Algorithmique et Science de Données

Yassir Beraich, Youssef Daoud

Avril 2020

1 Présentation des méthodes

Pour détecter les inclusions des polygones on a utilisé 2 algorithmes de base:

1.1 Crossing Number Algorithm (CNA)

Cette méthode compte le nombre de fois qu'un rayon à partir d'un point P traverse un bord de limite de polygone en le séparant à l'intérieur et à l'extérieur. Si ce nombre est pair, alors le point est à l'extérieur; sinon, lorsque le numéro de passage est impair, le point est à l'intérieur. C'est facile à comprendre intuitivement. Chaque fois que le rayon traverse un bord de polygone, sa parité entrée-sortie change (car une frontière sépare toujours l'intérieur de l'extérieur, non?). Finalement, tout rayon doit se retrouver au-delà et en dehors du polygone délimité.

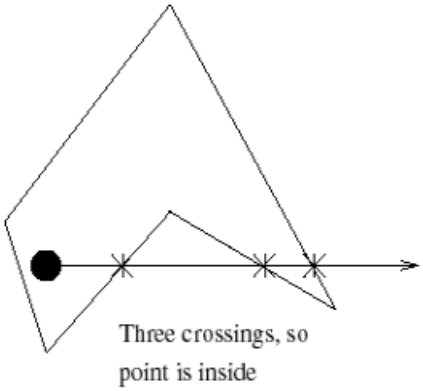

Figure 1: Crossing Test

On utilisera pour ceci un compteur **cn** qui s'incrementera de 1 à chaque fois que le rayon en prevenance du point étudié coupe un ségment du polygone objet de l'étude (voir la fonction **cn-est-dans** du fuchier **main.py**).

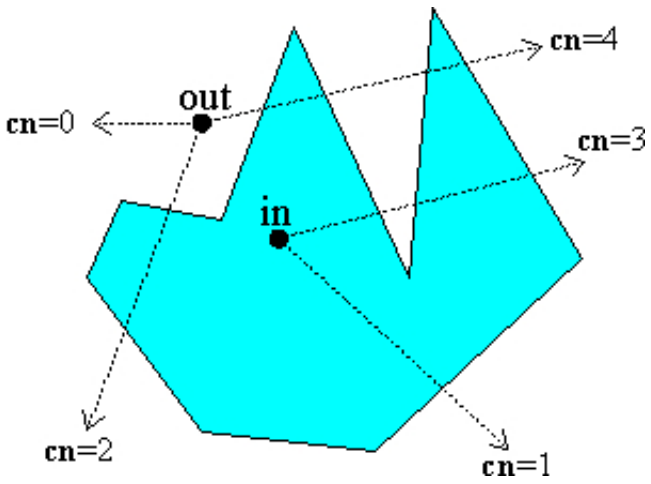

Figure 2: Exemple

1.2 Winding Number Algorithm

D'un autre côté, le nombre d'enroulement détermine avec précision si un point se trouve à l'intérieur d'un polygone fermé non simple. Pour ce faire, il calcule combien de fois le polygone s'enroule autour du point. Un point n'est à l'extérieur que lorsque le polygone ne s'enroule pas du tout autour du point, c'est-à-dire lorsque le nombre d'enroulement $wn = 0$. Plus généralement, on peut définir le nombre d'enroulement $wn(P, C)$ de toute courbe continue fermée C autour d'un point P dans le plan 2D. Soit la courbe 2D continue C définie par les points $C(u) = (x(u), y(u))$, pour $0 < u < 1$ avec $C(0) = C(1)$. Et soit P un point non sur C. $wn(P, C)$ est défini par :

