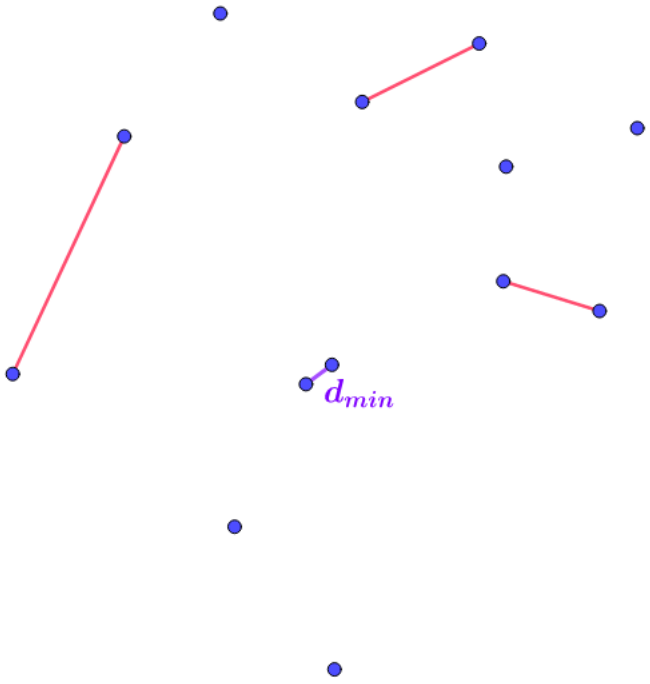
Algorithmique et structures de données

Rapport de projet



Bourseau Julien

Avril 2022

Table des matières

1 Introduction

1.1 Mise en contexte

On considère un nuage de points S dans un plan, il s'agit de trouver le couple de points de S tel que la distance entre ses points soit minimale. Ce problème fondateur en géométrie algorithmique possède de nombreuses applications, comme la prévention de collision entre deux objets proches (espace aériens ou maritimes).
Les nuages de points étant de grandes tailles pour ces applications, il est nécessaire de trouver un algorithme de complexité satisfaisante.

1.2 Objectif du rapport

Nous allons détailler les différents algorithmes construits lors du projet. Plus précisément, leurs particularités de conception, leur performance sur différents nuages de points ainsi que leur complexité algorithmique seront présentées.

2 Etude des algorithmes

2.1 Algorithme dit naïf

2.1.1 Idée directrice

La première intuition consiste à étudier toute les combinaisons possibles de points.
Pour chaque paire de points, on calcule la distance et on regarde si celle-ci est inférieure au minimum précédemment calculé. Si oui, on actualise ce minimum par cette distance.

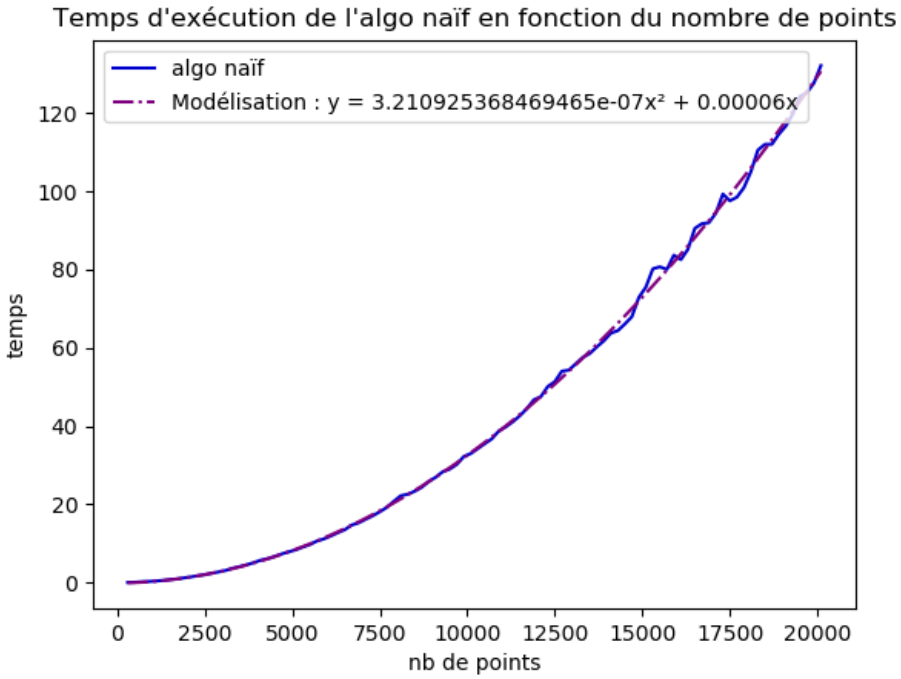
2.1.2 Complexité

Il est évident que cet algorithme n'est pas performant. S'il y a n points, alors il y aura n^2 combinaisons possibles à étudier. Plus formellement :
L'initialisation des variables ainsi que la mise en forme du résultat à renvoyer est en $O(1)$.
On fait n^2 tours de boucles, chaque tour étant en $O(1)$ (la fonction *distance.to* est en $O(1)$).
La complexité de l'algorithme naïf est donc bien en $O(n^2)$. L'utilisation d'un

tel algorithme semble alors extrêmement coûteuse pour un nombre de points conséquents.

