Rapport de projet algorithmique

Projet 3MMALGO 2020-2021

ELHAJ-LAHSEN Hugo, NEMO Maxime

Contents

Introduction

Ce document décrit notre projet d'algorithmique. Notre tâche était de résoudre un problème de recherche de composantes connexes dans l'espace. Nous présenterons ici nos méthodes ainsi qu'une analyse des performances en pratique.

Navigation du projet

Tout d'abord, merci pour votre temps! Le code est disponible dans le module `projet`.

```
projet  # Racine du projet.
|-- algos  # Nos algos différents.
|-- bench  # Module de benchmark, utilisé par nos tests.
|-- data   # Données.
|  |-- exemples  # Données exemples du sujet.
|  +-- test  # Nos propres données test.
```

```
|-- lib  # Utilitaires communs.
|  |-- data_structures  # Structures de données (union find...)
|  +-- geo  # Module géo
|-- main.py
|-- profiler.py  # Profiling du code
+-- test.py  # Tests basiques
```

Contexte

Le problème porte sur la recherche de composantes connexes dans l'espace.

On part d'un ensemble de points $\mathcal{P} \in \mathbb{R}^2$ et d'une distance d. Deux points $a, b \in \mathcal{P}$ sont reliés par une arête si $dist(a, b) \leq d$. Avec ces informations, il faut identifier le nombre de composantes connexes dans l'espace, ainsi que le nombre de points dans chaque composante connexe, et renvoyer un vecteur des tailles, triées dans l'ordre décroissant.

Résultats expérimentaux

Pour résoudre ce problème, nous avons utilisé différentes approches.

Approche canonique

Une première approche est un algorithme brute-force (`algos/brute_force.py`).

Pour identifier les composantes connexes, l'approche la plus simple est de parcourir tous les points. On crée un graphe. Pour chaque point a, on parcourra tous les autres points et on insère une arête si $dist(a, b) \leq d$. On peut au final parcourir le graphe et identifier les composantes connexes.

Cette première approche possède de pauvres performances, avec une complexité dans le meilleur et le pire cas en $O(n^2)$.

Pour les graphes de temps d'exécution qui vont suivre ;

- la couleur rouge représente une répartition de points très proche les uns des autres (une seule composante connexe), avec $10^3; 10^4; 7, 6.10^4$ points.

- la couleur bleu représente une répartition de points uniforme sur tout l'espace, avec $10^3; 10^4; 4.10^4; 7, 6.10^4$ points.

On remarque (figure 1) que l'algoritme devient quasi innutilisable dès un nombre de points $\geq 10^4$ points.

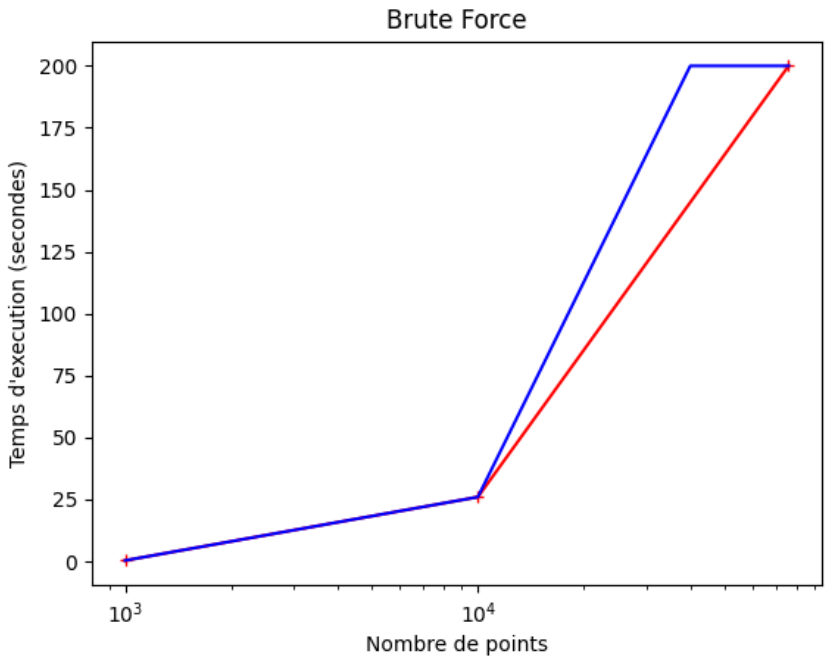

Figure 1: Temps d'execution - Brute-force | Les temps d'executions à 200s sont en réalité des temps trop longs pour être calculés (timeout)

