Hadj-azzem Anais
Cachet Théo

Projet Algorithmique : Compte rendu

Explication des choix pour la structure du code

- La première étape de l'algorithme était de créer les évenements de début et de fin pour chaque segment. Il nous a donc semblé qu'une classe *Event* serait adapté car celle ci nous permettrait de stocker un grand nombre d'information qui seront facile d'accés par la suite.

- Pour stocker ces événements, on utilise un tas. Les évenements y seront triés avec certains critères (définis dans __lt__), et l'ajout et la suppression d'élèment sera ainsi facilité (on enleve la racine à chaque fois par exemple).

- La suite de l'algorithme demande d'enregistrer les segments vivants. Nous avons utilisé pour cela une SortedList qui trie “automatiquement” nos segments vivants selon les critères définis par la méthode __lt__ de la classe *Segment*. Cela facilite nottament l'ajout et la suppression d'élèment.

- On utilise aussi deux “static attribu” : cache_x et current_point : le current_point sert a rendre dynamique la méthode de comparaison __lt__ de la classe segment. Le cache_x sert quant à lui à éviter les erreurs de calcul de clés de 10^-16 et permet un léger gain de performance.

- Un cache_segments est utilisé afin de ne comparer qu'une et une seule fois les intersections d'un même couple de segments.

Les problèmes rencontrés et les solutions trouvées

- Tout d'abord, nous avons eu des difficultés avec l'utilisation de l'outil SortedList. En effet, lorsque nous voulions enlever un segment de la SortedList, une erreur “Not in list” survenait, alors que lorsque l'on afficher l'intégralité de la SortedList, l'élèment était bien présent. Cela venait du fait que l'outil s'attendait à trouver l'élèment à une certaine place, et qu'il n'était pas à cette place ci. Nous avons résolu le problème en mettant à jour le point courant.

- Nous avons aussi eu des difficultés pour le calcul de clef. En effet, dans le cas d'une intersection, il est possible que le calcul de l'abscisse ait une erreur de 10^-16. Et ceci est la cause du problème précédemment expliquer. Pour pallier cela, nous avons crée un cache (appelé cache_x) qui enregistre le segment courant, l'ordonnee de l'intersection, et l'abscisse pour chaque intersection et chaque début ou fin de segment. Ainsi le calcul de clef est facilité : on ne

recalcule pas les abscisses qui sont déjà dans le cache (cf fonction calcul_clef du fichier segment.py), et cela limite les erreurs.

Les tests effectués

Nous avons créer une méthode “naive” qui trouve l'ensemble des intersections en testant l'intersection de chaque segment avec les autres.
Nous voulons comparer l'efficacité de cette méthode avec l'algorithme de Bentley Ottman. C'est le fichier test_efficacité.py qui effectue de cela.

On remarque que lorsque l'on traite peu de segment, par exemple pour simple.bo, simple_three.bo ou encore flat_simple.bo, il n'y a pas de grande différence, l'algorithme de Bentley Ottman est même moins performant.

En revanche, lorsque le nombre de segment devient important, le temps d'éxécution de Bentley Ottman est largement meilleur. On vérifie cela en exécutant le fichier test_efficacite.py sur exemples triangle_*.bo. On trouve :

Hadj-azzem Anais
Cachet Théo

Nombre de segment/ temps d'éxécution (s)	Méthode Naive	Bentley Ottman
2 flat_simple.bo	0.0007493495941	0.11083841323852
2 simple.bo	0.0007183551788330	0.12685513496398
3 simple_three.bo	0.0009019374847412	0.10049629211425
4 fin.bo	0.0011758804321289	0.10689473152160
31 triangle_b_1.0.bo	0.02610015869140	0.12485861778259
59 triangle_0.8.bo	0.048981428146362	0.15032219886779
87 triangle_b_0.5.bo	0.10635566711425	0.15091466903686
100 random_100.bo	0.24075603485107	1.6679081916809
111 triangle_h_1.0.bo	0.18250584602355	0.17130756378173
200 Random_200.bo	0.8007166385650	7.307536602020
363 triangle_h_0.5.bo	1.754608154296	0.458479642868
3003 Triangle_0.1.bo	99.4350938796	2.774857759475
5541 carnifex_h_0.5.bo	440.90224623680	5.162748098373
7353 triangle_b_0.1.bo	739.7097249031	6.696304559707

Ainsi, on remarque qu'à partir d'environ 100 segments traités, l'algorithme de Bentley Ottman est largement plus efficace. (Sauf dans le cas de random_100 et 200, cela est du au fait que la liste de segments vivants est très grande tout au long de

Hadj-azzem Anais
Cachet Théo

l'algorithme, ce qui fait perdre l'intêret de Bentley Ottman!)
Cela provient du faire que la compléxité de l'algorithme “naif” est $O(n^2)$, alors que la complexité de l'algorithme de Bentley Ottman est $O((n+k)\log(n))$, avec k le nombre d'intersection et n le nombre de segment.

Les améliorations à effectuer

En revanche, nous avons un problème dans le nombre de coupes dans un segment.
Nous n'avons pu résoudre cette erreur par manque de temps.
Et probablement bien d'autres améliorations à effectuer...