Convex hulls

Alexis Marouani
Rémi Grzeczkowicz

Février 2022

1 Introduction

Nous avons choisi de travailler en Python, utilisant :
• La bibliothèque numpy pour la manipulation de tableaux;
• La bibliothèque matplotlib pour l'affichage des graphiques.

Le sujet traité consiste à implémenter et comparer deux algorithmes qui prennent en entrée un ensemble de point et retourne son enveloppe convexe.

2 Datasets

2.1 Dataset A

Le dataset de type A permet de générer un carré plein discret, et d'effectuer une rotation. Ce type de dataset permet de visualiser une enveloppe convexe relativement basique où les 4 points extremaux sont censés être les seuls retournés par l'algorithme.

2.2 Dataset C

On remarque que sur les 4 types de datasets, seul C (génération de points dans un disque de rayon r) pourrait ne pas terminer et on a seulement une estimation probabiliste pour le temps de terminaison.
Nous avons donc choisi de faire deux types de datasets C :

- Un 1e respectant l'énoncé, en pseudo-code :

Data: n nombre de points, r rayon du disque
Result: C un tableau de n points aléatoires dans le disque de rayon r
while *longueur de* $C < n$ **do**
  Tirer un point dans le carré de côté $2r$ centré en 0
  if *Ce point est dans le cercle de rayon* r **then**
    Ajouter ce point à C
  end
end

Algorithm 1: generateC (1e version)

- Un 2e exploitant les coordonnées polaires :

Data: n nombre de points, r rayon du disque
Result: C un tableau de n points aléatoires dans le disque de rayon r
while *longueur de* $C < n$ **do**
  Tirer un nombre ρ entre 1 et r
  Tirer un angle θ entre 0 et 2π
  Ajouter le point $(\rho \times cos(\theta), \rho \times sin(\theta))$ à L
end

Algorithm 2: generateC (2e version)

Le dataset de type C est celui où l'enveloppe convexe sera généralement plus compliqué.

2.3 Dataset D

Pour l'algorithme du type de dataset D, on fait de même que pour l'algorithme ci-dessus, sans se donner le nombre ρ afin de se restreindre au cercle de rayon r.

Data: n nombre de points, r rayon du disque
Result: C un tableau de n points aléatoires dans le disque de rayon r
while *longueur de* $C < n$ **do**
  Tirer un angle θ entre 0 et 2π
  Ajouter le point $(r \times cos(\theta), r \times sin(\theta))$ à L
end

Algorithm 3: generateC (2e version)

Le dataset de type D admet une une enveloppe convexe constituée de tous les points, il permet de visualiser si l'algorithme fait des erreurs grossières ou non.

3 Sweeping

3.1 Algorithmes

On commence par écrire une fonction de base qui permettra de savoir si trois points A,B,C sont lus dans l'ordre horaire ou non. Pour cela on se base sur la propriété suivante :

$$\text{ABC triangle direct} \iff det(\overrightarrow{AB}, \overrightarrow{BC}) > 0$$

On peut ensuite écrire l'algorithme sweeping qui génére la partie haute de l'enveloppe convexe :

Data: Dataset de longueur n où les points sont triés de manière croissante selon les x
Result: Chain, la liste des points formant l'enveloppe convexe supérieure
Ajouter les deux premiers éléments de Dataset à Chain
for *i allant de 2 à n-1* **do**
  Soit p=Chain[-2] l'avant dernier élément
  Soit q=Chain[-1] le dernier élément
  Soit r=Dataset[i]
  while *p,q,r forme un triangle direct et Chain contient au moins deux éléments* **do**
    Enlever le dernier élément de Chain
    Soit p=Chain[-2] le nouveau avant dernier élément
    Soit q=Chain[-1] le nouveau dernier élément
  end
  Ajouter r à Chaine
end

Algorithm 4: sweeping pour la partie haute de l'enveloppe

On procède de même pour la partie inférieure et on peut compiler les différentes étapes :

Data: Dataset de taille n
Result: Enveloppe convexe
Trier Dataset de manière croissante selon les abcisses
Appeler sweeping pour la partie haute sur Dataset
Appeler sweeping pour la partie basse sur Dataset retourné
Renvoyer la concaténation des deux appels (avec éventuelle supression des termes extrêmaux si on veut éviter le doublon du premier et dernier élément)

