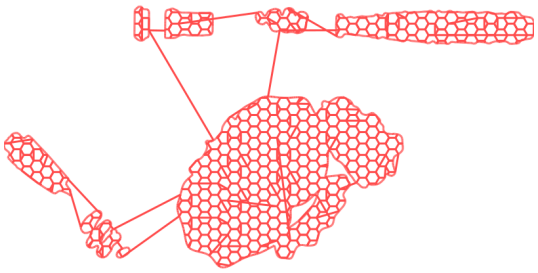
Projet d'algorithmique
Chemins eulériens

2018



1 Introduction

Cette année, le projet d'algo vise à réaliser un travail conjoint sur les graphes et les tables de hachage. On se propose de réaliser l'une des dernières étapes de calcul de chemins pour le contrôle d'une imprimante 3d.

On dispose en entrée d'un ensemble de segments dans le plan. Il est nécessaire pour l'imprimante de passer au moins une fois sur chacun d'entre eux. On cherche a réaliser un circuit passant par chaque segment. L'idée principale de l'algorithme est la suivante :

1. chargement du fichier de segments ;
2. création d'un *(multi-)graphe* : chaque extrémité de segment devient un sommet, chaque segment une arète ;
3. si le graphe est composé de plusieurs composantes connexes, alors on les relie entre elles en rajoutant des arètes formant un arbre couvrant les composantes ;
4. on continue ensuite en rajoutant des arètes jusqu'à ce que le degré de chaque sommet soit pair ;
5. enfin, on termine en calculant rapidement un circuit eulérien.

Bien entendu on cherche à ajouter les arètes les plus petites afin d'éviter le plus possible les surcoûts de déplacements.

2 Tables de hachage

Un problème apparait rapidement lorsque l'on réalise une analyse de coût asymptotique de notre algorithme. Si l'on considère n points du plan, alors le nombre potentiel de segments est en $O(n^2)$. Trier tous les segments à considérer du plus petit au plus grand coûte donc $O(n^2 \log(n))$. Il suffit alors d'un petit millier de points pour que ce coût pose problème.

On se propose donc d'utiliser une structure spéciale pour réaliser un parcours rapide des segments à rajouter.

Le principe est le suivant : on choisit une précision t (un flottant) qui correspond à la taille du côté d'un carré. On réalise un pavage du plan avec ces carrés et on les numérote à l'aide d'un couple de deux coordonnées. Si on considère un ensemble P de points du plan, on hache chaque point dans le carré qui le contient à l'aide de la fonction suivante : $h_1 : P \to (\mathbb{N}, \mathbb{N})$ telle que $h_1(p) = (\lceil p.x/t \rceil, \lceil p.y/t \rceil)$. La propriété intéressante est que tous les points qui se hachent dans un carré identique sont proches les uns des autres.

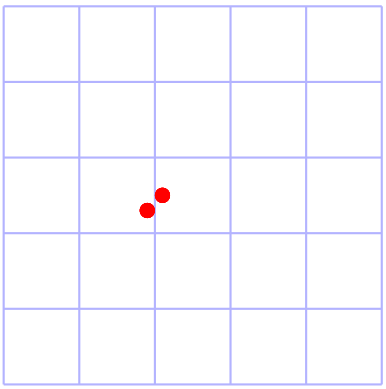


FIGURE 1 – Exemple de hachage

Attention toutefois, cette propriété n'implique pas que tous les points proches se hachent dans des carrés identiques. En effet, comme on le voit figure 1, deux points très proches peuvent se trouver de part et d'autre de la limite d'un carré.

La solution consiste donc à utiliser plusieurs fonctions de hachage (4) en décalant ou non le point de départ de $t/2$ sur x et sur y. On définit donc par exemple :

— $h_1(p) = (\lceil p.x/t \rceil, \lceil p.y/t \rceil)$,

— $h_2(p) = (\lceil (p.x + t/2)/t \rceil, \lceil p.y/t \rceil)$,
— $h_3(p) = (\lceil p.x/t \rceil, \lceil (p.y + t/2)/t \rceil)$,
— $h_4(p) = (\lceil (p.x + t/2)/t \rceil, \lceil (p.y + t/2)/t \rceil)$.

De cette manière, on est sûr que tout couple de points proches se hache au moins une fois dans un carré commun.

On utilise donc un tableau de 4 tables de hachage dont les clefs sont des couples de coordonnées identifiants des carrés et les valeurs sont des vecteurs de points. Tout point apparait 4 fois dans la structure, une fois dans chaque table de hachage.

Avec cette structure, on peut produire une séquence de segments de plus en plus grands de la manière suivante :

Entrées : points considérés
Sorties : itérateur sur des segments
$t \leftarrow$ précision de départ;
tables $\leftarrow$ {hasher(points, t)};
tant que *deux points sont en collision dans les dernières tables* **faire**
  $t \leftarrow t/2$;
  tables $\leftarrow$ tables $\cup$ {hasher(points, t)};
fin
pour chaque *jeu de tables, des dernières, aux premières* **faire**
  pour chaque *table parmi les 4 du jeu* **faire**
    pour chaque *clef* **faire**
      pour chaque *combinaison de valeurs associées à la clef* **faire**
        proposer le segment;
      fin
    fin
  fin
fin

3 Algorithme

Cette section détaille les algorithmes de graphe à implémenter. Parmi nos algorithmes, les deux premiers : reconnecter le graphe et rendre les degrés pairs nécessitent un parcours de segments potentiels, des plus petits aux plus grands.

