Algorithmique et structures de données
Projet

OUAZZANI CHAHDI Mohammed ZAKI Akram

Avril 2022

1 Introduction:

Le but de ce sujet est de trouver la paire de point de distance minimal dans un ensemble S des points dans un plan. Un algorithme naïf de recherche *Brute-force* pourra nous permettre de résoudre ce problème en complexité $O(n^2)$. Cela ne suffit pas pour traiter plusieurs points. Alors on cherche à résoudre ce problème en une meilleure complexité. Pour cela nous avons implémenté deux algorithmes. Le premier se base sur le paradigme *Diviser pour régner*, et le deuxième est un algorithme randomisé.

2 Présentation des algorithmes

2.1 Algorithme naïf:

2.1.1 Principe de l'algorithme:

L'algorithme naïf s'agit de comparer chaque point avec tous les autres points de S.

En utilisant les modules fournis on a remarqué qu'on fait plusieurs comparaisons entre des points qui sont très éloignés. Alors notre but était de d'éliminer ces comparaisons inutiles.

2.2 Diviser pour régner:

Le deuxième algorithme est basé sur le paradigme : diviser pour régner.

2.2.1 Principe de l'algorithme:

- On commence par trier la liste des points dans l'ordre croissant des abscisses et des ordonnées.
- On divise la liste triée dans l'ordre croissant des abscisses sur deux sous listes de taille $n/2$: gauche et droite.

- On calcul les distances minimales récursivement dans les deux sous listes et on ne garde que le minimum d de ces deux distances. Alors, s'il y a deux points de sorte que la distance entre eux est inférieur à d, on a forcément l'un est dans la première sous liste(gauche) et l'autre est dans la deuxième(droite) et les deux points se trouvent dans la bande verticale qui contient les abscisses comprises entre $x_{mid} - d$ et $x_{mid} + d$.

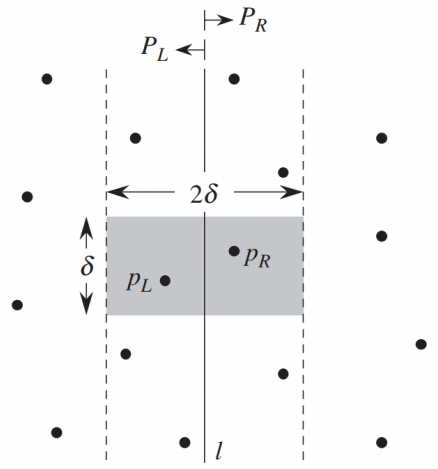

Figure 1: Exemple d'une bande
source: Introduction to Algorithms, Third Edition

- On cherche les deux points les plus proches dans la bande dont la distance est inférieur à d.
- On retourne les deux points.

2.2.2 Problèmes et améliorations:

On a essayé de trier la liste des points dans la fonction récursive qui retourne les points les plus proches, mais on s'est rendu compte que la complexité de la fonction récursive va être $O(nlog(n)^2)$ (car l'équation de récurrence du coût est $C(n) = 2C(\frac{n}{2}) + O(nlog(n))$ et d'après le Master théorème on aura :

$$C(n) = O(nlog(n)^2)$$

Du coup on a passé en paramètre de la fonction récursive la liste triée par rapport aux abscisses et on ne trie que la bande par rapport aux ordonnées.

2.2.3 Analyse de la complexité:

On note $C(n)$ le coût de l'algorithme récursif.
Pour la recherche dans la bande il suffit de considérer les distances d'un point avec les 7 points qui le suivent. Cela nous permet d'avoir Une complexité de recherche linéaire pour la bande.

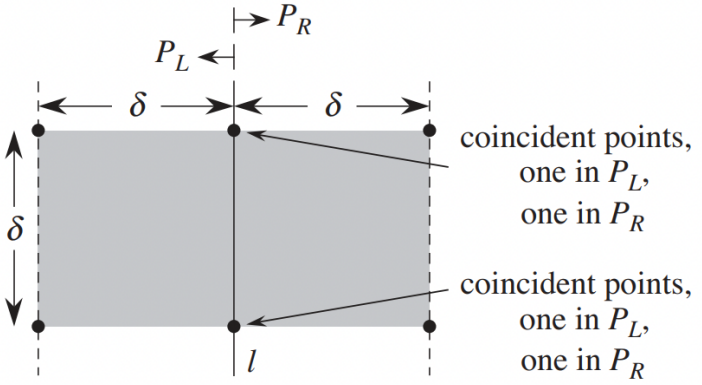

Figure 2: Linéarité de la recherche dans la bande
source : Introduction to Algorithms, Third Edition

Alors, les opérations hors récursion coûtent $O(n)$, les deux appels récursif dans la fonction coûtent $2C(\frac{n}{2})$, on a donc l'équation : $C(n) = 2C(\frac{n}{2}) + O(n)$ (et ceci pour les cas $n > 3$, le cas $n \leq 3$ a un coût de $O(1)$).
Et donc d'après le Master théorème on a :

$$\boxed{C = O(nlog(n))}$$

2.3 Un algorithme randomisé:

Le troisième algorithme qu'on a essayé d'implémenter était un algorithme proposé par *Michael Rabin*. Cet algorithme cherche à résoudre le problème en complexité linéaire $O(n)$

