on remarque que ce qui influe le plus sur la complexité et le temps d'exécution c'est la fonction RECHERCHE_VOISIN.

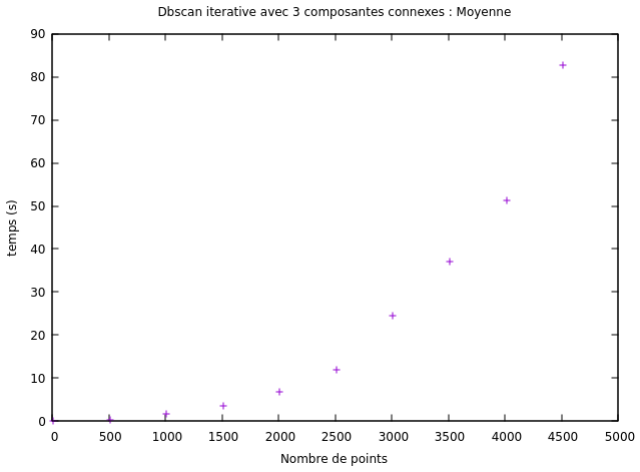
### 2.4.1 Nombre de composantes connexes fixé

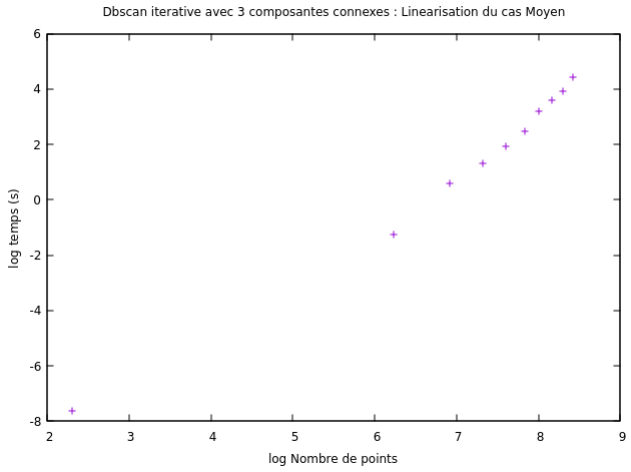
En fixant le nombre de composantes connexes à 3, et en ne variant que le nombre de points dans le CREATE_TEST (Voir 1.1) on obtient les courbes de performances suivantes.




On remarque que notre meilleur cas commence à avoir l'allure d'une droite pour des valeurs plus grande que 3 000 points. Dans le pire cas, la courbe a l'air en $O(n^\alpha)$ avec $\alpha > 1$ (on se doute évidemment que $\alpha = 2$).

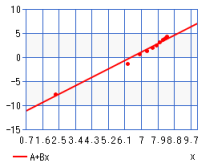
Pour fixer les idées, nous allons considérer la complexité en moyenne. Pour cela, on reprend les mesures précédentes et on remplace chaque groupe de points alignés verticalement par un point correspondant a la moyenne arithmétique.



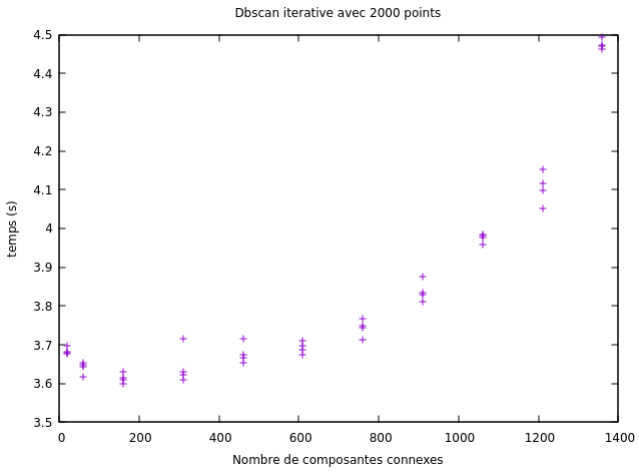

La régression linéaire donne un coefficient de corrélation de 0,99. On a donc bien une droite. On en déduit de $B = 1.94$ que le cas moyen est en $O(n^2)$.

