Algorithmique et Science de Données

Nabil Bensrhier, Redouane Yagouti

Mai 2019

1 Version préliminaire

1.1 Description

Principe de fonctionnement:

On considère que l'ensemble des points constitue un graphe. On modélise alors chaque composante connexe par un arbre dont les arêtes ne sont possibles que dans le cas où elles vérifient la condition de la distance.
Dans cet ordre d'idées, on a besoin de 4 fonctions réalisant le principe espéré:

la fonction: **ancetres_fils(distance, points)** prend en arguments **le dictionnaire** renvoyé précédemment et la liste des points donnée, et renvoie un autre dictionnaire. (**une table de hachage**), dont les clefs sont les représentant, et leurs valeurs sont des listes de points voisins et leurs voisins.
Dans ce cas, on parcourt la liste des points, tout on parcourt une deuxième fois afin de trouver ses représentants sous forme d'une liste **value**.
Ces représentants constituent bien des ancêtres possibles du point qui est considéré comme un sommet. Mais on enregistre pas tous les ancêtres, car l'objectif est le fait de minimiser le nombre de représentant.
Il va sans dire que le coût de cette fonction est exactement $O(n^2)$

Ensuite : **arbres_possibles(dictionnaire, points)** prend en arguments **la distance minimale** et la liste des points donnée, et renvoie un dictionnaire (**une table de hachage**), dont les clefs sont les points, et leurs valeurs sont des listes de points. chaque liste contient des sommets dans la composante connexe dans laquelle le point appartient.
Mais, dans ce cas on a une complexité au plus $O(n^2)$.

Dans un troisième temps, la fonction **union_arbres(ensembles, points)** prend en argument le dictionnaire renvoyé et fait l'union des arbres qui ont des points communs, tout en supprimant les arbres dont les sommets ont déjà existé dans d'autres arbres plus grands au niveau de nombre des sommets.
Dans cette fonction, on a essayé d'utiliser les ensembles afin de détecter l'union et l'intersection entre de ensembles. Et puisqu'on ne compare que des listes, on les transforme sous forme

des ensembles puis, on les compare de manière suivante:

$$nouvelle\_liste = \textbf{list}(\textbf{set}(\textbf{liste1}) \quad \textbf{symbol} \quad \textbf{set}(\textbf{liste2}))$$

symbol : peut-être **&** pour l'intersection ou $\|$ pour l'union.

Finalement, la fonction **tri_size(listes, points)**, sert à trier la liste donnée en argument selon la longueur de ses sous-listes tout en supprimant les listes qui se répètent.
En utilisant le même principe qu'avant, puis on trie d'un manière décroissante selon la taille de chaque sous-liste:
sorted(listes, key=len) : trie la liste d'un manière croissante.
listes.reversed() : renverse le résultat trouvé.
Cette dernière fonction a un coût:

$$O(\sum_{i=1}^{n} i) = O(\frac{n(n+1}{2}) = O(n^2)$$

En déduit que notre première méthode a un coût totale: $O(n^2)$

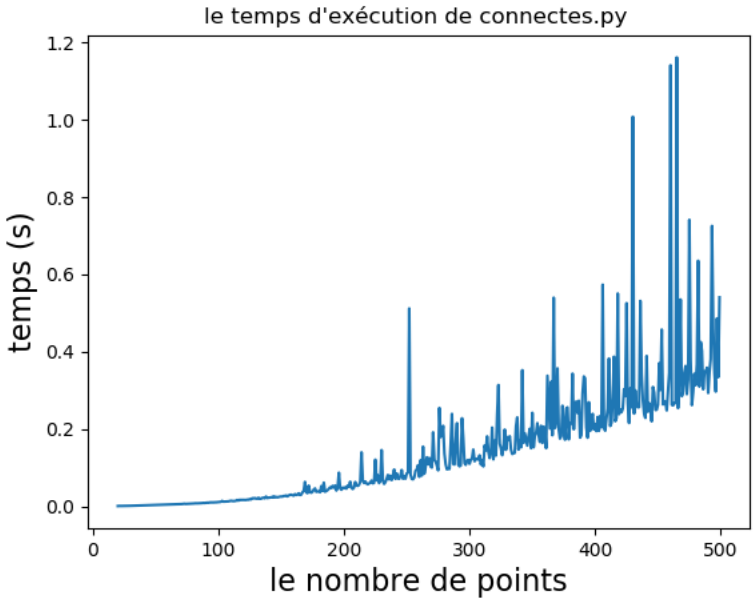

Figure 1: distance = 0.1

On remarque que le temps d'exécution pour 500 points peut atteindre 0.4 s qui est assez lent.

2 Version Améliorée

2.1 Diviser pour régner

La résolution précédente consiste à comparer les points du nuage deux à deux, ce qui donne une complexité en $O(n^2)$. On peut penser à réduire ce problème en faisant des divisions successives de notre nuage de points en 2 parties ou bien plus (4 par exemple), et résoudre les sous problèmes.

Soient N_1 et N_2 deux nuages de points différents, avec C_1 et C_2 leurs composantes connexes respectives. On défini la distance $d(N_1, N_2)$ entre N_1 et N_2 par $\min_{p_1 \in N_1, p_2 \in N_2} \mathrm{d}(\mathrm{p}_1, p_2)$.

On souhaite trouver l'ensemble C des composantes connexes de $N_1 \cup N_2$:

- Si $d(N_1, N_2) > s$ où s est le seuil, dans ce cas $C = C_1 \cup C_2$.

- Sinon, il faut calculer les distances entres les points de N_1 et de N_2 (distances inter-nuage après avoir calculé les distances intra-nuage).

On peut alors voir qu'il s'agit de résoudre des sous problèmes, mais en plus de ça il faut résoudre les problèmes de fusion qui auront des coûts importants pour notre cas.

Ce qu'on décide de faire dans un premier temps c'est de diviser le nuage de points en des carrées de dimension $\frac{s}{\sqrt{2}}$. Après on parcours l'ensemble des points pour voir quels points p_i $\in$carrée$_j$ est donc de trouver les composantes connexes à l'intérieur de chaque tableau.

Pour ne pas perdre de généralité, chaque carrée est entouré de plusieurs carrées voisins : 3 si dans le coins, sinon 5 si à côté des arêtes, sinon 8.

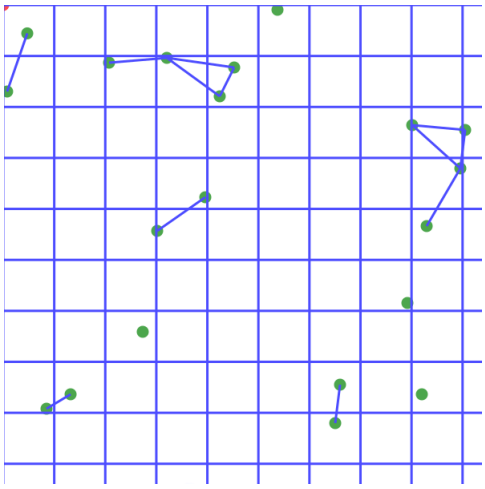

Figure 2: exemple$_1$

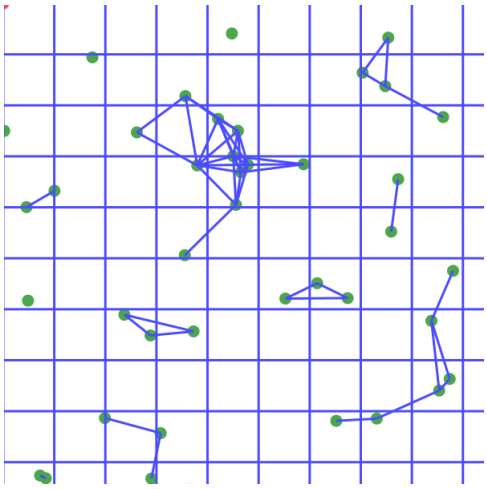

Figure 3: exemple$_2$

Sur les Figures 1, 2, 3 et 4 on fait une simple visualisation du problème pour trouver les composantes connexes. Le code de cette partie se trouve dans le fichier "connectes2.py".

