RAPPORT PROJET ALGORITHMIQUE ET STRUCTURE DE DONNÉES

Tailles des composantes connexe d'un ensemble de points sous contrainte d'une distance seuil

RANDRIAMORA Yevann
ROUSSEL Guillaume
Groupe 4

Remarque : Les algorithmes ont été lancés sur un pc portable sous Manjaro avec un i7-8750h, 16go de RAM et SSD 512Go. Nous avons pu observer des différences dans le temps d'exécution selon le système d'exploitation (Windows, Windows WSL, le processeur…). Les temps sont en général meilleurs sous Manjaro natif.

Remarque (bis) : Voici un lien pour accéder aux versions préliminaires 1 et 2 de notre algorithme.
https://drive.google.com/drive/folders/16hPr5iwhzYPjsL6cqwnYxqmiT9KUGDPG?usp=sharing

Introduction

Objectif : Trouver la taille de toutes les composantes connexes d'un ensemble de points dans le carré unité [0, 1] x [0, 1] avec une condition de distance seuil.

Au cours de ce projet, nous avons développé une première version de notre code qui fonctionne mais est assez lente, puis nous nous sommes intéressés à la structure de données Union-Find utilisable sur Python afin d'obtenir un code, bien que plus complexe, au temps d'exécution beaucoup plus court. Et enfin nous avons étudié l'utilisation des barycentres pour une condition de filtrage.

Première version

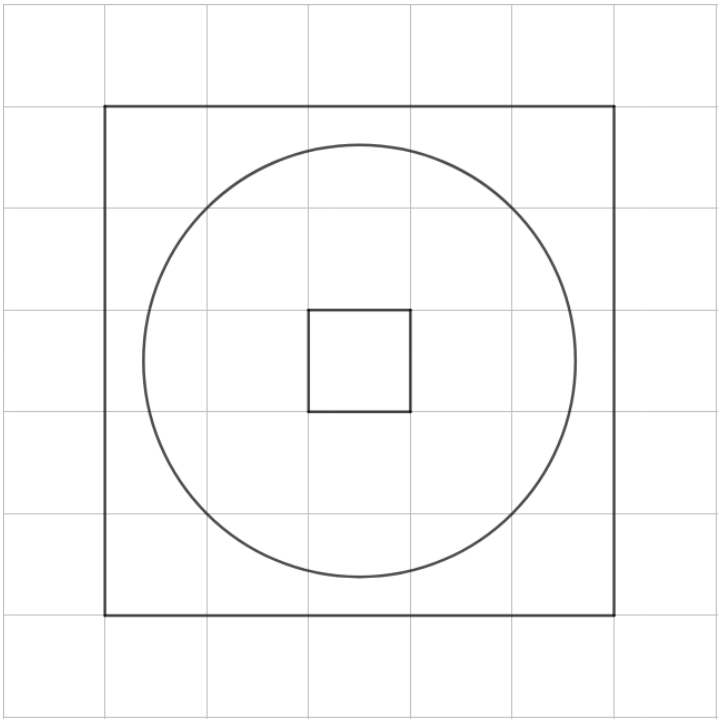
Dans un premier temps on divise le carré unité en un quadrillage de cases de diagonale de taille s, la distance seuil en entrée du programme. Ainsi tous les points présents dans une même case font partie d'une même composante connexe. On stocke les cases et leurs points dans un dictionnaire $composantes$. On repère les cases par des entiers en utilisant la formule suivante :

$$cote = \sqrt{2}$$
$$i = PartieEntiere(x / cote) + 1$$
$$j = PartieEntiere(y / cote) + 1$$

Avec (x, y) un point du carré unité

Nous avons utilisé des entiers car nous avons eu énormément d'erreurs d'arrondis. Cela nous menait donc à des erreurs de clé donc on ne détectait pas la composante. Cela permettait également d'avoir un code plus lisible. Il ne fallait plus que multiplier par $cote$pour obtenir les coordonnées du coin supérieur droit de chaque case. Cette étape est en $O(n)$: on parcourt une seule fois la liste de points, et on fait globalement la même séquence de calculs.

