Projet d'Algo 2020

Nathan Salleyrette et Fabien Vermeulen

Sommaire

Premier Algorithme

Principe:

- on prend les polygones un par un, on notera le polygone en question: poly. On note comme point de départ le tout premier point composant poly (ligne 97 en rouge). C'est arbitraire, on aurait pu prendre un autre point ou un point aléatoire pour les plus joueurs.

- A partir de ce point de départ, on fait appel à la fonction qui détecte les inclusions (ligne 98)

-Cette fonction detection_inclusion est particulière car elle renvoie TOUS les polygones dans lequel notre poly est inclus. Pour être exact, elle renvoie une liste contenant tous les polygones dans lequel poly est inclus

- Il ne restera plus qu'à trier (ligne 100) pour garder uniquement le plus petit polygone dans lequel poly est inclus

```
def main():
    """
    charge chaque fichier .poly donne
    trouve les inclusions
    affiche l'arbre en format texte
    """
    for fichier in sys.argv[1:]:
        polygones = read_instance(fichier)
        inclusions = trouve_inclusions(polygones)
        print(inclusions)

if __name__ == "__main__":
    main()
```

```
86  def trouve_inclusions(polygones):
87      """
88      renvoie le vecteur des inclusions
89      la ieme case contient l'indice du polygone
90      contenant le ieme polygone (-1 si aucun).
91      (voir le sujet pour plus d'info)
92      """
93      nb_poly = -1 # numéro du polygone dont on s'occupe
94      table_des_inclusions = [] # table contenant toutes les inclusions
95      for poly in polygones:
96          nb_poly += 1
97          point_de_depart = Point(list(poly.points[0].coordinates)) # ou sin
98          tab = detection_inclusion(polygones, point_de_depart, nb_poly)
99          table_des_inclusions.append(tab)
100     table_triee = tri_inclusion(table_des_inclusions)
101     return table_triee
```

Premier Algorithme

Détail des fonctions:

-Intéressons-nous à detection_inclusion: on va passer tous les polygones en revue pour savoir s'il contiennent poly. Pour cela, on dispose d'un test (des lignes 30 à 45)

-Ce test va regarder tous les segments de chaque polygone, si la formule mathématique de la ligne 41 est vraie, il va incrémenter de 1 la composante d'indice index_poly de la liste t. A la fin, si cette composante est impaire, cela veut dire que le point de départ est à l'intérieur du polygone numéroté index_poly. On vient donc de trouver un polygone qui contient notre poly.

-Enfin, cette fonction retourne la liste inclus_dans. Cette liste contient les index de TOUS les polygones dans lequel notre poly est inclus. Il faudra trier pour garder le plus petit.

C'est très coûteux en terme de temps de calcul car on regarde chaque segment de chaque polygone. De plus, on réitère cette étape pour chaque polygone. D'un point de vue complexité, on passe en revue les n polygones pour chaque polygone (donc n fois). Ainsi, on peut s'attendre à avoir du $O(n^2)$

```
22 def detection_inclusion(polygones, point, nb_poly):
23     """
24     renvoie tous les polygones dans lequel le polynome actuel (celui du segment)
25     est inclus
26     """
27     index_poly = -1
28     inclus_dans = [] # contient les polynomes dans lequel le polynome actuel (celui du se
29     t = [-1 for _ in range(len(polygones))] # contient le nombre d'intersections de chaqu
30     for poly in polygones:
31         index_poly += 1
32         for segment in poly.segments():
33             point1 = Point(list(segment.endpoints[0].coordinates))
34             point2 = Point(list(segment.endpoints[1].coordinates))
35             x1 = point1.coordinates[0]
36             y1 = point1.coordinates[1]
37             x2 = point2.coordinates[0]
38             y2 = point2.coordinates[1]
39             x = point.coordinates[0]
40             y = point.coordinates[1]
41             if (y1 > y) != (y2 > y) and (x < (x2 - x1)*(y - y1)/(y2-y1) + x1):  # s'il y
42                 t[index_poly] += 1
43     for j in range(len(t)):
44         if t[j] % 2 == 0 and j != nb_poly: #le point est dedans et on verifie que chaque
45             inclus_dans.append(j)
46     if len(inclus_dans) == 0:  # la liste est vide => c'est inclus dans rien
47         inclus_dans.append(-1)
48     return inclus_dans
49
```

Premier Algorithme

<u>Détails des fonctions:</u>

Pour déterminer si un point est inclus dans un polygone, on lance un rayon depuis point_de_départ de coordonnées (x, y) et on compte le nombre de segments du polygone que ce rayon intersecte. Si ce nombre est impair, alors le point est inclus dans le polygone. On peut donc en déduire que le poly auquel ce point appartient est lui aussi inclus dans ce polygone.

