Rapport Algorithmique et structures de données

Badaoui Abdessamad
Aghelias Nasr Allah

1 Problématique

Étant donné un ensemble fini de points dans un plan S cartésien, on cherche à implémenter des algorithmes efficaces pour trouver un couple de points distincts de S de plus court distance.
Ce type d'algorithme voit son utilité dans les transports aériens ou maritimes par exemple.

Formalisation du problème

Nous représenterons un point en python par une classe *Point*, qui contient toute un ensemble de méthodes utiles qui vont nous faciliter la manipulations des points et surtout la méthode distance_to(self, other) qui permet de calculer la distance entre deux points du plan. Les coordonnées des points peuvent être accédés grâce à l'attribue du point self.coordinates. L'ensemble des points considérés sera quant à lui stocké dans différentes fichiers.pts qui feront l'objet de tests de nos algorithmes.

2 Une première solution (algorithme naïf)

2.1 Description de l'algorithme

Cet algorithme a pour vocation de fournir un résultat de base au problème. Voici l'algorithme naïf calculant cette plus petite distance en considérant toutes les paires de points :

- **Entrée :** Tableau de points
- **Sortie :** Tuple donnant les deux points les plus proches et la distance min

```
1 Plus_proche_v1(Tab)
2         n := len(Tab)
3          d := Tab[0].disatnce_to(Tab[1])
4          idx1 := 0
5          idx2 := 1
6          for i := 0 to n - 1
7                 for j := 0 to n - 1
8                   if  Tab[i].disatnce_to(Tab[j]) < d:
9                                  d := Tab[i].disatnce_to(Tab[j])
10                                 idx1 := i
11                                 idx2 := j
12         return (Tab[idx1],Tab[idx2],d)
```

2.2 Évaluation de la complexité

On remarque que les lignes 9,10 et 11 sont exécutées n(n-1) fois avec n est le nombre de points. Par conséquent, on conclut sans peine que la complexité de notre première algorithme est quadratique $O(n^2)$.

On pourra très simplement améliorer un petit peu ce algorithme en divisant par 2 le nombre d'exécutions des lignes 9,10 et 11. En fait, puisque la distance est une application symétrique, il est donc inutile de tester deux fois la même distance.On peut donc se restreindre à l'un des deux cas où i ¡ j (ou l'inverse). L'algorithme dans ce cas devient donc :

```
1  Plus_proche_v1(Tab)
2          n := len(Tab)
3          d := Tab[0].disatnce_to(Tab[1])
4          idx1 := 0
5          idx2 := 1
6          for i := 0 to n - 1
7                  for j := i+1 to n - 1
8                      if  Tab[i].disatnce_to(Tab[j]) < d:
9                                  d := Tab[i].disatnce_to(Tab[j])
10                                 idx1 := i
11                                 idx2 := j
12         return (Tab[idx1],Tab[idx2],d)
```

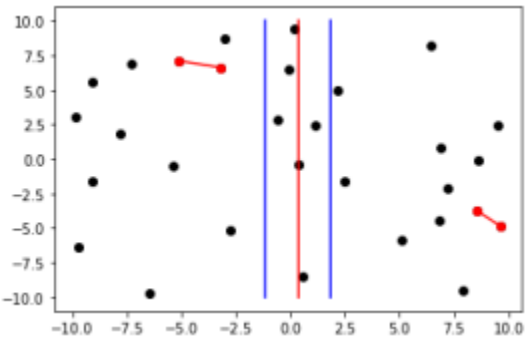
On a donc :les lignes 9,10 et 11 sont exécutées n(n-1)/2 fois .
Une autre piste d'optimisation et de changer la méthode distance_to dans le fichier point.py de telle façon qu'elle renvoie non pas la distance mais la distance au carrée, et comme ça on ne stocke que des distances au carré, et on fait la racine une fois pour toute à la fin.
Malgré ces optimisation la complexité est toujours quadratique.

3 Une deuxième solution (Diviser pour régner)

Soit S un ensemble de n points dans le plan.

3.1 Idée

On divise l'ensemble S verticalement i.e. suivant le critère des abscisses en le partageant en deux sous ensembles de même tailles.