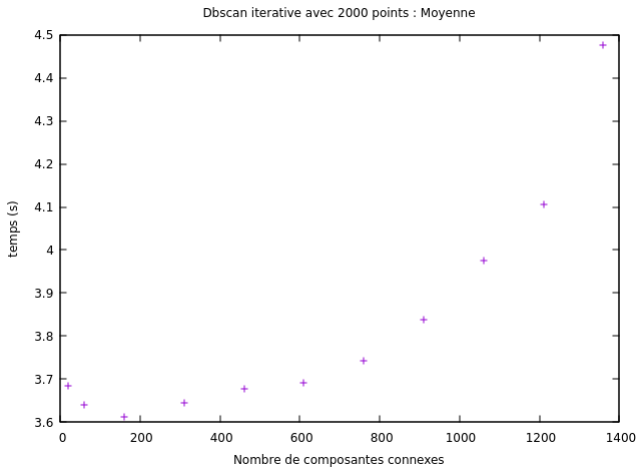
| function | value |
|---|---|
| mean of x | 7.109216002 |
| mean of y | 1.261950427 |
| correlation coefficient r | 0.9918425804 |
| A | -12.54464291 |
| B | 1.942069749 |




### 2.4.2 Nombre de points fixé

À présent, on fixe le nombre de points à 2000. Ce qui varie, c'est le nombre de composantes connexes. Les performances sont affichées ci-dessous avec le cas moyen.




On remarque l'existence d'une valeur critique minimale tel qu'après cette valeur le temps est un $O(k^2)$ avec $k$ le nombre de composantes connexes. En effet, pour un grand nombre de composantes connexes ou un nombre de composantes petit l'algorithme devient couteux.

# Chapitre 3

# Amelioration de l'algorithme

## 3.1 Arbres $kd$

Comme on a pu le remarquer précédemment le problème vient principalement de la fonction qui recherche le voisinage d'un point donné. Pour résoudre cela, il faut trouver une manière plus rapide et plus efficace de calculer le voisinage d'un point.

L'idée est de combiner l'algorithme précèdent avec une autre fonction RECHERCHE_VOISIN qui utilise les Arbres Kd pour calculer plus rapidement le voisinage.

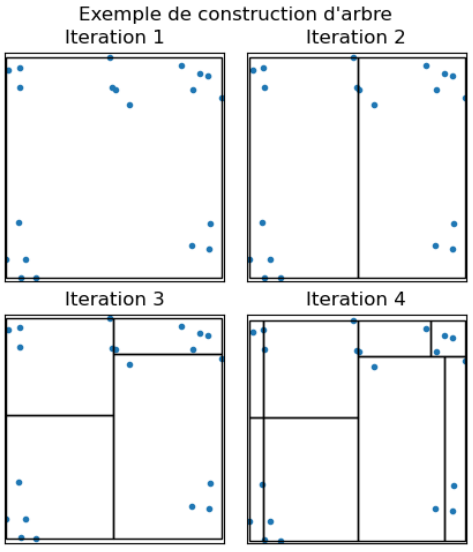
### 3.1.1 Principe

L'arbre $kd$ est comme son nom l'indique une structure de donné qui permet de partitionner l'espace avec des points de $k$ dimension. Dans la suite, on confondra les arbres $kd$ et les arbres $2d$.

Chaque point divise l'espace en deux sous-espaces. Pour simplifier l'explication plaçons nous dans un plan. Un point $(x_p, y_p)$ divise le plan selon Oy. Les points $(x, y)$ d'ordonné tel que $x_p > x$ sont stockés dans la branche gauche de l'arbre kd et les points $x_p \leq x$ dans la branche droite. Le prochain point $(x'_p, y'_p)$ divisera le plan selon Ox et ainsi de suite.

Étant donné un nuage de points $(x_i, y_i)$ il y a plusieurs moyens de choisir les points qui divisent l'espace _splitting point_ dans notre arbre : on choisit la méthode tel que chaque splitting point possède la médiane comme coordonnée selon x ou y. Les points donnés sont donc les feuilles de l'arbre inséré selon les règles mentionné ci-dessus.