2.1.3 Performance

On trace la performance de l'algorithme naïf pour des nuages de points de plus en plus grands, ainsi qu'une modélisation de degré 2.



La courbe nous confirme bien une complexité en $O(2)$. Ainsi, pour un nuage de 20000 points à peine, il faut à l'algorithme plus de deux minutes pour s'exécuter, ce qui est très peu efficient. De plus, il ne semble pas y avoir d'amélioration évidente.
Il est donc nécessaire de trouver un algorithme plus performant.

2.2 Diviser pour régner

2.2.1 Implémentation

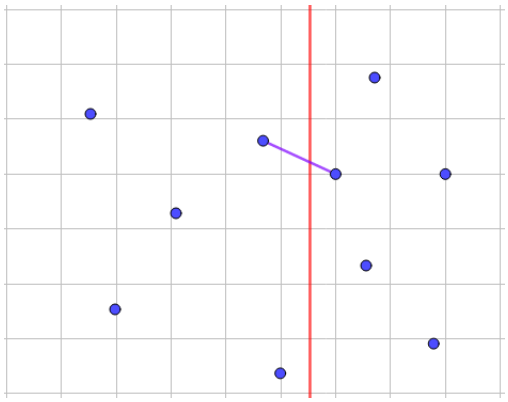
Pour plus d'efficacité, on va découper le problème initial en plusieurs sous-problèmes, résoudre ces derniers et combiner les solutions des sous-problèmes pour trouver la solution finale.

0. Initialisation De notre tableau de points initial, on crée deux tableaux tab_x et tab_y triés par abscisses et ordonnées croissantes.

1. Diviser Si tab_x et tab_y contiennent au moins 4 points (deux paires de points), on coupe le nuage de points en deux parties égales, et on retrie les 4 tableaux alors obtenus. On coupe ainsi récursivement les tab_x et tab_y jusqu'à la condition d'arrêt $n <= 3$.

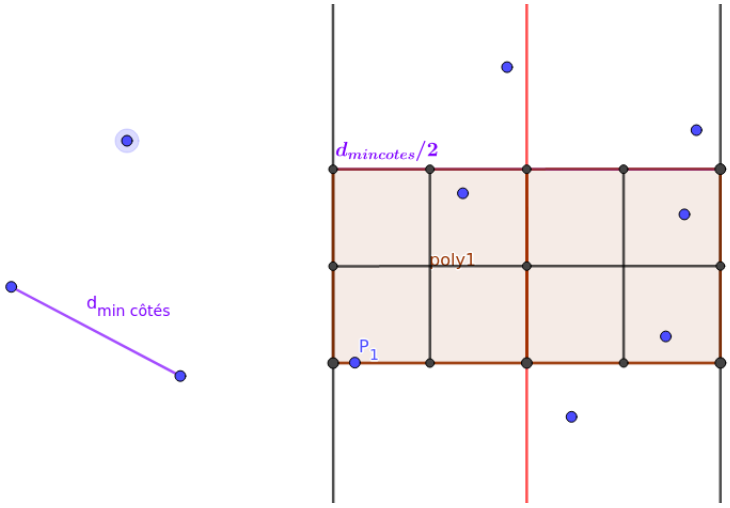
2. Régner Lorsque $n <= 3$, on résout le problème grâce à l'algorithme naïf, ce qui est peu coûteux car le nombre de points est alors faible.

3. Combiner On compare ensuite toute les distances trouvées pour ne garder que la minimale, que l'on notera d_{mincts}. Il ne faut pas oublier les couples de points étant dans deux tableaux distincts mais proche de la frontière, comme le montre l'exemple ci-dessous.



Comme on a déjà un candidat possible pour la distance minimale, $d_{min\ cotes}$, il suffit de regarder les couples de points inter-tableaux étant de distance inférieure à $d_{min\ cotes}$. Pour cela, on étudie la bande verticale de largeur

$2 * d_{min\ cotes}$ centrée en la frontière.
On utilise la fonction *dans bande* qui va retourner les points dans cette bande, puis on calcule la plus petite distance entre ces points grâce à l'algorithme naïf, qu'on notera $d_{minbande}$.
La solution finale est alors $min(d_{min\ cotes}, d_{min\ bande}$.
On peut cependant améliorer l'étape de recherche de minimum dans la bande.
Tout d'abord, il ne faut pas considérer chaque paire deux fois.
De plus, on sait que la distance minimale entre deux points du même côté de la frontière est $d_{min\ cotes}$, donc pour tout point P_1, s'il existe un point P_2 à une distance inférieure à $d_{min\ cotes}$ de P_1, alors il est dans le rectangle passant par P_1 ci-contre :



Le rectangle est divisé en 8 carrés de côté $d_{min\ cotes}/2$. On ne peut pas avoir plus d'un points par carré, car sinon la distance entre ces voisins seraient inférieures à $d_{min\ cotes}$, ce qui est absurde. On peut donc se contenter d'étudier les 7 points qui suivent P_1.