Ensuite, nous devons trouver les cases connexes de chaque case. Pour cela, nous prenons chaque clé/case du dictionnaire $composantes$, et nous regardons les 24 cases autour de cette case/clé (voir dessin ci-dessous). Pour cela, nous pré-trions nos points de nos 2 cases : soit par abscisses croissantes ou décroissantes, soit par ordonnées croissantes/décroissantes. Et selon la position de la case que nous voulons vérifier par rapport à la case de l'itération, nous choisissons 2 de ces possibilités de façon à favoriser les points proches. Nous avons commencé avec un tri-fusion "maison" puis sommes passés au *Timsort* (implémenté dans *sort()*) afin de pouvoir facilement passer en argument de celui ci de quoi trier nos listes par rapport à la deuxième composante plutôt que la première à l'aide de la fonction *itemgetter* du module *operator*, ou encore trier à l'envers, le tout en une seule ligne de commande.



Le schéma GeoGebra ci-dessus représente la case centrale lors d'une itération, et toutes les cases à trier. Le cercle est l'enveloppe de tous les cercles de rayon distance de la case centrale. Cela montre bien pourquoi il y a bien 24 cases (en pratique 20) à checker.

Si les 2 cases sont bien connexes, comme nous utilisons des ensembles, nous faisons des *updates* de ceux-ci. Cette opération est en $O(n)$.

Nous avons $O(n)$ cases. Chaque itération de la boucle *for* est en $O(1)$ (malgré une grande constante multiplicative devant car nous avons au moins 20 *if.* Chaque tri est en $O(nlog(n))$ en moyenne, et regarder si 2 composantes connexes sont donc en moyenne en $O(log^2(n))$. Donc cette étape est en $O(n * (2nlog(n) + log^2 n)) = O(nlog(n))$. Les deux tri permettent donc effectivement de gagner du temps.

La dernière étape consiste à profiter du fait que nous avons pas de disjonction entre le cas : case déjà traitée et case non traitée. Cela entraîne le fait que des points se situent dans plusieurs ensembles. Ainsi, on va donc parcourir les clés restantes avec 2 boucles *for* et si leurs ensembles ont une intersection non vide (opération en moyenne en $O(n)$ et en pire des cas en $O(n^2)$), on fusionne les ensembles. Et on effectue cela tant que le dictionnaire de retour change. Donc on est en $O(n^3)$.

Cet algorithme est donc en $O(n^3)$ (moyenne et pire cas).

Points positifs de cette version :

- Même si les 24 *if* peuvent clairement paraître barbares, cela nous permet de faire un tri personnalisé en fonction du cas dans lequel on se trouve et donc d'optimiser du temps de calcul
- Une grand partie des opérations sont en $O(n)$, les tris sont en $O(nlog(n))$et le check pour savoir si 2 cases sont connexes est en moyenne en $O(log^2(n))$
- Code simple à comprendre

Points négatifs de cette version :

- Code très long (pratiquement 500 lignes), donc fastidieux à corriger et à modifier
- Les 24 *if* sont synonymes d'agacement, d'oublis et de longue durée de correction et modification
- Dans le pire cas du check entre 2 cases (quand elles ne sont pas connexes), la complexité est en $O(n^2)$.
- La réunion globale des composantes est clairement en $O(n^3)$ (double boucle *for* dans un *while*). Donc on est très dépendant du nombre de points (3 min pour 100k points au minimum)

Deuxième version

Nous avons décidé de garder l'étape de division en cases du carré unité (la création des classes d'équivalences initiales de points).

On passe de 24 à 20 *if* en retirant les 4 cas limites qui correspondent aux cases à deux cases en diagonale de la case vérifiée car ils sont inutiles en pratique.

On utilise dorénavant la structure **Union-Find** qui nous permet de représenter des classes d'équivalences. Cette structure est très pratique car ses opérations de recherche (Find : trouve le père d'un élément et Union : unie 2 éléments) sont en $O(1)$ en moyenne et dans la pire des cas en $O(log(n))$. On indexe donc les t-uplets de coordonnées des cases par un indice. On ne manipule donc les points que lors du check de 2 composantes et lors du comptage de la taille des composantes. Les *children* d'Union-Find sont donc les coordonnées des cases (ou du moins leur indice correspondant).

Mais une différence notable est qu'on ne fait plus réellement d'opérations sur des ensembles de points, on ne fait qu'ajouter des indices dans une liste. C'est donc beaucoup moins coûteux.

Une seconde différence importante est que désormais, avec les 20 *if*, au lieu d' *update* des ensembles en cas de connexité entre 2 cases, on ne fait qu'ajouter un couple/t-uplet de la forme :

$((i,j),\ (i\ +\ k,\ j\ +\ l))$ avec $k,l\ \in \{-\ 2,\ \ -\ 1,\ 0,\ 1,\ 2\}^2$.