Si les coordonnées du point à tester sont (x, y) et qu'on note (x1, y1) (x2, y2) celles des deux extrémités d'un segment. Il y a intersection à deux conditions :

- L'ordonnée du point doit être dans la plage d'ordonnées du segment :

    (y1 > y) != (y2 > y)

- Ensuite le point doit être situé à droite du segment :

    x < ((x2 − x1) × (y−y1) / (y2−y1)) + x1

```python
for poly in polygones:
    index_poly += 1
    for segment in poly.segments():
        point1 = Point(list(segment.endpoints[0].coordinates))
        point2 = Point(list(segment.endpoints[1].coordinates))
        x1 = point1.coordinates[0]
        y1 = point1.coordinates[1]
        x2 = point2.coordinates[0]
        y2 = point2.coordinates[1]
        x = point.coordinates[0]
        y = point.coordinates[1]
        if (y1 > y) != (y2 > y) and (x < (x2 - x1)*(y - y1)/(y2-y1) + x1):  #
            t[index_poly] += 1
```

Premier Algorithme

Détails des fonctions:

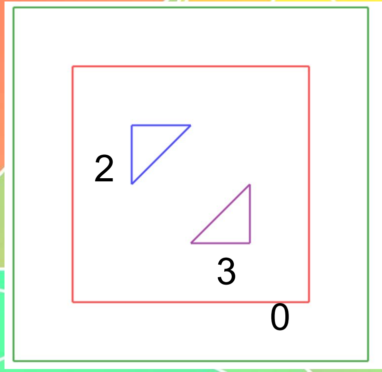
A l’issue de cette étape, on dispose d’une table des inclusions qui est en fait une liste de listes (ligne 99). Il ne reste plus qu’à la trier.

Pour l’exemple donné dans le sujet, on aurait:

table_des_inclusions =[[1], [-1], [0, 1], [0, 1]]

Or pour les 2 dernières listes, on aimerait qu’il reste un seul élément donc on va se servir d’un algorithme de tri (ligne 100)

```
86  def trouve_inclusions(polygones):
87      """
88      renvoie le vecteur des inclusions
89      la ieme case contient l'indice du polygone
90      contenant le ieme polygone (-1 si aucun).
91      (voir le sujet pour plus d'info)
92      """
93      nb_poly = -1 # numéro du polygone dont on s'occupe
94      table_des_inclusions = [] # table contenant toutes les inclusions
95      for poly in polygones:
96          nb_poly += 1
97          point_de_depart = Point(list(poly.points[0].coordinates)) # ou sin
98          tab = detection_inclusion(polygones, point_de_depart, nb_poly)
99          table_des_inclusions.append(tab)
100     table_triee = tri_inclusion(table_des_inclusions)
101     return table_triee
```



Exemple

Premier Algorithme

<u>Détail de l'algorithme de tri:</u>

-Quand notre poly est inclus dans 2 polygones "père", il y a forcément un père qui est inclus dans l'autre donc il suffit de regarder aux indices des pères celui qui contient la valeur -1. C'est le plus "gros" des deux donc on l'oublie. C'est donc l'autre père qui contient notre poly.

Dans les autres cas (n >2):
- Quand notre poly est inclus dans n polygones "pères", il suffit de regarder parmi ses pères, celui qui est inclus dans n - 1 polygones. On oublie toutes les autres valeurs et on garde seulement celle de ce père (c'est ce qu'illustre le test de la ligne 64)

Ce tri retourne tableau_final qui vaut dans le cas de notre exemple: tableau_final = [1, -1, 0, 0]

```python
def tri_inclusion(table):
    """
    un polygone peut etre inclus dans plusieurs
    polygone à la fois. Donc on trie pour garder
    le plus petit polynome dans lequel il est inclus
    """
    tableau_final = [-1 for _ in range(len(table))]
    for i in range(len(table)):
        if len(table[i]) > 1:
            possibilites = table[i] # tableau de longueur > 1 avec tous les
            trouver_petit_polygone(possibilites, i, table, tableau_final)
        else:
            petit_polygone = table[i][0]
            tableau_final[i] = petit_polygone
    return tableau_final
```

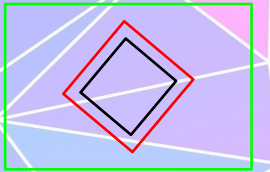
```python
51 def trouver_petit_polygone(tableau, indice, table, tableau_final):
52     """
53     fonction complexe à expliquer
54     mais elle participe au tri
55     (je l'ai testé sur des exemples et elle devrait fonctionner :/ )
56     """
57     if len(tableau) == 2:
58         for i in tableau:
59             if table[i][0] != -1:
60                 petit_polygone = i
61                 tableau_final[indice] = petit_polygone
62     else:
63         for j in tableau:
64             if len(table[j]) == len(tableau) - 1:
65                 petit_polygone = j
66                 tableau_final[indice] = petit_polygone
67
```

Premier Algorithme

Générateurs d'entrée:

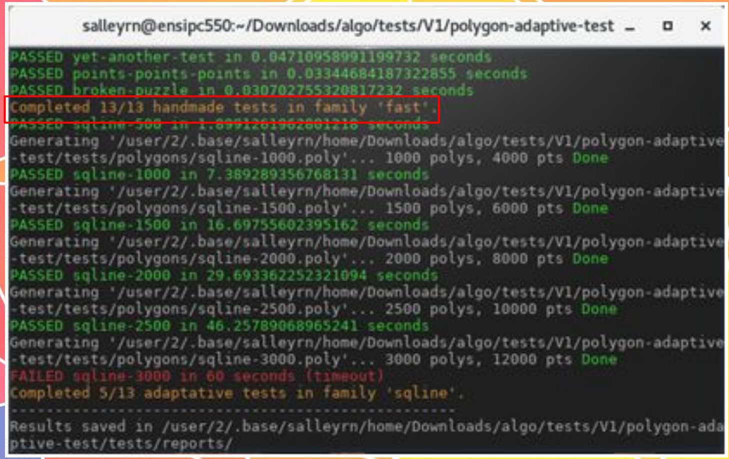
Nous avons testé cet algorithme sur les exemples de Guillaume Raffin.