Appliquons cette construction à notre exemple de la section 1.1 :

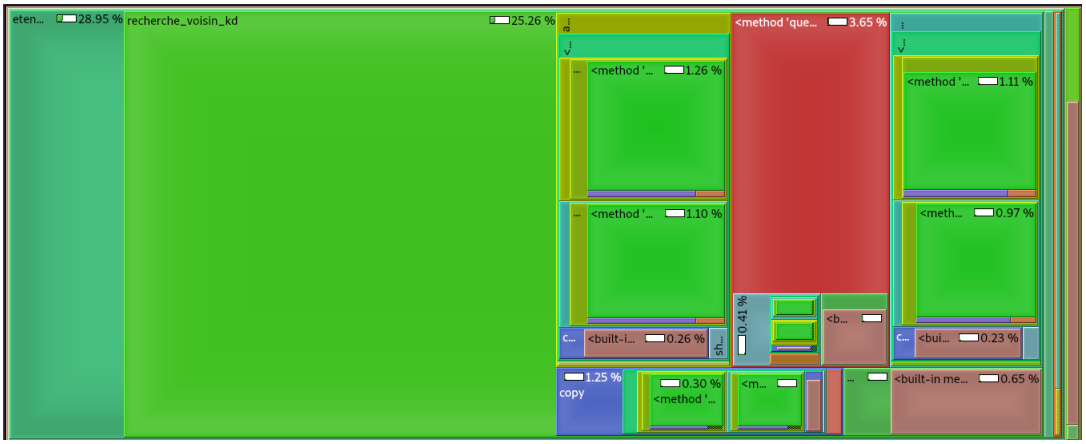


Ainsi, cette structure permet d'éviter de calculer la distance entre les points éloignés.

### 3.1.2 Performances

Même si j'avais une structure d'arbres *kd* qui fonctionne j'ai décidé d'arrêter cette approche. En effet, même si cette idée va fonctionner, le codage en python s'avère particulièrement pénible. Cependant à titre de comparaison avec l'algorithme ou le recherche de voisinage se fait de manière brut je fournie une comparaison entre les deux avec l'option cPROFILE de python3.

Le module SCIPY.SPATIAL nous fourni une fonction calcul de voisinage faite avec Python.




(a) Algorithme avec Arbre *kd*


(b) Algorithme sans Arbre *kd*

FIGURE 3.1.1 – Profile de Dbscan avec Arbre kd (a) et par recherche brute (b)

Ainsi l'utilisation des arbres *kd* permet de réduire significativement le temps de l'algorithme. En effet, sur les figures, on voit que ca passe de 97% a 25%. Sans oublier que le calcul de la distance avec DISTANCE_TO (fonction fournie dans le module GEO) n'est pas le plus rapide (80.15%).

## 3.2 Divide to Conquer : Union Find ?

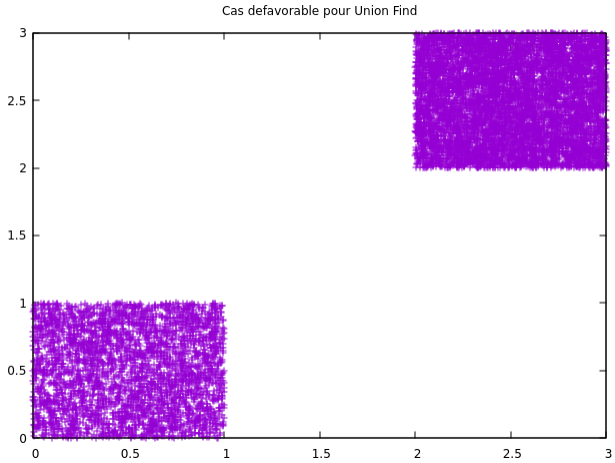
L'utilisation améliore potentiellement la complexité dans certains cas et rend le programme plus rapide. Cependant la complexité moyenne reste la même : en effet même si on sait calculer le voisinage d'un point plus

efficacement, pour répondre au problème il faut trouver les voisins de chaque point et revenir au principe d'un DBSCAN en marquant si le point a été visité ou non. Ainsi, la complexité est en $O(N^2)$ toujours ($N$ nombre de points).

Une méthode d'améliorer est de faire un diviser pour régner. Dans notre cas, diviser est simple : soit avec des Arbres Kd de nouveau ou avec une division en 'petit carré' de l'espace. Pour retrouver le résultat final une structure nommée Union Find sera utile.

