Rapport Projet Algo

Introduction

Étant donné un ensemble de points du plan et une distance (qu'on notera D dans la suite), l'objectif est de trouver les tailles des composantes connexes du graphe dont les sommets sont les points. Sachant que deux points de ce graphe sont connectés si et seulement si la distance euclidienne entre eux est inférieure ou égale à D.
On formalise le problème comme suit :
-$P = \{p1, p2, ..., p_n\}$ l'ensemble des points du plan.
-$D \in]0, 1]$ la distance limite.
-$G = (P, E)$ le graphe qui nous interesse, tel que $E = \{p_i \leftrightarrow p_j \ tq \ dist(p_i, p_j) \leq D \ \forall i \neq j \in [1, n]\}$ ($p_i \leftrightarrow p_j$ signifie l'arrete qui lie p_i à p_j) et $dist : [0, 1]^2 \times [0, 1]^2 \rightarrow [0, 1]$ la distance euclidienne entre deux points.
Dans la suite, nous allons proposer différents algorithmes qui répondent à ce problème.

-Entrée: un fichier qui contient la distance D dans sa premiere ligne, et n points (sous la forme x,y) chacun dans une ligne.
-sortie: un tableau trié par ordre décroissant, contenant les tailles des composantes connexes du graphe.

Premier algorithme

une tentative naïve, mais qui résout le problème. on considère à chaque fois un point, et on cherche parmi tous les autres points ceux qui sont à une distance inférieure ou égale à D du point considéré. une optimisation serait d'éviter de recalculer certaines distances.
Exemple : on considère 4 points p_1,p_2,p_3,p_4.
on stocke ces points dans un tableau. on peut considerer p_i et chercher ses voisins parmi tous les autres p_j tq $j \neq i$. Mais on peut faire mieux : on considère p_i, et on cherche ses voisins parmi tous les p_j tq $j > i$, cela diminuera la complexité des 2 boucles imbriqués en passant de n^2 à $\frac{n(n+1)}{2}$
Calculons la complexité de cet algorithme en fct de n (avec n le nombre de points).
On suppose que la compléxité du chargement de données dans le tableau est O(n). la compléxité de la fonction print_components_sizes est : -O(n) (lignes 27)
-

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c = c.\frac{n(n+1)}{2} = O(\frac{n(n+1)}{2}) \ \ (ligne 29 - 33)$$

Pour les lignes 36-44, la complexité est en O(n) puisque le while de la ligne 36 s'executera au plus n fois, car la taille du dictionnaire diminue à chaque iteration. Enfin un O(nlog(n)) pour

trier le tableau. on obtient donc :
$C(n) = O(\frac{n^2}{2})$. Avec la démarche naïve on va obtenir $O(n^2)$.
on remarque que dans cette méthode, la partie la plus couteuse est la création du dictionnaire, c.a.d la création d'une structure de données qui faciletera la recherche des voisins. c'est ce qui nous méne à l'algorithme suivant : les KDtree.

Deuxième algorithme: KDtree

on pourra procéder comme suit : chercher une structure de données, dont la construction n'est pas coûteuse, et qui permet de chercher les plus proches voisins d'un point efficacement.

KDtree est une structure de données qui sert à partitionner l'espace de manière à faciliter la recherche, l'insertion, la recherche des K plus proches voisins (KNNS : K nearest neighbors search) et sa variante : la recherche des points à une distance en plus "Radius" d'un point donné.

Cependant, le 'challenge' est le choix du point qui déterminera la droite de division : en fait, pour construire un arbre équilibré qui fournira des bonnes propriétés de recherche, il faut choisir, si on est sur les abscisses, le point dont l'abscisse est la médiane des abscisses. Pour faciliter la tâche, on trie le tableau des points qui restent selon la composante qui nous interesse, sachant qu'on alterne entre les composantes, et on prend le point qui se trouve au milieu du tableau trié; c'est celui qui sera dans le noeud, la partie gauche et droite du tableau seront utilisées pour créer le sous-arbre gauche et le sous-arbre droite du noeud.
Calculons la complexité de l'algorithme :
On a un O(n) pour charger les points depuis le fichier.
Construction de l'arbre (algorithme récursive ligne 76) :

$$
\begin{aligned}
C(n) &= 2c\left(\frac{n}{2}\right) + O(n\log(n)) \\
&= 2^{\log_2(n)} + n\sum_{i=0}^{\log_2(n)-1}\log\left(\frac{n}{2}\right) \\
&= n\left(1+\log_2\left(\frac{n^{\log_2(n)}}{2^{\frac{\log_2(n)^2-\log_2(n)}{2}}}\right)\right) \\
&= n\left(1+\log_2\left(2^{\frac{\log_2(n)^2+\log_2(n)}{2}}\right)\right) \\
&= n\left(1+\frac{\log_2(n)^2+\log_2(n)}{2}\right) \leq n\log_2(n)^2 \quad \text{à l'oo} \\
&\qquad donc \quad C(n) = O(nlog_2(n)^2)
\end{aligned}
\tag{1}
$$

