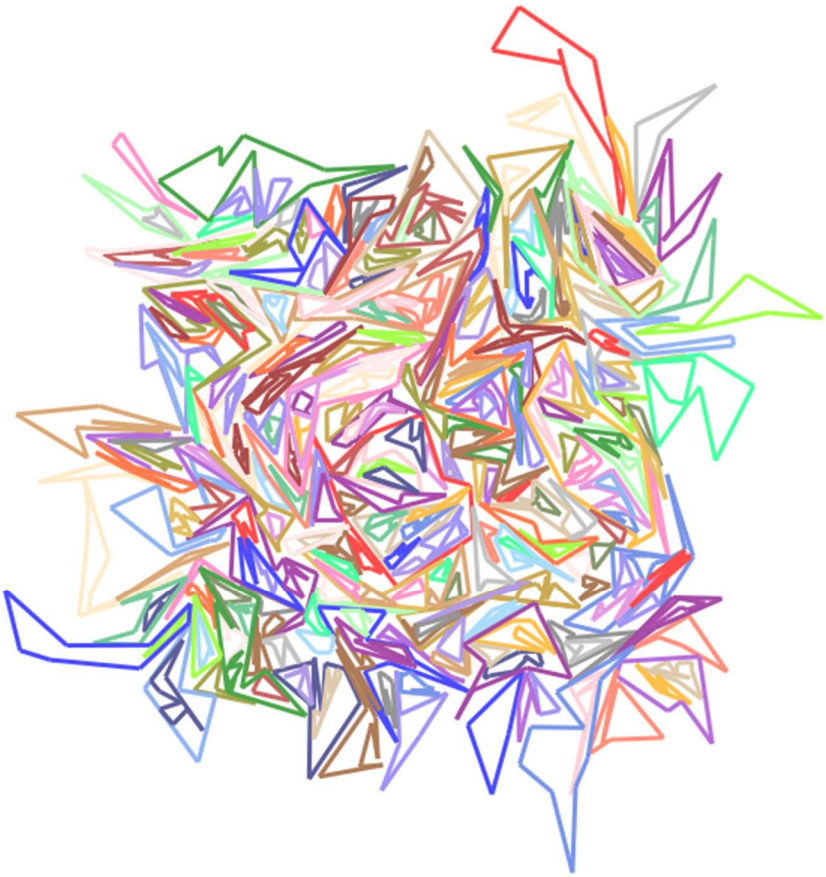
Dione Alan et Marmonier Yohann

Rapport du Projet d'Algorithmie

Sur la détection d'inclusion de Polygones en 2D



Grenoble INP - Ensimag
1ère année, 2019-2020

Table des matières

I. Introduction

Notre objectif était la détection d'inclusion de polygones au sein d'un repère cartésien de dimension 2. Pour chaque polygone, le programme doit indiquer le numéro du polygone dans lequel il est directement inclus ou -1 s'il n'est inclus dans aucun polygone.

Pour concevoir un algorithme capable de mener à bien cette tâche, un passage obligé est un algorithme de détection d'inclusion qui détermine si oui ou non un polygone A est strictement inclus dans un polygone B.

Etant donné que le sujet stipulait qu'il n'y a aucune intersection entre deux segments quelconques (mis à part deux segments consécutifs d'un même polygone), on comprend facilement que déterminer si un point du polygone A est à l'intérieur du polygone B revient à déterminer si tout le polygone A est à l'intérieur du polygone B.

Nous nous intéresserons donc dans un premier temps à l'algorithme dit ***Point in Polygon,*** un algorithme qui permet de déterminer si un point est dans un polygone.

Dans un second temps, nous examinerons la mise en place et la justification (théorique et expérimentale) d'une optimisation algorithmique que nous avons utilisée afin de réduire le coût élevé des détections d'inclusion : c'est **la méthode des Quadrants**.

Enfin, une fois que nous aurons optimisé le plus possible notre méthode de détection d'inclusion, nous tâcherons de **réduire au maximum le nombre de test d'inclusion** à effectuer afin d'atteindre une complexité minimum.

Dans ce but, et c'est la partie la plus fournie et complexe du projet, nous examinerons et comparerons l'implémentation de divers algorithmes et structure de données censés optimiser la recherche d'inclusion de polygones.

II. Algorithme *Point in Polygon (PiP)*

1. Algorithme de RayCasting

L'algorithme de **RayCasting** est l'algorithme mentionné par le sujet afin de traiter la question du PIP. C'est donc vers lui que nous nous sommes tournés en premier lieu.

Prenons le polygone **A** et le polygone **B** : nous voulons déterminer si le polygone **A** est inclus dans le polygone **B**, pour cela, nous regardons si un point **P** du polygone **A** est inclus dans le polygone **B.**

Pour ce faire, on va s'intéresser à la demi-droite **d** partant du point **P,** de vecteur directeur (-1, 0) ; donc la demi-droite partant de **P** allant « vers la gauche ».

Pour chaque segment de **B,** on va maintenant regarder s'il intersecte **d.** Si c'est le cas, on incrémente un compteur de segments croisés.

