Rapport du Projet Algo

Rapport du projet d'algorithmique et structures de données

Professeurs référents : M.Moritz MUHLENTHALER
M.Frederic WAGNER

Ensimag

Recherche des deux points les plus rapprochés

Maxime LI

Table des matières

1 Introduction

Lors de ce projet nous nous intéresserons à résoudre le problème de la Recherche des deux points les plus rapprochés.

Problématique On considère un ensemble S de n points dans le plan. On cherche un couple de points distincts de S de distance minimale. La sortie est donc un tel couple de points ; s'il y en a plusieurs il suffit d'afficher l'un d'entre eux.

Objectif On essayera de passer les trois tests présenter et pour ce faire réduire le plus possible le temps que prend l'algorithme pour trouver la paire de points la plus proche dans un ensemble de 1 million de points.

Présentation Dans la suite je vais vous présenter plusieurs algorithmes que j'ai implémenté pour me rapprocher et dépasser les limites de temps des tests. Mais aussi leurs fonctionnements et leurs complexités temporelles. Les complexités spatiales seront omises car n'influencent pas les résultats des tests.

Graphe Tous les graphes de performances ont été réalisés sur des ordinateurs de l'école. La liste du nombres de points utilisés est [20, 100, 1000, 10000, 20000, 40000, 60000, 80000, 10000] pour les échelles standards et [2^i for i in range(4,22)] pour les échelles logarithmiques.
4 tests différents sont réalisés pour chaque nombres de points et la moyenne est affichée.

2 Algorithme Paire Rapide

2.1 Fonctionnement

Principe On part de l'hypothèse qu'il est probable que si deux points sont très proches, alors leurs projections sur x ou y sont elles aussi très proches.

On va donc parcourir deux listes triés suivant x et y pour au maximum n itération. Lors de la i^{eme} itération on va coupler chaque points avec leur i^{eme} plus proche voisin suivant dans chacune des deux listes trié et tester leurs distances.

Par exemple X=[a,b,c,d] La liste trié des projections sur x
Lors de la 1^{re} itération on va tester (a,b), (b,c), (c,d)
Lors de la 2^{nd} itérations on va tester (a,c) et (b,d)

Condition d'arrêt À chaque tour on va aussi calculer Δx^i_{min} qui correspond à la distance minimale entre la projection sur x pour une paire testé durant l'itération. On fait de même pour Δy^i_{min}

La distance minimale(noté d_{min}) obtenue à la i^{me} itération est logiquement minoré par

$$d_{min} \leq \sqrt{(\Delta x^i_{min})^2 + (\Delta y^i_{min})^2} = D^i$$

De plus, toutes les distances calculées par la suite seront inférieures à cette valeur. Donc dès que notre d_{min} est inférieur à D^i on est sure d'avoir obtenue la bonne valeur de d_{min}.