On note S_G et S_D les sous-ensembles représentant les points dans le semi-plan gauche et le semi-plan droit.
On note d_S la distance minimale entre les points d'un ensemble S.
Alors, la distance minimale d_S est soit d_{S_G} soit d_{S_D} soit une distance d_{bande} des points situés entre les deux sous-ensembles S_G et S_D
On obtient d_{S_G} et d_{S_D} par un appel récursif. On note d_{min} le minimum entre d_{S_G} et d_{S_D}.
Il reste maintenant à voir s'il y a une paire de points l'un dans S_G et l'autre dans S_D telle que leur distance est inférieure à d_{min}. On va se focaliser sur la bande du milieu de largeur $2d_{min}$ pour éliminer les combinaisons de points dont la distance est supérieure à d_{min}

3.2 Description de l'algorithme

1ère étape
On représentera S sous forme d'un tableau de points.
On crée deux tableaux triés à partir de S :

- S_x un tableau des points de S triés dans l'ordre croissant des abscisses

- S_y un tableau des points de S triés dans l'ordre croissant des ordonnées

L'algorithme de tri utilisé doit être de complexité $O(n \log_2(n))$

$2^{\text{ème}}$ étape

Si $n \leq 3$, on appelle la fonction naïve et on renvoie le résultat.
Sinon, on partage le tableau S_x en deux tableaux S_{x_G} et S_{x_D} de même taille (à un point près).

$3^{\text{ème}}$ étape

On appelle notre fonction sur S_{x_G} et S_{x_D} pour récupérer d_G et d_D. On note d_{min} le minimum entre eux. Notre fonction renvoie un tuple contenant la distance minimale et les deux points en question. Les deux points sont a priori stockés avant de traiter la bande du milieu.

$4^{\text{ème}}$ étape

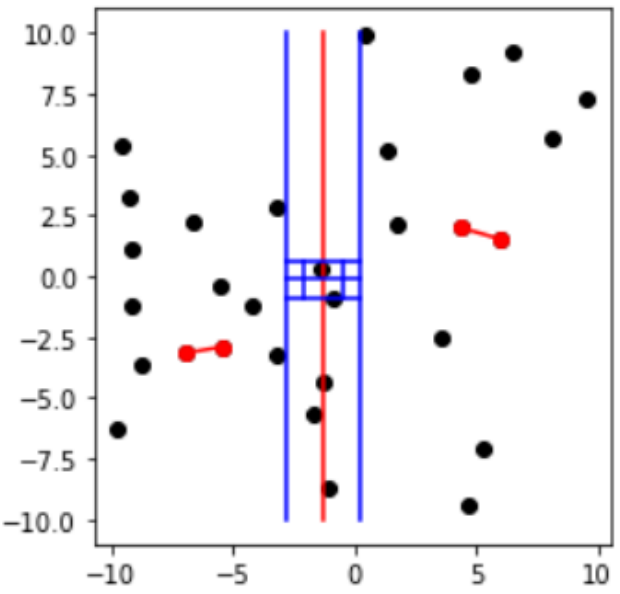
Soit x l'abscisse à l'aide duquel on va construire le rectangle du milieu qu'on va parcourir. x est calculé de la façon suivante.

```
Soit x un flottant
n := len(S)
si n%2 == 1 alors
        x := S[n//2].coordinates[0]
sinon
        x := (S[n//2].coordinates[0] + S[n//2 - 1].coordinates[0])/2
finsi
```

On note *bande* le tableau des points de S dont l'abscisse est dans l'inervalle $]x - d_{min}, x + d_{min}[$, trié dans l'ordre croissant des ordonnées.
Pour chaque point p de *bande*, on calcule la distance entre p et les 7 autres points qui lui sont consécutifs, et on note d'_{min} le minimum de toutes ces distances.
On renvoie à la fin le minimum entre d_{min} et d'_{min} et les deux points qui lui correspondent.

Explication du choix de 7 points

Pour chaque point p de *bande*, on construit le rectangle *cage* donc le côté inférieur passe par p. Il est de longeur $2d_{min}$ et de largeur d_{min}. S'il existe un point de distance inférieure à d_{min}, il sera forcément donc *cage*



On divise *cage* en 8 carré de côté $d_{min}/2$.
Supposons par l'absurde que chaque carré peut contenir plus de deux points.

