Boulhol Elodie
Eudeline Simon

Rapport du projet d'algo

Sommaire

I. Les différentes fonctions utilisées

1) Un mot sur les structures de données utilisées

Dans un premier, il nous a paru plus simple de travailler avec des structures de données différentes de celles fournies pour travailler sur les polygones. Nous avons décidé de représenter un polygone comme un vecteur de vecteurs à deux éléments (qui contiennent les coordonnées des points du polygone).
Cela nous à mener à créer la fonction suivante :

`vecteur_polygone(nom_fichier)` : Cette fonction est l'équivalent de read_instance mais adaptée à notre façon de représenter un polygone.

Dans un second temps, nous avons utilisé la structure de données Polygon fournie, notamment pour utiliser la méthode area(self), c'est pourquoi presque toutes nos fonctions ont deux versions : une adaptée à la première structure de données, et une adaptée à la deuxième structure.

2) Fonctions auxiliaires

Voici la liste de ces fonctions auxiliaires que nous avons créées:

`vecteur_polygone(nom_fichier)` : Prend en entrée le nom du fichier d'entrée et convertit ce fichier en un vecteur de polygones suivant la méthode présentée ci-dessus.

Les fonctions suivantes sont les fonctions les plus utilisées dans les algorithmes principaux :

`isLeft(segment, point) / isLeft2(segment, point)` : Prennent en entrée segment (vecteur constitué de 2 points) et un point, et renvoient True si le point est à gauche de la droite obtenue en prologeant le segment, False sinon.
Complexité : O(1)

`coupe_segment(segment, point) / coupe_segment2(segment, point)` : Prennent en entrée un segment et un point et renvoient True si la demi-droite horizontale partant de point et allant vers la droite coupe le segment, False sinon.
Complexité : O(1)

`inclusion_point(polygone, point) / inclusion_point2(polygone, point)` : Prennent en entrée un polygone et un point et renvoient True si le point est inclu dans le polygone, False sinon.
Complexité : O(k) avec k le nombre de points du polygone.

`aire_polygones(polygones)` : Prend en entrée un vecteur de polygones et retourne un vecteur tel qu'à l'indice i on ai un vecteur contenant : l'indice i, l'aire du polygone d'indice i ainsi que le polygone d'indice i.

Pour les deux fonctions suivantes, il existe à chaque fois deux versions : une version récursive et une version itérative, car Python n'est pas très adapté pour travailler en récursif et cela peut poser des problèmes sur des gros fichiers.

`insere(self, polygones, num_polygon, aire_poly, polygon) / insere_rec(self, polygones, num_polygon, aire_poly, polygon)` : Ce sont deux méthodes de la classe Noeud. Elles prennent en entrée l'arbre des polygones ainsi qu'un polygone (ainsi que leur numéro et leur aire) et insèrent ce polygone à sa place (il doit être le fils du polygone dans lequel il est inclus).

`complete_vect_inclu(pere, node, vect_inclusions / complete_vec(pere, node, vect_inclusions)` : Complètent le vecteur d'inclusions à partir du nœud et de l'indice du père donnés en entrée.

<u>Les fonctions suivantes ne sont pas utilisées dans les algorithmes principaux, mais servent pour calculer les performances de ces algorithmes :</u>

`generateur_fichier(nom_fichier, nb_poly)` : Génère `nb_poly` carrés dont le premier contient tous les autres, le deuxième tous les autres sauf le premier, etc. Ceci est stocké dans `nom_fichier`.

`generateur_polygones(nb_poly)` : Génère `nb_poly` carrés dont le premier contient tous les autres, le deuxième tous les autres sauf le premier, etc, et stock ça dans un vecteur.

`tracage_courbe{1...8}()` : Trace une courbe de performance du temps en fonction du nombre de polygones utilisés (utilise `generateur_polygones`). Cette fonction se trouve dans courbe.py.

`chrono(func, polygones)` : Chronomètre le temps d'exécution de func(polygones).

II. Les différents algorithmes

1) Descriptions des algorithmes

Les trois premiers algorithmes qui vont être présentés utlisent la première structure de données pour représenter les polygones, tandis que les deux derniers utilisent la structure de données Polygon fournie dans le dossier geo.

- `trouve_inclusions` : Notre premier algorithme `trouve_inclusions` est un algorithme naïf et très lent.
  Le principe est dans un premier temps d'ajouter dans un vecteur, pour chaque polygone, tous les polygones dans lequel il est inclus. Dans un second temps, on s'occupe de choisir lequel de ces polygones il faut garder, pour chaque polygone.
  La complexité en pire cas est un **O(k * n^3)** avec k le nombre maximal de points dans un polygone (dû aux appels à `inclusion_point`) et n le nombre de polygones.

