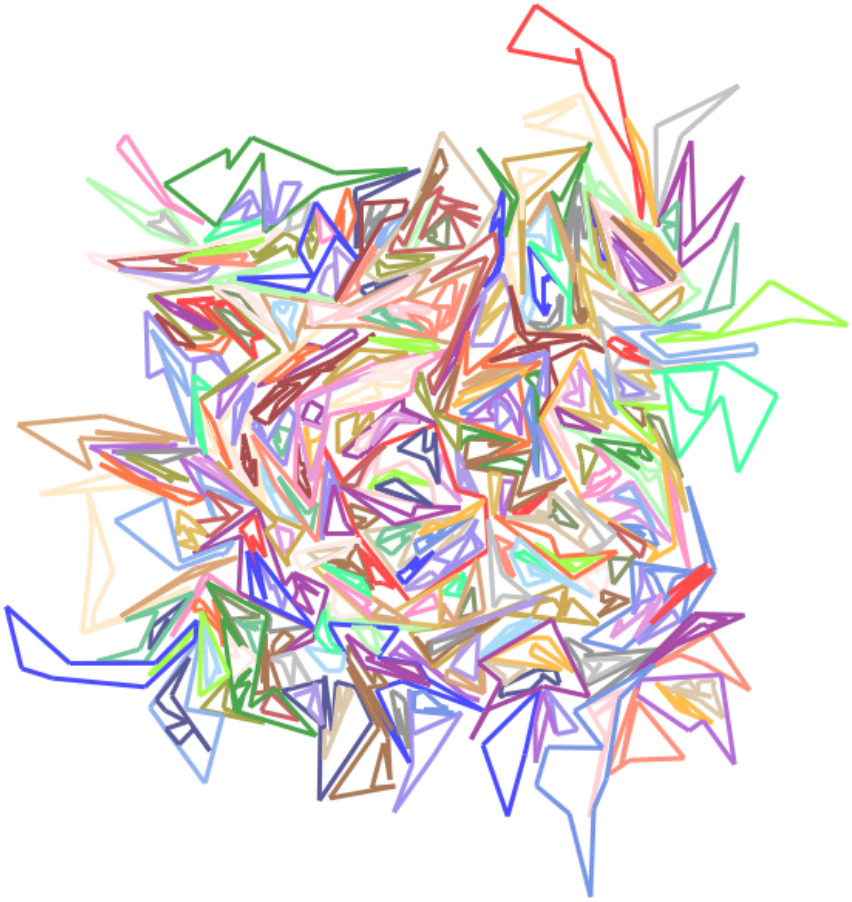


Rapport du projet d'Algorithmie 2023-2024

– Raphaël EUZEBY & Léonard HAVET

Aperçu

Durant ces dernières semaines le projet d'algorithmie a enflammé les discussions, "C'est quoi ton classement ?" " 100 000 polygones en 8 secondes ?". Le but était d'épater ses camarades et au passage prouver sa supériorité en termes de recherche d'algorithme.

Pour notre part nous avons implémentés deux algorithmes ayant des complexités similaires mais différents dans le fond.

Sommaire

Présentation du premier algorithme

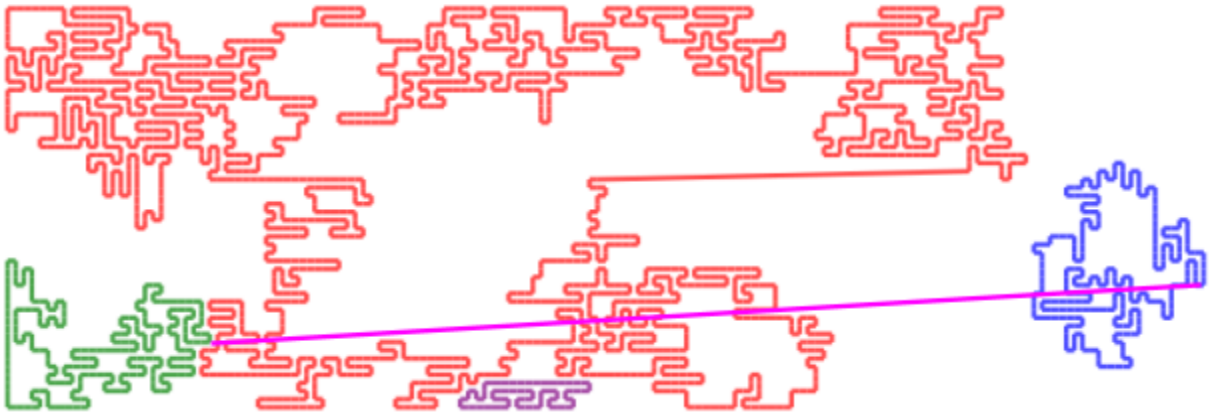
Idée générale :

Tracer un segment qui va d'un point du polygone jusqu'au point le plus loin et compte combien de fois il croise un certain polygone. Puis on répète ce procédé pour tous les polygones.

Approfondissement de l'idée :

La stratégie initiale consiste à choisir un polygone de référence et à le comparer à tous les autres polygones présents. Si un point du polygone A est situé à l'intérieur du polygone B, alors il est assuré que l'ensemble du polygone A est contenu dans le polygone B, étant donné l'absence d'intersection entre les polygones.