La compléxité de notre algorithme sera dans quelque cas en $O(Nlog(N))$ ($N$ nombre de points).

Cependant si on a un nombre $N$ tres grands et un nombre de composantw connexe $CC$ petit (mais non égal a 1) cette approche ne sera plus efficace :




En effet dans la figure de dessus, on a deux ensembles de points en mauve. Si $d = 0.5$ on aura deux composantes connexes. Dans ce cas (cas non-rare et fréquent) notre algorithme est en $O(N^2)$. Une manière de s'en convaincre est de penser comme se suit :

— un diviser pour régner normale forme les deux composantes connexe en mauve

— L'algorithme doit ensuite calculer la distance entre chaque couple de points de chaque composante connexes pour savoir s'il les unit dans une seule composante. La réponse étant non-car $d = 0.5$ on va calculer toutes les distances entre couples de points. Cela explique la complexité.

Pour cette raison, je n'ai pas développé un algorithme avec Union Find même s'il est plausible de résoudre ce cas particulier en regardant le barycentre par exemple ...

# Chapitre 4

# Triangulation de Delaunay : Version finale

## 4.1 Motivation

Le problème avec les versions précédentes, c'est qu'on commence à chercher depuis tout l'ensemble de points. Un prétraitement des points pourra être utile pour qu'on ne regarde qu'un sous-ensemble de relations entre les points donnés. L'idée est de faire le lien avec le cours de recherche opérationnelle.

Pour cela, notons notre ensemble de points $V$. On imagine qu'on les lie tous (même si en pratique, on ne va pas faire ca). On aura un ensemble d'arêtes $E$. A chaque arête de $E$ on lui associe son coût comme étant sa longueur euclidienne. On défini alors le graphe $G = (V, E)$.

**Définition 1.** Un arbre couvrant minimum euclidien ou **EMST** est un arbre couvrant minimum pour un graphe dont le coût des arêtes est la distance euclidienne

L'idée est de chercher les composantes connexes dans l'**EMST** du graphe. En effet on a la proprieté suivante sur les arbres couvrants :

**Proposition 2.** *Les composantes connexes sont incluses dans l'**EMST** du graphe $G$*

*Démonstration. (pseudo preuve)*

Sois une arête $e_i$ de l'EMST. On note $C$ le cycle formé par cette arête et d'autres arêtes non-présentes dans l'EMST. En se rappelant que l'EMST est un arbre couvrant minimum, une propriété qu'il possède est que chaque arête de $C$ possède une longueur supérieure à celle de $e_i$.

On en déduit que : s'il existe un chemin dans $G$ tel que toute arête possède une longueur inférieur a $d_{max}$ alors il y a un chemin dans l'EMST avec cette même propriété

Fin de la Preuve □

La question reste comment trouver un **EMST** du graphe. Pour cela, on utilise un théorème fondamental dans ce projet.

**Théorème 3.** *L'EMST est inclus dans la **triangulation de Delaunay** du graphe $G$*

## 4.2 Principe

La **triangulation de Delaunay** est une triangulation (relie les points pour former des triangles) telle que chaque point P du graphe soit à l'extérieur de tous les cercles circonscrits aux triangles de la triangulation.

La construction de la triangulation de Delaunay utilise un calcul de déterminant pour savoir si le point est à l'extérieur au cercle circonscrit ou non. L'implémentation est difficile et nécessite un algorithme diviser pour régner et de nombreuses astuces (flipping algorithm : on génère une triangulation et on flip les arêtes pour avoir une de Delaunay). Cependant le module SCIPY.SPATIAL propose de faire une triangulation de Delaunay.