- `trouve_inclusions2` : Ce deuxième algorithme est déjà moins naïf, sans pour autant être très optimisé.
  Le principe de déterminer en temps réel, pour chaque polygone, dans quel polygone il est inclus directement. Pour un polygone fixé, on utilise un booléen `appartient_deja` qui indique si on connaît un polygone dans lequel il est inclus. Si il appartenait déjà à un polyogne, et qu'on trouve un nouveau polygone dans lequel il est inclus, alors on fait un test d'inclusion entre les deux pour déterminer dans lequel le polygone initial est inclus directement.
  La complexité en pire cas est cette fois un **O(k * n^2)** avec k le nombre maximal de point dans un polygone et n le nombre de polygones.

- `trouve_inclusions3` : Cet algorithme est presque identique à trouve_inclusions2. Cependant, dans cet algorithme nous avons essayé de diminuer le nombre de tours de la deuxième boucle for en testant l'inclusion des polygones ‘‘dans les deux sens’’ et ainsi ne pas avoir à parcourir certains polygones.
  Cela ne s'est pas avéré être une bonne idée car nous avons dû multiplier le nombre d'appels à `inclusion_point` dans la deuxième boucle for, ce qui compense le nombre de tours de boucle économisés.
  La complexité en pire cas est encore une fois un **O(k * n^2)** avec k le nombre maximal de point dans un polygone et n le nombre de polygones. Cette complexité ne changera pas pour tout les autres programmes mais cela reste une complexité pire cas, qui ne met pas forcément en évidence que certains programmes sont plus optimisés que d'autres.

- `trouve_inclusions4` : Cet algorithme est le premier algorithme à utiliser la structure Polygon fournie. Nous avons utilisé cette structure pour utiliser la méthode area, et ainsi réaliser un tri fusion sur les aires de la liste de polygones.

Nous créons dans un premier temps `vect_aires`, qui est un vecteur contenant des vecteurs à trois éléments `[numéro du polygone, aire du polygone, polygone]`, qui est trié de manière décroissante sur l'aire, avec les polygones de plus grande aire en premier jusqu'aux polygones de plus petite aire.
Pour chaque polygone parcouru, il n'est pas utile de parcourir les polygones d'aire inférieure, donc on se contente de parcourir les polygones qui sont ''à sa gauche'' et dès qu'on en trouve un dans lequel il est inclus, il est nécessairement inclus directement dedans et il n'est alors pas utile de continuer à parcourir les autres polygones de plus grande aire.

- `trouve_inclusions5 :` Nous commencions à être à court d'idée pour optimiser nos algorithmes, nous avons donc essayé de concevoir une autre façon de représenter les inclusions entre polygones et un arbre est ce qui nous a paru le plus naturel. Chaque fils d'un nœud traduit l'inclusion de ce fils dans le nœud. Il fallait également utilisé le tri par aire qui a bien amélioré nos performances, c'est pourquoi dans les nœuds de l'arbre nous avons choisi de stocker le numéro du polygone et son aire.
  Il suffit d'insérer chaque polygone à sa place dans l'arbre et de lui fixer -1 comme racine. Pour chaque polygone, le polygone dans lequel il est inclus directement est alors son père. Nous convertissons ensuite cet arbre en vecteur d'inclusions.
  Les fonctions pour insérer un nouvel élément dans l'arbre et compléter le vecteur d'inclusions sont récursives et cela peut poser des problèmes si le fichier est trop volumineux.

- `trouve_inclusions6 :` C'est le même principe que `trouve_inclusions5` mais uniquement avec des fonctions itératives.

2) L’algorithme principal

L’algorithme que nous avons décidé d’améliorer le plus possible et que l’on considère comme notre meilleur algorithme est `trouve_inclusions4`, nous allons donc le décrire un peu plus en détail dans cette partie.

Comme dit précédemment, on commence par effectuer un tri sur les aires, puis on calcule le quadrant de chaque polygone pour pouvoir avoir accès à cette valeur dans la suite du programme.

La variable `saut` sert à traiter le cas particulier des polygones de même aire (qui est un cas très particulier nous en avons conscience), elle permet de sauter directement au premier polygone d’aire strictement plus grande que celle du polygone que l’on parcourt. Ainsi on est capable de trouver le vecteur d’inclusions de `sqline-50000.poly` (50000 carrés de même aire alignés) en moins de 2 secondes.

Dans la boucle while, on s’assure de tester l’inclusion du polygone parcouru uniquement avec les polygones suffisament proches de lui, c’est-à-dire, les polygones dont le quadrant s’intersecte avec son quadrant. Dès que le polygone est inclus dans un autre polygone, on sort de la boucle while car c’est nécessairement le polygone dans lequel il est inclus directement grâce au tri sur les aires.

III. Mesures de performances

1) Test via generateur_polygones

Pour cette première mesure de performances, nous utiliserons la fonction `generateur_polygones` décrite dans la partie I.2). Elle donne une bonne première idée de l'efficacité des algorithmes mais a cependant l'inconvénient de traiter un cas plutôt particulier (vecteur d'inclusions correspondant à `range(-1, nombre polygones – 1)`).