Premières optimisations

Parcours ordonné sur les points

Si deux points a et b sont reliés par une arête, il suffit de vérifier soit que a et b sont reliés, soit que b et a sont reliés. On peut alors économiser la moitié des parcours en ne regardant que dans un sens. Dans ce cas, on peut écrire l'ensemble des arêtes $\mathcal{V}$ comme:

$$\mathcal{V} = \{p_i, p_j \in \mathcal{P} \mid i < j \wedge dist(p_i, p_j) \leq d\}$$

Structure de données pour le graphe

Notre première approche pour stocker les voisins a été un tableau de voisin. Cependant, la structure était lourde pour notre problématique: il n'y a pas besoin d'avoir la conaissance du graphe entier, seulement des composantes connexes.

Nous nous sommmes alors orientés sur un Union-Find, où chaque classe d'équivalence représente un point. Quand deux points sont reliés, on unit les deux classes. Après avoir parcouru tous les points, nous faisons un parcours de l'union-find en comptant le nombre de points dans chaque classe.

Approche par quadtree

Dans l'algorithme canonique, pour savoir si un point est dans dans l'espace, il faut regarder par rapport a tous les autres points. On se retrouvera au final a regarder partout, même si, par exemple, la plupart des points ne sont pas proches du tout. Si il était possible de pouvoir grouper les points proches ensemble d'une manière peu coûteuse, on pourrait gagner en efficacité.

Une manière est d'utiliser un arbre pour partionner l'espace. De nombreux arbres existent: nous avons exploré de nombreux arbres et retenus le quadtree, le VPTree, et le MTree. Nos implémentations du MTree et du VPTree sont dans `etc/VPTree` et `etc/MTree`. Au final, nous avons retenu le quadtree pour deux raisons:

- la performance du vp-tree était trop pauvre dans notre cas.
- le MTree était difficile à implémenter. Nous l'avons gardée en solution de dernier recours, mais nous avons ensuite trouvé de meilleures méthodes.

Vous trouverez notre implémentation du quadtree dans `algos/quadtree.py`.