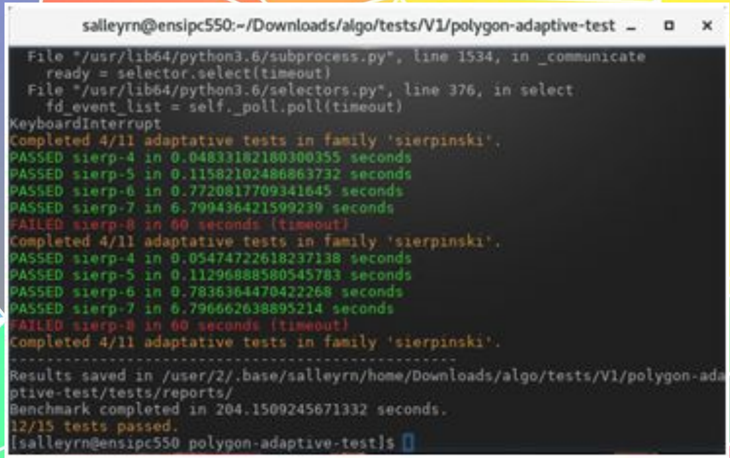
Nous validons les 13 tests de base contenant des figures similaires à celle ci:



On peut aussi visualiser les temps de notre algorithme pour Sqline qui contient des lignes de carrés.

On peut visualiser sur la seconde capture d'écran les temps mesurés pour le test des Triangles de Sierpinski.





Premier Algorithme

Générateurs d'entrée:

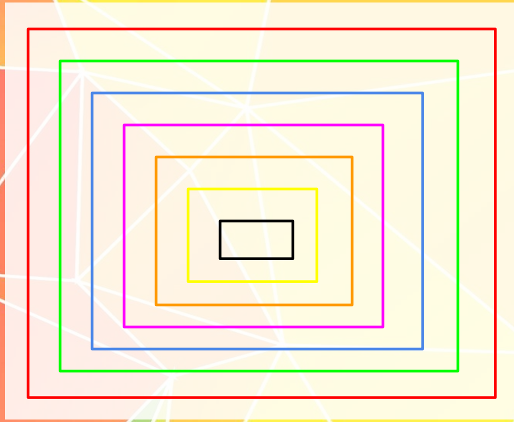
Dans l'optique de mesurer les performances de nos algorithmes, on utilise 2 générateurs de polygones: un pour le meilleur cas et l'autre pour le pire cas.

Pour les mesures dans le meilleur cas, on implémente la configuration 1 avec n polygones.

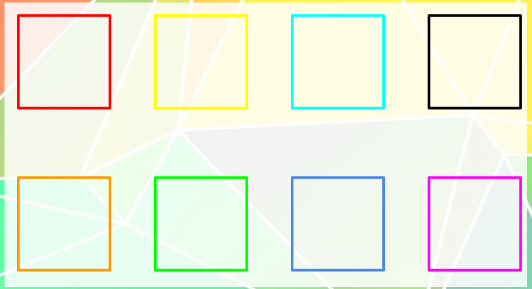
Pour les mesures dans le pire cas, on implémente la configuration 2 avec n polygones.

On va se servir des ces 2 entrées aux slides 11 et 20 pour vérifier nos conjectures sur les complexités des 2 algorithmes.

1



2



Premier Algorithme

Complexité théorique et conjecture

Ce premier algorithme semble très lent. Admettons qu'on ait n polygones. En effet, on sélectionne chaque polygone un par un (n opérations). Pour chaque polygone on passe en revue les n polygones pour voir s'il est inclus dans l'un d'eux. Donc on est sur une complexité en $O(n^2)$ avant de rentrer dans l'algorithme de tri que l'on soit dans le meilleur ou le pire cas.

Passons maintenant au tri:

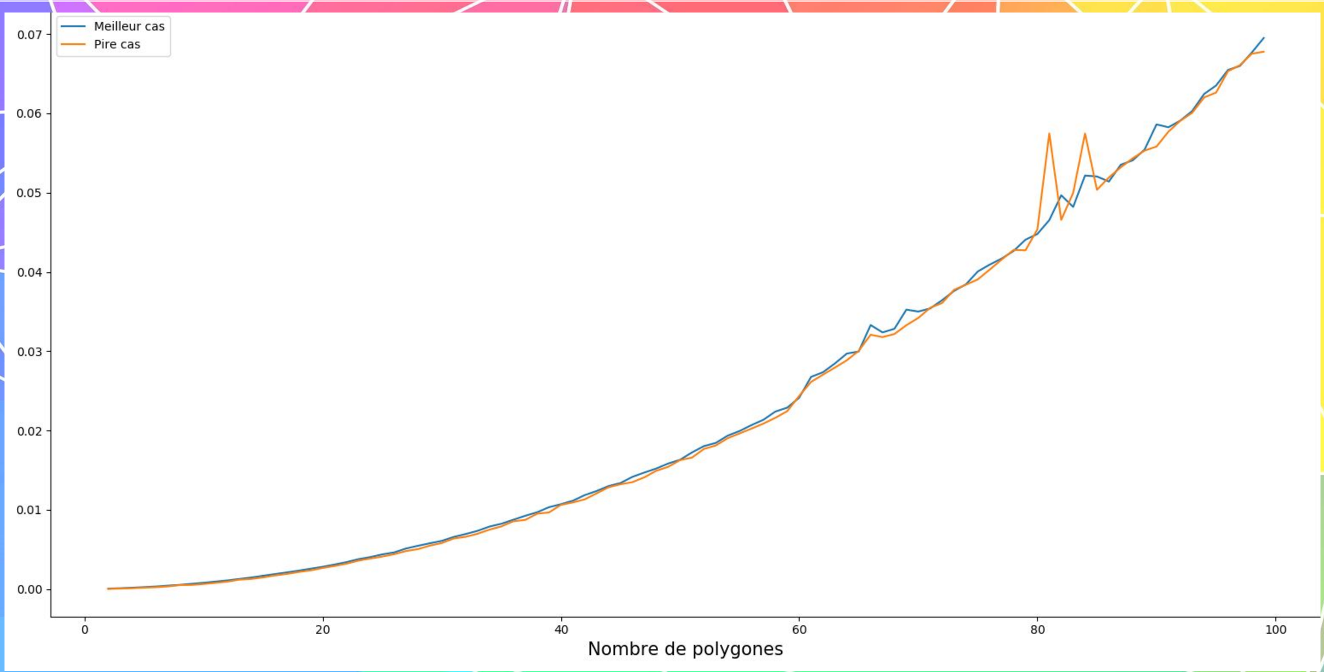
Dans le pire des cas, le polygone 0 est inclus dans les n-1 autres, puis le polygone 1 est inclus dans les n-2 autres, puis le polygone 2 est inclus dans les n-3 autres, etc… L'algorithme va donc passer en revue n-1 listes pour le polygone 0, puis il va passer en revue n-2 listes pour le polygone 1, puis il va passer en revue n-3 listes pour le polygone 2, etc… puis il y aura une opération pour le polygone n-1. La complexité vaut alors $n-1 + n-2 + n-3 + \ldots + 1 = n*(n-1)/2 = O(n^2)$ pour le tri du pire cas.

Tandis que dans le meilleur cas, table_des_inclusions = [[-1], [-1], [-1], … [-1]], il y aura 1 opération pour chaque -1, c'est-à-dire, n opérations en tout. La complexité vaut donc $O(n)$ pour le tri du meilleur cas.

Dans tous les cas, la détection des intersections possède une complexité en $O(n^2)$ donc quelle que soit la complexité du tri, on devrait avoir une complexité globale en $O(n^2)$ pour l'algorithme 1.

Premier Algorithme

<u>Vérifications expérimentales</u>: On mesure la performance en temps pour un nombre de polygones allant de 2 à 100. L'évolution est bien quadratique comme nous l'avons conjecturé.



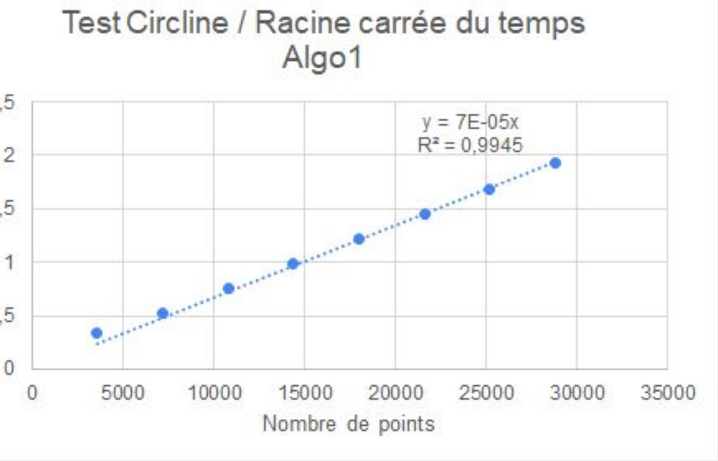
Résultats observés

A partir des tests de Guillaume Raffin, on relève les temps pour Sqline, Sqnest et Circline pour différents nombres de points en entrée. On obtient les courbes suivantes. Elle ressemblent vaguement à du n² mais pour en être sûr on effectue une régression (figure 3) et le coefficient de corrélation confirme notre conjecture

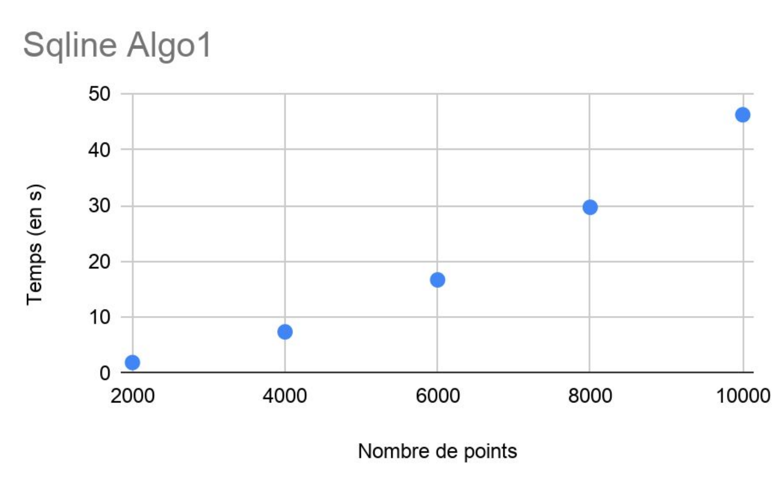
Complexité :