2.2.2 Calcul de complexité

L'étape d'initialisation est un tri par la fonction *sort*, qui a une complexité en moyenne de $O(nlog(n))$.
Pour l'étape diviser, on parcourt la moitié de tab_x pour faire les deux nouveaux tableaux, donc en $O(n/2)$, et on divise tab_y en deux en $O(n/2)$ aussi. Donc cette étape se faire en $O(n)$.

L'étape pour régner est en $O(1)$ car il n'y a que 3 points.
Il y a pour l'étape de combinaison au plus 7 calculs de distance pour chaque point, donc au plus 7n calculs de distance. Cette étape se fait en $O(n)$ opérations.

On divise un problème de taille n en 2 sous-problèmes de taille $n/2$, et le coût de gestion en dehors des appels récursifs est, comme on vient de le voir, en $O(n)$.
On a donc la formule de récurrence, en notant C la complexité :

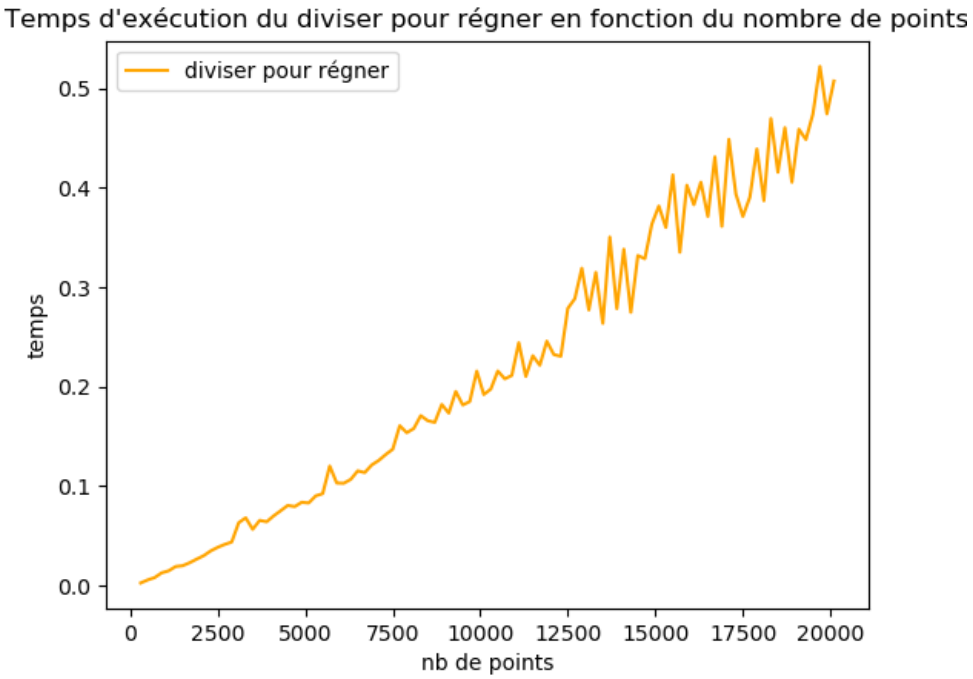
$$C(n) = 2 * C(n/2) + O(n)$$

Donc d'après le master théorème, la complexité de l'algorithme est $O(nlog(n))$.
Moins formellement, cela revient à dire qu'il y a $O(n)$ opérations par appels récursifs, et de l'ordre de $O(log(n))$ appels récursifs.

2.2.3 Performance

On se doute que cet algorithme est bien plus performant que l'algorithme naïf au vu de sa complexité :

La courbe croît légèrement plus vite que la fonction linéaire, on a donc bien le comportement voulu. Le programme s'exécute en moins d'une demi seconde, alors que l'algorithme dit naïf mettait 20 secondes.



3 Conclusion

Deux algorithmes ont été étudiés dans ce rapport répondant tout deux au problème donné, et bien que le premier semble inutile au vu de sa lenteur, il permet néanmoins de vérifier les résultats du second.
L'algorithme naïf ne peut passer tous les jeux de test au vu de sa complexité.
L'algorithme diviser pour régner passe les tests 0 et 1, il pourrait, en terme de performance, passer le test 2 sans timeout, mais il semblerait qu'une exception n'est pas prise en charge, rendant le résultat faux.