On note **e** l'état de **P** (« **intérieur** » ou « **extérieur** ») et on l'initialise à **extérieur.** Chaque segment intersecté changera alors logiquement son état. Son état final indiquera donc si le point **P** est à l'intérieur ou non de **B**.

Une fois tous les segments parcourus, on comprend que :

- Si le compteur est **pair** : alors l'état **e** aura changé d'état un nombre pair de fois : il sera donc à la fin à **extérieur**.
- Si le compteur est **impair** : alors l'état **e** aura changé d'état un nombre impair de fois : il sera donc à la fin à **intérieur**.

La parité du compteur indique le résultat du test d'inclusion.

Il nous reste à éclaircir la méthode utilisée pour détecter l'intersection entre un segment de **B** et **d**.

Soit **s** un segment de **B** reliant les deux points **P2** et **P3**. Le déterminant de l'équation du segment vaut :

$$(x - x_{P2})(y_{P3} - y_{P2}) - (y - y_{P2})(x_{P3} - x_{P2}) = 0$$

$$<=> \; x = \frac{(y - y_{P2})(x_{P3} - x_{P2})}{(y_{P3} - y_{P2})} + x_{P2}$$

L'ensemble des points de la demi-droite **d** sont les points de la même ordonnée que **P** et d'abscisse inférieure à celle de **P.** On cherche donc un point de la même ordonnée que **P** qui appartient à **s,** soit qui résout l'équation de **s.**

Il nous faut tout d'abord vérifier si l'ordonnée de **P** est bien entre les deux ordonnées des points **P2** et **P3**. Si tel est le cas, il ne nous reste plus qu'à savoir s'il existe un point ayant une abscisse inférieure à **P** qui résout l'équation de **s.**

Pour cela, il suffit de regarder si :

$$x_P < \frac{(y_P - y_{P2})(x_{P3} - x_{P2})}{(y_{P3} - y_{P2})} + x_{P2}$$

La complexité de cet algorithme est calculée sur la base du nombre de segments n_B. Comme on doit tester l'intersection pour chaque segment de **B,** et que le test d'intersection en lui-même est uniquement composé d'instructions élémentaires, la complexité temporelle est en $O(n_B)$.

2. Algorithme du Winding Number

Le deuxième algorithme auquel nous nous sommes intéressés est le **Winding number** algorithme. Nous avons rapidement été amenés à l'examiner étant donné qu'il représente dans la littérature l'alternative classique au **RayCasting** algorithme.

Nous ne détaillerons pas son implémentation étant donné que son explication est complexe. La principale différence avec l'algorithme du **RayCasting** réside dans les résultats pour les polygones complexes qui sont plus précis. Néanmoins, les polygones que nous devons examiner sont tous simples.

En bref, cet algorithme consiste à compter le nombre de fois que les segments du polygone **B** s'enroule autour de **P :**

- On rajoute +1 si le segment s'enroule « dans le sens contraire des aiguilles d'une montre »
- -1 s'il s'enroule dans « le sens des aiguilles d'une montre ».

Comment savoir si un segment s'enroule dans le sens des aiguilles d'une montre ou dans le sens inverse ? Il suffit de :

- regarder l'ordonnée de **P :** elle doit être comprise entre celle de **P2** et **P3**
- remplacer par **P** dans l'équation du segment **s :** si le résultat est positif alors **P** est à gauche du segment et celui-ci s'enroule dans le « sens contraire à celui des aiguilles d'une montre ». Si le résultat est négatif, alors **P** est à droite du segment et celui-ci s'enroule dans le « sens des aiguilles d'une montre ».

Cet algorithme s'exécute également en $O(n_B)$ puisqu'il faut tester l'enroulement pour chaque segment de **B.** Toutefois, la constante multiplicative de la complexité est différente de l'algorithme précédent ce qui induira une différence de temps d'exécution.

3. Algorithme de Triangulisation

Nous avons par la suite pensé à une méthode de **triangulisation** – avant de nous rendre compte que c'était bien plus complexe que ça en avait l'air.

Notre idée était de partitionner le polygone **B** en triangles. Ainsi, il suffisait de tester l'inclusion de **P** dans chaque triangle de **B** : si **P** est dedans alors la détection d'inclusion est positive, si **P** n'est dans aucun des triangles alors elle est négative. **Note** : L'inclusion d'un point dans un triangle s'effectue en temps constant et est triviale.