Le principe de l'algorithme est donc le suivant :

1. On génère la triangulation de Delaunay $D(G)$ du graphe $G$

2. On crée un graphe $G'$ représenté en python par un dictionnaire
3. On parcourt les triangles de $D(G)$ et on calcul la longueur $\ell$ des côtés des triangles :
   (a) Si $\ell \leq d_{max}$ on ajoute les deux extrémités dans $G'$
   (b) Sinon on continue
4. On fait un $dfs$ (depth first search ou parcours en profondeur) sur $G'$ pour avoir les composantes connexes

L'algorithme est illustré par le schéma suivant :

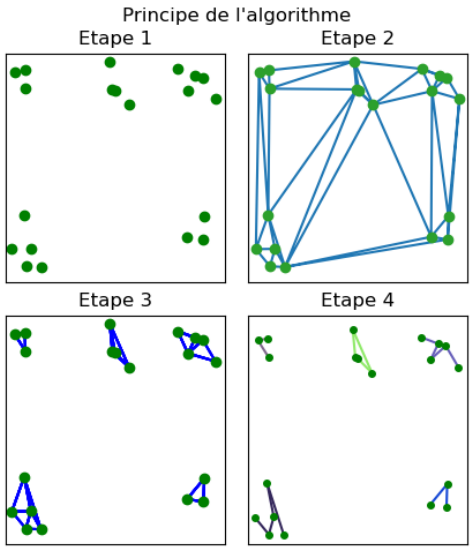



FIGURE 4.2.1 – Principe de la solution avec la triangulation de Delaunay

Dans l'Étape 1, on vous montre les points, dans l'Étape 2 on fait la triangulation de Delaunay sur ces points, dans l'Étape 3, on enlève les segments de longueur plus grande que $d_{max}$ et dans l'Étape 4, on fait le parcourt en profondeur sur le graphe.

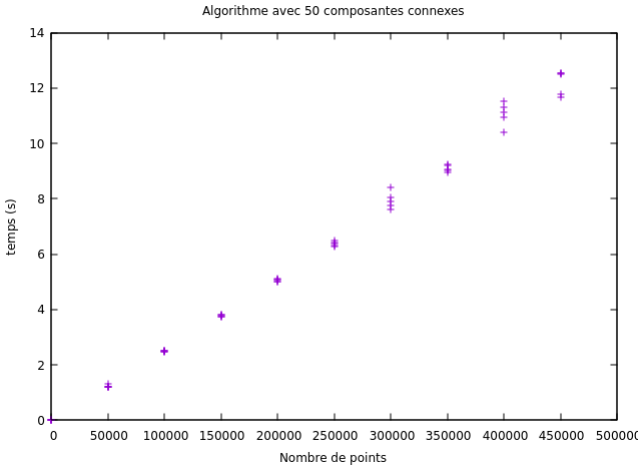
## 4.3 Performances

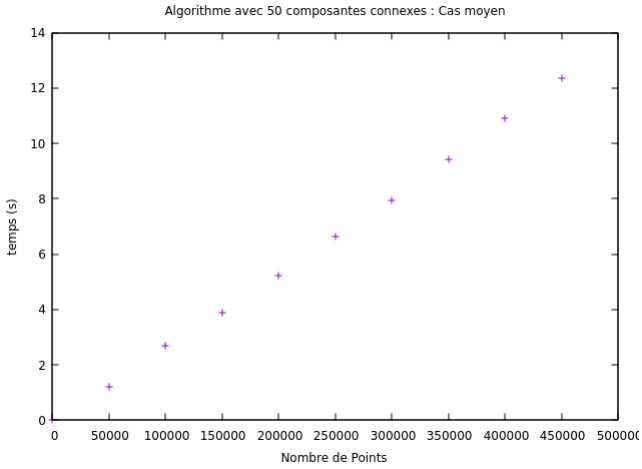
On commence par donner une propriété utile :

**Proposition 4.** *La triangulation de Delaunay de $N$ points dans le plan contient $O(N)$ triangles.*

### 4.3.1 Nombre de composantes connexes fixé : ditribution RANDOM des points

Dans cette partie, on fixe le nombre de composantes connexes. On fait varier le nombre de points. La distribution des points dans le plan est considérée à ce stade aléatoire. Les coordonnées $x$ et $y$ sont générées avec RANDOM.RANDOM() de python3.