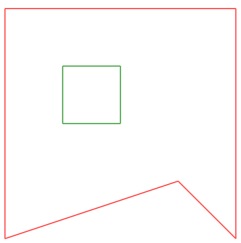
Ainsi, notre problème se ramène sur la détermination de l'appartenance d'un point à un polygone. Pour ce faire, nous créons un segment à partir d'un point du polygone A, s'étendant jusqu'au point le plus éloigné du point d'origine (défini comme étant celui pour lequel la distance euclidienne par rapport au point (0,0) est maximale). Ce point maximal, noté M, est défini comme étant $max_i(points_i) + Point([1,1])$, établissant ainsi le segment S entre M et le point de A.



Vérification du polygone vert en traçant un trait rose jusqu'au point le plus loin appartenant au polygone bleu

Pour compter cela on utilise la méthode du counterclockwise qui vérifie si les points sont de parts et d'autres d'un segment et ainsi permet de vérifier si deux segments se coupent. Si les segments se coupent on ajoute 1 à un compteur. Lorsqu'un segment S croise un segment du polygone B, nous incrémentons un compteur de 1.

Après avoir parcouru tous les segments du polygone B, si le compteur est pair, cela indique que le point (et par extension le polygone A) n'est pas inclus dans le polygone B. En itérant sur l'ensemble des polygones, nous pouvons ainsi dresser une liste des polygones contenant le polygone A. Dans cette liste on peut choisir le plus petit qui le contient en vérifiant sur tous les points lequel est le plus proche du point choisi de A, alors ce sera le plus petit polygone

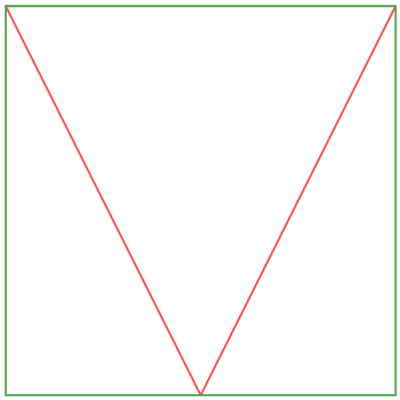


Seul le vert possède une liste d'inclusion, qui contient le rouge

Explication des modifications et des optimisations

Bounding box

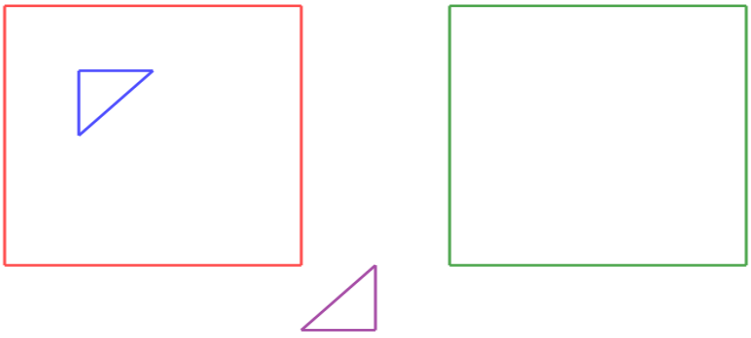
Il est superflu de vérifier si un polygone est inclus dans tous les autres polygones. Par exemple, un carré de 6 unités de côté ne peut jamais être inclus dans un triangle ayant une base de 1 unité de largeur. Afin de simplifier cette vérification, on utilise une "bounding box", c'est-à-dire un rectangle minimal englobant le polygone. Cette bounding box offre une représentation simplifiée du polygone, facilitant ainsi les calculs d'inclusion. Ensuite, on peut trier les polygones soit par leur aire, soit par leur largeur (pour diminuer le temps de calcul). Ce tri permet de réduire la complexité des opérations de test d'inclusion. Effectivement maintenant une liste des polygones est effectuée triée par leur largeur on n'a plus qu'à vérifier les polygones qui sont à droite dans la liste, ceux qui ont une largeur de "bounding-box" plus grande.



En vert la bounding box, en rouge le triangle

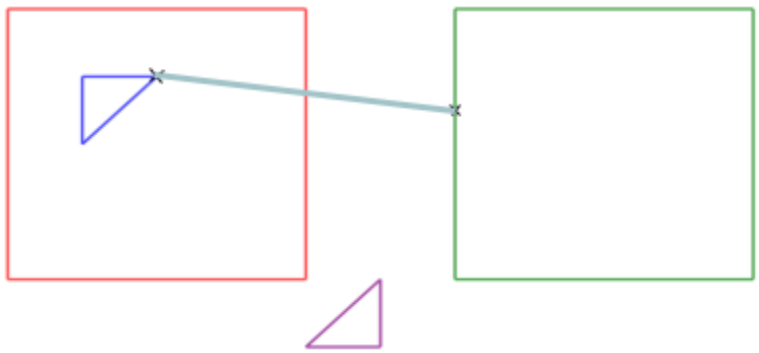
Position des polygones

On peut continuer à éliminer plein d'autres polygones pour éviter de les tester.



Certes le vert est plus large mais pourquoi vérifier si le bleu est inclus dans le vert ?

Pour optimiser le test d'inclusion de polygone, une méthode efficace consiste à comparer les positions des points extrêmes des polygones A et B. Tout d'abord, on peut vérifier si le point le plus à gauche du polygone A a une abscisse supérieure à celle du point le plus à droite du polygone B. Si cette condition est satisfaite, le polygone A ne peut pas être inclus dans le polygone B. De manière similaire, examiner si le point le plus bas du polygone A a une ordonnée supérieure à celle du point le plus haut du polygone B permet aussi de confirmer que le polygone A n'est pas inclus dans le polygone B. Cette approche peut être généralisée pour les côtés droite et bas, offrant ainsi une méthode rapide et efficace pour pré-déterminer la non-inclusion entre deux polygones.