Ici, le tracé des résultats pour 100, 200, 300 et 400 carrés (Figure 1).

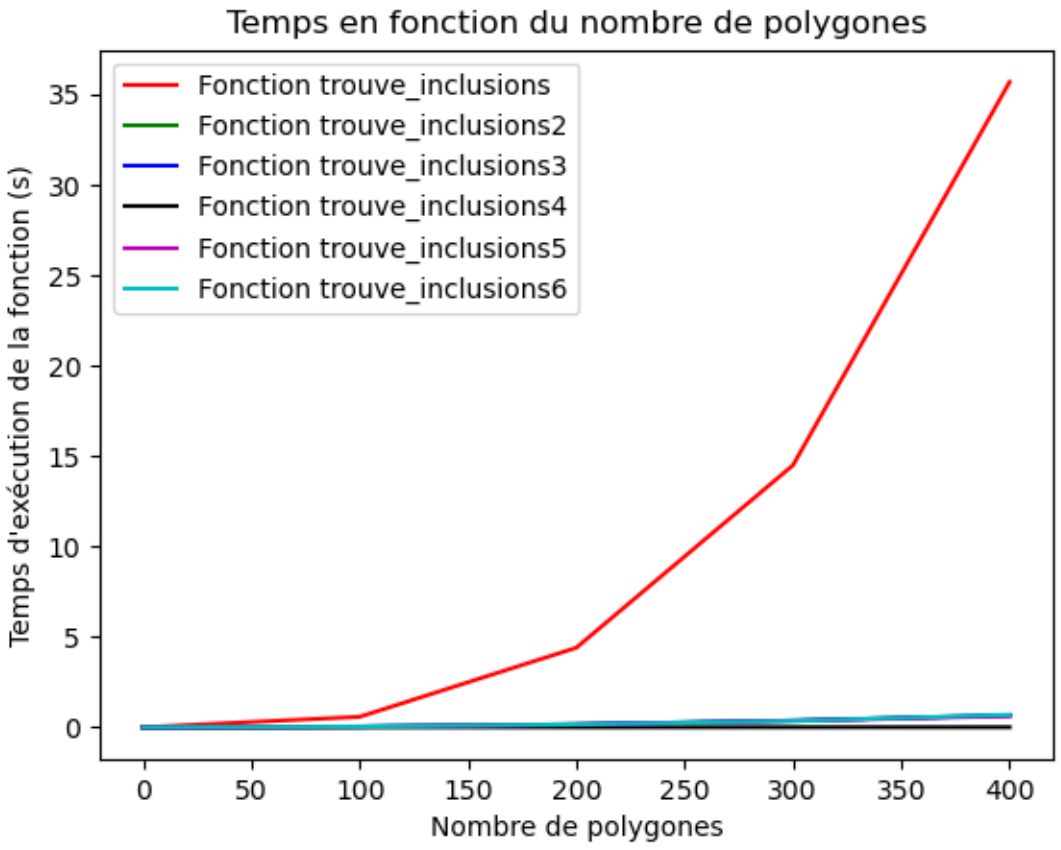

Figure 1

A partir de maintenant, nous excluerons `trouve_inclusions` des tests car il est beaucoup trop long à s'exécuter.

Voici le tracé des résultats pour un nombre de carrés allant de 0 à 2000 (Figure 2).