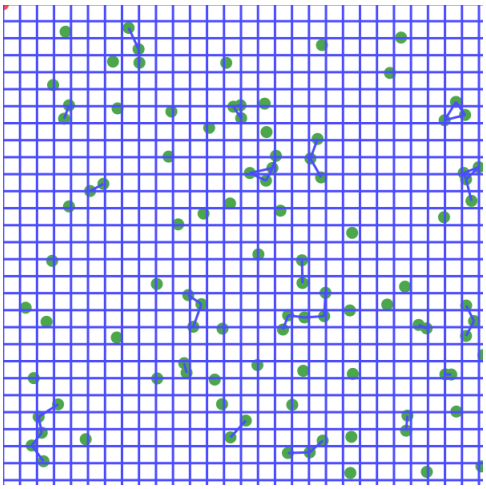

Figure 4: exemple$_3$

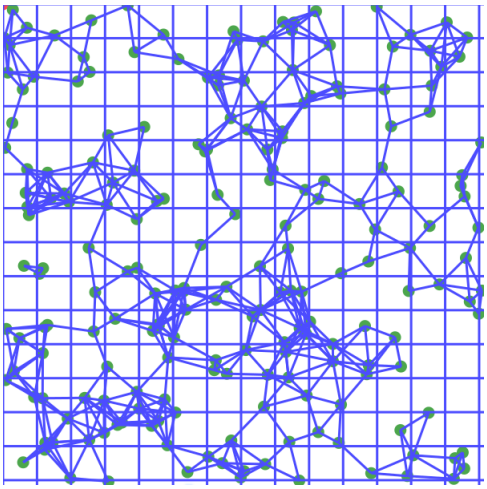

Figure 5: exemple$_4$

2.2 Analyse de coût :

On notera par C le coût du problème à n points. On note $m = [\frac{1}{s}]^2$ qui est le nombre de carrées dans la composition initiale et F le coût de la fusion

On a :

$$C(n) = mC(\frac{n}{m}) + F(m, n)$$

D'après le Master Theorem, on déduit que la complexité du programme sera plus emportée par les coût de la fusion car $F(n, m) \geq O(n^{(1+\epsilon)})$.

Alors on peut déduire une borne inférieure et une borne supérieure de cette complexité :

$$nlog_2(n) \leq C(n) \leq n^2$$

La complexité de F dépend de m, et donc dépend du seuil choisi, ainsi cette méthode peut être idéale pour certaines valeurs de m où on sera obligé de diviser notre nuage en un grand nombre de points (mais pas trop grand sinon on se jette dans le premier cas et donc on atteint la quadrature).

Le temps d'exécution est à 0.008 ce qui montre que le programme s'est amélioré, il est plus rapide 50 fois que le premier.