On ne vérifie plus si le bleu est inclus dans le vert.

Différence entre le deuxième algorithme et le premier

Idée générale :

On arrête de tracer un trait pour vérifier les intersections, on va plutôt vérifier si le segment est à droite et si les extrémités du segments sont de part et d'autre du point choisi. Ainsi en “traçant une droite horizontale” on saura quel segment cela croise.

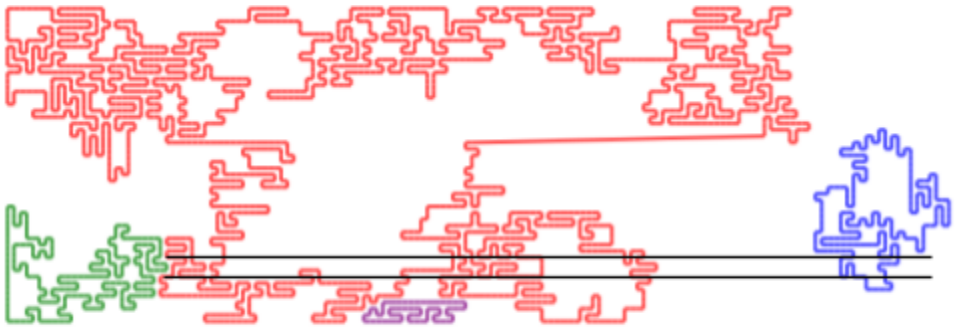
Approfondissement de l’idée :

Le principal problème du premier algorithme réside dans les points ou les droites parallèles au segment créés. C’est pour cela que le deuxième algorithme est plus pratique.

De nouveau on va chercher à vérifier si un point est dans un polygone, pour cela on itère sur les segments du polygone B à tester. Lors de l'itération sur ces segments, ion analysera la position relative de leurs extrémités par rapport au point d'intérêt. Les segments dont les deux extrémités se trouvent à gauche, au-dessus ou au-dessous du point peuvent être immédiatement exclus de la vérification. On se retrouve maintenant avec quelques

segments dont au moins une des extrémités est à droite du point et possède une extrémité avec une ordonnée supérieure et l'autre ordonnée inférieur au point.

Ensuite, pour chaque segment retenu, on calcule le point d'intersection avec la droite horizontale passant par le point du polygone A. Si l'intersection se trouve à droite du point, on incrémente un compteur. Après avoir itéré sur tous les segments, si le compteur est pair, cela indique que le point, et par conséquent le polygone A, est inclus dans le polygone B.



Le programme ne vérifie qu'entre les deux lignes noires pour le point le plus à droite du polygone vert

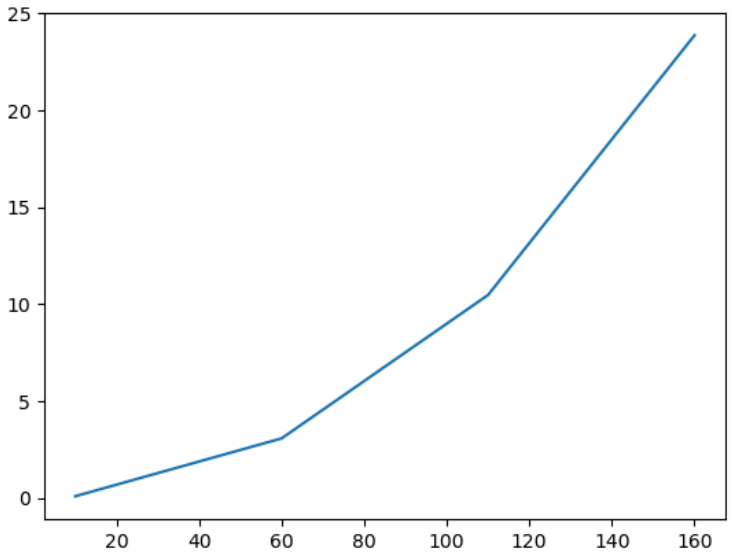
Comparaison en complexité et en temps

Pour les deux algorithmes il faut itérer sur tous les polygones pour créer le vecteur d'inclusion complet, de plus pour les deux algorithmes il faut créer la liste des polygones à vérifier il faut alors créer une bounding-box et classer par largeur les polygones (avec une insertion dans la liste par dichotomie) puis pour affiner la liste il est possible de vérifier si les polygones ne sont pas trop éloignés. Puis pour les polygones qui restent dans la liste de vérification il faut itérer sur les segments du polygone et augmenter le compteur jusqu'à trouver le plus petit.