2.2 Pseudo-code

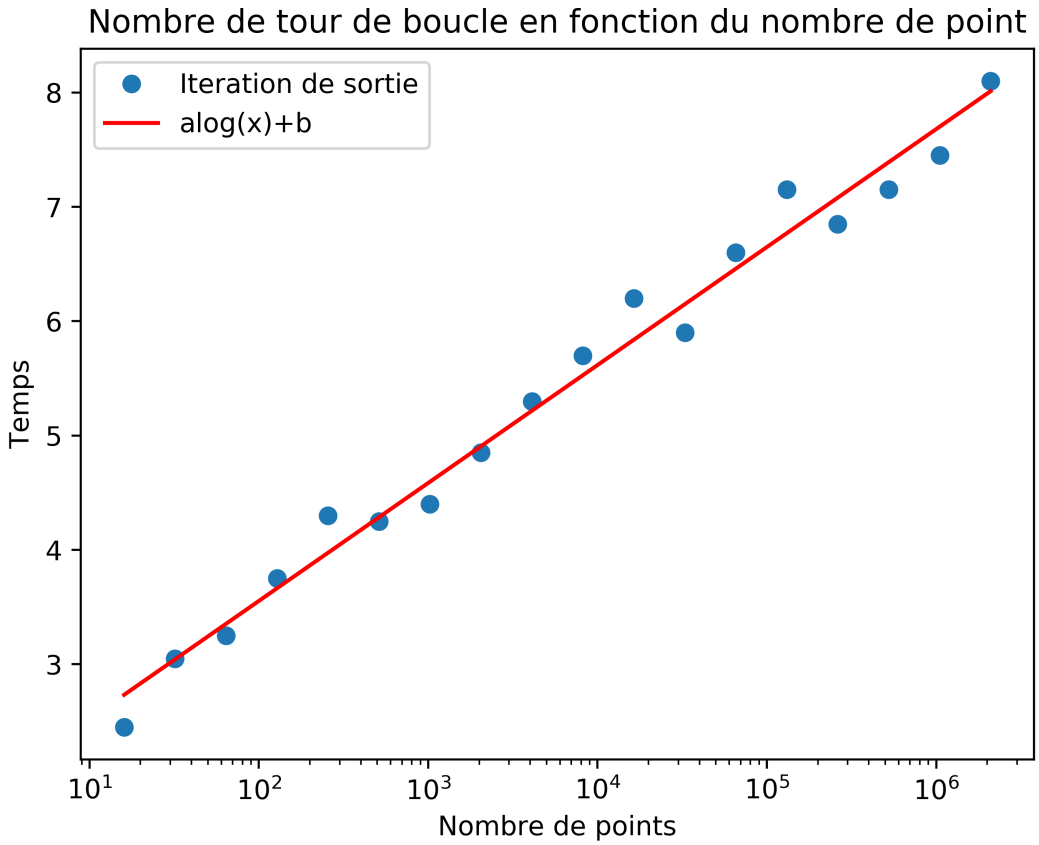
Algorithm 1 Rapide Paire

$S = \{P_1, P_2 ... P_n\}$
X = liste trié par x de P
Y = liste trié par y de P
$d_{min} = \infty$
for i in range(1,n) **do**
  $\Delta x_{min}, \Delta y_{min} = \infty$
  for j in range(1,n-i) **do**
    $\Delta X = X[j+i].x - X[j].x$
    $\Delta Y = X[j+i].y - X[j].y$
    $\Delta x_{min} = min(\Delta x_{min}, \Delta X)$
    On sauvegarde la distance et le couple de point si la distance $< d_{min}$
    $\Delta X = Y[j+i].x - Y[j].x$
    $\Delta Y = Y[j+i].y - Y[j].y$
    $\Delta y_{min} = min(\Delta y_{min}, \Delta Y)$
    On sauvegarde la distance et le couple de point si la distance $< d_{min}$
  end for
  if $d_{min} < \sqrt{(\Delta x^i_{min})^2 + (\Delta y^i_{min})^2}$ **then**
    return le couple sauvegardé
  end if
end for