On observe une complexité en O(n²) car la relation avec la racine carrée du temps est linéaire

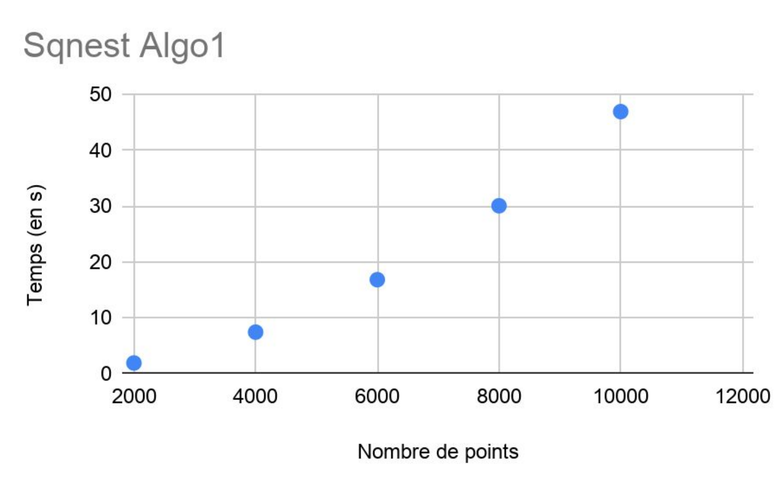
3



1



2



Bilan pour l’Algorithme 1

- Fiable et précis. Nous avons rencontré peu d’erreurs avec ce programme.
- Algorithme très lent. C’est son gros point faible. Pour un grand nombre de points, il est vite largué. De plus on peut voir sur la slide 11 que la durée explose lorsqu’on a beaucoup de polygones en entrée

Second Algorithme : version tri par aire

On souhaite simplifier le premier algorithme en faisant intervenir l'aire de chaque polygone. En effet, un polygone ne peut être inclus dans un autre polygone possédant une plus petite aire, on peut par exemple:

- classer les polygones par aire croissante. Dès qu'on trouve un polygone dans lequel on est inclus, on s'arrête. Si on en trouve aucun, on insère un -1.
- vérifier qu'un des points qui composent le petit polygone se trouve bien à l'intérieur du plus grand.

Second Algorithme

Principe:

- on prend les polygones un par un, on notera le polygone en question: poly. On note comme point de départ le tout premier point composant poly (ligne 66). La fonction trouve_inclusions n'a quasiment pas changé depuis l'algo 1. On a juste enlevé le tri qui est inutile maintenant que l'on classe les polygones par aires.

- A partir de ce point de départ, on fait toujours appel à la fonction qui détecte les inclusions (ligne 67 en rouge)

-Désormais, la fonction detection_inclusion renvoie le polygone avec la plus petite aire dans lequel poly est inclus s'il existe. Sinon, il renvoie -1. On stocke tous ces résultats dans table_des_inclusions.

- Cette fois, table_des_inclusions sera déjà triée et on peut effectuer un print de cette liste, tout se passera bien.

```
def main():
    """
    charge chaque fichier .poly donne
    trouve les inclusions
    affiche l'arbre en format texte
    """
    for fichier in sys.argv[1:]:
        polygones = read_instance(fichier)
        inclusions = trouve_inclusions(polygones)
        print(inclusions)
```

```
55 def trouve_inclusions(polygones):
56     """
57     renvoie le vecteur des inclusions
58     la ieme case contient l'indice du polygone
59     contenant le ieme polygone (-1 si aucun).
60     (voir le sujet pour plus d'info)
61     """
62     nb_poly = -1 # numéro du polygone dont on s'occupe
63     table_des_inclusions = [] # table contenant toutes les inclusions
64     for poly in polygones:
65         nb_poly += 1
66         point_de_depart = Point(list(poly.points[0].coordinates)) # on prend le
67         nb = detection_inclusion(polygones, point_de_depart, poly, nb_poly)
68         table_des_inclusions.append(nb)
69     return table_des_inclusions
```

Second Algorithme

Détail des fonctions:

-Intéressons-nous à detection_inclusion: on va passer tous les polygones en revue pour savoir s'il contiennent poly. Pour cela, on dispose d'un test (des lignes 30 à 45)

-Dans ce second algorithme, on ajoute un test sur les aires (ligne 32) (en rouge) et on effectue les opérations suivantes seulement si ce test est vrai. Cela devrait nous faire gagner du temps de calcul. En dehors de ça, le principe de ce second algorithme est le même que pour le premier. On stocke juste l'aire minimale aux lignes 45 et 48 (en bleu)

