INFO602, L3 Informatique, Algorithmique II
Leçon 5, Géométrie algorithmique

Jacques-Olivier Lachaud
LAMA, Université Savoie Mont blanc
`https://www.lama.univ-savoie.fr/wiki`
(suivre INFO602)

6 avril 2020

1 Géométrie algorithmique

Ce domaine étudie les algorithmes pour résoudre des problèmes géométriques, souvent formulés dans l'espace Euclidien. On trouve beaucoup d'applications de ces algorithmes en conception assistée par ordinateur, infographie, analyse d'image, ingénierie (calcul numérique sur des structures), robotique, jeux vidéo. Nous ferons ici plutôt de la géométrie dans le plan, même si beaucoup de notions s'étendent à l'espace. Attention, les algorithmes deviennent souvent difficiles dans l'espace.

On supposera que nous disposons d'un type assez précis pour représenter les nombres réels, même si une grosse difficulté des algorithmes géométriques est souvent liées à des erreurs numériques. Notamment, on sent bien que la notion "un point est-il sur une droite" est très sensible à la moindre erreur numérique. Les implémentations bien faites d'algorithmes géométriques font très attention aux imprécisions et utilisent les stratégies suivantes :

- favoriser les opérations qui n'augmentent que peu la taille mémoire nécessaire pour représenter les nombres : $+$ et $-$ notamment, $\times$ si nécessaire, $/$ à éviter.
- définir des zones de résultats où on sait qu'on n'a pas besoin d'être précis.
- utiliser des nombres à précision arbitraire
- ne pas utiliser des valeurs mais juste déterminer le bon signe

La bibliothèque CGAL est une bibliothèque C++ qui contient beaucoup d'algorithmes géométriques sûrs.

On peut aussi se placer dans le plan discret des entiers et ne manipuler que des entiers. Ce domaine de la géométrie est appelé géométrie discrète (*digital geometry* en anglais).

1.1 Notions élémentaires

On notera $\mathbb{R}^2$ le plan Euclidien. Un *point* $p = (p_x, p_y)$ du plan est caractérisé par ses coordonnées p_x et p_y, deux nombres réels. (On les notera parfois $p.x$ et $p.y$ comme en C.) Un *segment* $[p, q]$ est un sous-ensemble du plan caractérisé par ses extrémités p et q. Il est constitué de tous les points "entre" p et q. Plus formellement, si $0 \leq \alpha \leq 1$, $p = (1 - \alpha)p + \alpha q$ est entre p et q. On dit que p est une *combinaison linéaire convexe* de p et q. Le segment est donc l'ensemble des combinaisons linéaires convexes de p et q. C'est d'ailleurs l'enveloppe convexe de p et q.

La *droite* (pq) est définie comme le segment $[p, q]$, sauf que α peut prendre n'importe quelle valeur. Le *rayon* $[pq)$ est obtenu pour des valeurs $\alpha \geq 0$, tandis que le rayon $(pq]$ est obtenu pour des valeurs $(1 - \alpha) \geq 0$. Si l'ordre de p et q est important, on parlera du segment orienté $[\overrightarrow{pq}]$.

La longueur du segment $[p, q]$ est la distance euclidienne entre les deux extrémités p et q, ou de manière équivalente la norme-2 du vecteur $\vec{pq}$. On utilise le théorème de Pythagore pour déterminer que

$$L([p, q]) = d(p, q) = \|q - p\| = \sqrt{(q.x - p.x)^2 + (q.y - p.y)^2}.$$

Dans la suite on voudra déterminer si un point est à gauche ou à droite d'un segment orienté, si deux segments s'intersectent, etc. On voudra aussi déterminer quels sont les points les plus proches d'un autre, calculer des enveloppes convexes, etc.

1.2 Orientation et produit en croix

On veut déterminer si deux segments orientés $[\overrightarrow{pq}]$ et $[\overrightarrow{pr}]$ sont dans l'ordre trigonométrique, ou, dit autrement, si le point r est "à gauche" du segment orienté $[\overrightarrow{pq}]$.

Une solution est de calculer l'équation de droite (pq) puis de vérifier la hauteur de r par rapport à celle de la droite. C'est correct mathématiquement mais assez instable numériquement.