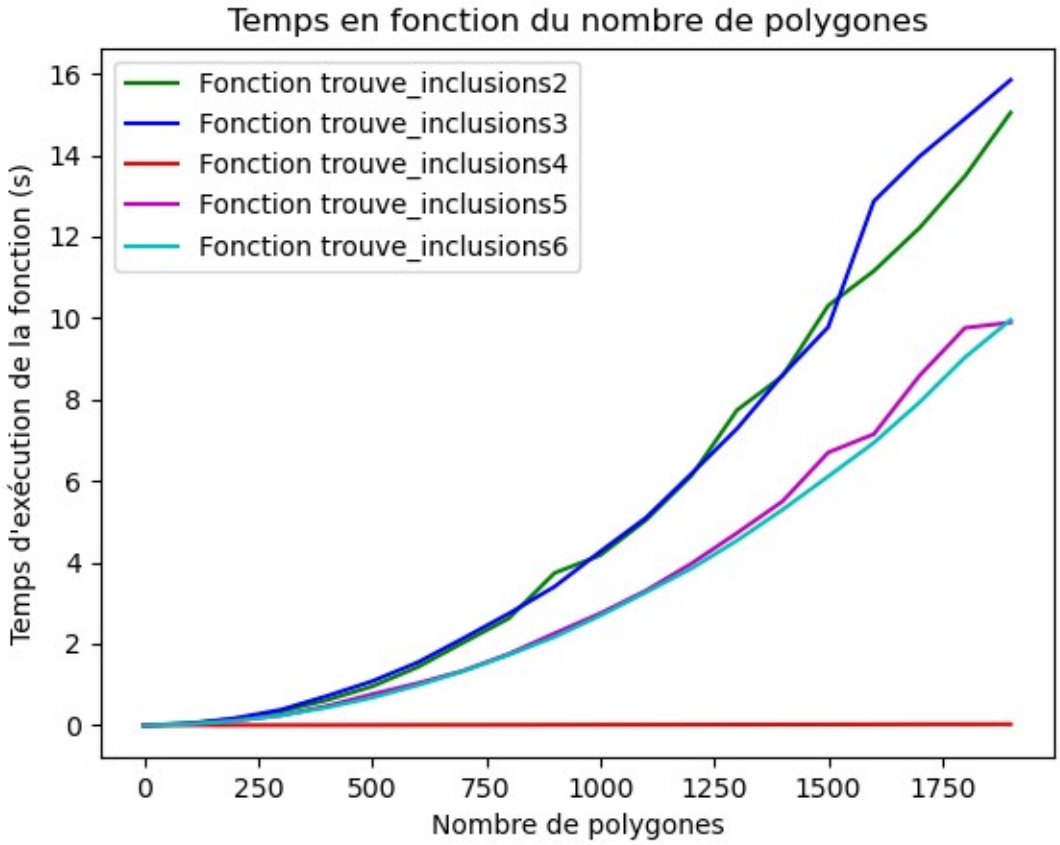

Figure 2

Ce premier test est un cas particulier de fichier qui joue en la faveur certains algorithmes, il faut donc tester nos algorithmes avec d'autres fichiers.

2) Test sur différents fichiers

Pour ces prochains tests, nous allons majoritairement utiliser ceux mis en ligne par Guillaume Raffin. Pour le test suivant, nous avons utilisé `sqline` appliqué aux fonctions `trouve_inclusions{2...6}`. (Figure 3).

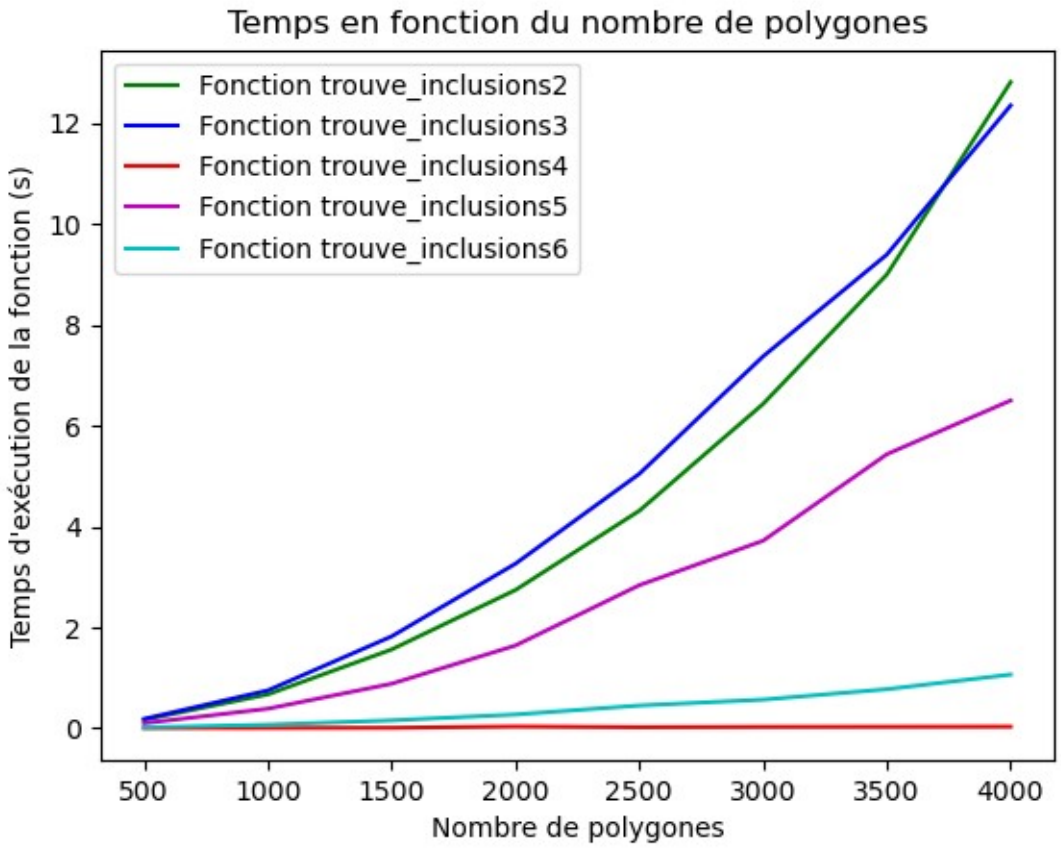

Figure 3

On se rend alors compte que ce sont les fonctions `trouve_inclusions4` et `trouve_inclusions6` qui réalisent les meilleurs temps.

On décide alors de réexécuter le même test seulement pour ces deux fonctions, et pour un nombre de polygones bien plus important (Figure 4).