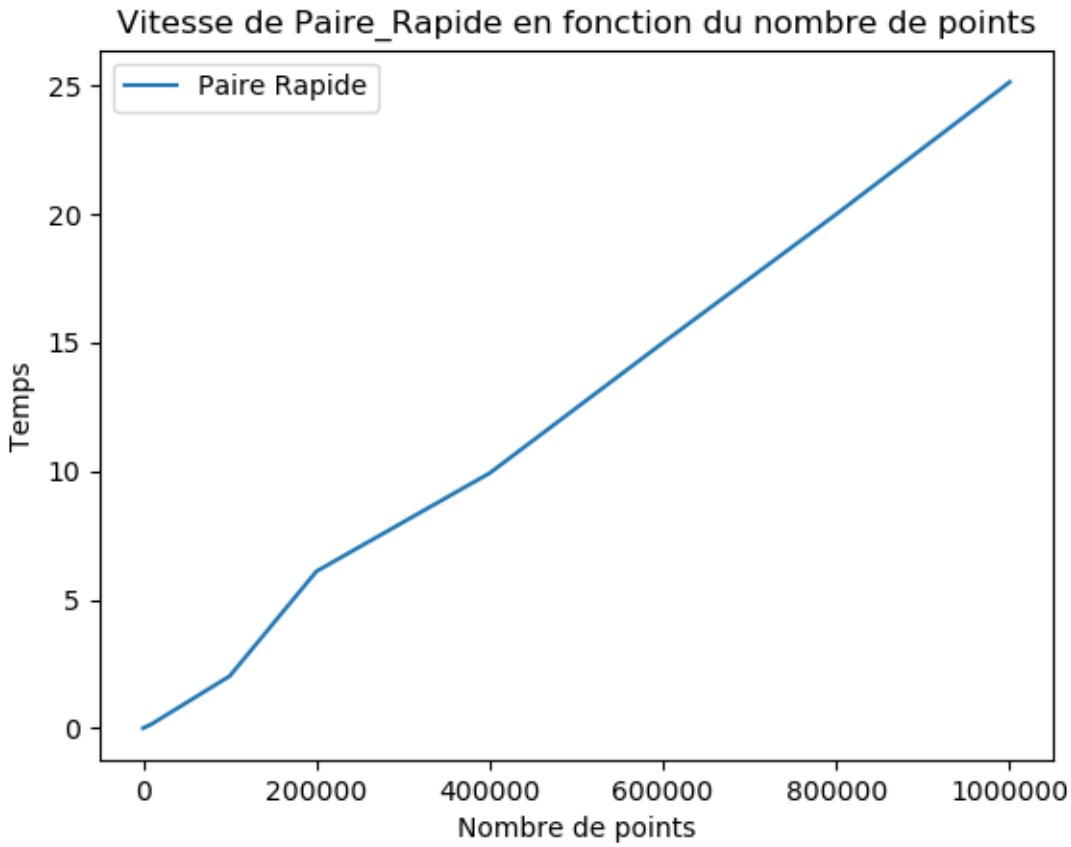
2.3 Performance

Complexité pire cas Dans le pire cas, l'algorithme parcours les n itérations et la complexité est donc en O(n^2).

Complexité en pratique En pratique l'algorithme ne parcourt jamais toutes les itérations. Voir le nombre d'itération est extrêmement faible comparer au nombre de points.



Sur le graphique on peut voir que le nombre d'itération semble suivre avoir une complexité logarithmique avec un coefficient directeur assez faible.
Lors d'une itération on calcule la distance de n-i couple donc la complexité par itération est en O(n)

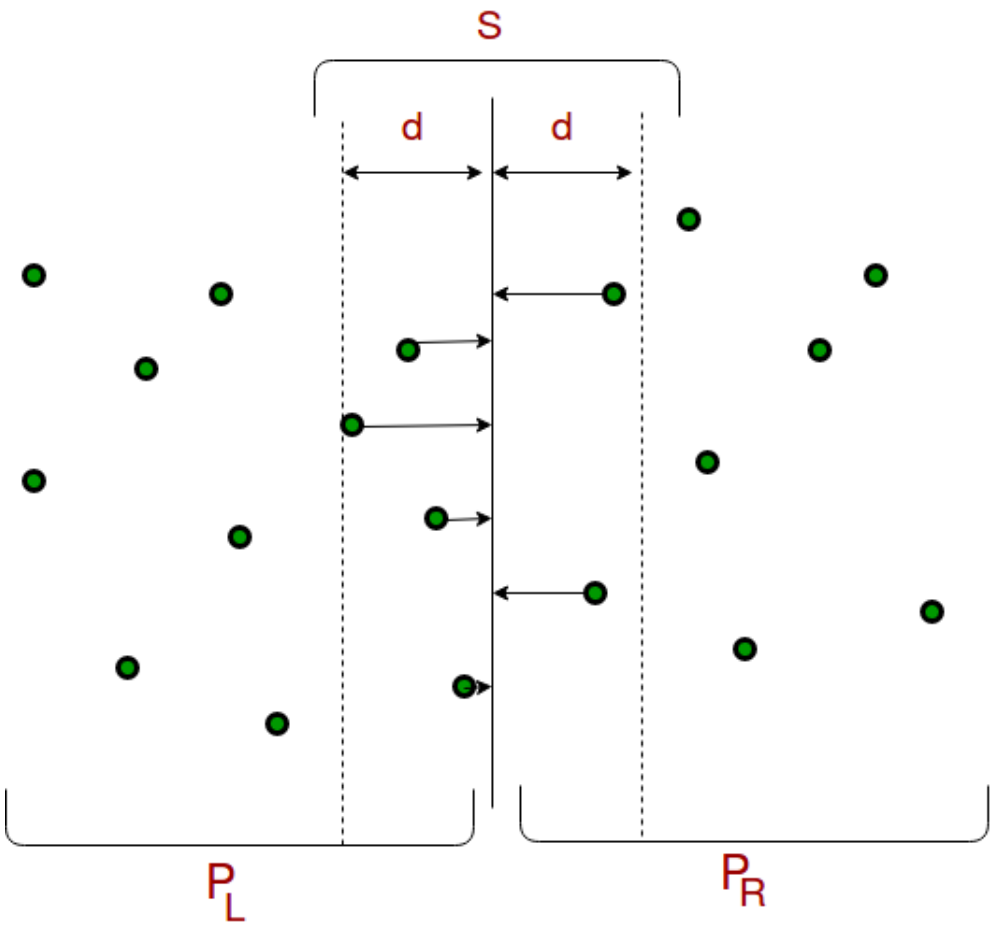


On se retrouve enfin avec une complexité en O(nln(n))

3 Diviser pour régner

3.1 Fonctionnement

Principe L'algorithme repose sur un principe de diviser pour régner sans réelle différence de la version présenté en cours.