-Enfin, cette fonction retourne la liste inclus_dans. Cette liste contient l'index du polygone avec la plus petite aire dans lequel notre poly est inclus. Plus besoin de trier en sortie de cette fonction !

```
22 def detection_inclusion(polygones, point, poly, nb_poly):
23     """
24     renvoie le polygone avec la plus petite aire dans lequel le polynome actuel (celui
25     est inclus
26     """
27     index_poly = -1
28     inclus_dans = [] # contient le plus petit polygone dans lequel le polynome actuel (
29     for poly2 in polygones:
30         index_poly += 1
31         nb_intersect = 0
32         if abs(poly2.area()) > abs(poly.area()): # on teste seulement les polygones ave
33             for segment in poly2.segments(): # on teste chaque segment de poly2
34                 point1 = Point(list(segment.endpoints[0].coordinates))
35                 point2 = Point(list(segment.endpoints[1].coordinates))
36                 x1 = point1.coordinates[0]
37                 y1 = point1.coordinates[1]
38                 x2 = point2.coordinates[0]
39                 y2 = point2.coordinates[1]
40                 x = point.coordinates[0]
41                 y = point.coordinates[1]
42                 if (y1 > y) != (y2 > y) and (x < (x2 - x1)*(y - y1)/(y2-y1) + x1):  # s
43                     nb_intersect += 1
44             if nb_intersect % 2 == 1 and len(inclus_dans) == 0: # si on est inclus dans
45                 min_area = abs(poly2.area()) # l'aire minimale
46                 inclus_dans.append(index_poly)
47             elif nb_intersect % 2 == 1 and abs(poly2.area()) < min_area: # si on est de
48                 min_area = abs(poly2.area())
49                 inclus_dans[0] = index_poly
50     if len(inclus_dans) == 0: # si on est inclus dans aucun polygone
51         inclus_dans.append(-1)
52     return inclus_dans[0]
```

Second Algorithme

Détails des fonctions:

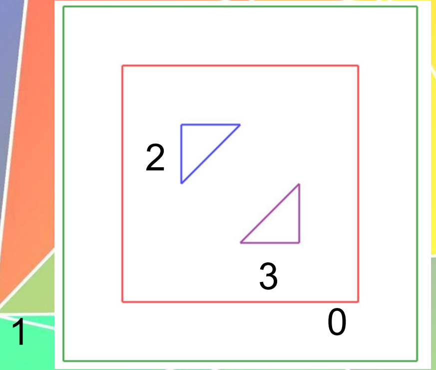
A l’issue de cette étape, on dispose d’une table des inclusions complète que l’on peut retourner (ligne 68 en rouge et 69)

Pour l’exemple donné dans le sujet, on aurait:

table_des_inclusions = [1, -1, 0, 0]

La table est déjà triée. On devrait donc gagner du temps de calcul mais nous allons vérifier cela expérimentalement.

```
54
55 def trouve_inclusions(polygones):
56     """
57     renvoie le vecteur des inclusions
58     la ieme case contient l'indice du polygone
59     contenant le ieme polygone (-1 si aucun).
60     (voir le sujet pour plus d'info)
61     """
62     nb_poly = -1 # numéro du polygone dont on s'occupe
63     table_des_inclusions = [] # table contenant toutes les inclusions
64     for poly in polygones:
65         nb_poly += 1
66         point_de_depart = Point(list(poly.points[0].coordinates)) # on prend le t
67         nb = detection_inclusion(polygones, point_de_depart, poly, nb_poly)
68         table_des_inclusions.append(nb)
69     return table_des_inclusions
70
71
```



Exemple

Second Algorithme

<u>Générateurs d'entrée (Rappel) :</u>

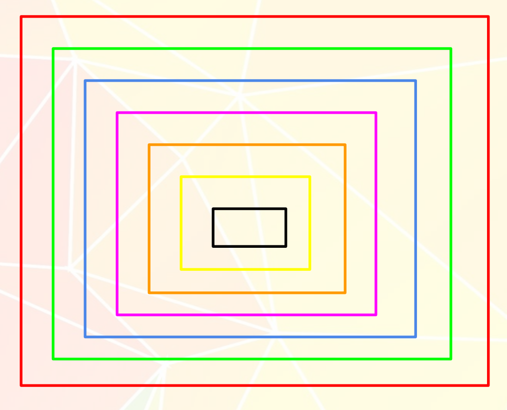
On garde les 2 générateurs de polygones suivants: un pour le meilleur cas et l'autre pour le pire cas.

Pour les mesures dans le meilleur cas, on implémente la configuration 1 avec n polygones.

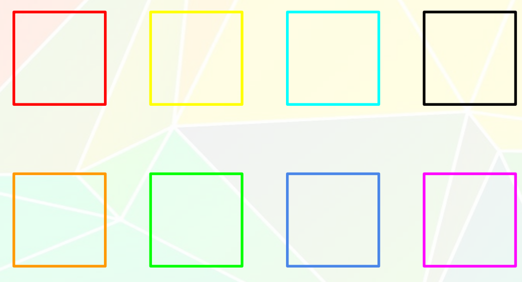
Pour les mesures dans le pire cas, on implémente la configuration 2 avec n polygones.

On se sert des ces 2 entrées aux slides 11 et 20 pour vérifier nos conjectures sur les complexités des 2 algorithmes.