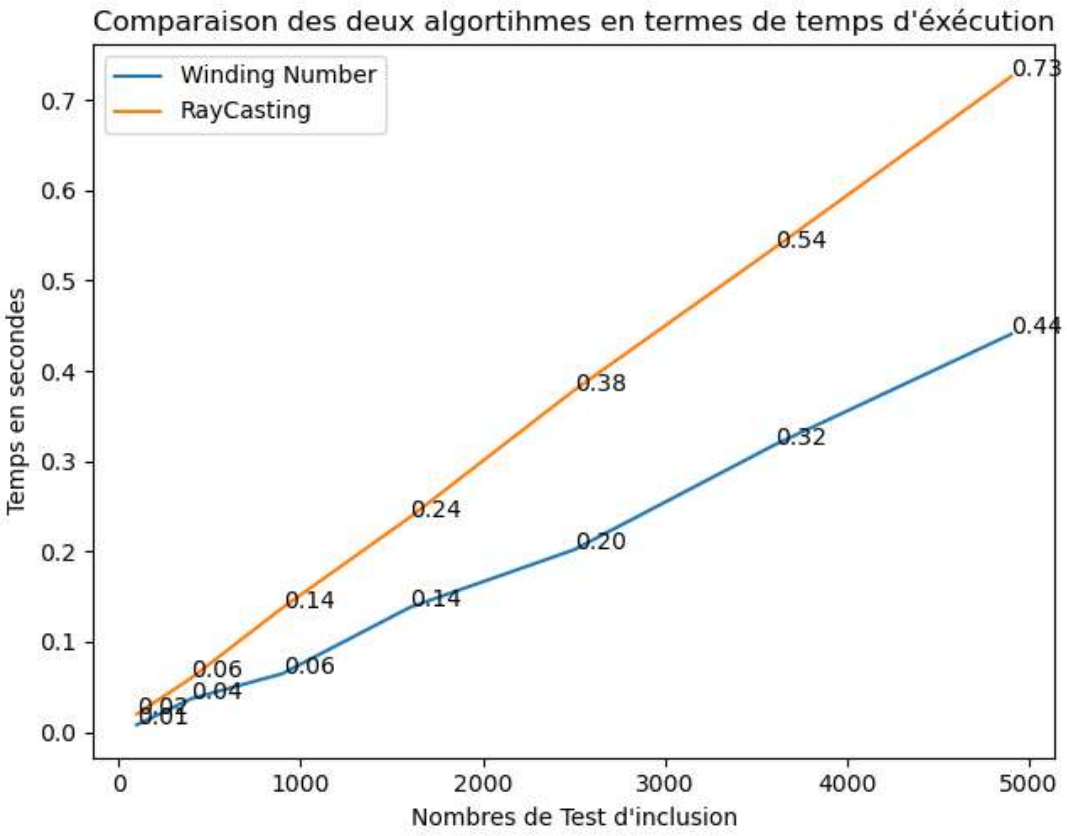
Nous avons pensé à une partition simple en prenant tout triplets de sommets consécutifs de **B.** Mais bien sûr, cette méthode pêche pour certains polygones non convexes où des triangles ainsi obtenus peuvent donner sur l'extérieur du polygone.

Pour pallier cette difficulté, nous avons dans un premier temps distingué les polygones convexes des concaves. Nous avons alors utilisé la **triangulisation** pour les polygones convexes et le **RayTracing** pour les concaves.

Pour déterminer la convexité d'un polygone, nous avons commencé par mettre tous les polygones « dans le sens inverse à celui des aiguilles d'une montre » (le sens dans lequel les points sont donnés) grâce à une fonction du module **geo**. Une fois cette étape effectuée, il suffit de regarder si l'angle formé par chaque triplet de points consécutifs est dans le sens inverse à celui des aiguilles d'une montre. Si tel est le cas alors le polygone est convexe.

Au final, déterminer si un polygone est convexe coûte presque autant que de déterminer si **P** appartient à l'un des triangles de **B** (cout linéaire). Nous n'avons donc pas retenu cet algorithme.

4. Comparaison : RayTracing et Winding Number



Les deux algorithmes ont à peu près la même complexité (à la constante près). L'algorithme **Winding Number** est tout de même plus rapide. La différence de vitesse se justifie par le nombre de calculs et comparaisons à effectuer.

Pour les trois dernières abscisses, l'écart entre les scores temporels des deux algorithmes est multiplié de :

- 1.41 pour passer de 1600 à 2500 (pour un nombre de tests d'inclusion multiplié par $\frac{2500}{1600} = 1.5625$)
- 1,44 pour passer de 2500 à 3600 (pour un nombre de tests d'inclusion multiplié par $\frac{2500}{3600} = 1.44$)
- 1,33 pour passer de 3600 à 4900 (pour un nombre de tests d'inclusion multiplié par $\frac{4900}{3600} = 1.361$)

L'évolution de l'écart temporel est assez proche de l'évolution du nombre de tests d'inclusion effectués. C'est bien ce que nous pensions trouver : les deux algorithmes ont la même complexité à la constante multiplicative près.