On trie les points suivants leurs projections sur l'axe x et on lance l'algorithme récursif avec la liste triée en entrée. Si la liste contient moins de 3 éléments, on teste toute les paires et on renvoie la paire la plus proche.
Sinon l'algorithme va séparer la liste en deux partie égale et se lancer récursivement sur les deux demi-listes P_L et P_R, comparer les deux paires obtenue et choisir la paire la plus proche. Il ne faut pas oublier de vérifier les points à une distance de d_{min} de la frontière.

3.2 Pseudo-code

Algorithm 2 Diviser pour régner DPR

```
S = {P_1, P_2...P_n}
X = liste trié par x de P
function algo_dpr(Liste)
    if len(Liste)≤3 then
        return algo_bruteforce(Liste)
    end if
    couple1 = algo_dpr(Liste[ :len(Liste)//2])
    couple2 = algo_dpr(Liste[len(Liste)//2 +1 :])
    couple3 = algo_bruteforce (Points au centre)
    Mettre le couple le plus rapproché entre les couples 1, 2, 3 dans couple
    return couple
end function
couple = algo_dpr(X)
```

3.3 Performance

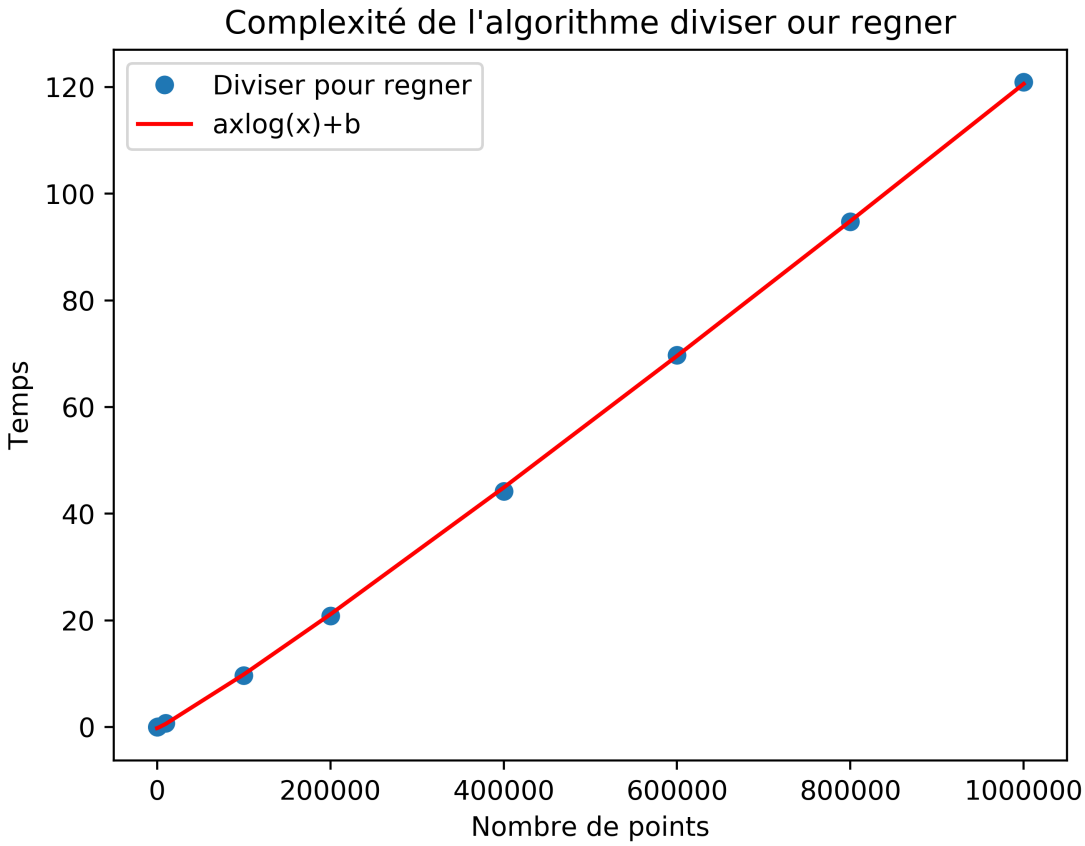
Master theorem

$$C(n) = aC\left(\frac{n}{b}\right) + f(n)$$

Dans notre cas le problème est divisé en 2 sous problèmes donc a=2
La taille de chaque sous-problème est de $\frac{n}{2}$ donc b=2
Dans le pire des cas les n points se trouve au centre l'$algo_b ruteforce$(points au centre) est en O(n). La fusion et la division de la liste sont en O(1). Donc f(n) est en O(n)