Algorithm 5: Sweeping

Pour tester les algorithmes, nous avons utilisé les outils suivants :
- Visualisation graphique
- Bon nombre de points dans l'enveloppe convexe pour les datasets de type A et D
- Sortie identique à l'algorithme développé en partie 2

3.2 Complexités

- Le 1e tri a une complexité en $\mathcal{O}(n \log n)$

- L'ajout d'un point à Chain effectue au plus m opérations élémentaires où m est le cardinal maximal de Chain : $\mathcal{O}(m)$

- Cet ajout est effectué une et une seule fois pour chaque point du Dataset, la construction de Chain a donc une complexité en : $\mathcal{O}(n \times m)$

Ce cardinal pouvant être égal à n, on obtient une complexité en $\mathcal{O}(n^2)$

Cependant, on constate que dans la construction de Chain, chaque élément du Dataset est ajouté une et une fois et est supprimé au plus une fois, les opérations d'ajout et de supression étant les seuls que l'on effectue si on met de côté les affectations et les appels à la fonction élémentaire qui sont exécutés uniquement conjointement à une supression ou un ajout. On en déduit une complexité en $\mathcal{O}(n)$

Finalement on obtient une complexité en $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$

3.3 Tests de complexité sur les différents Datasets

A la section précédente, on a réussi à s'affranchir de la taille maximale de l'enveloppe convexe, on devrait donc observer des performances relativement similaires sur les différents types de Datasets.

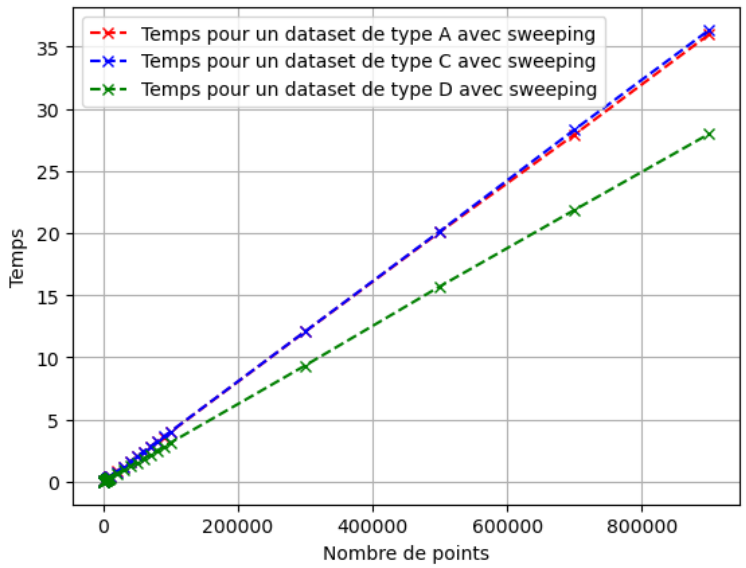

Figure 1: Temps d'exécution en fonction du nombre de points pour différent type de Dataset

En figure 1, les temps d'exécution de sweeping sur les différents types de Dataset.

On remarque que l'algorithme est plus rapide pour D car aucune supression de point n'est effectué i.e. la boucle While de notre algorithme ne s'exécute jamais.

On remarque qu'on obtient une complexité qui semble linéaire malgré un tri au début.
Notre hypothèse est que les opérations Python faites lors de sweeping qui elles sont une compléxité linéaire sont beaucoup plus lourdes que le tri effectué avec la fonction numpy.argsort(), implémenté et optimisé directement en C.

Pour s'en convaincre, nous avons verifié que les tris étaient bien en $\mathcal{O}(n \log n)$:

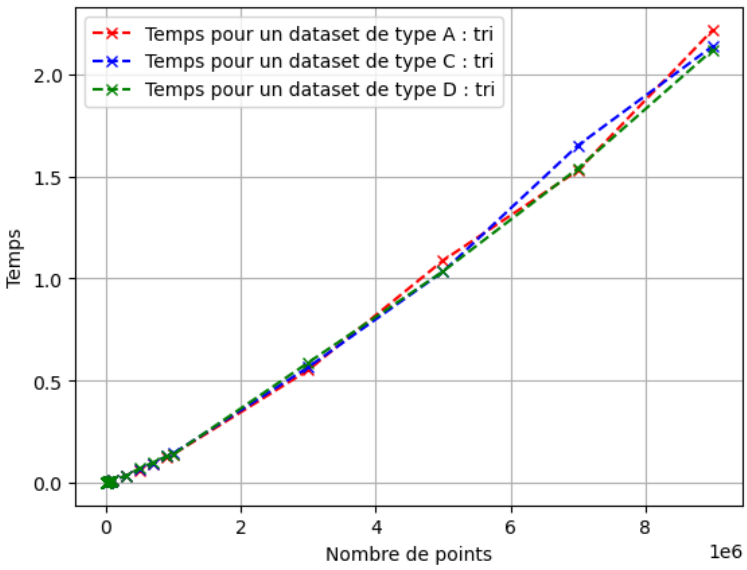

Figure 2: Vérification de la complexité du tri

Puis nous avons verifié que les tris prenaient un temps négligeable par rapport aux autres opérations :

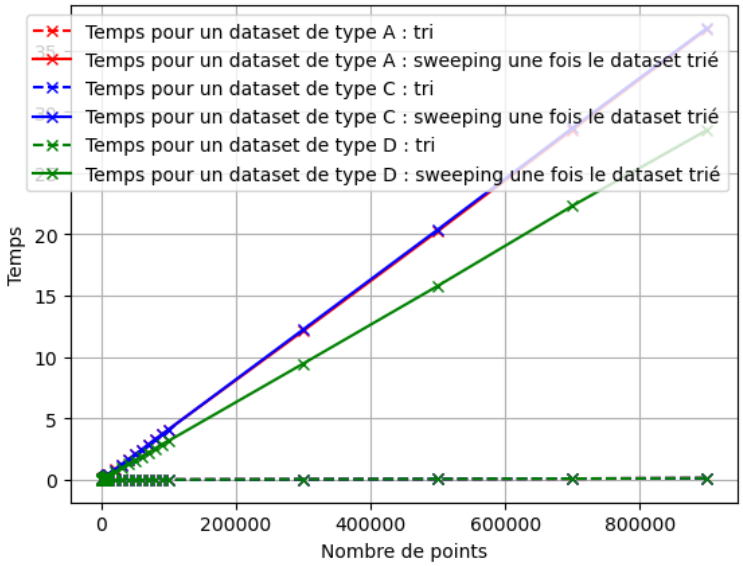

Figure 3: Comparaison du temps pour le tri et du temps pour sweeping une fois les datasets triés en fonction du nombre de points

4 Output-sensitive algorithm

4.1 Median

L'algorithme naïf consiste à trier à liste et renvoyer l'élément du milieu. Cependant, celà nécessite de trier. L'algorithme s'effectue donc avec une complexité $\mathcal{O}(n \log n)$.
Il faut donc réfléchir à un autre algorithme. Une solution connue est l'algorithme "médiane des médianes". Il fonctionne de cette manière :

- On coupe la liste en sous liste de 5 éléments. On trie chaque sous-liste (en temps constant donc car ces sous-listes ne comportent que 5 éléments) et on renvoie la médiane. Le nombre de 5 est choisie de manière optimale car il est impair (la médiane est donc facile à renvoyer) et se trie facilement (choisir 3 ou 7 rallongerait la durée de l'algorithme car 3 nécessiterait de trier trop de liste et 7 de trier des listes trop longues).
- On obtient donc une liste des médianes de chaque sous-liste et on execute de manière récursive l'algorithme pour trouver la médiane de cette sous-liste.
- On a alors une valeur qui sert de piveau. On réexécute l'algorithme avec la liste la plus longue parmis la liste des élements en dessous du pivot et celle des éléments au dessus du pivot.

En pseudo-code, voici ce que ça donne :

Data: Dataset de longueur n où les points sont dans un ordre quelconque
Result: Médiane de ce datatset
if *la taille du datatset est inférieure à 5* **then**
| On retourne la médiane calculée normalement (en triant puis retournant l'élément du milieu)
end
else
| on applique l'alogirthme de selection avec pour entrée dataset, et len(dataset)//2
end

Algorithm 6: La fonction médiane

On précise alors l'algorithme de sélection:

Data: Un dataset, et un pivot
Result: Le pivot-ième plus petit élement de dataset
if *la taille du datatset est inférieure à 5* **then**
| On retourne la médiane calculée normalement (en triant puis retournant l'élément du milieu)
else
| On calcule la liste liste_mediane, la liste des médiannes des sous liste de taille 5 (ou moins).
| On applique l'algorithme de sélection récursivement pour calculer la mediane_des_medianes. On utilise pour pivot len(liste_mediane)//2.
| On définit la liste upper (resp. under) comme la liste des élements du du dataset supérieurs (resp. inférieurs) à la médiane_des_médianes.
| **if** *pivot* $<$ *len(under)* **then**
| | On applique l'algorithme de sélection avec la liste under comme dataset et on garde le même pivot.
| **else if** k = *len(under)* **then**
| | On retourne mediane_des_medianes
| **else**
| | On applique l'algorithme de sélection avec la liste upper comme dataset et on prend k-len(under)-1 comme pivot.
| **end**
end

Algorithm 7: L'algorithme de sélection

On vérifie alors que la complexité de l'algorithme de détermination de la médiane est linéaire. Pour cela on trace le graphique suivant.

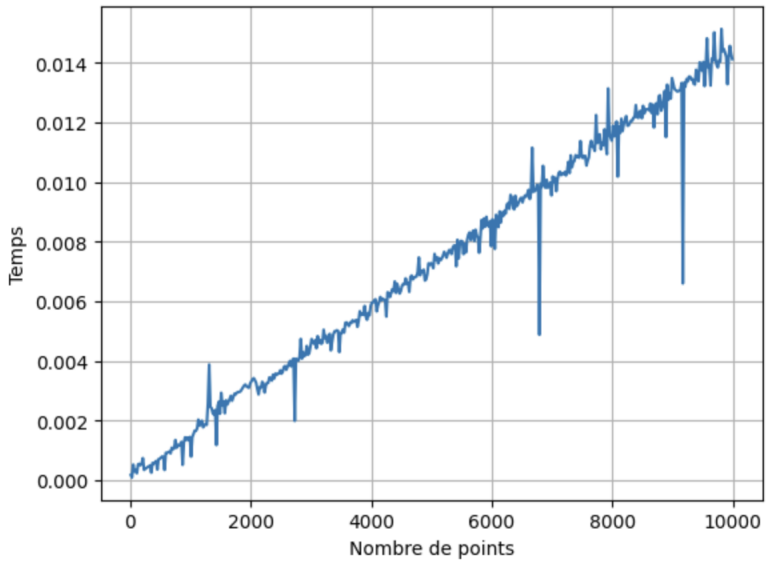

Figure 4: Variation du temps d'éxécution de l'alogrithme de la médiane en fonction du nombre de points pour un dataset de type A

4.2 Main algorithm

On commence par définir un alogorithme qui nous dit si le poit P est au dessus de la droite formée par les points Q et R.

Data: Un point P(x_p,y_p), un point Q(x_q,y_q), un point R(x_r,y_r) avec $x_q < x_r$
Result: Un boléen qui vaut True si P est au dessus de la droite formée par Q et R, False sinon
On définit f comme la fonction telle que $f(a) = \frac{y_r - y_q}{x_r - y_q}(a - x_r) + y_r$
if $y_p >= f(x_p)$ **then**
  On retourne True
else
  On retourne False
end

Algorithm 8: Position relative d'un point par rapport à la droite formée par deux autres points

À partir de là, on peut définir l'algorithme qui va nous donner les deux points formant le seigment le plus haut de l'enveloppe convexe.

Data: un Dataset
Result: Les deux points qui forment le seigment le plus haut de l'enveloppe convexe de Dataset
On calcule la médiane de Dataset grâce à l'algorithme détaillé précedement.
On initialise les deux premiers points : p1 étant le premier point rencontré dans l'odre de Dataset situé à gauche de la médiane et p2 le premier point rencontré dans l'ordre de Datasert situé à droite de la médiane.
On ajoute p1 (resp. p2) à la liste des points à gauche (resp. à droite) déjà explorés et on le supprime de Dataset.
On mélanche Dataset pour prendre les points au hasard.
while *Dataset n'est pas vide* **do**
  On prend p3, le premier point de Dataset et on le supprime de Dataset. On l'ajoute dans la liste correspondante (i.e. : à gauche s'il est à gauche de la médiane, et à droite sinon).
  if *p3 est au dessus de la droite formée par p1 et p2* **then**
    if *p3 est à gauche de la médiane* **then**
      On cherche parmis les points déjà explorés à droite de la médiane, celui qui forme la plus grande pente avec p3 (on note ce point p4).
      p1 devient p3
      p2 devient p4
    else
      On cherche parmis les points déjà explorés à gauche de la médiane, celui qui forme la plus petite pente avec p3 (on note ce point p4).
      p1 devient p4
      p2 devient p3
    end
  end
end
On retourne p1 et p2

Algorithm 9: Recherche du plus haut seigment de l'enveloppe convexe d'un Dataset

On vérifie alors que la complexité d'ajouter un point se fait bien en $\mathcal{O}(k)$ avec k le nombre de point déjà explorer.