1



2



Second Algorithme

Complexité théorique et conjecture

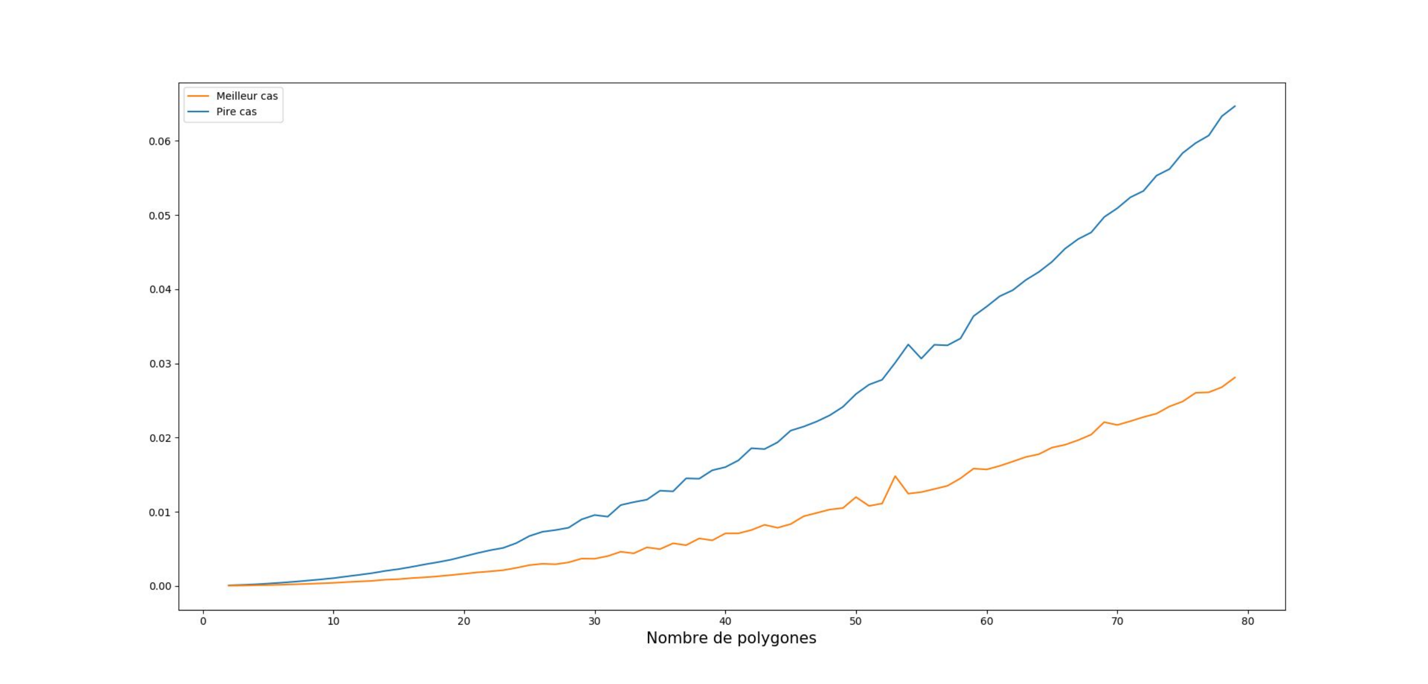
Ce second algorithme semble plus rapide que le précédent. Admettons qu'on ait n polygones composés respectivement de i1,...in segments (dans les tests de Guillaume Raffin le nombre de segments vaut 4 pour les tests avec des carrés (sq) et 360 pour les tests avec des cercles (circ))

On prend cet ensemble de n polygones. Dans ce cas, l'algorithme va tester l'inclusion de chaque polygone dans tous les polygones ayant une aire supérieure à la sienne. En tout, le nombre total de tests vaudrait (i2 + … + in) + (i3 + … + in) + … + in. On peut simplifier en i2 + 2i3 + 3i4 + … (n - 1)*in . On peut encore simplifier cette expression dans le cadre de nos exemples car tous les polygones ont le même nombre de segments: i1 = i2 = i3 = … = in = C. On obtient alors un coût qui vaut C*n*(n-1)/2 ce qui représente une complexité en $O(n^2)$.

Ca alors ! La complexité asymptotique serait la même que celle du premier algorithme ? Vérifions tout cela avec des expériences pour en avoir le coeur net.

Second Algorithme

<u>Vérifications expérimentales</u>: On mesure la performance en meilleur cas et pire cas pour un nombre de polygones allant de 0 à 80. L'évolution est quadratique comme conjecturé. On remarque que le pire cas explose pour des grands valeurs de polygones.



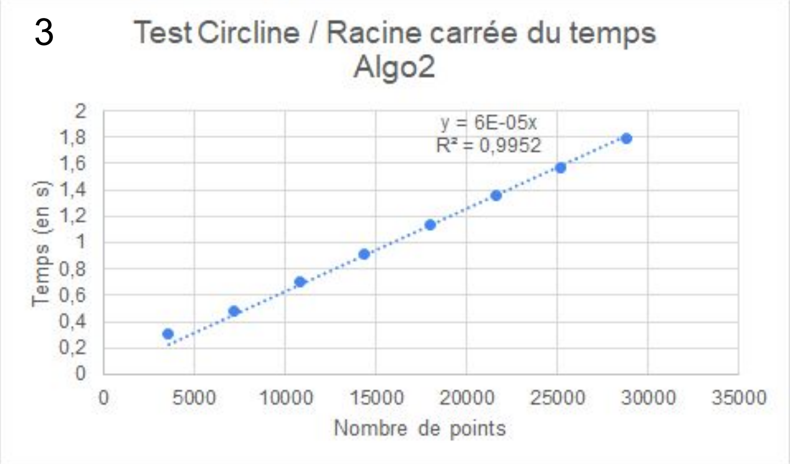
Résultats observés

A partir des mêmes tests de Guillaume Raffin, on relève les temps pour Sqline, Sqnest et Circline pour différents nombres de points en entrée. On obtient des courbes semblables à celles de l'algo 1 mais si on note les valeurs on remarque que l'algo 2 est plus rapide. On vérifie dans la figure 3 qu'on a bien affaire à du O(n²)