Une meilleure idée est le produit en croix. Soit deux vecteurs $\vec{u} = (u_x, u_y)$ et $\vec{v} = (v_x, v_y)$. Leur produit en croix (en fait leur déterminant) est :

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_x \\ u_y \end{pmatrix} \times \begin{pmatrix} v_x \\ v_y \end{pmatrix} = u_x v_y - u_y v_x = \det(\vec{u}, \vec{v}) = \begin{vmatrix} u_x & v_x \\ u_y & v_y \end{vmatrix}. \quad (1)$$

On note que $\vec{u} \times \vec{v} = -\vec{v} \times \vec{u}$. En fait, cette quantité représente l'aire *signée* du parallélogramme de côtés $\vec{u}$ et $\vec{v}$.

Pour revenir à l'orientation, il suffit donc de faire le produit en croix $\vec{pq} \times \vec{pr}$. Si le signe est positif, r est à gauche. Si le signe est négatif r est à droite. Si le signe est 0, r est sur la droite (pq).

Fonction ORIENTATION(**E** p, q, r : *Point*) : *réel*;
Début
  $u \leftarrow q - p$;
  $v \leftarrow r - p$;
  Retourner $u.x$ * $v.y$ - $u.y$ * $v.x$;

1.3 Comment déterminer si deux segments sont sécants ?

Un segment traverse une droite ssi ses deux extrémités sont de chaque côté. Deux segments se coupent ssi : 1) soit chaque segment traverse la droite contenant l'autre, ou 2) une extrémité d'un segment appartient à l'autre segment.

Cela nous donne donc le pseudo-code de l'Algorithme 1.

1.4 Polygones

Un *polygone* est une courbe du plan, refermée sur elle-même, composée d'une suite de segments de droites consécutifs appelés *côtés* du polygone. Les points reliant deux segments consécutifs sont les *sommets* du polygone. Un polygone est dit *simple* lorsque seuls les segments consécutifs s'intersectent. Il est clair que l'on peut caractériser un polygone par la séquence de ses sommets. Un polygone sera donc souvent représenté sous forme d'un tableau de points ou d'une liste de points.

Si le polygone est simple, par le théorème de Jordan, il coupe le plan en deux domaines dont l'un est fini, appelé *interieur* du polygone, l'autre est infini, appelé *estérieur*. Les segments eux-même forment le *contour* du polygone. Voilà quelques problèmes classiques que les outils précédents peuvent résoudre assez simplement.

Exercices

1. Si $P = (p_1, \ldots, p_n)$ est un polygone simple, montrer comment calculer son aire en un temps linéaire en n. Aide : que vaut ORIENTATION(p_1, p_i, p_{i+1}) ?
2. Donner un algorithme quadratique pour déterminer si un polygone est simple. Notez que le meilleur algorithme possible est dans $\Theta(n \log n)$.

Algorithme 1 : Algorithme décidant si deux segments s'intersectent (temps constant).

```
// Sachant que r est sur la droite (pq), détermine si r appartient au segment [pq].
Fonction SUR-SEGMENT( E p, q, r : Point ) : booléen ;
début
    Retourner min(p.x, q.x) ≤ r.x ≤ max(p.x, q.x) et min(p.y, q.y) ≤ r.y ≤ max(p.y, q.y) ;
// Détermine si les segments [p1, p2] et [p3, p4] s'intersectent.
Fonction INTERSECTION-SEGMENTS( E p1, p2, p3, p4 : Point ) : booléen;
début
    d1 ← ORIENTATION(p3, p4, p1);
    d2 ← ORIENTATION(p3, p4, p2);
    d3 ← ORIENTATION(p1, p2, p3);
    d4 ← ORIENTATION(p1, p2, p4);
    si ((d1 < 0 et d2 > 0) ou (d1 > 0 et d2 < 0)) et ((d3 < 0 et d4 > 0) ou (d3 > 0 et d4 < 0))
     alors Retourner Vrai ;
    sinon si d1 = 0 et SUR-SEGMENT(p3, p4, p1) alors
        Retourner Vrai ;
    sinon si d2 = 0 et SUR-SEGMENT(p3, p4, p2) alors
        Retourner Vrai ;
    sinon si d3 = 0 et SUR-SEGMENT(p1, p2, p3) alors
        Retourner Vrai ;
    sinon si d4 = 0 et SUR-SEGMENT(p1, p2, p4) alors
        Retourner Vrai ;
    sinon Retourner Faux ;
```

1.5 Intersection dans une soupe de segments

Etant donné un ensemble de segments, on veut savoir s'il existe (au moins) une intersection. Le meilleur algorithme est en $\Theta(n \log n)$. C'est un algorithme utilisant une technique de balayage, assez classique en géométrie algorithmique.