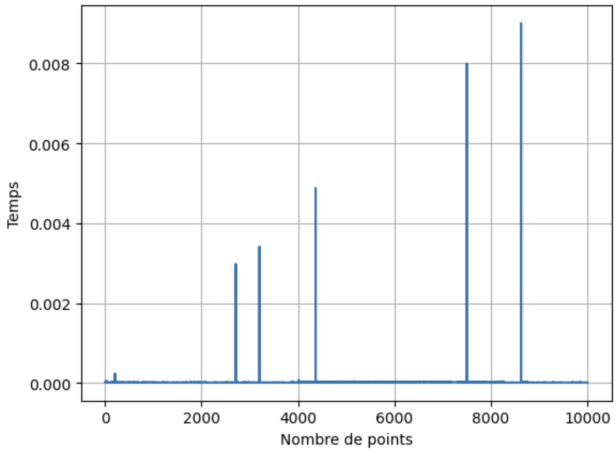

Figure 5: Variation du temps de recherche de la base en fonction du nombre de point déjà ajoutés pour un dataset de type C de 10000 points et de rayon 10

Enfin, on peut obtenir la liste des points composant l'enveloppe convexe supérieure, dans l'odre en executant l'algorithme suivant de manière récursive :

Data: Un Dataset
Result: La liste des points formant l'enveloppe convexe supérieur de Dataset, dans l'ordre de droite à gauche
if *Dataset ne contient que 1 point* **then**
| On retourne ce point
else if *Dataset ne contient que 2 points* **then**
| On retourne ces deux points, triés dans l'ordre croissant des x
else
| On note l et r les points p1 et p2 retourné par l'algorithme précédent.
| On note left, la liste des points dont l'absisse est inférieure ou égale à celle de l.
| On note right, la liste des points dont l'absisse est supérieure ou égale à celle de r. On recommence l'algorithme sur la liste left et sur la liste right.
end

Algorithm 10: Recherche de l'enveloppe convexe supérieure d'un Dataset

On remarquera que pour calculer l'enveloppe convexe inférieure, il suffit d'éxécuter l'algorithme précédent en retournant le Dataset. Pour vérifier le résultat, on pourra tracer l'ensemble des points ainsi que son enveloppe à l'aide du module MatPlotLib. À titre d'exemple, voici ce que l'on obtient pour un Dataset de type A et un Dataset de type C:

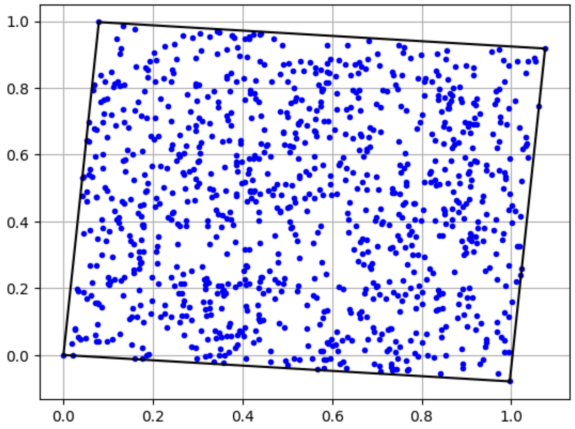

Figure 6: Tracé de l'enveloppe convexe d'un Dataset de type A pour 1000 points

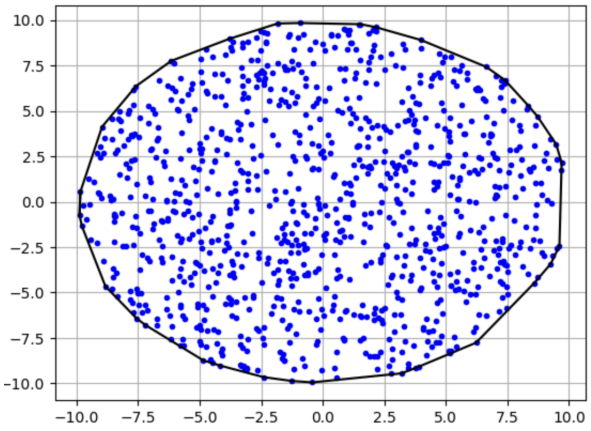

Figure 7: Tracé de l'enveloppe convexe d'un Dataset de type C pour 1000 points et un rayon de 10