Même si les points semblent être parfaitement alignés la complexité est en $O(Nlog(N))$. En effet le temps d'exécution $T$ de notre algorithme s'écrit comme :

$$T = \underbrace{a \times Nlog(N)}_{\text{chercher la triangulation de Delaunay}} + \underbrace{b \times O(N)}_{\text{Parcourir les triangles et enlever les points éloignés}} + \underbrace{c \times O(\overbrace{N}^{O(N)\text{ arêtes}} + \overbrace{N}^{N\text{ sommets}})}_{\text{Parcours en profondeur}}$$

**RQ :** On a $O(N)$ arêtes d'après la proposition 4

Cependant et vu la distribution aléatoire des points, chercher la triangulation de Delaunay se fait rapidement. Ainsi, on a $b >> a$ ou $c >> a$. Il est difficile de voir alors le $Nlog(N)$.

### 4.3.2 Nombre de composantes connexes fixé : Pire cas , Delaunay et Parabole

On cherche maintenant a faire en sorte que le $a$ de la section précédente domine. Si tel est le cas on pourra voir un $NlogN$.

Pour cela utilisons le théorème suivant :

**Théorème 5.** *Le pire cas du calcul de la triangulation de Delaunay, sans diviser pour régner, est en $O(N^2)$. $N$ étant le nombre de point.*

*Ce pire cas est réalisé quand les points sont sur le même coté d'une parabole. Donc on a affaire à des points de la forme :*

*$(x_1, x_1^2), (x_2, x_2^2), ..., (x_N, x_N^2)$ avec les $x_i$ de même signes.*

*Démonstration. (pseudo preuve)*

On trie les points par abscisse croissante. $x_1 < x_2 ... < x_N$. Cela est possible, car on a un nombre fini de points.

On suppose que la triangulation de Delaunay a été faite pour $(x_{i+1}, x_{i+1}^2), (x_{i+2}, x_{i+2}^2), ..., (x_N, x_N^2)$. On la note $D_{i+1}$. On veut ajouter $(x_i, x_i^2)$ à cette triangulation $D_{i+1}$.

Pour cela considérons $(\alpha, \beta, \gamma) \in \{x_1, ..., x_{i-1}\}^3$ tel que $\alpha < \beta < \gamma$. Soit $C$le cercle circonscrit au triangle de sommet $(\alpha, \beta, \gamma)$.

Le point $(x_i, x_i^2)$ est dans $C$ ssi $V = det(\begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^2+\alpha^4 \\ 1 & \beta & \beta^2 & \beta^2+\beta^4 \\ 1 & \gamma & \gamma^2 & \gamma^2+\gamma^4 \\ 1 & x_i & x_i^2 & x_i^2+x_i^4 \end{bmatrix}) = 0.$

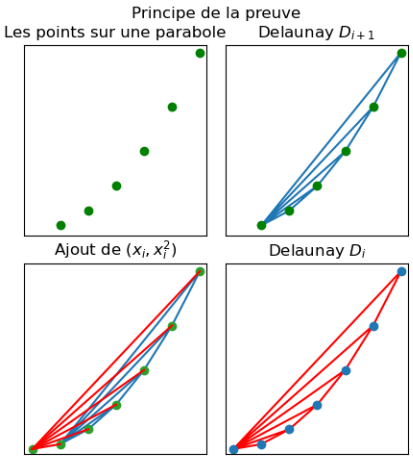
Le calcul avec la calculatrice donne $V = (\alpha+\beta+\gamma+t) \times (\alpha-\beta) \times (\alpha-x_i) \times (\alpha-\gamma) \times (\beta-\gamma) \times (\beta-x_i) \times (\gamma-t)$

Vu que les points sont distincts on a $V \neq 0$. Les $(\alpha, \beta, \gamma)$ étant quelconque, on en déduit que $(x_i, x_i^2)$ est à l'extérieur de tout cercle circonscrit aux triangles de la triangulation. Ajouter le point $(x_i, x_i^2)$ ajoute donc $N - i$ cotés à la triangulation $D_{i+1}$.

Ainsi passer de la triangulation $D_N$ à $D_i$ nécessite l'ajout de $\sum_{i=1}^{N}(N-i) = O(N^2)$. C'est d'ici dont vient notre complexité quadratique.