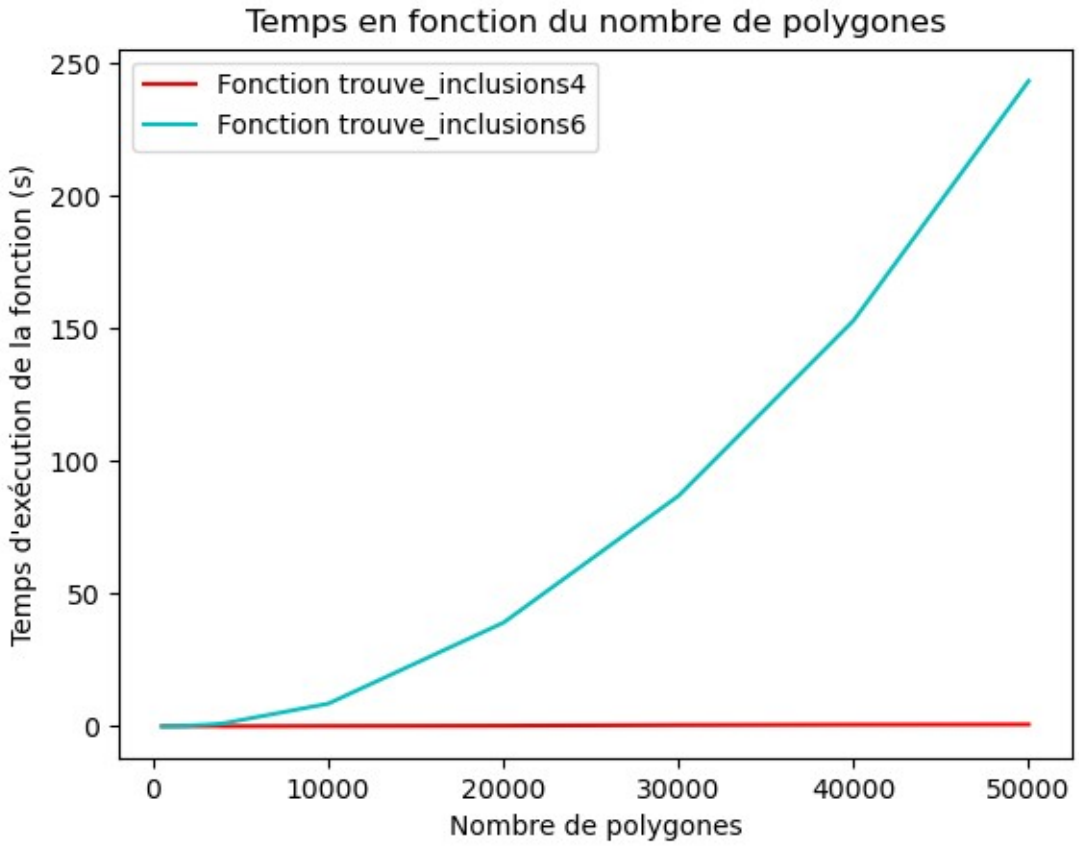

Figure 4

Ce test nous a donc permis de comparer ces deux fonctions. Ainsi, on se rend compte qu'il y a une très grande différence concernant le temps d'exécution des deux fonctions. Pour conclure sur ce test, c'est la fonction `trouve_inclusions4` qui traîte le plus rapidement (et de loin) une ligne de carré. En d'autres termes, nous avons réussi à bien optimiser `trouve_inclusions4` dans ce cas particulier.

Pour ce prochain test, nous allons suivre le même schéma que précédemment mais cette fois sur le test `sqgrid` (Figure 5).