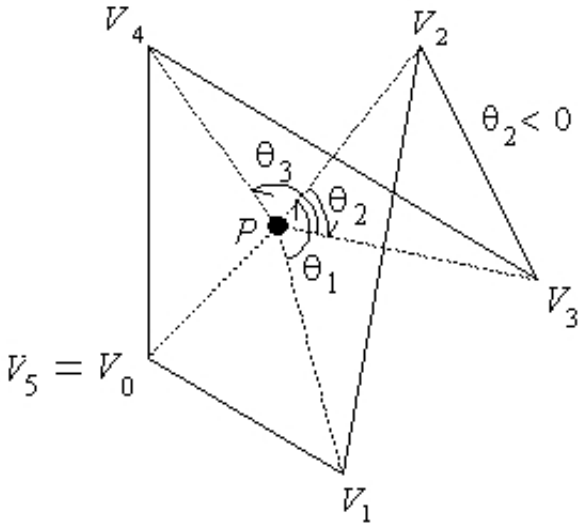

Figure 3: Exemple-winding

$wn(P, C) = \frac{1}{2\pi} \sum_{i=0}^{n-1} \theta_i$

ce qui donne: $wn(P, C) = \frac{1}{2\pi} \sum_{i=0}^{n-1} arccos(\frac{(V_i - P).(V_{i+1} - P)}{|(V_i - P)|.|(V_{i+1} - P)|})$

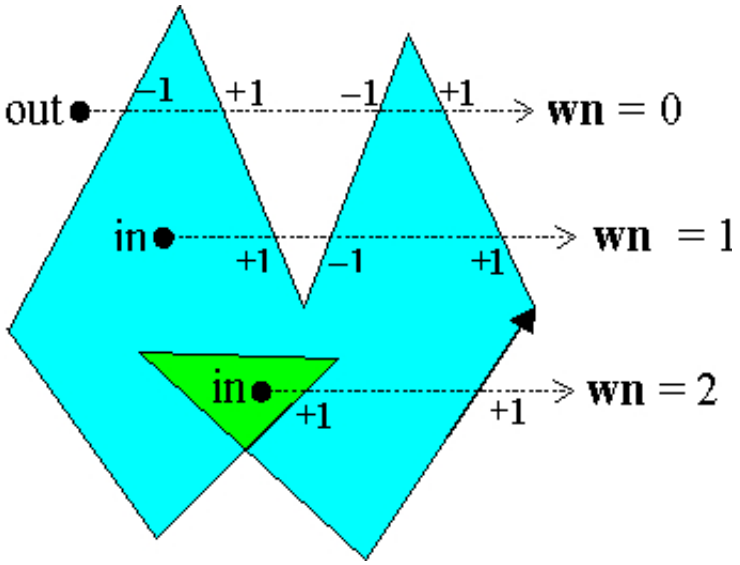

Figure 4: Exemple

Pour éviter l'usage de fonctions trigonomeetriques on a utilisé la méthode **is-oriented -clockwise()** de la classe polygon .Cependant, il doit être appliqué différemment pour les bords ascendants et descendants, ainsi selon les cas et selon si $V_iV_{i+1}P$ est orienté dans le sens antihoraire ou horaire wn va être decrémenté ou incrémenté par 1 (voir la fonction **wn-est-dans(point,polygone)** dans le fichier **(wn-main.py)**).