2.3.1 Principe de l'algorithme:

On commence par choisir aléatoirement $\sqrt{n}$ points de l'ensemble des points S et on cherche la distance minimale d par l'algorithme naïf. On divise le plan en une grille des cases de taille $d \times d$. En définissant une case remplit comme une case qui contient au moins un point. On parcourt les cases remplit. Pour chaque point dans une case remplit. On le compare avec ces points voisins. On définit un point voisin comme étant un point dans la même case, ou une case voisine. Le pseudo-code se présente comme suit

$S_{initial} \leftarrow PointsAleatoires(source = S, taille = \sqrt{n})$
$d \leftarrow RechercheBruteforce(S_{initial})$
$CasesHash \leftarrow CreerHashTable(S, d)$ ▷ Création de la grille
$d_{min} \leftarrow 0$
for $Case in CasesHash$ **do**
  if $Case$ is remplit **then**
    $S_{recherche} \leftarrow Case + CasesVoisine(Case)$
    $d \leftarrow RechercheBruteforce(S_{recherche})$
    if $d < d_{min}$ **then**
      $d_{min} \leftarrow d$
    end if
  end if

2.3.2 Analyse de l'algorithme et limitations

Le tri des $\sqrt{n}$ points au début est linéaire de complexité $O(\sqrt{n}^2) = O(n)$

En notant n_i le le nombre de points dans la i-ème case. On trouve que le coût est :

$$C(n) = \sum_i O(n_i)^2$$

Cette somme dépend fortement de d, donc de la distribution des points dans le plan. Dans le cas ou on a une bonne distribution, l'algorithme est très performant.

3 Comparaison des performances des algorithmes

Les courbes suivantes présentent les performances temporelles de nos algorithmes. La première compare l'algorithme naïf qui est en $O(n^2)$ avec l'algorithme de *Diviser pour régner* qui est en $O(nlog(n))$. La deuxième compare ces algorithmes avec l'algorithme randomisé.

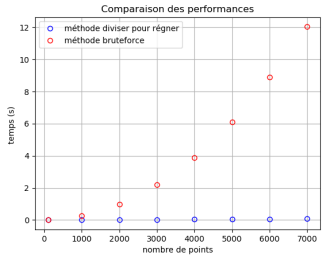

Figure 3: Bruteforce

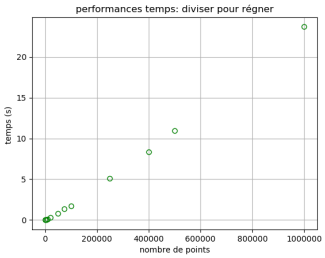

Figure 4: *Diviser pour régner*

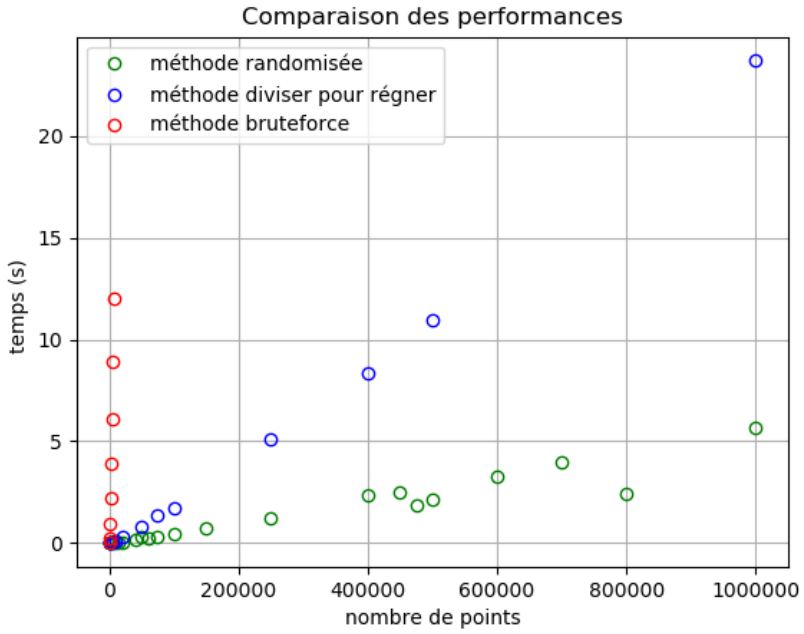

Figure 5: Algorithme randomisé

4 Conclusion

L'algorithme randomisé donne de bonnes performances. Néanmoins, à cause de sa forte dépendance de la distribution des points, on a choisit d'utiliser l'algorithme *Diviser Pour régner*. Le travail sur cette problématique était intéressant. Ça nous permet de plonger dans le monde de la recherche d'une solution optimale d'un problème concret. Ça nous a aussi permet de travailler en groupe, et échanger les idées afin d'avoir une bonne solution.