5 Comparaison des algorithmes

5.1 Comparaison des résultats

Pour comparer les résultats des deux algorithmes, on peut utiliser un petit programme fonctionnant comme tel :

Data: Un entier N donnant le nombre de comparaison souhaitées
Result: Un pourcentage de similitude sur les comparaisons
On tire un type de Dataset au hasard.
On tire le nombre de poinst du Dataset au hasard et, si nécessaire, un rayon.
On obtient la liste des points composant l'enveloppe selon les deux algorithmes. On trie ces listes et on les compare.
On répète le procédé N fois.

Algorithm 11: Algorithme de comparaison des résultats

Après avoir fait tourner cet algorithme plusieurs fois, pour N=1000, on obtient systématiquement 100% de similitude. On peut donc affirmer que, dans tous les cas, les deux algortihmes nous donnent les mêmes résultats.

5.2 Comparaison des complexités

Pour comparer les complexités, en fonction de l'algorithme et du Dataset, on trace le graphique en Figure 8.

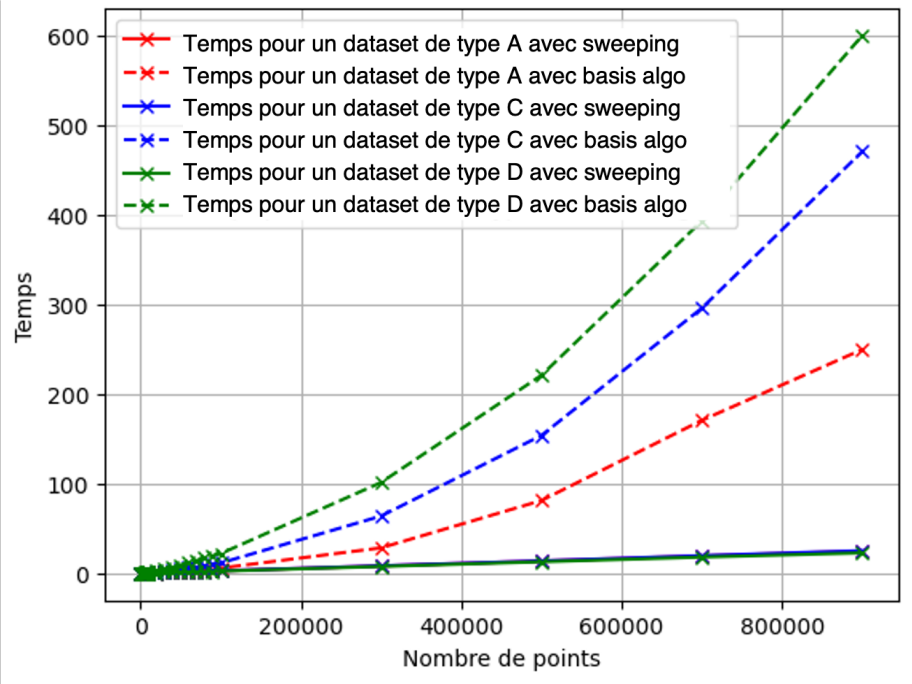

Figure 8: Variation du temps d'éxécution en fonction du nombre de points pour différents ensembles avec les deux algorithmes (ndlr : les tracés vertes représentent le type de Dataset D)

Le deuxième algorithme dépend fortement du type de Dataset. Cela est du au fait que plus les seigments de l'enveloppe sont longs, plus l'algorithme est rapide car il exclut des points à chaque étape. Ainsi, un dataset de type A qui possède une enveloppe composée de 4 grands seigments exlut rapidement la majorité des points là où un dataset de type C possède une enveloppe avec de petits seigments. L'algorithme doit donc être exécuté plus de fois pour le dataset de type C. De manière générale, l'algorithme sweeping semble être indépendant du nombre de seigment composant l'enveloppe là où le deuxième alogrithme est dépendant de ce facteur.

L'algorithme de sweeping semble donc être bien plus efficace. Cependant, il ne faut pas oublier que la méthode de tri utilisée est très efficace. Sa complexité ne se voit donc pas vraiment en pratique.