III. Méthode des Quadrants

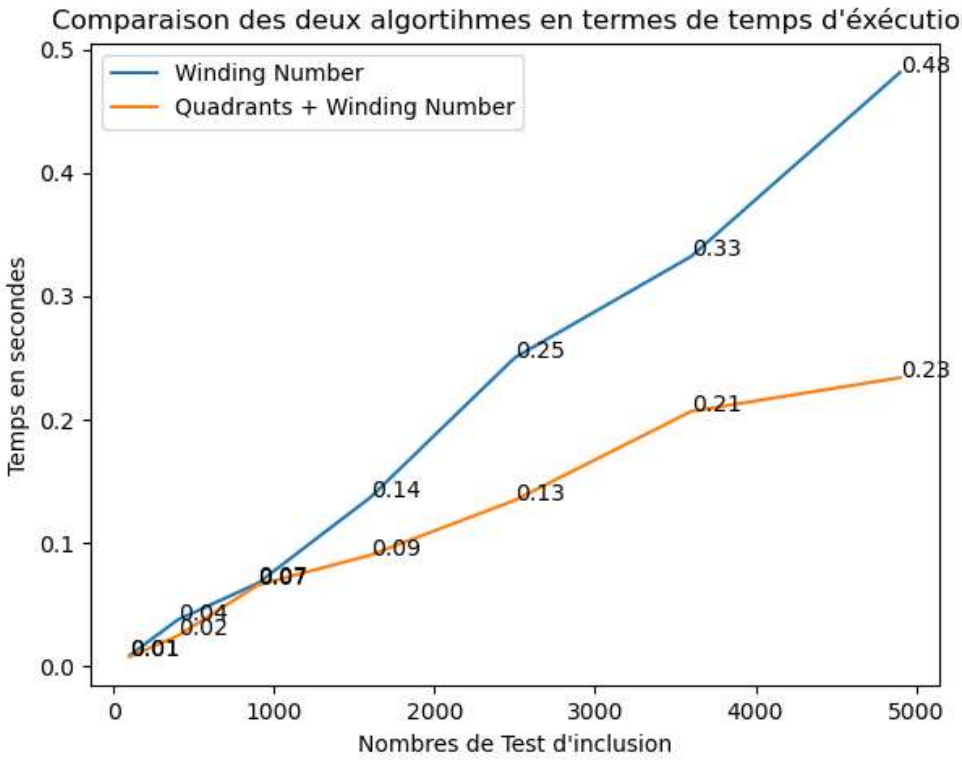
Afin d'optimiser la détection d'inclusion entre deux polygones, nous avons eu l'idée de chercher en premier lieu à savoir si le rectangle minimum les contenant étaient inclus l'un dans l'autre.

L'algorithme pour trouver le plus petit rectangle encadrant un polygone est donné dans le module **geo :** il suffit de rajouter les points un à un au rectangle en ne gardant que les abscisses et ordonnées minimales et maximales. Pour chaque point du polygone, il y a ainsi 4 tests à faire. La complexité pour générer le rectangle englobant d'un polygone ayant *n* points est ainsi de O $(4 \times n)$.

Par la suite, tester l'inclusion entre deux rectangles est en O (4).

Sachant que tester l'inclusion entre deux polygones avec un des algorithmes étudiés précédemment coûte bien plus. En comptant grossièrement le nombre d'instructions élémentaires de l'algorithme **Winding Number**, on obtient du O $(10 \times n)$.

Ainsi, il est bien plus avantageux de tester d'abord l'inclusion des rectangles avant de tester l'inclusion classique. En effet, si l'inclusion des rectangles renvoie faux, on est alors sûrs que l'inclusion classique renverra faux aussi.



Voilà sans surprise le résultat de cette optimisation très sympathique.

IV. Réduction du nombre de tests d'inclusion

Nous entrons maintenant dans la partie la plus dense du projet. Nous allons examiner les idées trouvées afin de diminuer au minimum le nombre de tests d'inclusion à effectuer. A noté que lorsqu'on parle de tests d'inclusions, on parle désormais de test de quadrants suivi, ou non, de test classique utilisant le **Winding Number** algorithme.

1. Approche naïve

L'approche naïve, dite en force brute, est sûrement la plus intuitive et celle qui nous est naturellement venue en tête en cherchant une solution.

Basiquement on va effectuer le test d'inclusion pour tous les couples possibles de polygones. Voici le début de cet algorithme :

```
input : La liste polygones de taille n
input : Une fonction permettant de faire un test d'inclusion entre deux
        polygones
1 listeInclusions ←[[] × n] ;
2 for poly1 in polygones do
3 |   for poly2 in polygones do
4 |   |   if poly1 != poly2 and estInclus(poly, poly2) then
5 |   |   |   Ajouter poly2 à la liste des polygones qui incluent poly1 dans
  |   |   |    listeInclusions
6 |   |   end
7 |   end
8 end
```

Une fois cette liste d'inclusions obtenue, il nous reste à trancher entre les inclusions directes et indirectes dans le cas des polygones inclus par plusieurs autres polygones.

Pour ce faire, nous allons regarder le nombre de polygones englobant chaque polygone englobant. En effet, si le polygone **B** inclus directement le polygone **A**, alors il aura exactement n – 1 polygones englobants ; n étant le nombre d'englobant de **A**.

Voici donc la fin de l'algorithme :

```
   input : La liste polygones de taille n
   input : On sait dans quels polygones chaque polygone est inclus
 1 for poly1 in polygones do
 2 |   if poly1 est inclu dans 0 polygone then
 3 |   |   La réponse pour ce polygone est -1
 4 |   end
 5 |   if poly1 est inclu dans 1 polygone then
 6 |   |   La réponse pour ce polygone est le numéro du polygone incluant
 7 |   else
 8 |   |   for polyEnglobant dans la liste des polygones englobant poly do
 9 |   |   |   if polyEnglobant est inclu par un polygone de moins que poly
   |   |   |    then
10 |   |   |   |   La réponse pour ce polygone est le numéro de
   |   |   |   |    polyEnglobant
11 |   |   |   end
12 |   |   end
13 |   end
14 end
```

Cet algorithme est très naïf puisqu'il réalise un maximum de tests d'inclusion. En effet, il en fait $(n-1) \times (n-1) \approx n^2$.