Premier algorithme:

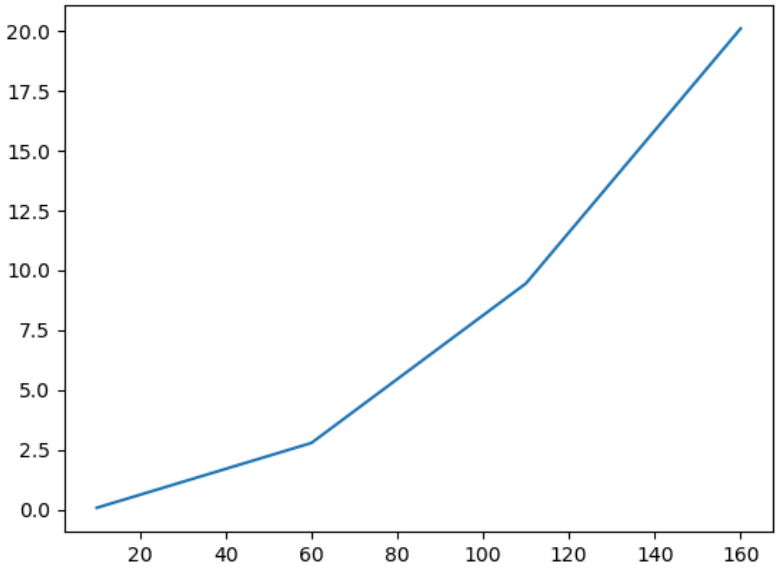
Pour le polygone A, il faut itérer sur les segments du polygone et vérifier si le segment se croisent, si le compteur est pair à la fin de l'itération il faut passer au prochain polygone sinon le polygone est désigné comme le plus petit contenant A. En somme c'est du $O(nm)$ dans le pire des cas avec n le nombre de polygones et m le nombre moyen de segment et dans le meilleur des cas $O(m)$.

La liste créée est très utile lorsque les polygones sont séparés ou de tailles similaires. Par ailleurs l'insertion par dichotomie (donc dans un tableau trié) permet d'éviter un tri en fin pour tout le tableau (en $O(nlog(n))$).



Graphe représentant le temps (en s) en fonction du nombre de carrés imbriqués (x100)

Dès qu'on rajoute la liste triée:

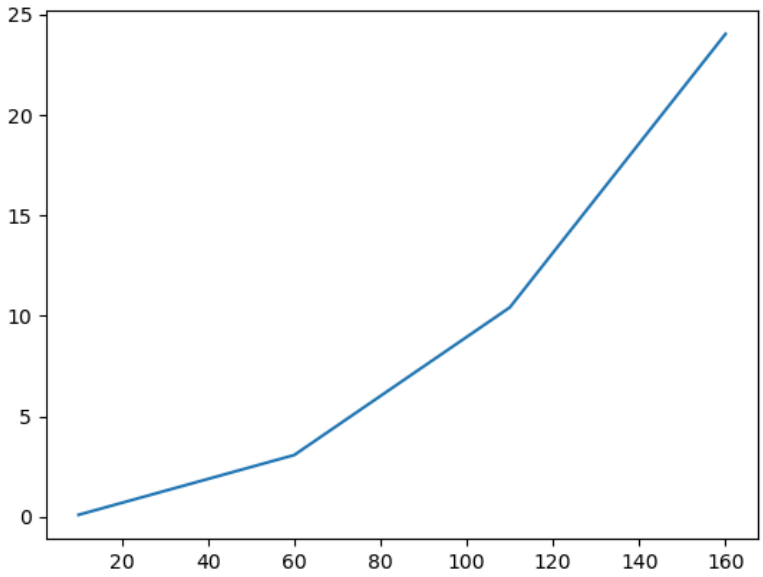


Graphe représentant le temps (en s) en fonction du nombre de carrés imbriqués (x100)

Deuxième algorithme:

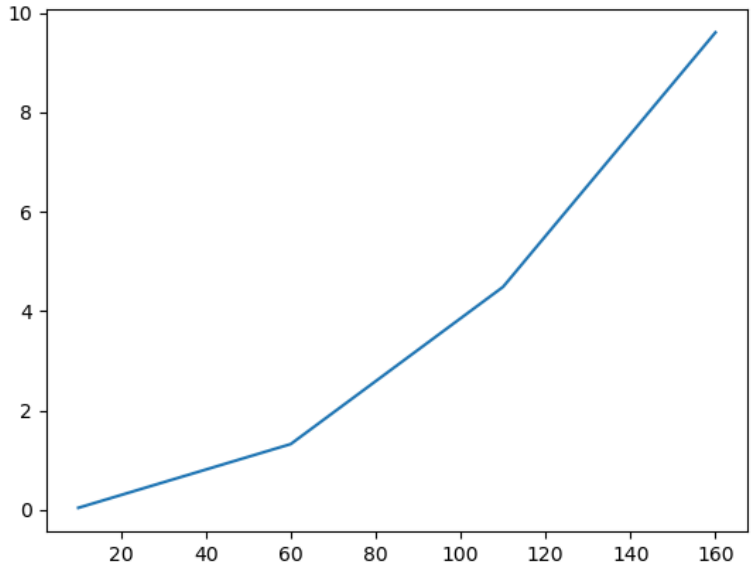
Pour le polygone A, il faut itérer sur les segments du polygone et vérifier si le segment se possèdent les bonnes caractéristiques alors uniquement on incrémente le compteur, si le compteur est pair à la fin de l'itération il faut passer au prochain polygone sinon le polygone est désigné comme le plus petit contenant A. En somme c'est du $O(nm)$ dans le pire des cas avec n le nombre de polygones et m le nombre moyen de segment et dans le meilleur des cas $O(1)$. Comparé au précédent, on n'essaie que très peu de segments.

Sans l'optimisation de la liste triée, les deux programmes font sensiblement les mêmes temps.