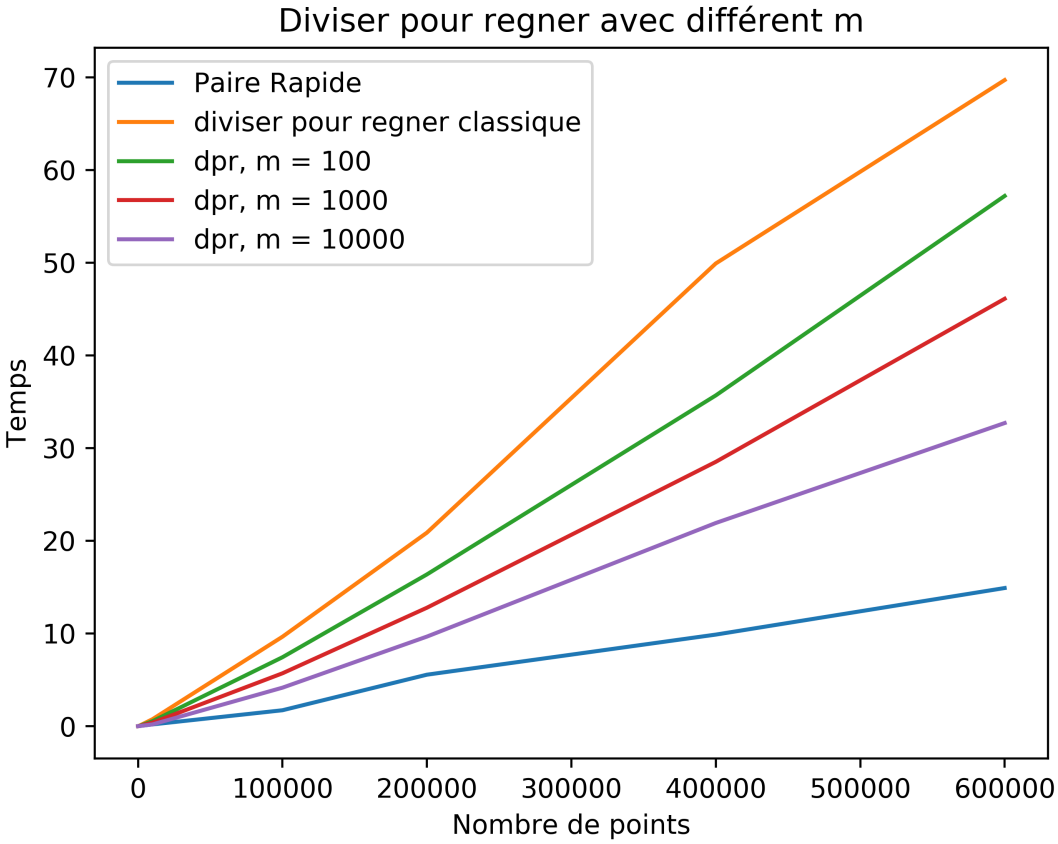
$$C(n) = 2C\left(\frac{n}{2}\right) + O(n)$$

D'après le master theorem puisque les feuilles et les racines ont le même poids. On a une performance temporelle en Θ(nlog_2(n))
Il ne faut pas oublier de prendre en compte le sort du début. Python utilise un quicksort qui à une complexité en O(n^2) dans le pire des cas, mais en O(nln(n)) expérimentalement. Donc la complexité reste inchangée.



3.4 Amélioration avec Rapide Paire

Principe On utilise notre 1er algorithme pour remplacer les utilisations de l'$algo\_brute force$. De plus, on peut changer le seuil limite de points pour la condition d'arrêt.