Remarque :
Il existe un algorithme complexe qui trouve la médiane en O(n), c'est la médiane des médianes. Dans ce cas, on obtient : $C(n) = O(nlog_2(n))$

Trouver les plus proches voisins : ici c'est un peu compliqué, dans un arbre équilibré cette operation a un coût de $O(log(n))$ pour la recherche d'un seul plus proche voisin, mais dans

notre cas on ne sait pas combien notre point admet de voisins, c'est une variable aléatoire qui peut prendre {1,...,n-1} n-1 valeurs. Au pire cas, tous les points sont dans la même composante connexe. on éxécutera pour chaque point, la fonction 'recherche_voisins' qui coutera $O(nlog(n))$ car tous les autres n-1 points sont voisins du point considéré : les lignes 88-104 nous couterons $O(n^2log(n))$.

la complexité de l'algorithme sera donc $O(n^2log(n))$. On remarque donc que cette approche ne fonctionne pas si la distance D et n sont très grandes.

On comparera entre les KDTree que nous avons implémenté et les cKDTree du module scipy et on verra leurs performance à la fin.

3ème algorithme : Méthode de partition du plan avec matrice

Le principe est simple, au lieu de considérer à chaque fois un seul point, on considère plusieurs points de manière à ce qu'on sera sur du fait que tous ces points sont liés entre eux. On partitionne le plan en carrés tel que la distance la plus grande entre 2 points appartenant à un carré soit inférieur ou égale à D : c'est la diagonale. On considère donc les carrés de diagonale $d = D$, c.a.d de coté $c = \frac{D}{\sqrt{2}}$. On crée une matrice dont chaque coefficient (i,j) représente un carré et Matrice[i][j] repésente un tableau qui contient les points appartenant au carré d'indice (i,j). On insère les points dans la matrice. Maintenant il faut chercher une méthode qui nos faciletera la fusion des parties des composantes connexes. Pour cela, on crée une classe Collection_de_point qui contiendra après l'execution de la fonction 'Partition_avec_matrice', et pour chaque point, l'ensemble des points de la composante connexe à laquelle il appartient. Calculons la compléxité de cet algorithme.
-Création de la matrice : On pose $nb\_carre\_par\_ligne = m$

Alors la création de la matrice se fera en $O(m^2)$

Or $m = int(\frac{1}{c}) \approx \frac{\sqrt{2}}{D}$ donc on crée la matrice en $O(\frac{1}{D^2})$ (ligne 57)

-Insertion des points dans la matrice ; $O(n)$ (ligne 59-60)

-Pour les lignes 62-65, le calcul exacte de la complexité est complexe et depend fortement de la distribution des points, on considère donc une destribution uniforme tq chaque carré contient le même nonmbre de points en moyenne, c.a.d $\frac{n}{m^2}$ points. Ce qui fait que la complexité des lignes (62-65) est $O(n)$.

-Pour les lignes 69-83, on considere le pire cas, c.a.d quand toutes les cases de la matrice sont pleines, on considére aussi que chaque carré contient $\frac{n}{m^2}$ points. Dans ce cas, la fonction 'conn_exists' coûtera au pire cas $(\frac{n}{m^2})^2 = \frac{n^2}{m^4}$.

Enfin, dans la ligne 83 on fait appel à la fonction 'add_collection', qui coûtera soit $O(1)$ soit $O(\frac{n}{m^2})$, encore une fois cela depend de la distribution, on suppose que 'add_collection' coûtera toujours $O(\frac{n}{m^2})$ même si cette supposition est grossière. on obtient à la fin (sachant que $m = \frac{\sqrt{2}}{D}$)

$$C(n) = O(n) + O(\frac{1}{D^2}) + O(n^2(10[\frac{n^2D^4}{4} + \frac{nD^2}{2}]))$$

~(10 car on a 10 mouvements possibles)

On remarque l'apparition du terme $O(\frac{1}{D^2})$ dans l'initialisation de la matrice. On verra dans la dernière partie du rapport que lorsque $D \to 0$, ce terme tend vers l'infini rapidement ce qui provoque l'echec de l'algorithme dans la ligne 57...
Pour surpasser ce problème, nous avons decider d'optimiser cette méthode en évitant de créer les carrés inutiles : c.a.d les carrés vides.

Méthode de partition de plan optimisée avec dictionnaire :

Au lieu de créer la matrice décrite dans la section précédente, on crée un dictionnaire, les clés sont les indices des carrés $\{0, 1, ..., m^2 - 1\}$, la valeur d'une clé est une liste contenant les points appartenant au carré qui admet cette clé comme indice. Comme on est pas autorisé à importer les defaultdic, on utilise un bloc try/exept KeyError. on obtient donc le dictionnaire en O(n) au lieu de $O(\frac{1}{D^2} + n)$ pour la méthode précédente.

Donc si $D \to 0$, cette méthode ne sera pas influencée mais on perd quelque chose comme même, on doit vérifier si les indices calculés des voisins appartiennent au dictionnaire. on a résolu donc le problème de l'échec de la méthode quand $D \to 0$

Courbes de complexité:

Dans cette section , on va commencer par tracer, pour différents seuils, les courbes d'evolution du temps en fonction du nombre des points n, pour les 4 algorithmes : KDTree, cKDTree, Partition_avec_matrice et Partition_avec_dictionnaire.