Graphe représentant le temps (en s) en fonction du nombre de carrés imbriqués (x100)

Dès qu'on rajoute la liste triée:



Graphe représentant le temps (en s) en fonction du nombre de carrés imbriqués (x100)

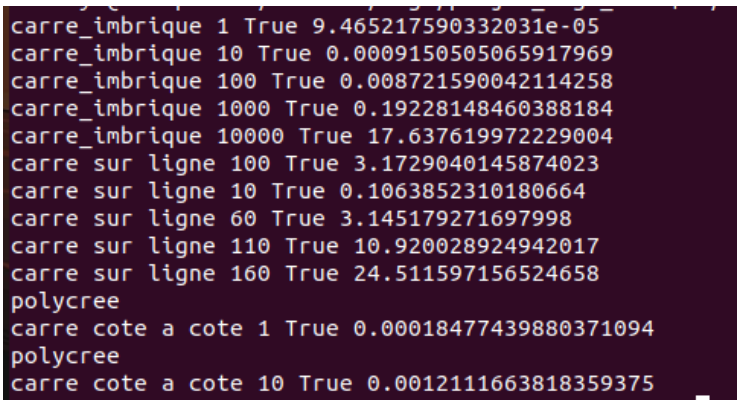
On remarquera qu'avec cet algorithme nous sommes placé 56 sur le test 0

48. algo_bomom_petersci en 5040 ms (commit 2b55277ec353122a9a81729ff61e221b6bf2cd11)
49. algo_berradah_contentb en 5040 ms (commit c568f1c898af55d1278191d8a0c03d30b80dd7cf)
50. algo_adebayon_georgeel en 5040 ms (commit d3a4847f308f364258d2a12548e595375eba4759)
51. algo_bagneaua_boissetv en 6040 ms (commit f71af68d20d97f88fa8ec86d95c170855efca954)
52. algo_aadroujm_sbaih en 6040 ms (commit a6bbfd9c62dd8c5b59c0482bc22b9bf2b31a3f80)
53. algo_arocke_dydar en 6040 ms (commit b2f1d90bbe7429520c4dc7bddd96f9421dd0842f)
54. algo_coustonl_duclossj en 6040 ms (commit 69b816fc25dace30c0c69c77b4aa7ea985a7c0a2)
55. algo_elhaouaa_madakc en 6040 ms (commit 3c726f3d4cebff61349026bfc0b04a08c4ace875)
56. algo_euzebyr_havetl en 6040 ms (commit ec902d896753389d7671b616ef5448d3eafec3f0)
57. algo_oughellf en 6041 ms (commit 2cba168af8e585f91dd6e665c408b072b49a4ac4)
58. algo_kassaoua_zaroualh en 7040 ms (commit 3ba835552626c4f6ec18087029ab3bf0fdc232cb)
59. algo_cousileo_richaadr en 7040 ms (commit d0d1d706740b0d3f0ed992416a020a6c693198d2)
60. algo_beint_raballac en 7040 ms (commit 255d651a6bcc5be59c8dcc6257e5f3b382ed9a5c)
61. algo_ardisson_fastierj en 7040 ms (commit b3d7bfb51d895e45f9bcc0884be7ef658227e12e)

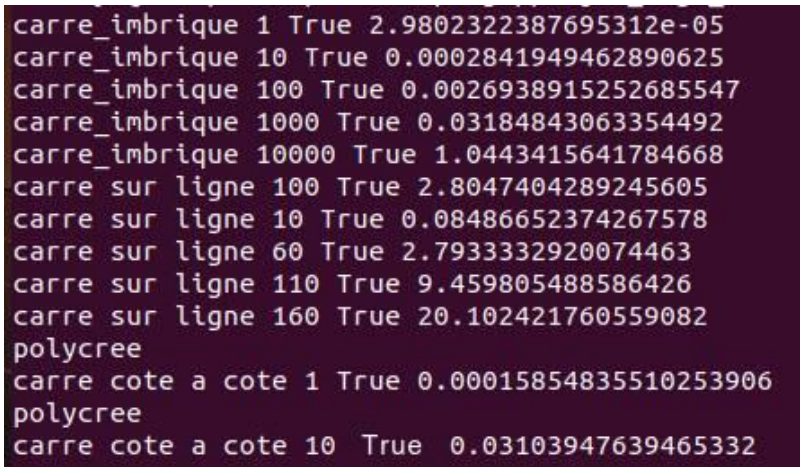
Petite fierté

Dans ce contexte, les tests effectués sont sélectionnés de manière arbitraire pour fournir une base de comparaison significative entre ces différents algorithmes. Ces tests sont choisis pour évaluer les performances relatives des différentes méthodes en fonction de leur temps d'exécution et de leur véracité.