On peut voir que coupler l'algorithme Rapide Paire avec diviser pour régner permet d'améliorer les performances de diviser pour régner. Plus le m est élevé plus l'algorithme est rapide. Cependant l'algorithme Rapide Paire reste plus rapide que diviser pour régner peut importe l'algorithme utilisé

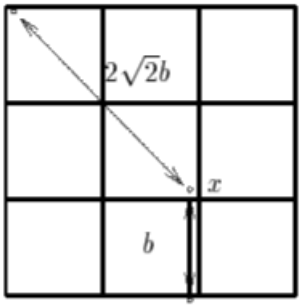
3.5 Conclusion

Certes l'algorithme divise pour regner à une meilleur complexité dans le pire cas, et un complexité en pratique égale à Paire Rapide O(nln(n)). Cependant Rapide Paire est beaucoup plus rapide en pratique.

4 Algorithme grille

4.1 Fonctionnement

On va lancer l'algorithme Paire Rapide sur un nombre de points m pour obtenir une approximation grossière de d_{min} en calculant la distance minimum des m points que l'on note b.
On va créer une grille dont les cases sont des carrés de côté b. Cette grille prendra la forme d'une table de hashage, dont la clé dépend des coordonnées des points sur la grille (En pratique clé = (int(x/b)«20)+int(y/b), puisque 1«20 > 1M, on est sure que deux case ne peuvent pas avoir la même clé.)
Chaque case de la table de hashage contient une liste et on va ajouter tous les points de S dans ces listes en fonction de leur coordonnée.
On va enfin appliqué Paire Rapide à chaque liste contenant au moins un élément concaténé au 8 listes adjacents (en pratique juste celle de droite, de haut et en haut à droite car les autres aurons déjà testé le reste).
Il suffira ensuite de choisir la paire la plus proche issue des algorithmes.



Comme le montre la figure prendre les 8 cases adjacents suffit pour avoir tout les points de distance au moins égale à b>d_{min}

4.2 Pseudo-code

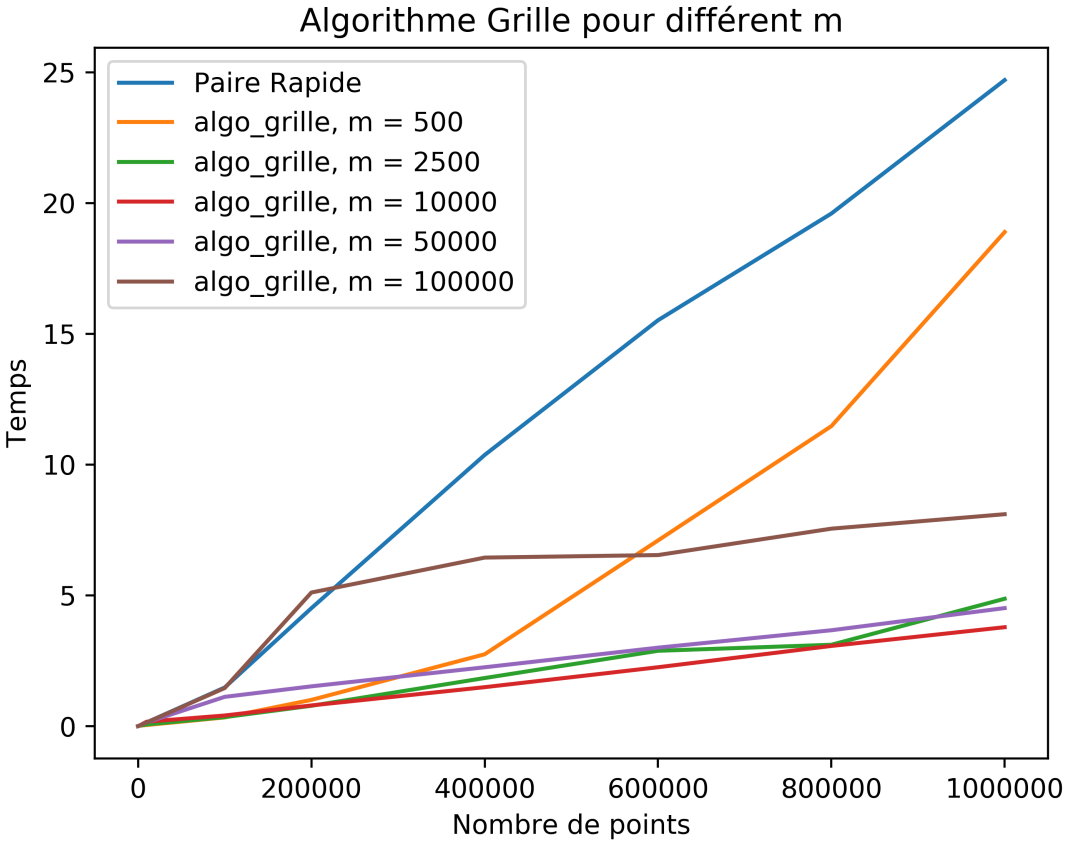
Algorithm 3 Algo Grille

```
S = {P_1, P_2...P_n}
d_min, couple = paire_rapide(S[ :m])
b = d_min
grille = dict()
ajoute(S, grille)                          ▷ On ajoute tout les points de S à grille
for i in grille.keys() do
    groupe = adjacent(i)                   ▷ On créer une liste avec i et ses adjacents
    d, c = paire_rapide(groupe)
    if d_min > d then
        d_min, couple = d, c
    end if
end for
return couple
```