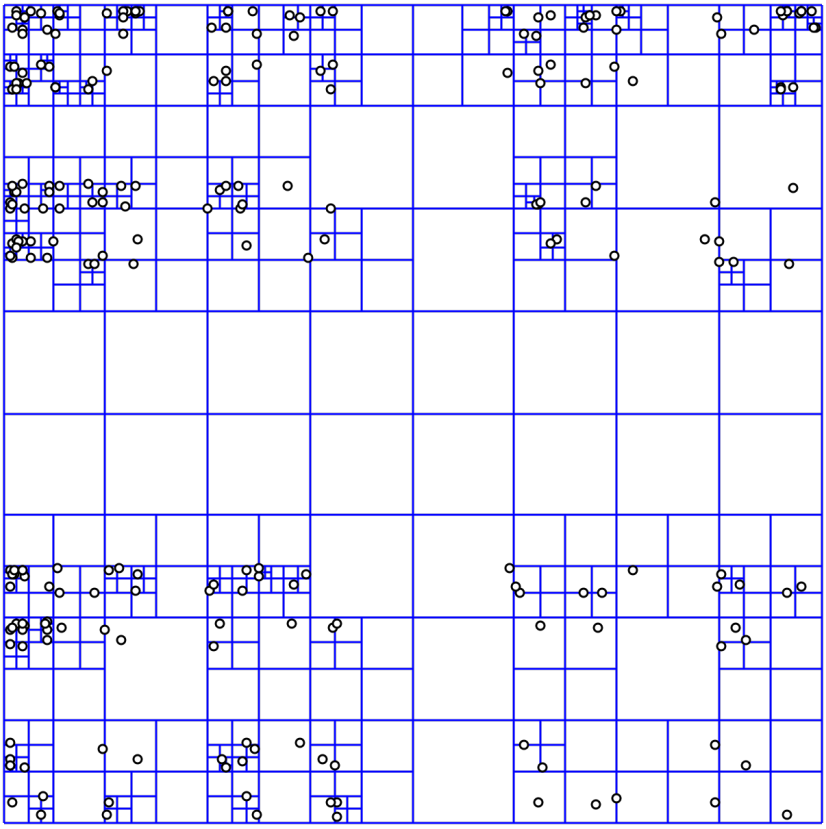

Figure 2: Visualisation d'un quadtree

L'insertion dans un quadtree se fait en temps $O(n\,log(n))$. Nous obtenons un partionnement rapide de l'espace. Il est ensuite utile pour accélérer la recherche.

Après création du quadtree, pour chaque point, nous recherchons ses "voisins", c'est-à-dire les points auquel il est relié. Pour ce faire, on query le quadtree avec le carré circonscrit au cercle de diamètre d du point. Tous les quadtree qui ne sont pas en intersection avec ce carré, et donc tous les points dans ces quadtree, peuvent être éliminés.

Pour les points retenus, on doit finalement tester un à un la distance pour voir si le point est dedans ou en dehors du cercle, de la même manière que la méthode canonique.

Le défaut principal de cette algorithme est sa complexité en pire cas. En effet, si la distribution des points est plutôt dense autour d'un point (tous les points sont proches) alors la complexité devient $O(n^2)$ puisque le quadtree permet de "renvoyer" tous les points voisins potentiellements proches (il faut donc vérifier si ils sont proches ou non.). (Figure 3)