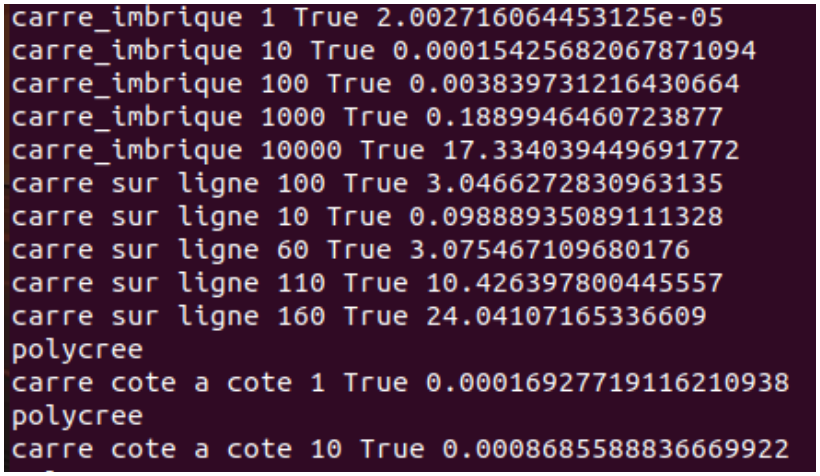
Par ailleurs, voici d'autres comparaisons pour élargir l'analyse et explorer plus en détail les performances des algorithmes. Les détails sur la création et l'exécution de ces tests supplémentaires seront présentés dans la suite.



Premier algorithme sans aucune amélioration ^



Premier Algorithme avec l'amélioration de la liste triée ^



Deuxième algorithme sans optimisation ^

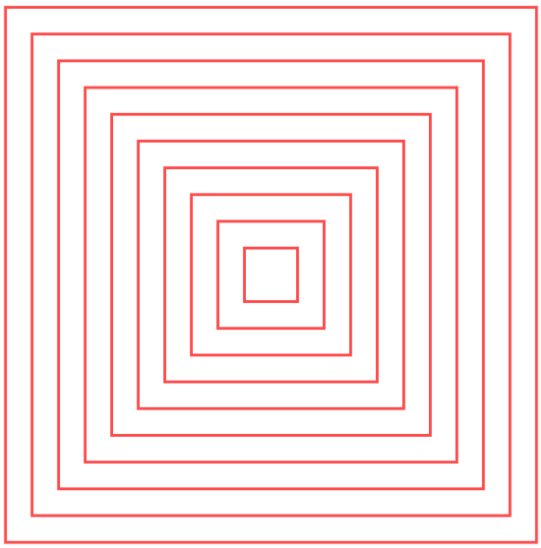
```
carre_imbrique 1 True 2.193450927734375e-05
carre_imbrique 10 True 0.00021982192993164062
carre_imbrique 100 True 0.00118255615234375
carre_imbrique 1000 True 0.02387070655822754
carre_imbrique 10000 True 0.9631547927856445
carre sur ligne 100 True 1.3159642219543457
carre sur ligne 10 True 0.040389299392700195
carre sur ligne 60 True 1.3231091499328613
carre sur ligne 110 True 4.487083435058594
carre sur ligne 160 True 9.608137130737305
polycree
carre cote a cote 1 True 0.0001513957977294922
polycree
carre cote a cote 10 True 0.012715339660644531
```

Deuxième algorithme avec optimisations ^

Génération de fichiers de tests

Génération de fichiers test à l'aide de la géométrie

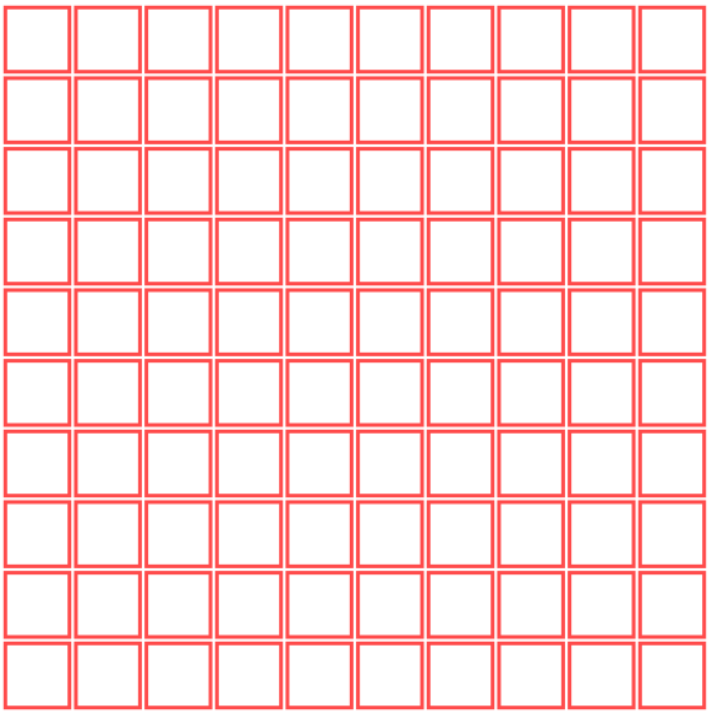
La géométrie offre une méthode précise et fiable pour générer des fichiers de test dont les résultats sont prédictibles. Par exemple, en utilisant des carrés imbriqués les uns dans les autres et numérotés de manière croissante, il est possible d'assurer une relation d'inclusion cohérente entre eux. Plus spécifiquement, si l'on considère un carré numéroté i , on peut faire qu'il soit toujours inclus dans le carré numéroté $i - 1$.