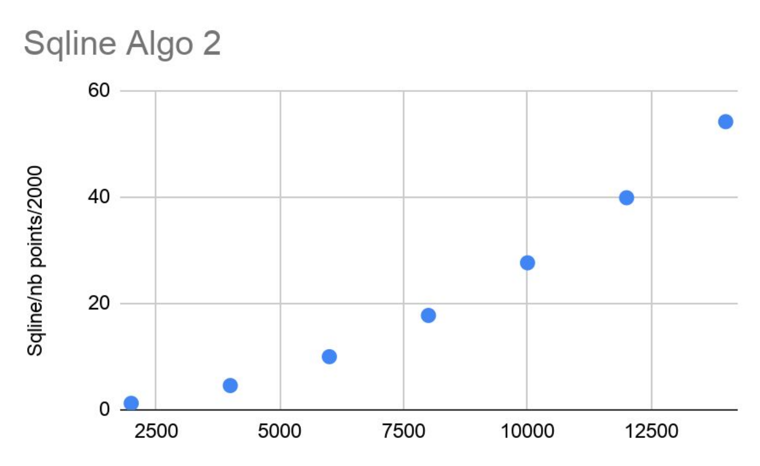
Complexité :

On observe une fois de plus une complexité en O(n²) car la relation avec la racine carrée du temps est linéaire

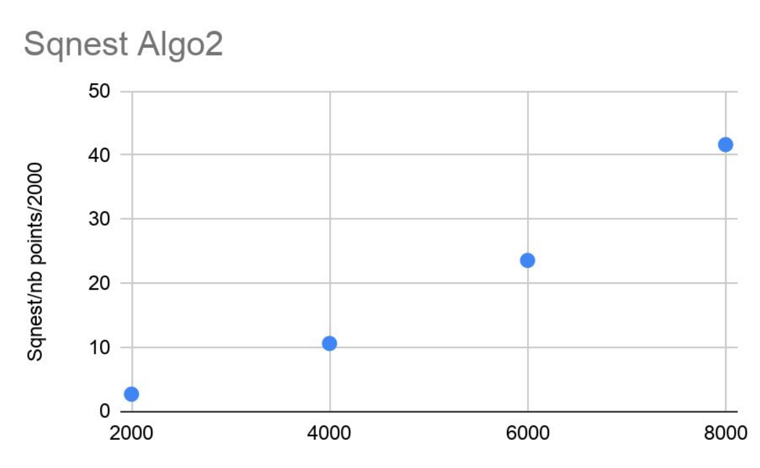
3



1



2



Comparaison des performances des 2 algorithmes

Algo 1	Sqline		
	nb points	nb poly	Temps (s)
	2000	500	1,8991
	4000	1000	7,3893
	6000	1500	16,6976
	8000	2000	29,6934
	10000	2500	46,2579
	12000	3000	timeout
	Sqgrid		
	nb points	nb poly	Temps (s)
	6400	1600	18,2903
	10000	2500	45,1361
	14400	3600	timeout

Algo 2	Sqline		
	nb points	nb poly	Temps (s)
	2000	500	1,1453
	4000	1000	4,4882
	6000	1500	9,9139
	8000	2000	17,6712
	10000	2500	27,5664
	12000	3000	39,8012
	14000	3500	54,1068
	16000	4000	timeout
	Sqgrid		
	nb points	nb poly	Temps (s)
	6400	1600	11,332
	10000	2500	27,6157
	14400	3600	56,8564

On remarque que le second algorithme est plus performant que le premier sur les tests Sqline et Sqgrid

Bilan pour l’Algorithme 2

- Plus rapide que l’algorithme 1. Large gain de temps grâce au tri par aires.
- Un peu moins fiable que l’algorithme 1 (Sûrement à cause des calculs d’aires ou des arrondis). Nous avons rencontré certaines erreurs que nous n’avions pas eu avec l’algorithme 1.

Conclusion globale

Durant ce projet, nous avons réalisé un premier programme pour répondre à la problématique. Puis nous avons cherché les améliorations possibles. C'est pourquoi, nous avons perfectionné notre premier algorithme en utilisant le tri par aire afin de gagner du temps de calcul. De plus, nous avons calculé les complexités de ces deux algorithmes qui sont toutes les deux de $O(n^2)$. Nous avons mesuré expérimentalement les performances de nos algorithmes et nous avons vérifié nos conjectures grâce à plusieurs tests. On remarque que le second algorithme est bien plus rapide que le premier. Nous avons même calculé que le second algorithme allait 1,67 fois plus vite que le premier algorithme sur certains tests.

Problèmes rencontrés et remerciements

<u>Problèmes rencontrés :</u>

- Dans un premier temps, nous avions utilisé la méthode “intersection_with” du fichier segment.py appartenant au module geo (qui est utilisée dans le CM3) car un exemple était donné en commentaire (ligne 23). Cependant, elle n’était pas définie. Donc, le test0 échouait mais nous ne savions pas pourquoi.
- Le travail à distance était handicapant. Nous avons effectué tous les tests par machine virtuelle et c’était très lent. (De plus, Fabien habite en campagne profonde…)

Un grand merci à Guillaume Raffin pour ses tests et aux Bug Busters de manière générale pour leur aide face aux difficultés créées par le travail à distance.