Fin de la preuve □

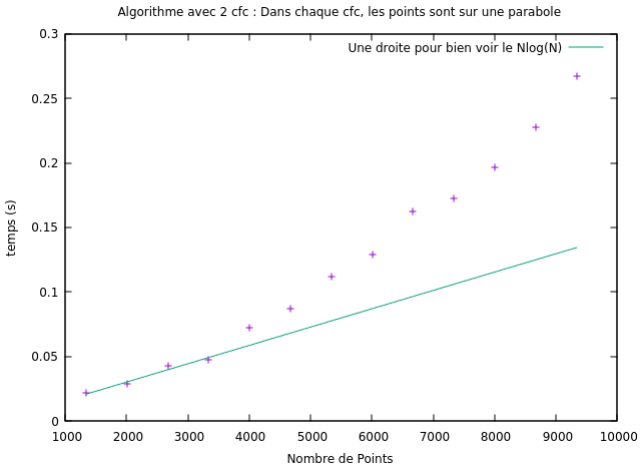
On présente une illustration graphique du théorème :

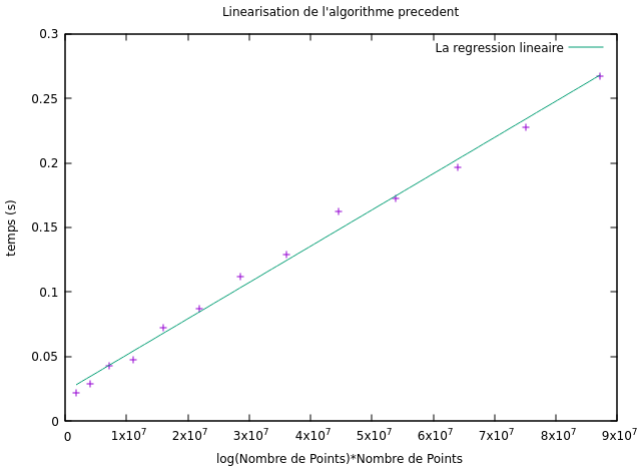



Dans notre test, il suffit alors lors de la génération des points aléatoires de créer aléatoirement une abscisse $x$ et en déduire $y = x^2$ par exemple.

Le théorème 4 assure que la complexité de la création de la triangulation sera en $O(N^2)$ s'il n'y a pas de diviser pour régner. Cependant l'algorithme de SCIPY.SPATIAL utilise un diviser pour régner avec de nombreuses astuces. La complexité sera alors en $O(NlogN)$. On fixe le nombre de composantes fortement connexes ($cfc$) à 2.

10000, nbre_de_point) start =




time.

### 4.3.3 Nombre de composantes connexes variable

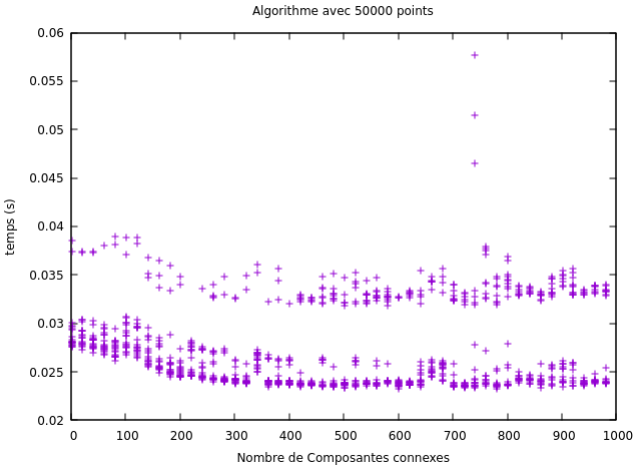
Dans cette partie, on fait varier le nombre de composantes connexes. Dans un premier temps, fixons le nombre de points a 50 000.

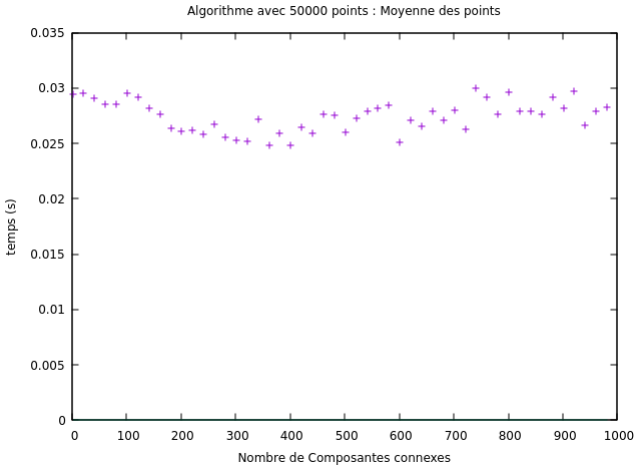
La force de cet algorithme avec la triangulation de Delaunay, c'est qu'il est indépendant du nombre de composantes connexes.