De plus, il nécessite par la suite de parcourir la liste des englobants de chaque polygone pour déterminer lequel l'englobe directement.

Dans le pire des cas, en notant s_i le nombre de segments du polygone i, on effectue à peu près $4 \times n + n \times 4 \times \sum s_i \times c$ instructions élémentaires, avec c le nombre d'instructions élémentaires à exécuter pour savoir dans quel sens s_i s'enroule autour d'un point :

- $4 \times n$ instructions pour calculer les rectangles minimums englobant les polygones
- Pour chacun des n polygones, on s'intéresse à l'inclusion dans le polygone i :
  - on fait le test des rectangles qui coûte 4 instructions (résultat négatif)
  - on applique le Winding Number algorithme qui coute $s_i \times c$ instructions

En définitive on obtient une complexité de $O(n \times 4 \times \sum s_i \times c)$

2. Approche par tri de taille

L'optimisation de réduction du nombre de test d'inclusion, de notre point de vue, la plus simple à trouver consiste à prétraiter les polygones en les triant par taille.

Notre tri ne se base pas sur l'aire d'un polygone, mais sur la taille de la diagonale de son rectangle encadrant, qui est plus rapide à calculer étant donné qu'on calcule déjà les rectangles encadrants minimaux.

On est complètement sûr qu'un polygone **A** ayant une diagonale inférieure à celle du polygone **B** ne peut qu'être inclus ou non dans celui-ci, mais ne peut en aucun cas l'inclure. De ce fait, cela élimine un grand nombre de tests d'inclusions à faire.

De plus, si les polygones sont triés par taille et qu'on examine le prochain plus gros polygone, on est sûr que s'il y a inclusion, celle-ci sera minimale.

Voici le pseudo-code correspondant :

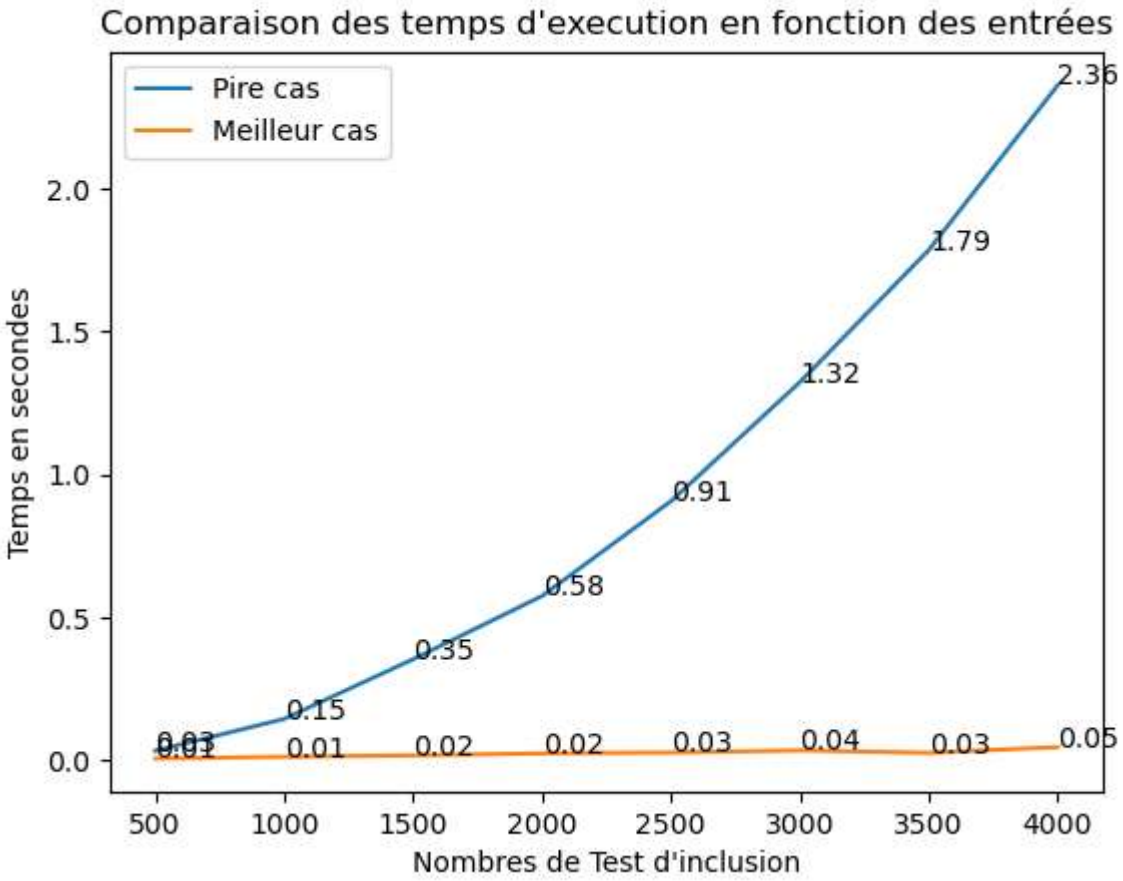
```
input : La liste polygones de taille n
input : Une fonction permettant de faire un test d'inclusion entre deux
        polygones
On trie polygones sur la base de la taille de la diagonale du rectangle
 encadrant minimal
for poly1 in polygones do
    for poly2 in polygones à partir du polygone après poly1 do
        if poly1 est inclu dans poly2 then
            La réponse pour poly1 est le numéro de poly2
        end
    end
end
```