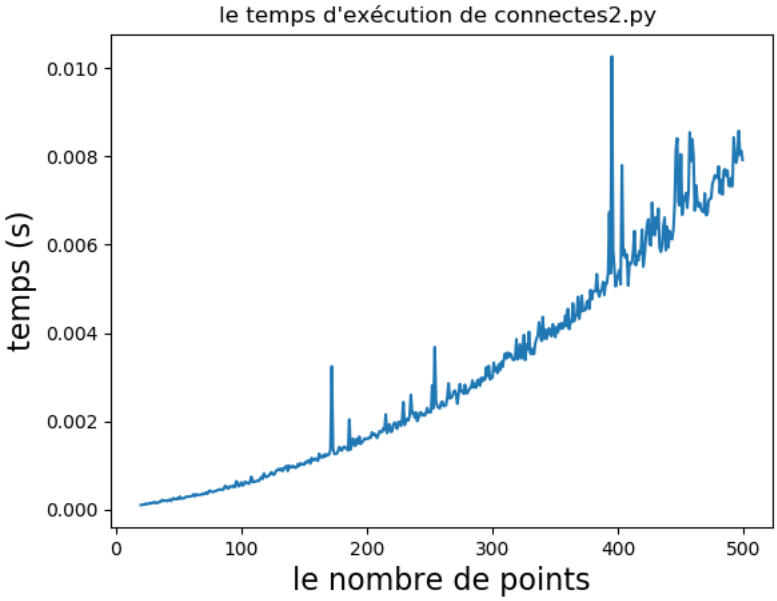

Figure 6: distance = 0.1

3 Une Autre Version

3.1 Chercher un $O(nlog_2(n)$: "connectesXY.py"

Dans la méthode qui précède on se jette souvent dans les coûts quadratiques, ceci est bien évidemment à cause des opérations de fusion. Dans cette partie on a pensé une nouvelle méthode pour éviter le plus de tomber dans des cas critiques de fusion, ou bien le cas échéant dans des fusions aisée.

La première étape est de trier les points selon les abscisses, et de les stocker dans un tableau $SortedX$, de même pour les ordonnées on les trie et on les mets dans un tableau $SortedY$.

L'idée est de parcourir les abscisses en succession (c'est à dire comparer $SortedX_i$ avec $SortedX_{i+1}$, une fois qu'on trouve que $d(SortedX_i, SortedX_{i+1}) > s$ alors on est sur qu'il n'y aura pas d'inter-composantes entre les points avant $SortedX_i$ et les points après $SortedX_{i+1}$. On peut constater sur les figure 5 et 6 que la séparation par le segment est bien cohérente.

Après cette étape on continue la même chose tout au long du nuage des points après $SotredX_i$ par la même opération. Une fois qu'on a fini des abscisses on passe aux ordonnées, mais cette fois ci on ne compare que entre le nuage de points déjà calculé par la séparation selon les X. On refait exactement la même opération sur ces sous ensemble mais selon les ordonnées.

On peut voir sur les figures 7 et 8 ce dont on parle :

On poursuivant ainsi on peut trouver toutes les composantes connexes.

MAIS il peut y avoir un problème, c'est que ces opérations peuvent ne pas mener à aucune division ou séparation comme dans cette figure 9 :

Comme vous le remarquez sur Figure 12 il n'y a aucune séparation ni au niveau des x

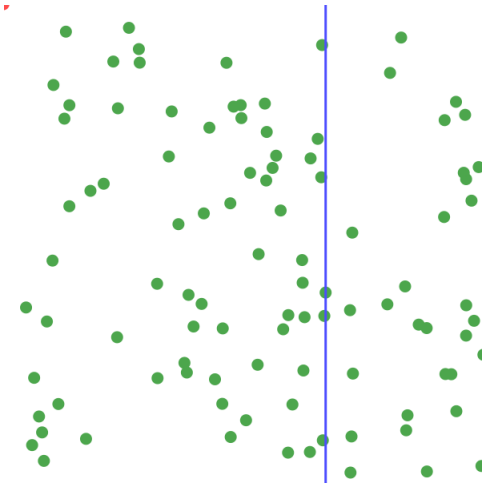

Figure 7:

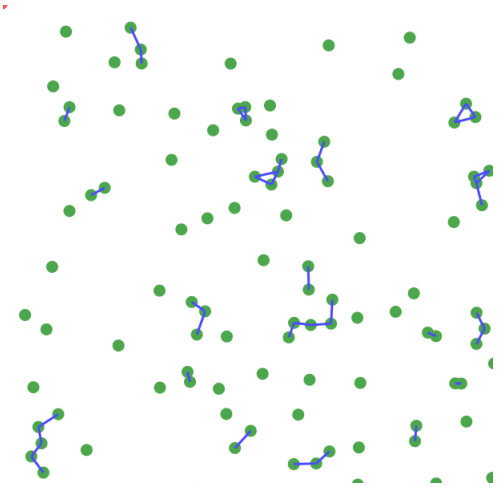

Figure 8:

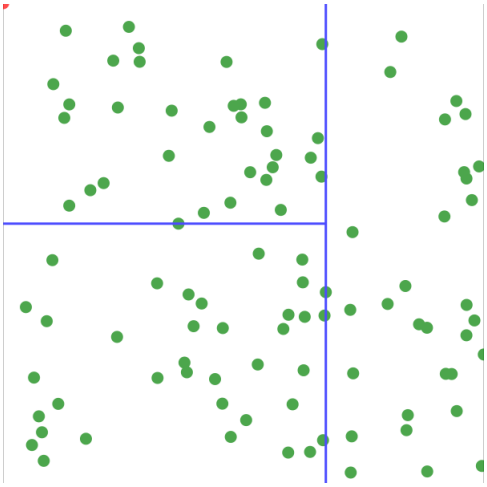

Figure 9:

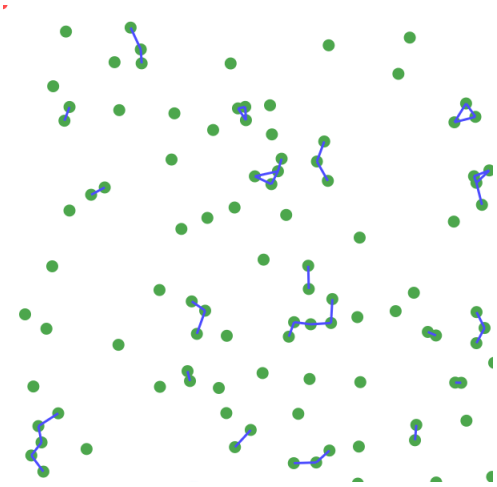

Figure 10:

ni au niveau des y. Ceci nous mène alors à diviser notre nuage de points en deux parties selon les abscisses : on prendra la médiane pour avoir un même nombre de points des deux côtés. Et on test notre méthode, si ça marche pas on divise simultanément selon les x et les y jusqu'à ce que ça devient possible.

On peut proposer un autre amélioration vu qu'on sera amené à chercher les inter-composantes, c'est qu'on va faire un chevauchement entre les classes divisée d'un seuil s pour que la comparaison inter-classe soit d'ordre linéaire.

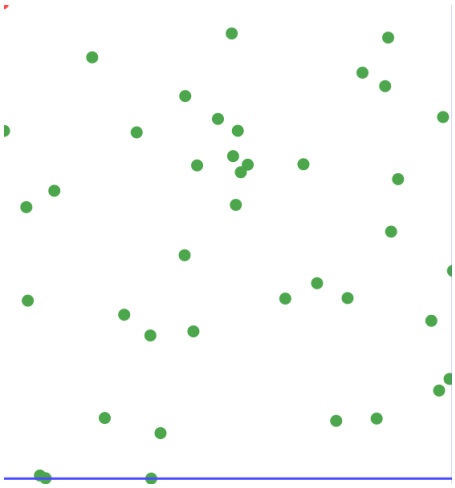

Figure 11:

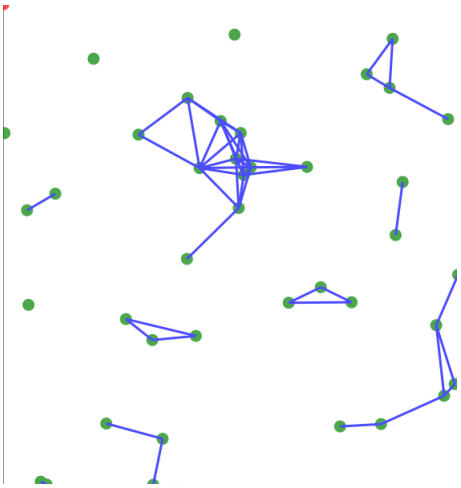

Figure 12:

3.2 Analyse du coût :

On peut constater que le coût de cette méthode sera selon les cas un $O(nlog_2n)$ ou bien un $O(n^2)$ au pire des cas. On peut aussi voir que si le nuage de points est plus dispersé cette méthode serait plus rapide que la précédente vu qu'on pourra trouver des divisons assez rapidement.