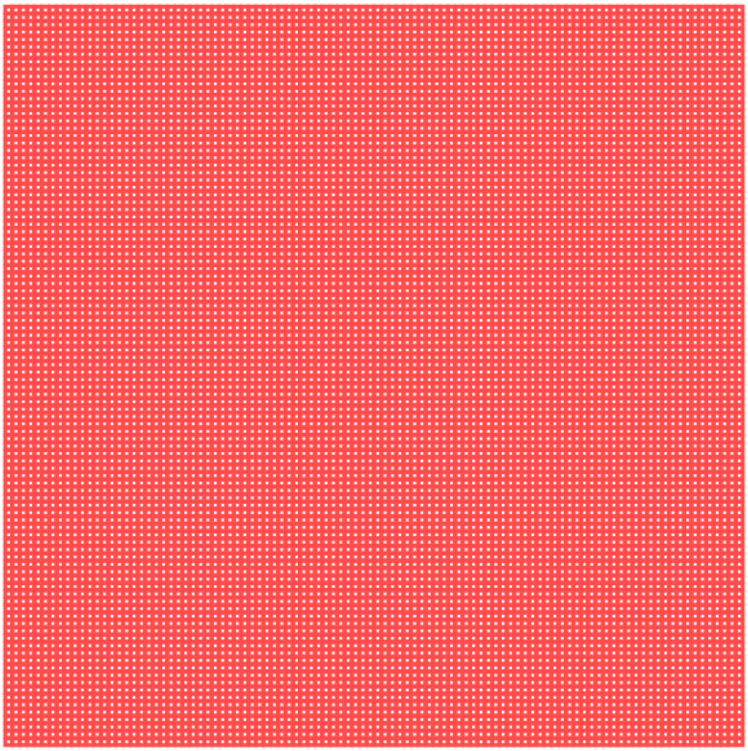
10 carrés imbriqués les uns dans les autres ^