Cela nous forme donc une pile de couple/t-uplet de cases.

Une dernière différence est que désormais on ne checkera plus 2 fois la même case. En effet, on stocke dans un dictionnaire (les coordonnées sont la clé, la valeur est *True*). Ainsi, lorsqu'une case traitée apparaît à un instant donné dans les 20 *if* d'une autre case, alors on n'entre pas dans ses *if* à elle.

Pour obtenir le résultat, on a plus qu'à dépiler la pile progressivement et de faire une union entre les 2 éléments du couple. L'Union-Find se chargeant donc pour nous de faire la réunion de toutes les cases entre elles. Cette étape est en $O(n)$(la recherche des indices des cases est en $O(log(n))$.

Pour trouver les tailles, il ne reste plus qu'à parcourir tous les children d'une composante et à sommer leur taille. Cette étape est en $O(n)$.

Cet algorithme est donc en $O(nlog(n))$en moyenne, et en $O(n^2)$dans le pire des cas.

Points positifs :

- Nous avons clairement réduit notre temps d'exécution et sommes désormais en $O(nlog(n))$.
- Nous avons donc clairement moins de check entre 2 cases, ce qui réduit le nombre d'appels de fonction en $O(n^2)$ ou $O(nlog(n))$ ou $O(log^2(n))$.
- Beaucoup plus de fonctions sont en $O(n)$: y compris l'union des cases qui étaient jusque-là notre plus gros problème.

Points négatifs :

- Code toujours très long
- Toujours 20 *if*
- Même si le cas moyen est en $O(log^2 n)$, le check de 2 cases est toujours dans le pire cas (cases non connexes) en $O(n^2)$.
- Plus compliqué à comprendre

Troisième et dernière version

Pour réduire le temps en du check entre 2 composantes, il nous faut donc trouver une condition nécessaire sur la connexité de 2 cases. Nous avons donc utilisé le barycentre. Pour chaque case, nous calculons le barycentre des coins d'une case. (le milieu de la case) et nous prenons le point max_ide la case la plus éloignée du barycentre.

Si nous faisons pour 2 cases à checker, on a alors la condition nécessaire à la connexité suivante :

$distance(max_1, max_2) <= (2 * distance)^2$, avec les max_i les points les plus éloignés du barycentre de leur case respective *i*.

Calculer le barycentre des coins est en $O(1)$, calculer le max est $O(n\,logn\ +\ n\ )\ =\ O(n\,log\,n)$. Donc si on fait ça pour les 2 cases, on obtient à la fin un pré-filtrage en $O(nlogn)$. Nous n'avons donc plus d'étapes en $O(n^2)$.

Nous avons également apporté d'autres optimisations. En effet, pour faciliter la correspondance entre indice et le t-uplet de coordonnées des cases, on a dû rajouter 2 dictionnaires : un qui a comme clés les t-uplets coordonnées des cases et en valeur leur indices correspondants, l'autre leurs indices correspondants en clé et les t-uplets coordonnées en valeur. Ainsi, la recherche de l' indice ou des coordonnées d'une case dans l'Union-Find se fait se fait toujours en $O(1)$.

Cet algorithme est donc en $O(nlog(n))$, à côté, de nombreuses étapes en $O(n)$ ou $O(1)$ ont été remplacées par des étapes en $O(1)$.

Points positifs :

- $O(nlog(n))$, beaucoup d'étapes en coût constant
- L'algorithm gère donc très bien le pire des cas dans le check de 2 cases

Points négatifs :

- Toujours long
- Toujours 20 *if*
- Comme on a rajouté énormément d'éléments pour améliorer les opérations d'un point de vue temporel, on a donc une moins bonne gestion spatiale/mémorielle

Améliorations que l'on pourrait encore apporter :

- Pour enlever les 20 *if*, on pourrait utiliser les tableaux de Karnaugh avec 2 sorties pour savoir si on trie par abscisses ou par ordonnées, de façon décroissante ou non. Les opérations logiques étant logiquement plus rapides, on aurait un code beaucoup plus court, sans aucune disjonction de cas. On perdrait cependant la facilité de compréhension du code car tout est bien séparé.
- En faisant des calculs de distance, on manipule des flottants avec énormément de chiffres après la virgule. Python fait donc énormément d'arrondis, et cela peut mener à des erreurs de code. Même si on évite le pire en travaillant avec des distances au carré, on pourrait quand même y faire attention en prenant en compte le zéro machine.