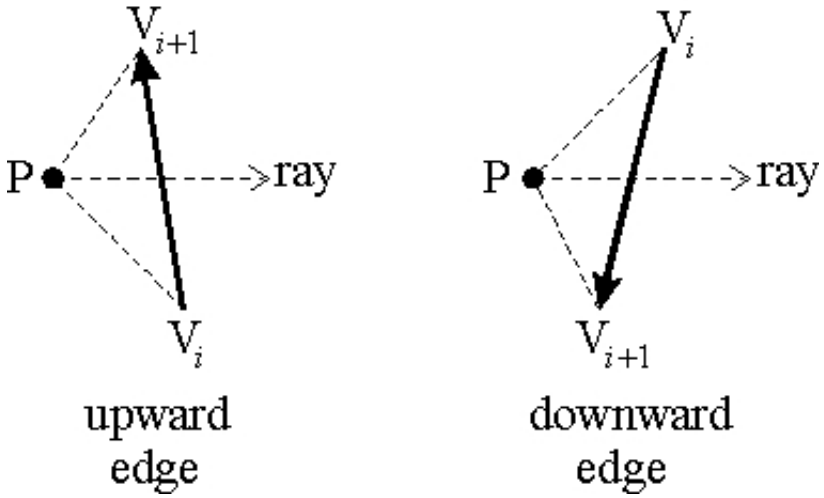

Figure 5: exemple bord ascendant - bord descendant

2 Coût des deux méthodes :

Puisque dans les deux méthodes on fait un seul parcours sur les segments du polygone. Alors, le coût des deux méthodes est : **O(n)** avec n le nombre de segments du polygone.
Pour tester le coût des deux fonctions(**cn-est-dans(point, polygone)**) et **wn-est-dans(point,**

polygone) on a utilisé un générateur de polygones convexes aleatoires puis et a l'aide d'un point fixé on a tracé les courbes permettant d'observer l'évolution du temps de l'éxecution des deux fonctions en fonction des nombres des points des polygones généres. Notons que l'unité du temps d'éxecution est $10^{-4}s$.

2.1 Coût du Winding number algorithm

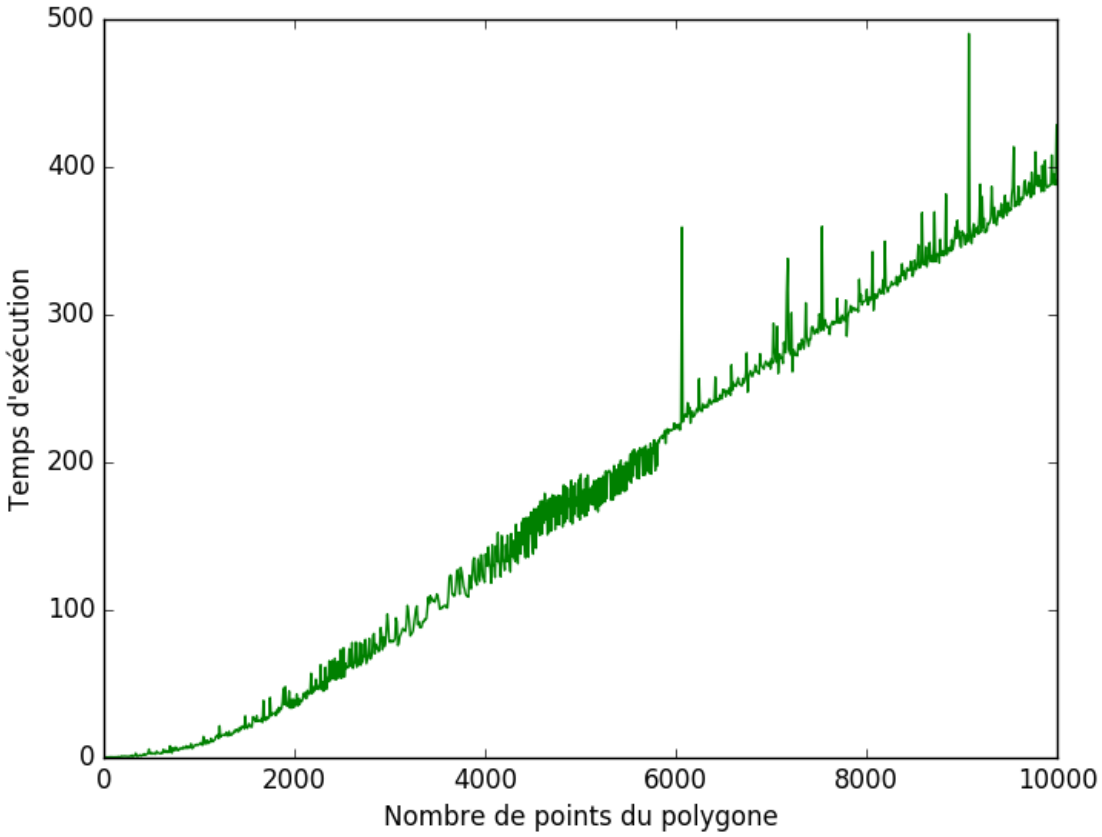

Figure 6: Evolution du temps d'execution (en $10^{-4}s$) de la fonction $wn - est - dans(point, poly)$ en fonction du nombre des points du polygone