1000 carrés imbriqués ^

groupes de 100 carrés imbriqués sur une ligne espacé de 10 unités ^



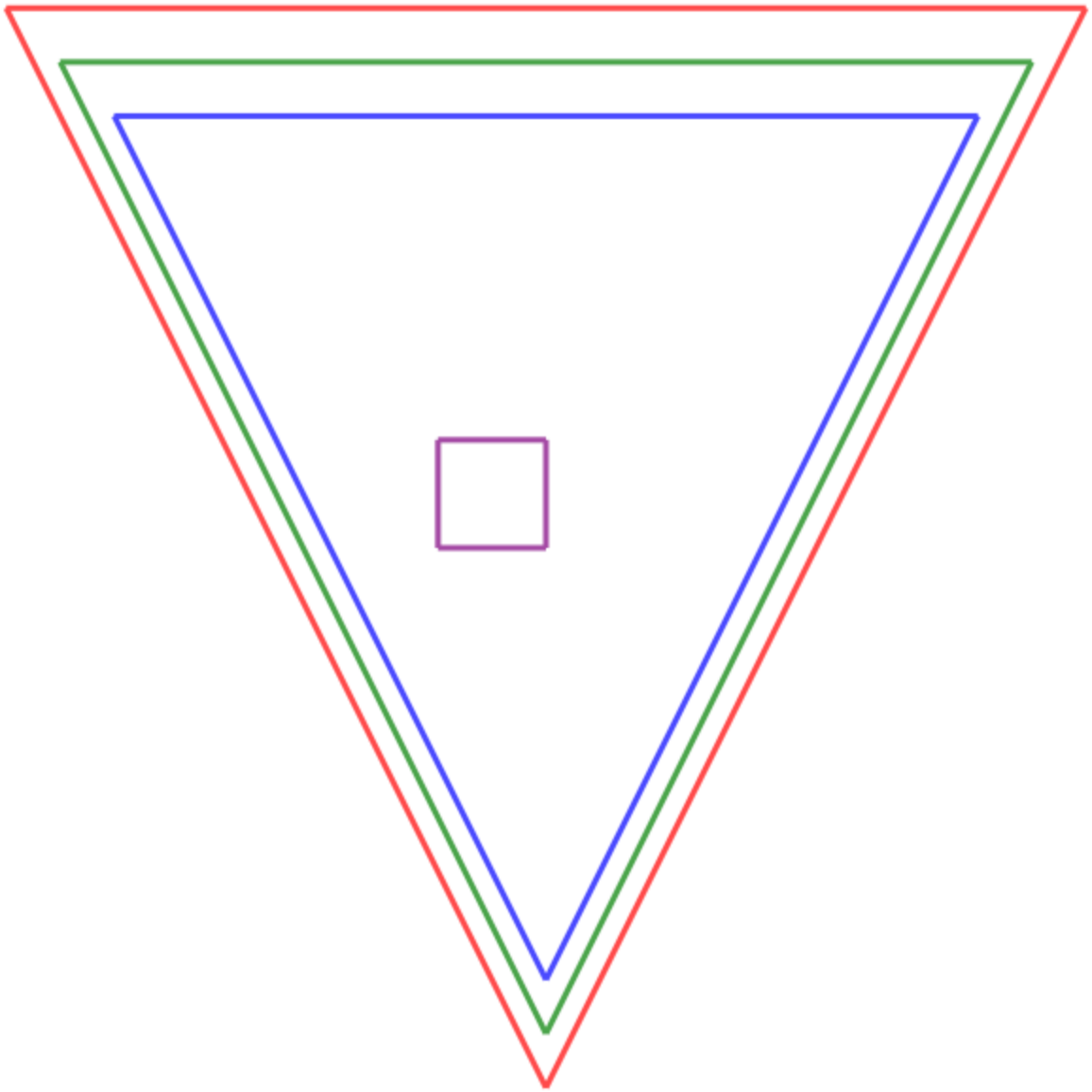
Carrés disposés dans le plan ^



Groupes de 100 carrés imbriqués disposés dans le plan ^

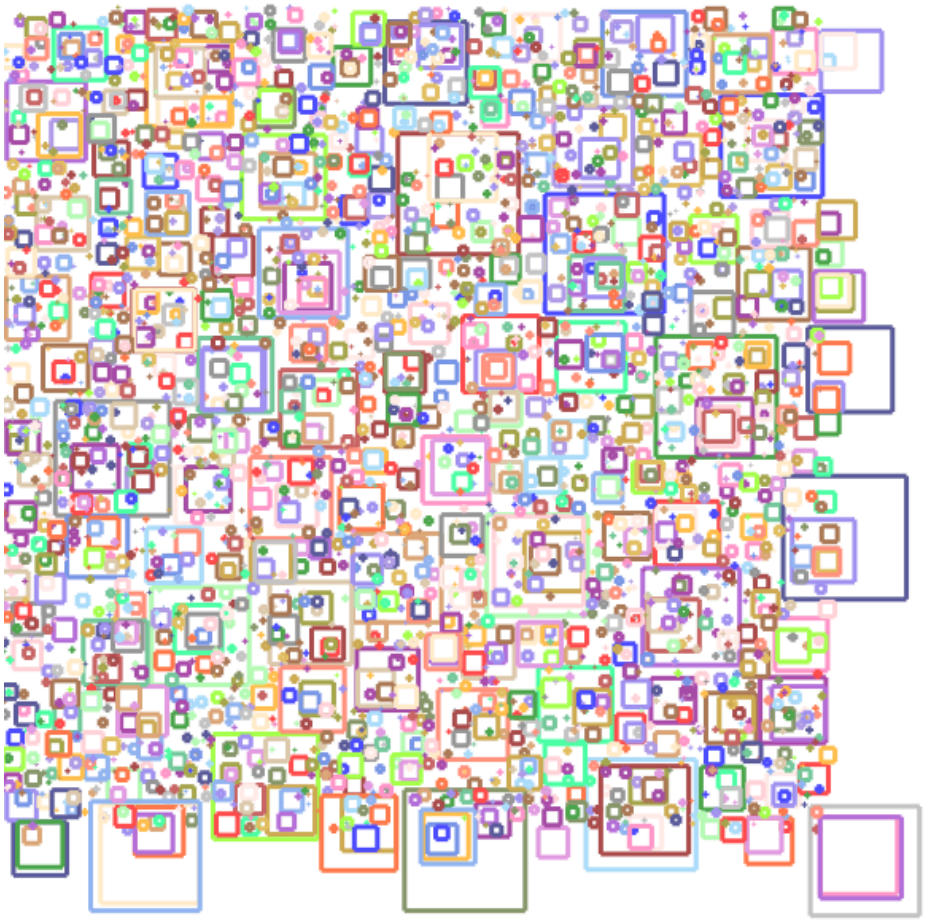
Cependant, la plupart des tests que j'ai réalisés étaient conçus pour évaluer des cas spécifiques, tels que l'inclusion d'un carré dans un triangle. Ces expérimentations m'ont révélé une limitation dans mon deuxième algorithme : celui-ci effectuait des vérifications sur les deux côtés du point choisi, plutôt que de se concentrer exclusivement sur le côté droit. En conséquence, dans certains scénarios, le carré n'était pas correctement identifié comme étant inclus dans les triangles, soulignant ainsi la nécessité d'ajuster et d'optimiser davantage l'algorithme pour améliorer sa précision et sa fiabilité dans diverses situations géométriques.

Il aurait été possible d’effectuer les mêmes tests avec des triangles équilatéraux (imbriqués, sur ligne, dans l’espace).



Génération de fichiers test à l'aide du hasard

Lorsqu'on envisage de générer des fichiers de test de manière aléatoire, une première approche intuitive consiste à positionner aléatoirement un polygone dans le plan et à vérifier ensuite s'il présente des intersections avec d'autres polygones. Dans le cadre de mon premier algorithme, j'avais déjà mis en œuvre la méthode ccw (counter clockwise), ce qui m'a permis de l'intégrer efficacement à cette nouvelle méthode de génération aléatoire. Si le polygone initialement placé présente des intersections avec d'autres formes, des tentatives de repositionnement sont effectuées. Toutefois, si après trois essais le polygone ne parvient pas à être correctement positionné sans intersection, il est alors remplacé. Cela permet d'éviter de continuer à essayer de placer un polygone trop grand qui ne pourrait être dans le plan car il y aurait déjà trop de polygones.



Art2.png ; 2000 carrés placés dans le canva

Conclusion

Dans le cadre de la première année, le projet d'algorithmie s'est imposé comme l'un des défis majeurs, prenant de l'ampleur à mesure que la date limite approchait. Il était difficile de passer à côté des discussions animées autour du thème des polygones. Ce projet a été particulièrement stimulant et enrichissant grâce à l'introduction d'un leaderboard et à l'émulation que cela a engendré.

Ce projet a été une véritable révélation quant à l'importance de l'optimisation, même sur des détails apparemment mineurs, pour garantir des performances efficaces lors de la manipulation de grands ensembles de données. Face à la gestion de tests impliquant plus de 1000 polygones, chaque optimisation réalisée a contribué de manière significative à accélérer le programme.

Enfin, cette expérience nous a été bénéfique, notamment en ce qui concerne la gestion du temps et la répartition des tâches au sein de l'équipe.