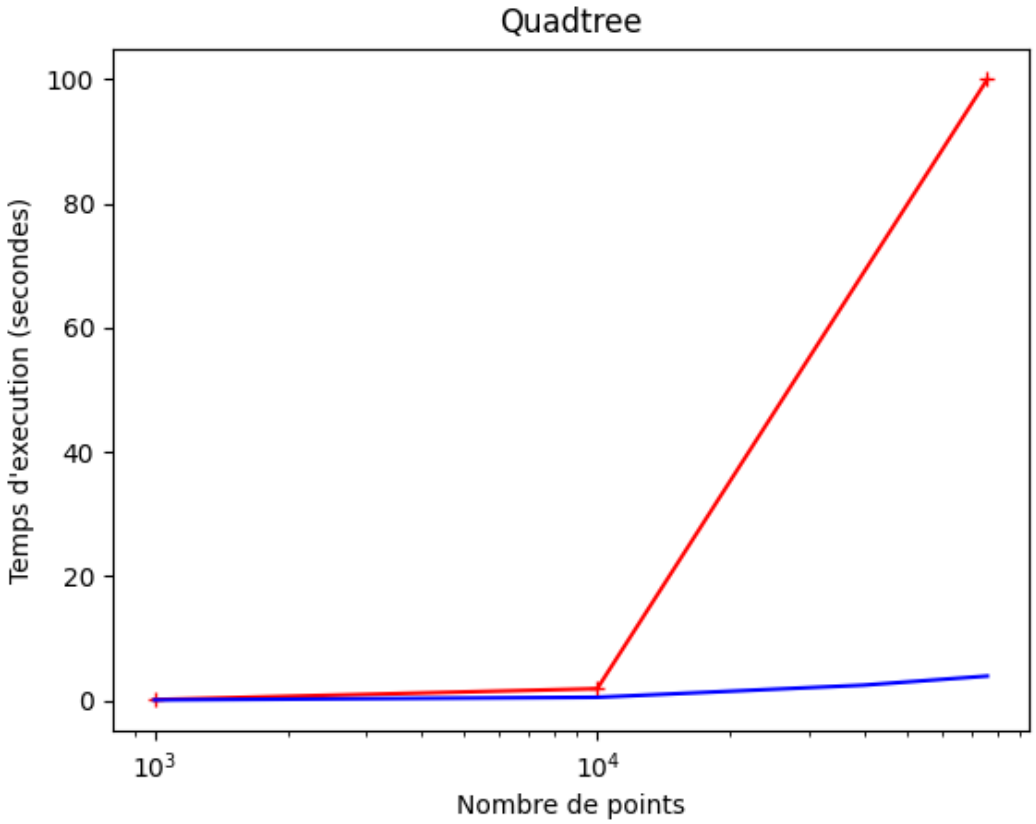

Figure 3: Temps d'execution - QuadTree

Diviser pour régner

L'utilisation d'arbre n'étant pas sufisante, il a fallu trouver une autre méthode pour passer les tests 2 et 3. On a donc cherché à utiliser un algorithme "diviser pour régner". Vous trouverez cet algo dans `algos/divise_regne.py`.

L'idée est d'utiliser un quadrillage de taille $\frac{d}{\sqrt{2}} \times \frac{d}{\sqrt{2}}$. Cela permet d'assurer que tous les points dans une même case sont à une distance $dist(a, b) \leq d$. (Figure 4 : tous les points dans la case sont à une distance plus petite que d du point B qui est un cas limite)

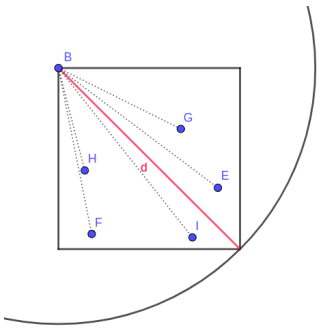

Figure 4: Visualisation d'une case

Il ne reste plus qu'à régner, ce qui consiste ici à regarder cases voisines pour former des composantes connexes multi-cases.

On regarde le point C appartenant à la case orange (Figure 5). En considérant que l'on règne en parcourant chaque case orange de gauche à droite, puis de haut en bas, on peut alors réduire le nombres de cases voisines à regarder à celles en vert, et les demi-cases en bleu.

La méthode que l'on a utilisé est en fait de regarder uniquement les cases en vert, et de former des comportantes connexes entre les cases vertes et la case orange.

On réitère ensuite le processus entier (diviser et régner) avec un cadrillage décallé en diagonale de $d/2$. Ceci permet de de parcourir les cases bleus.

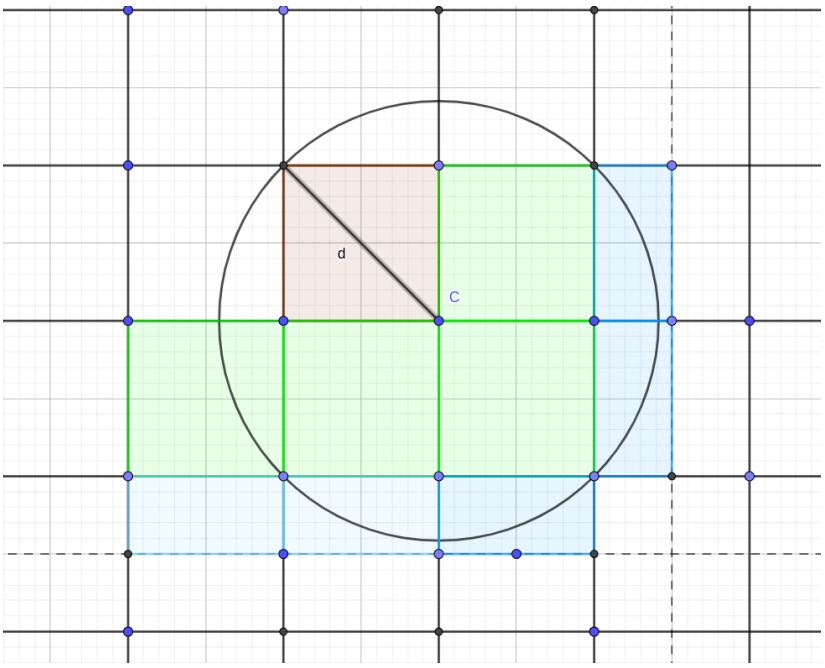

Figure 5: Illustration des cases voisines à examiner

Cette méthode nous a alors permis de passer tous les tests.