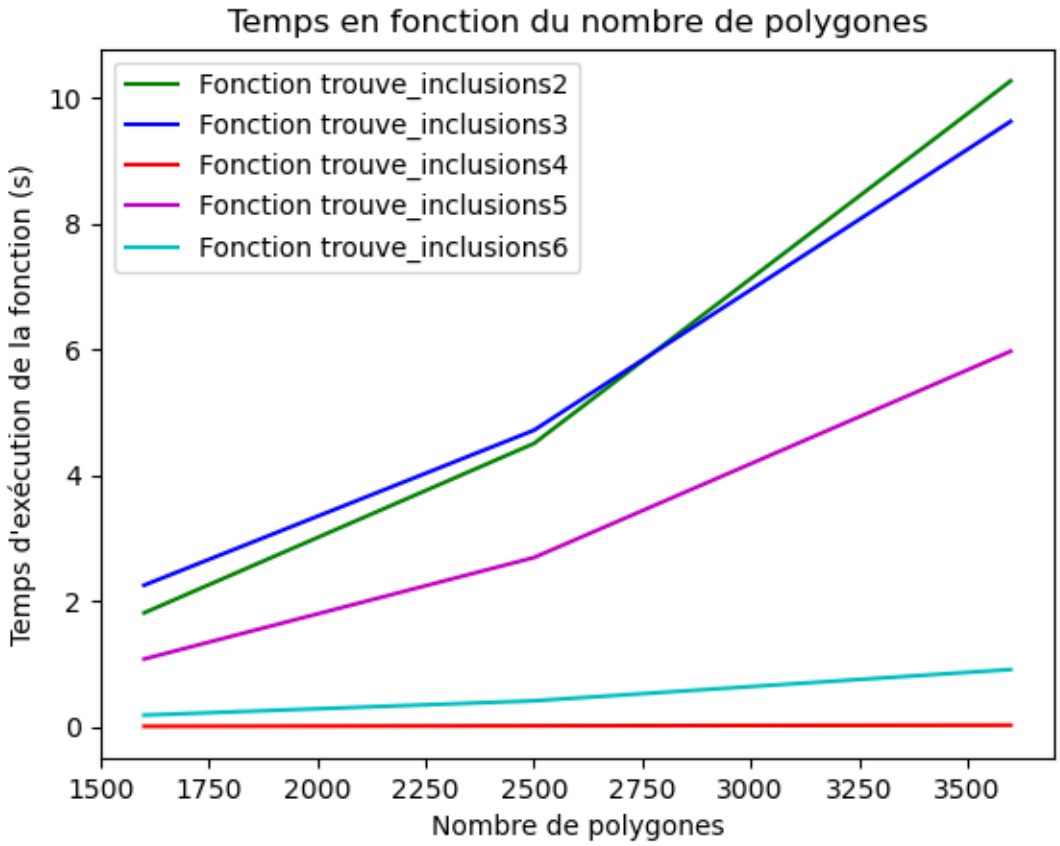

Figure 5

Une fois de plus, les fonctions `trouve_inclusions4` et `trouve_inclusions6` se démarquent des autres fonctions. En effet, les autres fonctions sont très lentes car elles mettent déjà 10 secondes pour 3500 polygones. Cela pourrait ne paraître pas si long, cependant il s'agit ici de cas simples puisque l'on a une grille de carrés de même aire.

Voici alors ce que donne le même test, pour les fonctions `trouve_inclusions4` et `trouve_inclusions6` pour un nombre plus important de polygones (Figure 6).

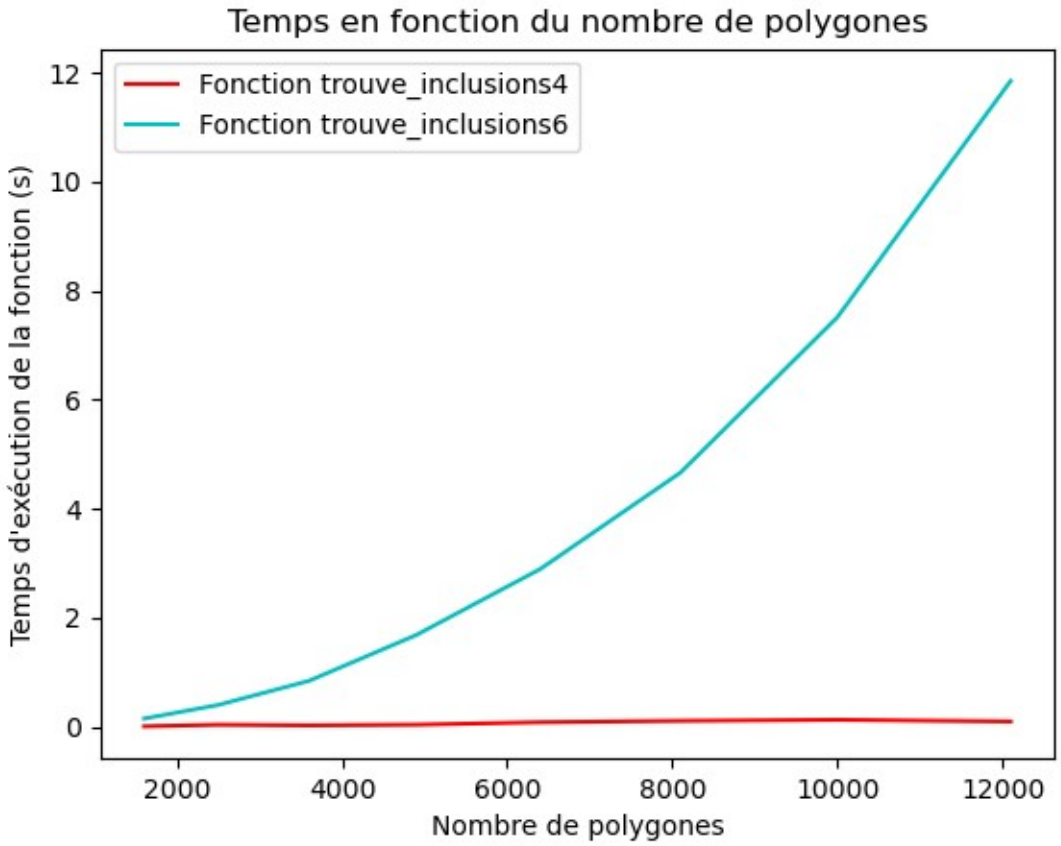

Figure 6

On se retrouve dans le même cas que pour le test `sqline`. On a une très grande différence de temps d'exécution et notre fonction `trouve_inclusions4` est bien plus efficace que les autres pour gérer une grille de carrés de même aire. C'est un autre cas particulier pour lequel nous avons bien optimisé `trouve_inclusions4`.