Analyse de performance sur différentes instances

Avec les fichiers fournis :

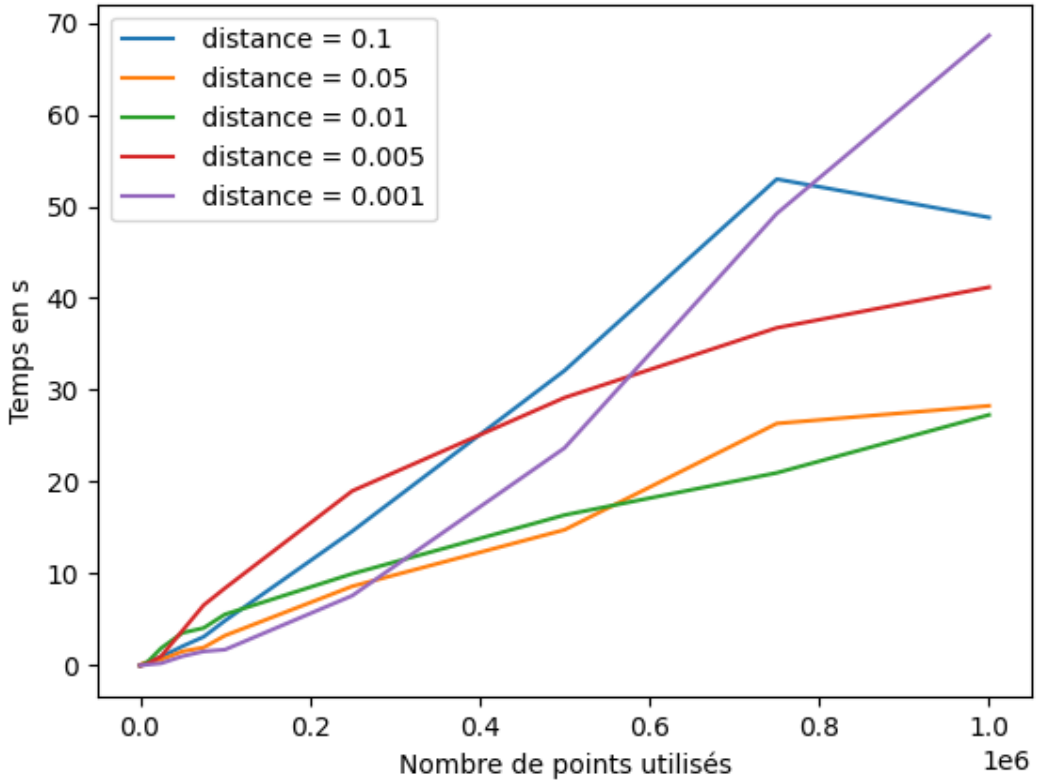
En faisant une moyenne sur 10 exécutions :

exemple_1.pts	exemple_2.pts	exemple_3.pts	exemple_4.pts
0.631s	0.645s	0.652s	0.66s

Avec mes propres fichiers :
Les nombres de points utilisés sont : [100, 1000, 5000, 10000, 25000, 50000, 75000, 100000, 250000, 500000, 750000, 1000000].

Les distances utilisées sont : [0.1, 0.05, 0.01, 0.005, 0.001].
Les points sont générés selon une loi uniforme avec la fonction *random.random()*.

Evolution du temps d'exécution de l'algorithme selon le nombre de points et la distance seuil :



Selon le graphe, on peut clairement voir qu'on est en complexité $O(n\ log(n))$, voire en $O(\ log^2 n)$selon la "chance" qu'on a avec les distributions de points. On peut également voir que les résultats dépendent également de la distance fournie.
En effet, plus elle est petite, plus le nombre de cases augmentent, et donc plus il y aura de check entre composantes. Pour pallier ce problème on peut créer un algorithme qui ne passe plus par des cases mais travaille directement avec les points, ou encore modifier la taille des cases pour les rendre plus grandes.
De même, si elle devient assez grande, on a beaucoup plus de points dans une case, donc il y aurait plus de comparaisons entre les points (pour savoir si les cases sont connexes ou non). Il faudrait donc réduire la taille des cases.