En prenant les exemples 10×10.poly et e2.poly:
- **main.py** qui utilise la méthode **crossing-number** s'éxecute respectivement en 1.3 ms, 0,57 ms.
- **wn-main.py** qui utilise la méthode **winding-number** s'éxecute respectivement en 1.2 ms, 0.56 ms.

2.2 Coût du crossing number algorithm

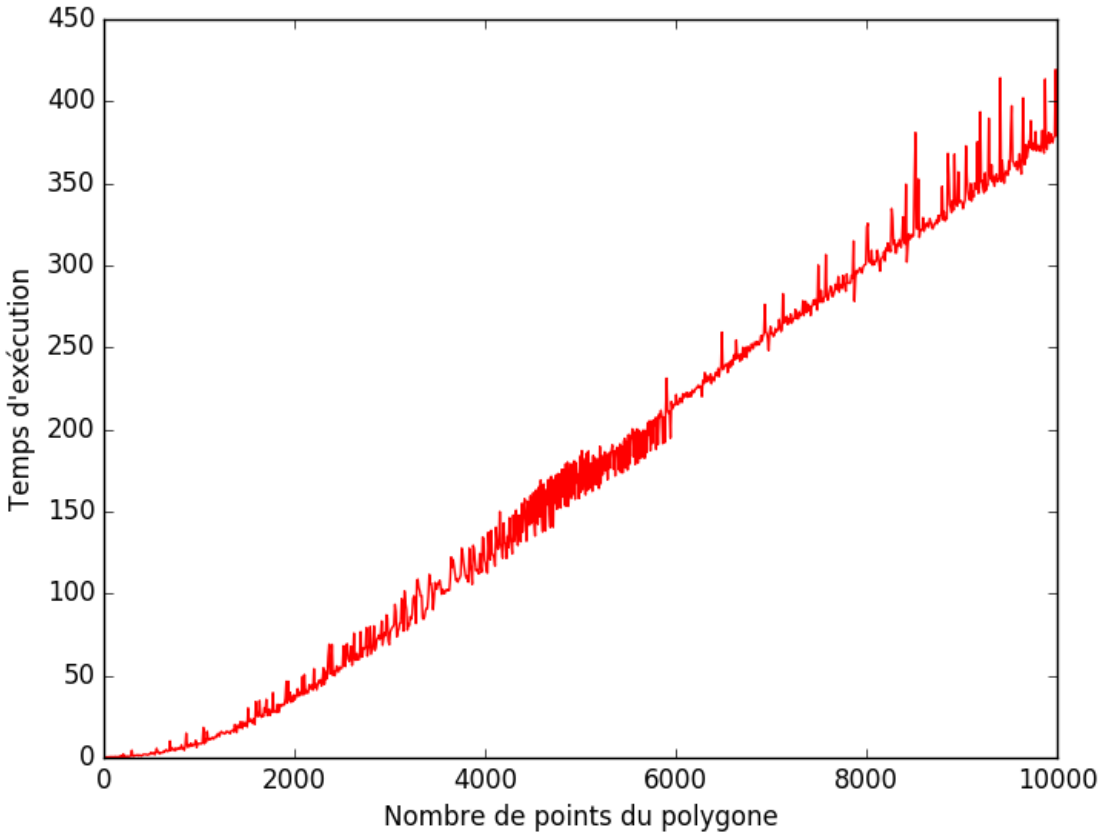

Figure 7: Evolution du temps d'execution (en $10^{-4}s$) de la fonction $cn - est - dans(point, poly)$ en fonction du nombre des points du polygone