Pour le test suivant, il s'agit de l'exécution de `circgrid`. Lors de ce test, nous testons l'efficacité de nos fonctions pour le traîtement d'une grille de cercles (Figure 7).

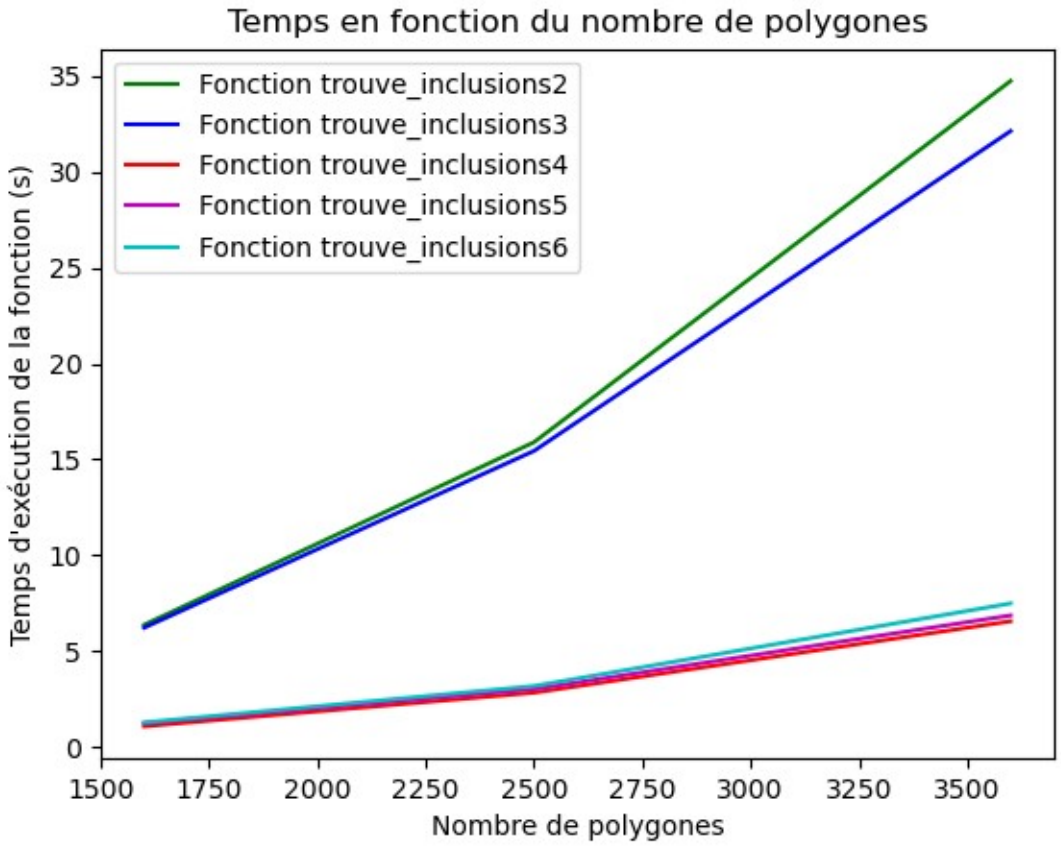

Figure 7

On en conclut alors deux choses différentes :

- Tout d'abord, dans toutes nos fonctions, gérer l'inclusion d'un cercle est plus longue que l'inclusion d'un carré. En effet on triple au moins de le temps d'exécution de chaque fonction, excepée `trouve_inclusions5.` Cette fonction est donc plus efficace lorsque les polygones sont des cercles.

- Ensuite, c'est le troisième test pour lequel les fonctions `trouve_inclusions2` et `trouve_inclusions3` sont très lentes et pas du tout efficaces. On en conclut donc qu'elles ne sont clairement pas optimisées.

Pour finir, nous avons passé le test `sierp`. Dans ce test, les polygones sont des triangles tous inclus dans un grand triangle. On décide de tracer sur ce graphique le temps d'exécution en fonction du niveau de récursion (Figure 8).