Cet algorithme est plus intelligent puisqu'il nécessite de réaliser dans le pire des cas $1 + 2 + 3 + \cdots + (n-2) + (n-1) = \frac{n \times (n+1)}{2} \approx \frac{n^2}{2}$ itérations.

Ce pire scénario est atteint lorsque qu'aucun polygone n'est inclus dans un autre.

Dans le meilleur des cas, lorsque tous les polygones sont directement imbriqués les uns dans les autres, chaque polygone trouvera immédiatement son incluant, et il y aura seulement n tests d'inclusion à effectuer.

Voici la démonstration expérimentale de l'adaptation de son algorithme à son entrée :



Les pires cas sont composés de carrés mis les uns à côté des autres, sans jamais d'inclusions. Les meilleurs cas sont composés de carrés parfaitement imbriqués les uns dans les autres.

En moyenne, le polygone incluant se situera à la moitié du parcours du reste des polygones potentiellement incluants, on obtiendra donc en moyenne :

$$\frac{1+2+3+\cdots+(n-2)+(n-1)}{2} = \frac{n\times(n+1)}{4} \approx \frac{n^2}{4}$$ tests d'inclusions.

Si on mesure la complexité du programme en termes de nombre de test d'inclusion, cet algorithme du tri de diagonale est en $O\left(\frac{n^2}{4}\right)$.

3. Approche par structure d'arbre

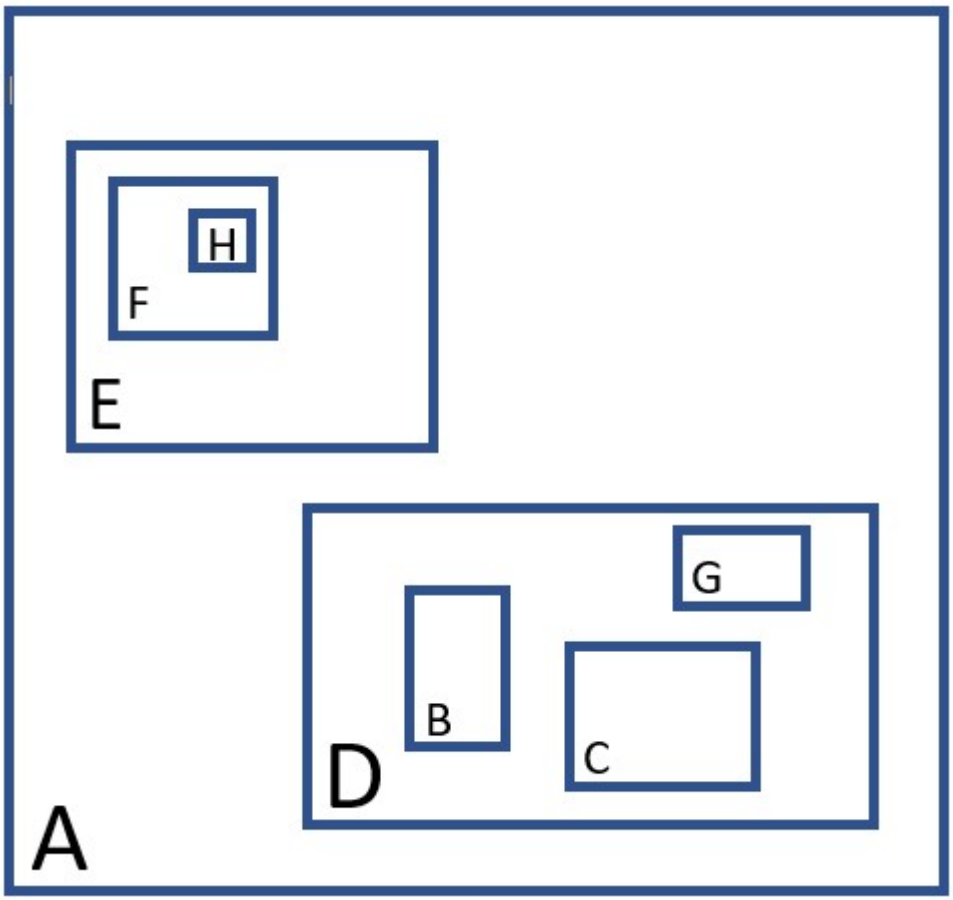
Cette approche est assez particulière. L'idée de départ était d'optimiser au maximum la structure de données à utiliser. Nous avions alors eu l'idée d'utiliser une liste triée. Notre clé de tri était la relation d'inclusion. Nous voulions insérer le polygone dans la liste en fonction de sa place dans la « chaîne d'inclusion » des polygones.