Afin de créer des fichiers de test (contenant uniquement des points sans seuil), on utilisera le fichier 'gene_points.py' qui se trouve dans le repertoire 'cr_n_'. ce programme python prend comme entrée sur la STDIN le nombre de divisions voulu sur l'échelle logarithmique de l'intervalle $[10^{3.5}, 10^{6.2}] \approx [3\ 000, 1\ 600\ 000]$ (vous pouvez changer ces bornes dans le fichier). Pour nos tests, on choisira 50 comme nombre de divisions. On aura donc 50 fichiers 'crX.pts' (dans le repertoire cr_n_) avec $\{n_0 < n_1 < \cdots < n_{49}\}$ points générés aléatoirement.(Nous avons supprimer les ficiers crX.pts puisque la taille du compressé dépasse la taille autorisée par TEIDE).

On tracera donc pour chaque fichier, pour différents seuils et pour les 4 algorithmes les courbes de complexité.

Pour lancer les 4 algorithmes sur les fichiers de test on utilise le programme 'test.py' qui génére à partir des fichier 'crX.pts' qui se trouvent dans cr_n_, des fichiers 'result_D' dans le repertoire 'resultat_test', et pour tracer les différentes courbes on utilise le programme 'trace.py' qui se trouve dans le repertoire 'resultat_test'. 'trace.py' prend en argument sur la STDIN un fichier 'result_D' et trace sa courbe dans le reperoire 'courbes_resultats'

Pour éviter d'avoir un test qui prendra des heures voir des jours, on utilise la fonction 'timeout' (https://stackoverflow.com/a/13821695) implémentée dans 'test.py' avec un timeout de 60 secondes. On utilise ensuite un boolean qui nous indiquera, pour un seuil et un algorithme

fixé, quand ce dernier dépassera le timeout en un certain fichier contenant n_i points, pour éviter de lancer l'algorithme dans les itérations suivantes pour les fichiers contenant $n_{j>i} > n_i$

Interpretations :

Les resultats théoriques sont vérifiées, l'algorithme 'Partition_avec_matrice' échoue quand $D \to 0$. Cela est expliqué par le terme $O(\frac{1}{D^2})$ qui apparaît dans la complexité de l'algorithme, autrement dit, si le D est de l'ordre de 10^{-5}, alors cet algorithme devra créer une matrice avec 200 000 000 cases.

Cependant l'algorithme 'Partition_avec_dictionnaire' réussit même si $D \to 0$.

Mais ce dernier n'est pas toujours plus performant que 'Partition_avec_matrice' pour D entre 0.5 et 0.001, puisque si la création de la matrice ne prend pas beacoup de temps, alors l'accés au carrés est immediat, par contre pour 'Partition_avec_dictionnaire' il faut vérifier à chaque accés de carré si son indice existe ou pas dans le dictionnaire.

On remarque l'échec des KDTree et cKDTree quand $D \to 1$ à cause du terme $O(n^2 log(n))$ qui apparaît dans la complexité des KDTree et cKDTree si $D \to 1$ comme expliqué dans la section des KDTree ci-dessus.

On remarque aussi que les cKDTree importé du module scipy sont moins efficaces que les KDTree que nous avons implémenté quand D est grand. Cependant les cKDTree sont largement mieux que les KDTree quand $D \to 0$. On en déduit que le fait d'implémentation des KDTree en python pure et cKDTree en C++ (ce qui rend les cKDTree très rapide) n'est pas la seule différence entre les deux structures de données...

Exploitation des résultats :

Nous allons utiliser des conditions sur D et n pour écrire le fichier 'connectes.py', la version finale qui décide quel algorithme utiliser sur une entrée donnée, en se basant sur les courbes de complexité. Puis on tracera la courbe de complexité finale de notre algorithme qui utilisera les 3 algorithmes 'KDTree', 'Partition_avec_matrice' et 'Partition_avec_dictionnaire', à l'aide des programmes 'test_optim.py' et 'trace_optim.py'. (pas de cKDTree à cause des restrictions sur l'importation des modules).

les courbes qui se trouvent dans le repertoire 'courbes_finales' montrent que la versions finale réussit dans presque tous les seuils, grâce à la combinaison des trois algorithmes.

Conclusion :

Le projet nous a montré comment les algorithme sont sensibles aux types des entrées, et qu'il faut avoir une idée sur l'entrée afin de décider quel algorithme utiliser. Nous avons vu plusieurs algorithmes qui répondent au problème, certain ne figurent pas ici comme:
-l'utilisation de LSH (locality sensitive hash) pour hasher les points dans des packets avec une forte probabilité d'avoir les points voisins dans le même paquet, ce qui est une solution

fortement recommandée quand la dimention des points est grande.
-Union Find qui était implementé mais qui n'a pas donné des bonnes résultats.

Groupe 7 : El Asli Adnane | Nasmane Abdelhadi