3 Coût des autres fonctions:

En ce qui concerne la fonction $proj(point, seg)$ du fichier **main.py**, on paie un coût fixe : **O(1)**.
Par rapport à la fonction $est - inclus(poly1, poly2)$ qu'on trouve dans les 2 algos, elle a le même coût que les fonctions $wn - est - dans(point, poly)$ et $cn - est - dans(point, poly)$: **O(n)**. En fait, vu l'hypothèse du sujet qui précise qu'il n'y ait pas d'intersections entre les segments des polygones, alors il suffit de tester l'inclusion d'un seul point pour vérifier l'inclusion d'un polygone donné dans un autre.

3.1 Fonction de tri:

Pour le tri, on a utilisé la méthode de "diviser pour régner". En fait, on passe par trois phases:
Diviser : diviser notre problème en des sous problèmes.
Régner : résoudre les sous problèmes.
Combiner : combiner les résultats des sous problèmes pour arriver au résultat du plus grand problème.

Pour ce faire, on utilise deux fonctions :

triFusion(polygones, inf, sup): assure la phase "diviser".
L'argument **(polygones)** est une liste de triplets.. Dans chaque triplet, on a le polygone, son indice et la valeur absolue de sa surface. En fait, on cherche à trier la liste par rapport au 3ème élément(la valeur absolue de la surface). Alors :
Entrée : Une liste polygones[0...n-1] et deux indices entiers **inf** et **sup** avec
$0 \leq inf \leq sup \leq n-1$
Sortie : La liste des polygones triée par rapport au 3ème élément (la valeur absolue de la surface).

Fusion(polygones, inf, sup, moyenne) : assure les phases "régner" et "combiner".
Entrées : Une liste polygones[0...n - 1] et trois indices entiers **inf**, **moyenne** et **sup** avec $0 \leq inf \leq moyenne \leq sup \leq n-1$. La liste est triée entre les indices inf et moyenne et entre les indices moyenne + 1 et sup.
Sortie : La liste est fusionnée triée entre inf et sup.
Calculons la complexité des deux fonctions :

-L'efficacité de l'algorithme **Fusion** vient du fait que deux listes triées peuvent être fusionnées en temps linéaire. La fusion qui consiste en un parcours linéaire de la liste de longueur n est en $O(n)$.

-A l'intérieur de **triFusion**, on appele la même fonction deux fois avec la liste initiale découpé en deux. En plus, on appelle la fonction **Fusion**. Alors, on obtient cette relation de réccurence pour la complixité :

$$C(n) = C(\left\lfloor \tfrac{1}{n} \right\rfloor) + C(\lceil \tfrac{1}{n} \rceil) + O(n)$$

En utilisant le Master Theorem, on trouvera que la complexité de **triFusion** est : $O(nln(n))$.
Remarque:
Le coût du tri fuison est **O(nln(n))** dans tous les cas; le meilleur, le moyen et le pire.

3.2 La Fonction (trouve-inclusions):

trouve-inclusion(polygones) : cette fonction consiste à construire au début deux listes de longueur len(polygones). La première(le vecteur des inclusions) ne contient que des -1, alors que la deuxième contient des triplets. Dans chaque triplet, on a le polygone, son indice et la valeur absolue de sa surface. Après, on trie cette liste de manière croissante selon la valeur absolue de la surface des polygones. Ensuite, on parcourt la liste triée. On prend le i-ème polygone et on teste s'il est inclus dans les autres dont la surface est plus grande. Au moment où on trouve le polygone père on met le numéro du polygone père à l'indice du fils dans le vecteur des inclusions et on sort de la deuxième boucle(While). Puis, On passe au (i+1)-ème polygone.

Remarque :
- Lorsqu'on cherche le père de chaque polygone, on parcourt jusqu'à l'avant dernier élément parce qu'après les avoir trié par rapport à leur surface, on est sûr que le dernier n' a pas de

père.
- Puisque les deux fonctions **wn-est-dans** et **cn-est-dans** ont le même coût, alors la complexité est la même pour la fonction **trouve-inclusion** des deux fichiers **main.py** et **wn-main.py**.