Ainsi, nous aurions pu l'insérer de manière dichotomique. Il aurait ensuite suffit de lire la liste du début à la fin pour retrouver les inclusions en temps constant.

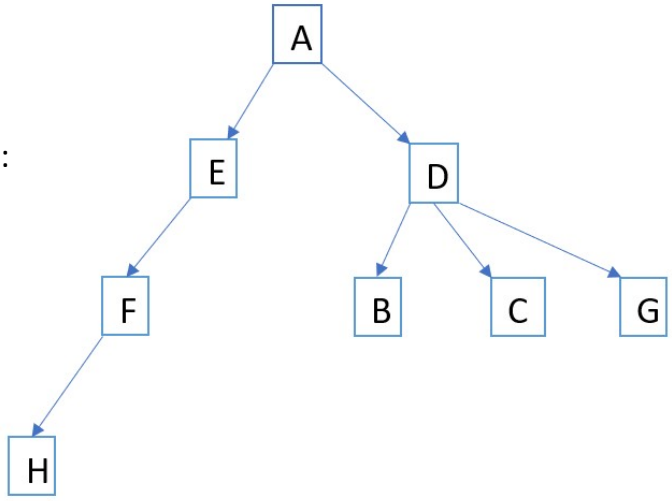
Cette structure est irréalisable puisque la « chaine d'inclusion » des polygones n'est pas unique. Elle peut se diviser en deux sous-chaines si deux polygones sont disjoints mais tout d'un inclus dans le même polygone.

Pour essayer de retrouver cette idée tout en esquivant cette difficulté, nous avons pensé à un arbre. Chaque nœud représenterait un polygone **A**, chacun de ses fils seraient des polygones directement inclus dans **A.**

Ainsi, considérons les polygones suivants :



Nous obtiendrons
un arbre de la forme :



Par soucis de simplification de la construction d'un tel arbre, nous avons commencé par trier les polygones sur la base de la longueur de la diagonale de leur rectangle minimal et ce de manière décroissante. Ainsi, tout nouveau polygone ajouté à l'arbre sera une feuille.

Nous avons également séparé nos polygones en sous-ensembles représentant les chaines d'inclusions de polygones. Ainsi chaque sous-ensemble aura son propre arbre. Voici l'algorithme correspondant :

input : La liste *polygones* de taille n
input : Une fonction permettant de faire un test d'inclusion entre deux polygones

```
On trie polygones sur la base de la taille de la diagonale du rectangle
  encadrant minimal, de manière décroissante
listeEnsemble ← []
for poly in polygones do
    for ensemble in listeEnsembles do
        if poly est inclu dans ensemble[0] then
            poly est ajouté à ensemble
            estAjoute ← True
            break;
        end
    end
    if estAjoute is False then
        poly est ajouté comme le premier élément d'un nouvel ensemble
          dans listeEnsemble
    end
end
```

Il nous reste à écrire la classe Nœud et à remplir nos arbres. Nous avons décidé de construire les arbres de manière récursive. Voici, en python, la classe Nœud :

```python
class Noeud:
    def __init__(self, poly):
        self.poly = poly
        self.fils = []

    def ajout(self, poly, vecteur_reponse):
        est_ajoute = False
        for f in self.fils:
            if est_inclus(poly, f.poly):
                f.ajout(poly, vecteur_reponse)
                est_ajoute = True
                break
        if not est_ajoute:
            self.fils.append(Noeud(poly))
            vecteur_reponse[poly.numero] = self.poly.numero

```

Il suffit par la suite de placer les polygones dans l'arbre :

```python
for ensemble in listes_ensembles:
    racine = Noeud(ensemble[0])
    for i in range(1, len(ensemble)):
        racine.ajout(ensemble[i], vecteur_reponse)
```

Cette approche pourrait être redoutable avec la bonne entrée. En effet, supposons que nous ayons une entrée où chaque polygone inclus directement exactement 2 autres polygones. L'arbre ainsi construit serait un arbre binaire équilibré.

Si cet l'arbre est de plus construit en restant équilibré pendant sa construction, alors l'efficacité est maximale.

Ainsi, pour cette configuration de données d'entrées optimale de n polygones, il faudrait :

$$0 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + \cdots = \sum_{i=1}^{n} \lceil log_2(i) \rceil$$

tests d'inclusions. Malheureusement, les seules entrées que j'ai à ma disposition ne me permettent pas de démontrer expérimentalement l'efficacité de cet algorithme.

4. Approche par tri des ordonnées

Après avoir envisagé plusieurs possibilités d'optimisation, nous en sommes venus à l'idée de trier les polygones selon leurs ordonnées maximales afin de tester l'inclusion avec les polygones situés seulement au-dessus.

Pour ce faire nous cherchons l'ordonnée maximale de chaque polygone, en parcourant ses points et nous l'ajoutons à une liste grâce à la fonction `insort_right` du module `bisect` de python. Cela nous permet de l'insérer de manière dichotomique dans la liste.

Ainsi, le tri des ordonnées se fait en $O(n \times \log(n))$ avec $n = \sum_{polygone} nombre\ de\ points$

Ainsi, la partie du tri des ordonnées n'est clairement pas la plus couteuse, intéressons-nous maintenant aux tests d'inclusions.