4.3 Performance

Complexité Paire Rapide de m point à une complexité de O(mln(m)).
La complexité de la création de la grille et de l'insertion des points est de O(n) car on insère n points dans une table de hachage (coût d'insertion =O(1))

En partant du principe que l'approximation de d_{min} obtenue à partir de Paire Rapide est assez bonne. On peut majorer le nombre de points dans une cellule à 4(un vers chaque sommets) et donc majorer la complexité de chaque Paire Rapide(groupe) dans la boucle à un Paire Rapide sur un maximum de 16 points(4 cases). Donc chaque itération de la boucle peut être ramenée à O(1) car 16«n.
La boucle de recherche du minimum est donc en O(n)car il y a au maximum n cases remplies.
La complexité totale est donc en O(mln(m)+n)



On peut voir que pour un m trop faible (Exemple m = 500) notre hypothèse d'avoir au maximum 4 points par cases ne tient pas.
Cependant si le m est trop grand la composante O(mln(m)) prend beaucoup de temps.
La valeur optimale de m se rapproche vers m=10000. Pour m«n on passe à une complexité de O(n)

4.4 Conclusion

L'algorithme Grille avec m = 10000 est beaucoup plus efficace que Rapide Paire. Cependant avec la Sandbox, on obtient des temps plus de deux fois plus long.

Cependant le passage d'un algorithme en O(nln(n)) à un algorithme en O(n) améliore grandement les performances. L'algo grille est plus efficace que Paire Rapide.

5 Algorithme Crible

5.1 Fonctionnement

Cet algorithme repose aussi sur une table de hashage qui simule une grille. Cependant cette fois à chaque ajout de point, on va calculer directement le nouveau d_{min}. Et si il a changé on va recréer une grille ayant pour côté le nouveau d_{min}. Ainsi chaque case ne peut contenir qu' un point, et il suffit toujours de regarder les 8 cases adjacents pour calculer le nouveau d_{min}