En effet, la fonction pour calculer la triangulation de Delaunay ne dépend que du nombre de points et leur distribution dans le plan.

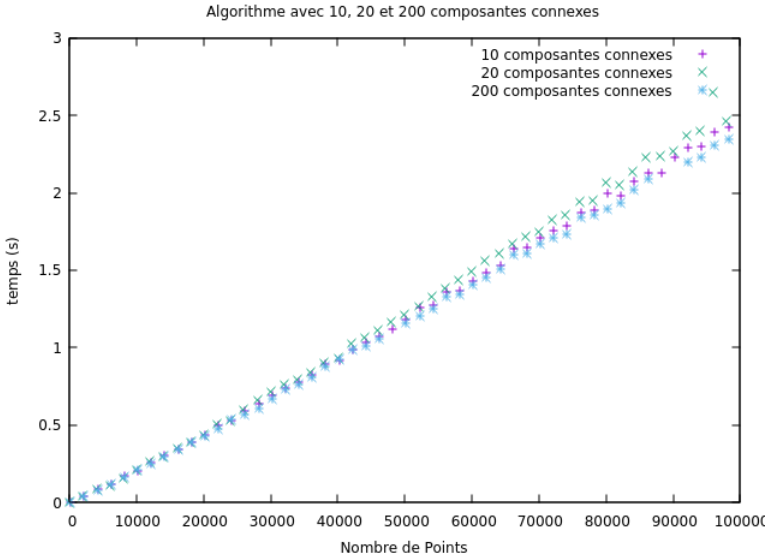
La fonction qui enlève les points éloignés ne dépend que du nombre de triangles formé par la triangulation de Delaunay.

Le parcours en profondeur dépend du nombre de sommets ($= N$) et du nombre d'arêtes. ($= O(N)$ (proposition 4)).



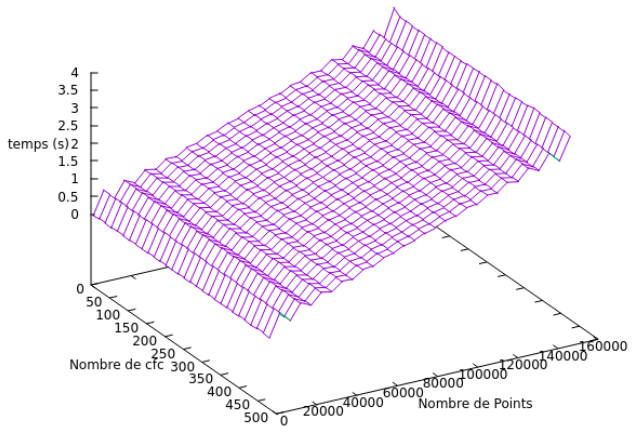

En faisant varier à nouveau le nombre de points, on peut aussi tracer un schéma comme celui de dessus pour s'en convaincre.

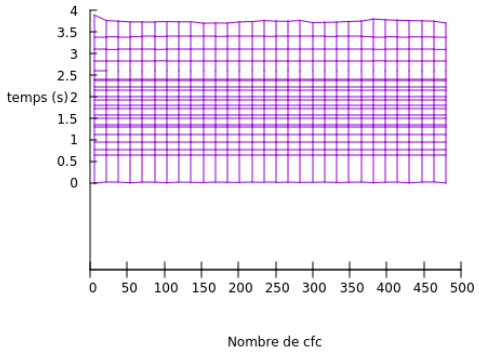



On peut remarquer l'entremêlement des points vers 40000 puis l'inversion de l'ordre de ces points vers 70000.

### 4.3.4 Graphe en 3D

Une représentation possible est en 3D en faisant varier le nombre de points et le nombre de composantes fortement connexes (cfc) en même temps.




En projection sur le plan, on voit des lignes horizontales. Cela rejoint l'idée d'un temps constant en fonction du nombre de composantes connexes.