Calculons la complexité de cette fonctions:
Soit n le nombre de polygones.
- Pour les 5 premières lignes dont on initialise "tab" et "liste", on paie $O(n)$.
- Le coût du tri fusion est : $O(n\ln(n))$
- Jusqu'à le moment, on paie en total $O(n\ln(n))$
- Maintenant, on rentre dans les deux boucles imbriquées. Puisqu'on a un "While", alors on va distinguer le pire et le meilleur des cas. Soit m_i le nombre de segment du i-ème polygone, $m = max(m_i)$ et $c_i = O(m_i)$ le coût de la fonction **"est-inclus(poly-j, poly-i)"** qui teste si un polygone est inclu dans le i-ème polygone.

Le pire des cas: Lorsqu'aucun polygone n'a un père.
Alors le coût de ces deux boucle imbriquées est :

$$O(\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} c_j) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} O(m_i) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} O(m)$$

Ce qui donne:

$$O(m)\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = O(m\frac{n(n-1)}{2}) = O(n^2m)$$

Alors, le coût total de la fonction "**trouve-inclusions**" dans ce cas est :

$$O(n^2m) + O(nln(n)) = O(n^2m)$$

En testant notre algorithme sur un tel cas(Aucun polygone n'a un père, vous trouverez le générateur de ce genre de polygones dans le fichier **genere-polygones-sans-père.py**), on dessine la courbe du temps par rapport au nombre des polygones.
Remarque : Le test est sur des carrés.

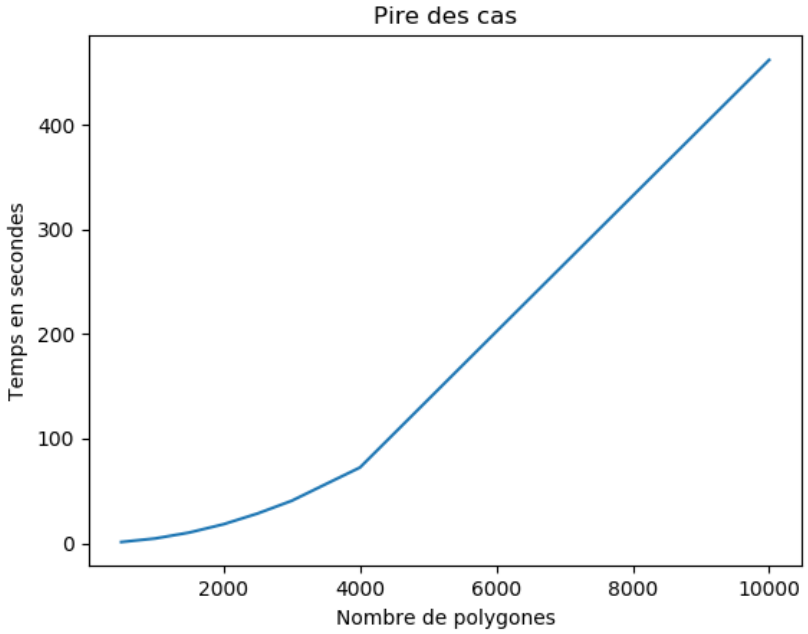

Figure 8: Le temps d'exécution de notre algo sur le pire des cas

Le meilleur des cas: des polygones imbriqués.
Dans ce cas, on trouve que le père du i-ème polygone est le (i+1)-ème polygone. Ce qui veut dire qu'on trouve le père au premièr pas de la boucle "while". Alors, le coût des deux boucles imbriquées de notre algorithme est :

$$O(\sum_{i=0}^{n-2} c_j) = \sum_{i=0}^{n-2} O(m_i) = \sum_{i=0}^{n-2} O(m) = O(nm)$$

Alors, le coût total de la fonction "**trouve-inclusions**" dans ce cas est :

$$O(nm) + O(nln(n)) = O((n(ln(n) + m))$$

En testant notre algorithme sur un tel cas(des polygones carrés imbriqués, vous trouverez le générateur de ce genre de polygones dans le fichier **genere-polygones-imbriqués.py**), on dessine la courbe du temps par rapport au nombre des polygones.