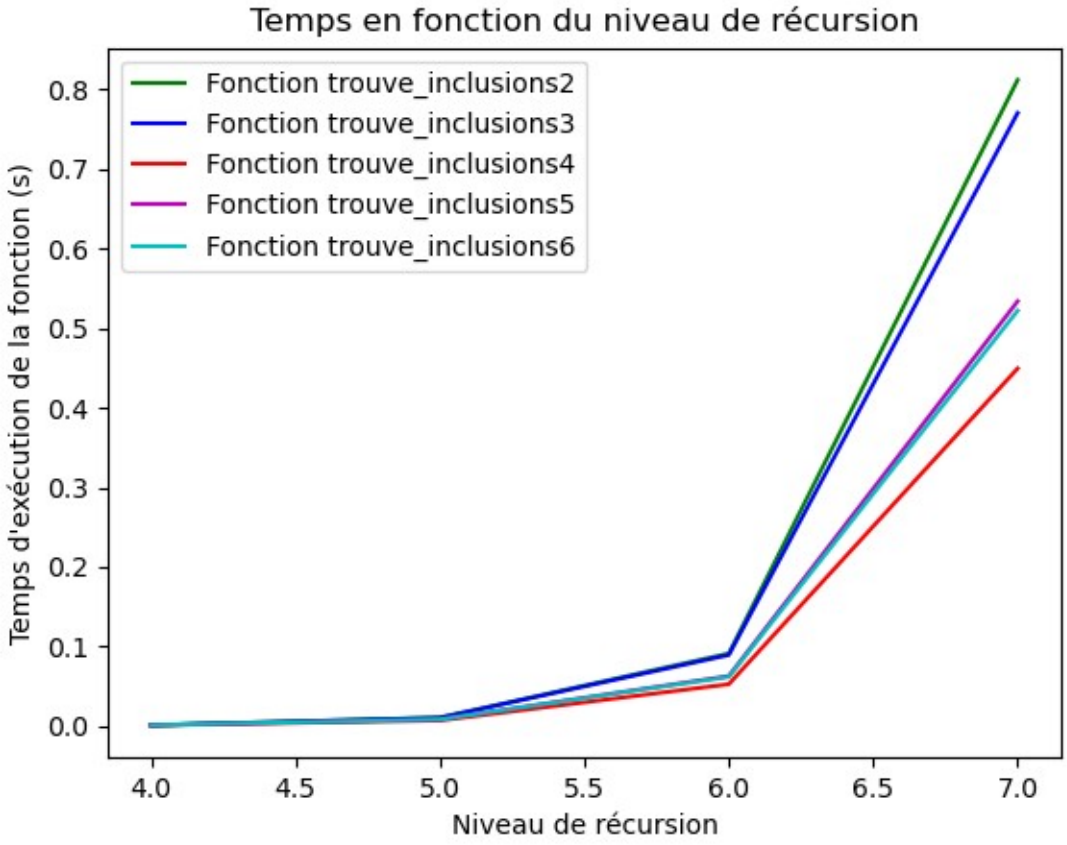

Figure 8

On retrouve ici l'ordre « habituel » des fonctions, ce qui nous permet de conclure sur le fait que la fonction que nous avons réussi à le plus optimiser est la fonction `trouve_inclusions4` (et heureusement puisque c'est celle que nous avons le plus travaillé). En revanche les fonctions `trouve_inclusions{2, 3}` sont à banir car beaucoup trop lentes.

IV. Idées non réalisées

1) Quadrillage de l'espace

Une idée visant à améliorer le fonctionnement de `trouve_inclusions4` est d'effectuer un pré-traitement des données pour réduire le nombre de polygones à tester pour chaque polygone. Le tri par les aires fait parti de ce pré-traitement, mais nous avions également pensé divisé l'espace en petites cases contenant un nombre réduit de polygones. Ainsi il n'est pas utile pour chaque polygone de tester l'inclusion avec les polygones en dehors de cette case. Voici le bout de code qui servait à cela :

```
quadrillage = [[quadrants[vect_aires[0][0]], [vect_aires[0]]]]
for i_polygon in range(1, nb_poly):
    num_polygon, aire_poly, polygon, = vect_aires[i_polygon]
    for case in quadrillage:
        if case[0].intersect(quadrants[i_polygon]):
            case[0].update(quadrants[i_polygon])
            case[1].append(vect_aires[i_polygon])
            break
    quadrillage.append([quadrants[i_polygon], [vect_aires[i_polygon]]])
```

Figure 9

Cependant l'utilisation des quadrants provoquent des erreurs (certains polygones ne sont pas mis dans les bonnes cases) dans des situations bien précises mais qui arrivent fréquemment avec des fichiers contenant beaucoup de polygones.

2) Fichiers quelconques

En voyant les résultats des tests 1 et 2 du Gitlab, nous constatons que notre algorithme manque d'efficacité dans certains cas. Nous pensons que cela est dû au fait que l'on s'est principalement concentré sur les tests proposés par Guillaume Raffin pour améliorer notre algorithme, or ces tests comportent beaucoup de cas particuliers et peu de cas quelconques. Notre algorithme est donc très efficace sur ces tests mais traîter les cas particuliers a tendance à ralentir le traîtement des fichiers quelconques. C'est donc ce qu'il nous resterait à améliorer : des cas quelconques plus globaux.