Prenons un polygone **A.** Examinons les polygones dont l'ordonnée maximale est au-dessus de la l'ordonnée maximale de **A :**

- soit leur rectangle minimal n'inclut pas celui de **A,** auquel cas le test est effectué en temps constant et on continue à monter
- soit leur rectangle minimale inclut celui de **A.** Dans ce cas, on teste l'inclusion complète. On s'arrête si elle fonctionne.

Cette partie de l'algorithme est encore une fois la plus coûteuse.

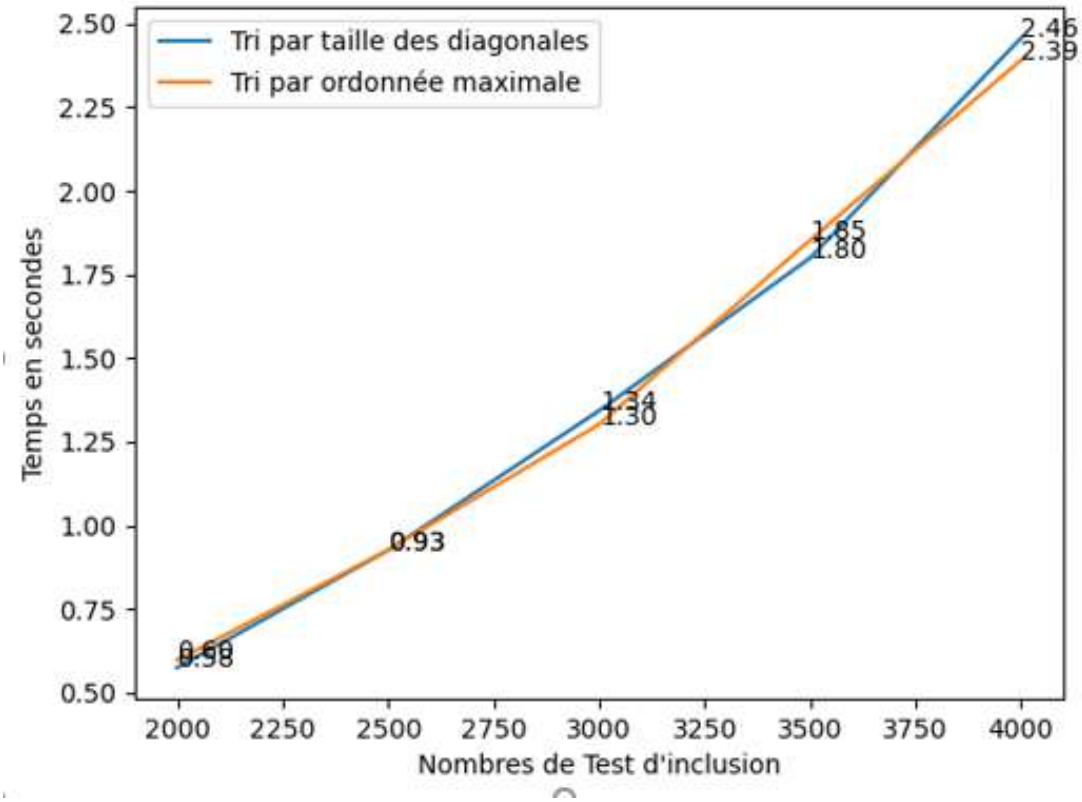
Dans le meilleur des cas, les polygones sont parfaitement imbriqués les uns dans les autres et le premier polygone rencontré sera le bon. Dans ce cas, le nombre de tests d'inclusion à effectuer est de n.

Dans le pire des cas, tous les polygones sont disjoints. Dans ce cas, le nombre de tests d'inclusion à effectuer est de $1 + 2 + 3 + \cdots + (n-1) + n \approx \frac{n^2}{2}$.

En moyenne, on dira qu'on monte d'environ la moitié des polygones restants avant de trouver le bon, soit :

$\frac{1+2+3+\cdots+(n-2)+(n-1)}{2} = \frac{n\times(n+1)}{4} \approx \frac{n^2}{4}$ test d'inclusions.

Cela nous rappelle fortement les complexités de **l'algorithme de tri par taille**. Voici une comparaison expérimentale des deux algorithmes sur les données d'entrées les moins avantageuses auxdits algorithmes.



Les deux algorithmes ont des temps d'exécution quasiment identique, c'est bien ce que nous avions prévu.

5. Une approche plus exotique

La dernière approche à laquelle nous avions pensé est plus originale que les autres et irréalisables en pratique pour une question de mémoire !

L'idée est de discrétiser l'ensembles des polygones afin de placer leurs points dans un tableau 2 dimensions. Malheureusement, le tableau aurait rapidement atteint une taille impraticable dû au fait des nombreux points du plan repère non utilisés.

Cette approche nous aurait permis de nous déplacer dans le tableau comme dans la réalité. Nous ne savons pas précisément comment nous aurions implémenté nos tests d'inclusions sur ce genre de structure mais ce ne sont pas les optimisations qui doivent manquer de ce côté-là.