5.2 Pseudo-code

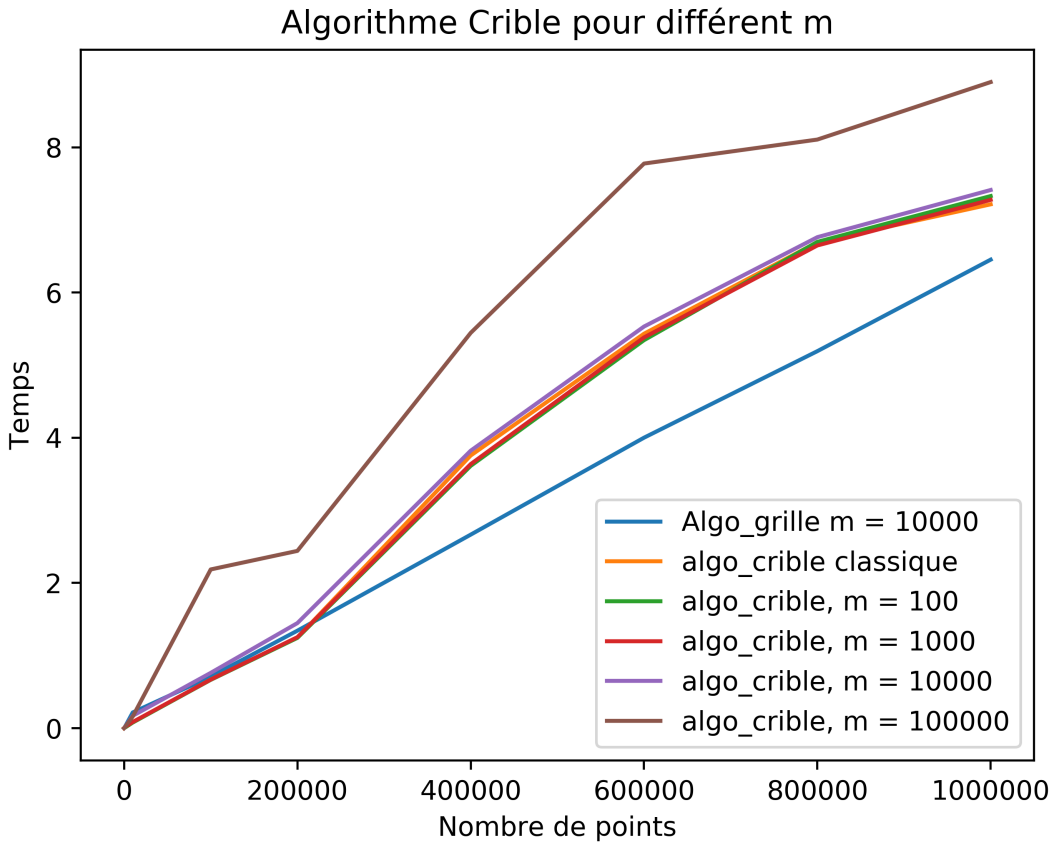
Algorithm 4 Algo Crible

```
S = {P_1, P_2...P_n}
d_min, couple = paire_rapide(S[ :m])
grille = creation_grille(S[ :m], d_min)  ▷ Créer la grille de côté d_min et insère les m 1^er points
for i in range(m,len(S)) do
    d, c = ajoute(i, grille)             ▷ Ajoute i à la grille et retourne la distance à son plus
                                         ▷ proche voisin d et le couple (i,voisin )
    if d_min > d then
        d_min, couple = d, c
        grille = creation_grille(S[ :i], d_min)        ▷ Redimensionnement de la grille
    end if
end for
return couple
```

5.3 Performance

Complexité temporelle Puisqu' on utilise des table de hashage en python, le coût d'ajout et du calcul de la distance avec le plus proche voisin sont en O(1).
Le coût de la fonction *creation_grille* dépend du nombre d'élément à insérer dans la table de hashage et est donc en O(n).
On pose h, le nombre de fois ou on doit redimensionner la table de hashage et on a une complexité temporelle en O(hn).

Amélioration On veut essayer de réduire le h le plus possible, donc on utilise comme pour l'algorithme grille une approximation grossière de d_{min} en calculant la distance minimal de m points grâce à Paire Rapide.
Cependant on se rend compte que réaliser l'algorithme sur les m premiers points est peu coûteux pour m petit car on recréer une table avec seulement m élément et non n et seul les redimensionnements avec un nombre de point élevé est coûteux. C'est pourquoi calculer la distance minimale est peu utile sur un pc disposant d'un stockage rapide.



En pratique La sandbox prend beaucoup plus de temps pour la création et l'utilisation d'une table de hashage (facteur > 3). Donc en pratique il est utile d'utiliser l'algorithme Paire Rapide. (Sans l'approximation de départ l'algorithme ne passe pas le test 3, alors que l'algorithme passe avec m=500).

L'algorithme reste cependant plus lent que Algo Grille donc l'algorithme final choisi est l'algorithme Grille.

6 Autre amélioration

6.1 Utilisation de la distance au carré

Au lieu de calculer la distance euclidienne à chaque fois pour les comparaison. On privilégiera la comparaison de la distance au carré qui permet d'économiser un opérateur racine carré à chaque fois.
On ne calcule la distance que pour avoir la longueur des côtés pour les grilles dans algo grille et algo crible.

6.2 Passage d'un struct à un tuple

Il est avantageux même si cela n'a pas été réalisé de passer d'un struct Point à un tuple avec juste les coordonnées.

6.3 Clé pour la table de hashage

Le hash d'un int est le int en lui même donc avoir un int en clé permet de gagner du temps de calcul du hash. Dans notre cas clé = (int(x/b)«20)+int(y/b). Et puisque 1«20 > 1M, on est sûr que deux cases ne peuvent pas avoir la même clé.

6.4 Utilisation de différent algorithme selon l'entrée

On remarque que les performances des algorithmes utilisant une table hashage dans la Sandbox sont réduite comparé à ceux n'en utilisant pas.

Donc si le nombre de points d'entrée est supérieur à m=10000. On utilise l'algo grille. Cependant si le nombre de points est inférieur à m, on utilisera Paire Rapide.