On considérera donc deux versions de ces algorithmes, en fonction des itérateurs utilisés sur les segments :

— une version itérant sur un nombre quadratique de segments ;
— une version itérant sur les segments hashés à l'aide de l'algorithme présenté section 2.

Un booléen passé en argument de chaque fonction nous permettra de choisir le type d'itérateur à utiliser.

3.1 Arbre

Première étape donc, on commence par rajouter des arètes jusqu'à n'avoir plus qu'une seule composante connexe. L'algorithme est le suivant :

Entrées : G : Graphe
soit C l'ensemble des composantes connexes de G;
pour chaque *segment* (p_1, p_2)*donné par l'itérateur* **faire**
  si p_1 *et* p_2 *appartiennent à deux composantes différentes* **alors**
    ajouter (p_1, p_2) aux arètes de G;
    fusionner les deux composantes correspondantes de C;
  fin
  si $|C| = 1$ **alors**
    retourner
  fin
fin

La difficulté principale porte sur la représentation des composantes connexes. On utilisera une structure d'*Union-Find*[1] dont le code est fourni.

3.2 Degrés pairs

Seconde étape, on ajoute des arètes jusqu'à ce que tous les degrés soient pairs. C'est à cause de cette étape que le graphe peut devenir un multi-graphe, une arète pouvant désormais apparaitre plusieurs fois.

On utilise l'algorithme glouton suivant :

Entrées : G : Graphe
impairs $\leftarrow 0$;
pour chaque *sommet* s *de* G **faire**
  si s *est de degré impair* **alors**
    impairs $\leftarrow$ impairs $+1$;
  fin
fin
tant que *impairs* $\neq 0$ **faire**
  soit (p_1, p_2) le segment suivant;
  si p_1 *et* p_2 *sont de degrés impairs* **alors**
    ajouter (p_1, p_2) à G;
    impairs $\leftarrow$ impairs - 2;
  fin
fin

1. `https://fr.wikipedia.org/wiki/Union-find`

3.3 Circuit eulérien

Dernière étape : le circuit eulérien. La propriété clef est la suivante : dans un graphe où tous les sommets ont un degré pair, si on part d'un point de départ et que l'on avance dans n'importe quelle direction (un seul passage par arète), alors on revient toujours à son point de départ. Il est donc très simple de trouver un cycle mais malheureusement celui-ci ne passe pas forcèment par toutes les arètes. On peut néanmoins répéter l'opération jusqu'à ce que toutes les arètes du graphes soient utilisées. On dispose alors d'un ensemble de cycles qu'il faut fusionner.

Un choix judicieux des différents points de départs permet une fusion facile et le coût asymptotique de votre algorithme devra être très faible.

4 Code fourni

On vous fournit un petit module de géometrie *geo*. Le fichier *geo_graph.py* est à compléter ; les fichiers *geo/point.py*, *geo/segment.py*, *geo/quadrant.py*, *geo/tycat.py* et *geo/union.py* sont complets et ne devraient pas nécessiter de changements. Vous êtes bien entendu libres de créer autant de fichiers que nécessaire.

On vous fournit également deux fichiers principaux :

— affichage.py : affiche dans terminology un fichier de segments ;

— chemins.py : lance les calculs sur un ensemble de fichiers.

Nous vous demandons de *NE PAS MODIFIER* l'interface en ligne de commande de *chemins.py* car elle sera utilisé par votre correcteur pour réaliser des tests de performance. Normalement, aucune modification de ce fichier n'est requise.

4.1 Tycat

On vous fournit une méthode *tycat* très puissante, vous permet de débogguer plus facilement votre code et également d'animer vos algorithmes.

Elle réalise l'affichage graphique des objets ou itérables passés en argument. Chaque argument dispose de sa propre couleur. Les objets affichables sont :

— les points ;

— les segments ;

— les graphes.

4.2 Graphes

On vous fournit une structure de graphe (constructeur complet fourni). Un graphe est une table de hachage de sommets. Chaque clef est un sommet

du graphe, et à chacun d'entre eux est associée une valeur : un vecteur d'arètes. Chaque arète est un couple (ou un vecteur) de 2 points.

On peut par exemple récupérer le nombre de sommets de degrés impairs de la manière suivante :

```
impairs = sum(1 for e in self.vertices.values() if len(e) % 2 == 1)
```

Trois fonctions sont à compléter :

— `reconnect`
— `even_degrees`
— `eulerian_cycle`

5 Travail attendu

On vous demande d'implémenter en python l'ensemble des algorithmes et structures proposées en complétant le code fourni. Vous réaliserez une série d'évaluations des performances des algorithmes ainsi qu'une analyse de vos résultats. Réaliser des expériences est un exercice difficile duquel vous n'avez pas l'habitude et il convient d'y allouer du temps.

Vous serez essentiellement évalués sur un rapport de quelques pages décrivant vos expériences et vos conclusions. Ce rapport sera au format *pdf* et uploadé sur teide dans la même archive que votre code.

Il y a de nombreuses possibilités de mieux optimiser les choses que l'algorithme basique décrit dans ce document. Toute modification (évaluée bien sûr) est la bienvenue.

Le travail sera réalisé dans des équipes d'au plus 2 étudiants et à rendre au plus tard le dimanche 29 avril 2018 à 23h59.

Nous vous rappellons que le projet participe à la note du module d'algo. Toute copie de code sera sanctionnée.