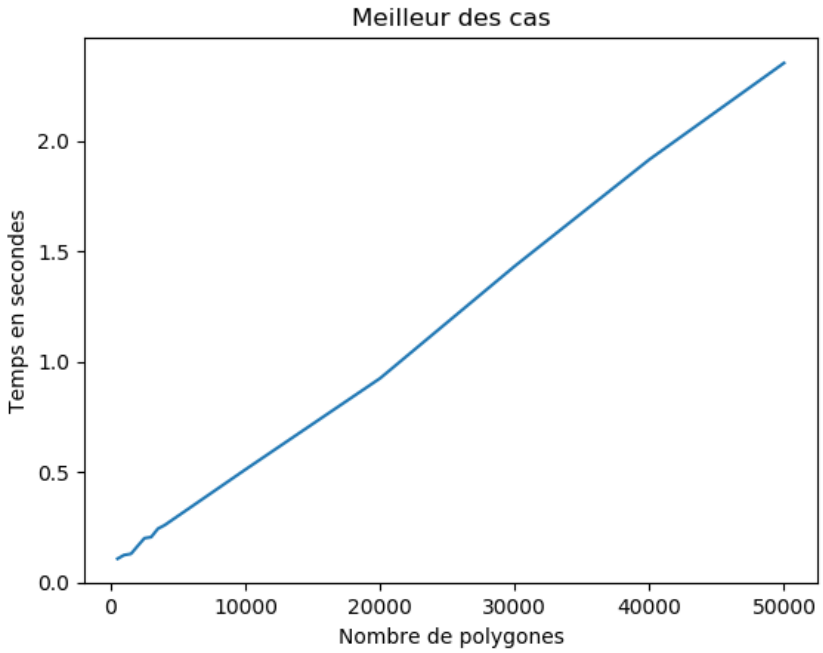

Figure 9: Le temps d'exécution de notre algo sur le meilleur des cas

Pour bien voir la différence entre les deux cas, on réunit les deux en un seul graphe.

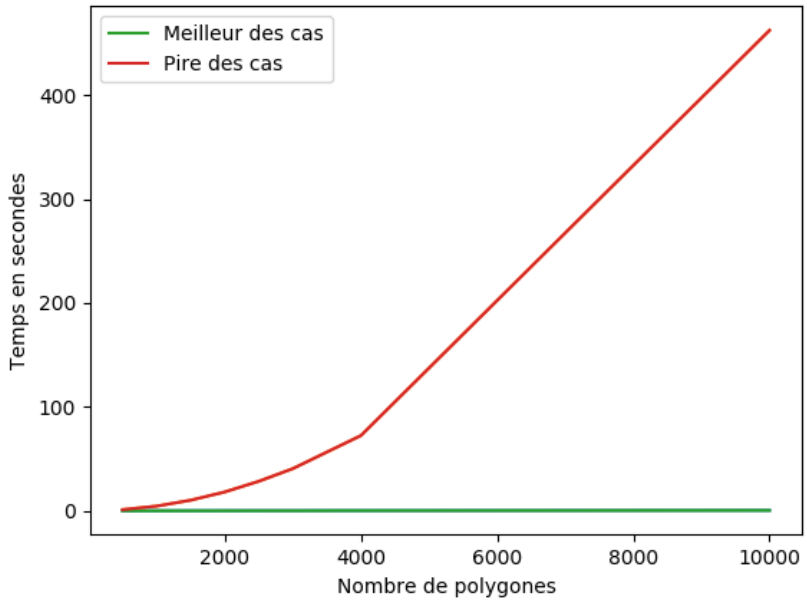

Figure 10: Les deux cas précédents en un seul graphe