La complexité de cet algorithme dépend fortement de la distance d et de la répartition des points, et du nombre de points n.

En effet, plus les points sont proches, plus on va retrouver de points dans une même "case" du cadrillage, donc plus l'étape "diviser" sera efficace. Et alors le cout sera proche de $O(log(n))$ ($\approx$ coût de l'union).

Cependant, on peut creer un exemple particulier tel que, connaissant l'ordre de parcours des points de notre algorithme, on puisse faire $(\frac{n}{2})^2$ calculs de distance. Donc le coût en pire cas est $O(n^2)$.

Mais, dans des cas aléatoires (au sens pas crées spécifiquement pour contrer cet algorithme), alors l'algorithme est suffisement performant pour passer tous les tests.

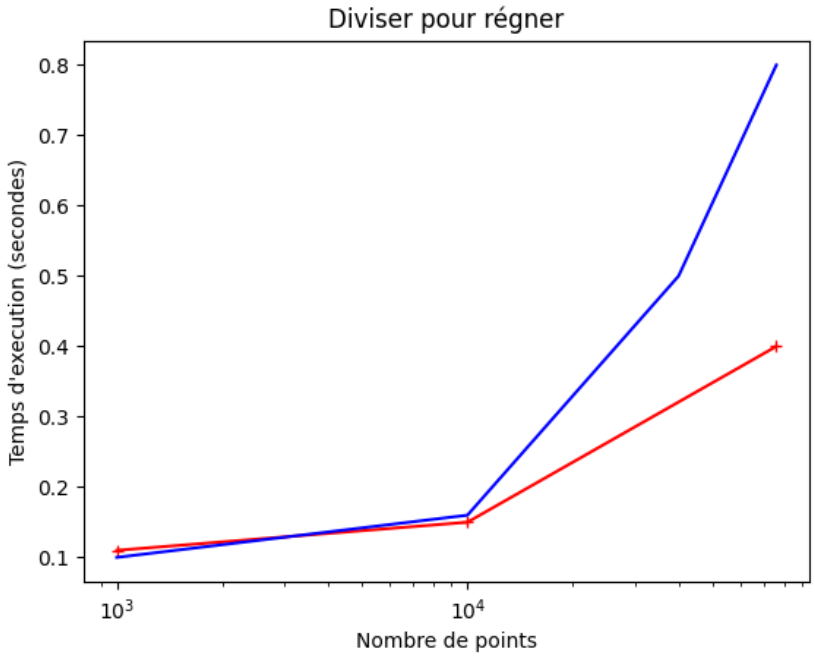

Figure 6: Temps d'exécution - Diviser pour régner

Remarque : Plus la distance d est faible, plus le coût pour créer les cases est élevé. Il faudrait en toute rigueur limiter à un d_{min} notre algorithme, mais, on sait que les tests 0 à 3 ont tous une limite d_{min} assez grande pour qu'on puisse décider de ne pas implémenter algorithmiquement une limite, ce qu'on a vérifié expérimentalement.

Améliorations possibles

Dans notre méthode actuelle, il faut qu'on crée deux grilles, dont une décalée de $d/2$. On pourrait parer ce problème en utilisant une grille de carrés deux fois plus petits.

Nous avons exploré la possibilité de fusionner notre méthode diviser pour régner avec des quadtree. En pratique, le coût de création du quadtree était trop important pour des petits montants de points.

Plusieurs de nos recherches n'ont pas abouti à un résultat satisfaisant. Vous pouvez les trouver dans les autres fichiers du dossier `algo/`.

Conclusion

Ce projet aura été l'occasion pour nous de nous confronter à 3 techniques totalement différentes sur la manipulation et structuration de données, mais également l'occasion d'apprendre à traduire du pseudo-code provenant de papier de recherche en code python. Nous avons pu mettre en pratique les compétences apprises lors du cours d'algorithmie.

Merci pour votre temps!