Alors il existe une paire de points dans un carré dont la distance est inférieure à $d_{min}/2$ (diagonale du carré)

Donc il existe une paire de points dans un semi-plan S_G ou S_D (puique les carrés sont construits de façon à être dans un plan ou bien l'autre) i.e. il existe une paire de points dans l'un des semi-plans dont la distance est inférieure strictement à d_{min} ce qui est absurde.

On conclut donc que chaque carré contient au plus un point. Par conséquent, il suffit de comparer chaque point p de *bande* avec les 7 points qui le suivent pour déterminer d'_{min}

3.3 Complexité

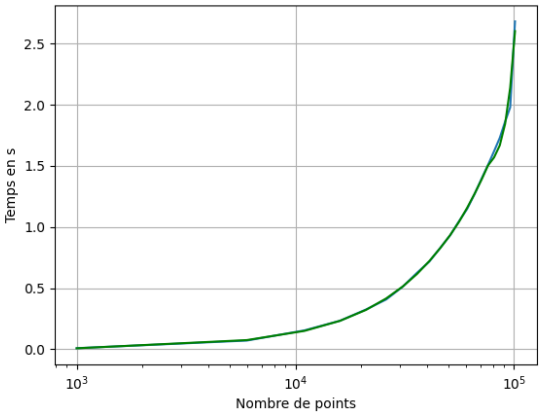
Le tri des tableaux se fait en $O(n \log_2(n))$
Le parcours de bande se fait en $O(n)$
Et lorsque l'on applique le master theorem on obtient un temps en $O(n \log_2(n))$

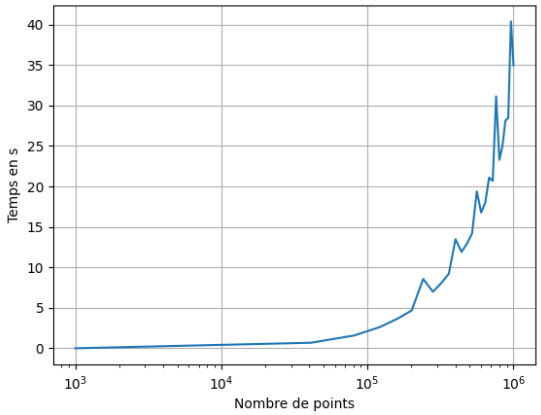
4 Courbes de performance

On trace les courbes de performances (temps de l'exécution du programme en fonction du nombre de points) à l'aide du code suivant :

```
from time import time
import matplotlib.pyplot as plt
import time
import main
from random import uniform
import numpy as np

NOMFICHIER = "exemple_6.pts"

x = [1000]
i = 0
while x[i] <= 100000:
        x.append(x[i]+10000)
        i += 1

def generer(n, nomfichier):
        fichier = open(nomfichier, "w")
        for _ in range(n):
        x = uniform(-10, 10)
        y = uniform(-10, 10)
        print(f"{x}, {y}", file=fichier)
        fichier.close()


def clock(n, nomfichier):
        generer(n, nomfichier)
        points = main.load_instance(nomfichier)
        debut = time.time()
        main.plus_prochesv2(points)
        fin = time.time()
        return fin - debut

y = [clock(x[i], NOMFICHIER) for i in range(len(x))]
y2 = [np.log10(n) for n in x]

plt.plot(x, y2)
plt.xscale("log")
plt.show()
```

On obtient les courbes suivantes :





On constate que l'allure des courbes est parabolique lorsque l'échelle de l'axe des abscisses est logarithmique. Donc, notre programme a en fait une complexité de $O(n \log_2(n))$
On constate que la deuxième courbe a des points aberrant lorsque le nombre de points est compris entre 10^5 et 10^6. Ceci est dû au fait qu'on n'a pas suffisamement calculé la performance du programme sur suffisamment d'échantillons de points de cet intervalle. Comme on doit couvrir $9 \cdot 10^5$ points pour lesquels la performance dure entre 2s et 40s, l'exécution du programme prendra un temps très grand. Néanmoins, on peut discerner l'allure de $n \log_2(n)$ malgré ces points aberrant.

P.S. : Le code ci-joint doit être exécuté avec le nom du fichier contenant les points en ligne de commande. Il renvoie le couple de points les plus proches dans le fichier.