Le principe est de trier les extrémités des segments suivant une direction, mettons x. Ensuite, on maintient une structure ordonnée T des segments actifs (intersectés par la droite de balayage courante), dont l'ordre est l'ordre des points d'intersection sur la droite de balayage. En général, on choisit un arbre rouge-noir ou un ABR pour cette structure.

Ensuite, deux segments s'intersectent si et seulement si à un moment dans le balayage ils sont côte à côte dans la structure et qu'ils s'intersectent. On a donc seulement à regarder à chaque insertion de segment (lorsqu'une extrémité gauche croise le balayage) et à chaque suppression de segment (lorsqu'une extrémité droite croise le balayage) si autour de ce segment il y a une intersection. Il est inutile de regarder ailleurs.

Le code final requiert $O(n \log n)$ pour le tri initial, puis $\log n$ par extrémité traitée à cause du temps pour réaliser l'insertion ou la suppression dans l'arbre ordonné T.

Exercices

1. Exhibez un ensemble de n segments qui induit le plus d'intersections possible entre ces segments.
2. En déduire la complexité minimale en pire cas d'un algorithme pour déterminer toutes les intersections d'un ensemble de segments.
3. Donnez un algorithme qui a cette complexité.

1.6 Convexité et enveloppe convexe

Dans n'importe quel espace euclidien (et donc notamment $\mathbb{R}^2$ et $\mathbb{R}^3$), un ensemble C de cet espace est dit *convexe* ssi $\forall p, q \in C, [pq] \subset C$. Le carré, le losange, le rectangle, le triangle, le disque sont des exemples d'ensembles convexes. Ces ensembles ont des propriétés remarquables et beaucoup d'algorithmes sont facilités lorsque l'on sait que la donnée est convexe en entrée. Par exemple, l'algorithme GJK – algorithme très populaire de détection dynamique de collisions utilisé dans les jeux 3D — détermine les collisions entre ensembles convexes de manière très rapide.

Parfois, on approche une forme géométrique complexe par un ensemble convexe. Ainsi, c'est ce que l'on fait lorsqu'on utilise des boîtes englobantes. Par exemple, avant d'aller tester si réellement deux formes complexes s'intersectent, on teste d'abord si leurs deux boîtes englobantes s'intersectent.

Un ensemble convexe englobant une forme S mais plus précis que la boîte englobante est l'*enveloppe convexe de* S, notée $\text{Conv}(S)$. Il est défini comme l'intersection de tous les ensembles convexes qui contiennent S. On peut aussi le définir comme l'intersection de tous les demi-plans qui contiennent S. Au sens où il est contenu dans tous les autres convexes contenant S, c'est le plus petit convexe qui contient S.

Comme beaucoup d'algorithmes utilisent la convexité, il est important de savoir calculer efficacement l'enveloppe convexe d'un ensemble. Nous regardons ici des algorithmes qui calculent l'enveloppe convexe d'un ensemble de points.

L'enveloppe convexe d'un ensemble de points $\{p_0, \ldots, p_{n-1}\}$ a certaines propriétés. Il s'agit d'un polygone simple dont les sommets doivent être trouvés parmi les p_i. On peut le voir (en 2D) comme un élastique que l'on lâche autour des points et qui se reserre au plus court.

Chacun de ses côtés vérifie la propriété que tous les autres points sont sur la gauche de ce côté. Utilisez cette propriété pour trouver un algorithme qui affiche la liste des côtés de l'enveloppe convexe sous forme de paires de points (pas forcément dans l'ordre). L'algorithme naïf ainsi obtenu est en $\Theta(n^3)$.

Il y a beaucoup d'algorithmes pour calculer l'enveloppe convexe de points dans le plan : méthodes incrémentielles en triant les points de gauche à droite, la méthode diviser pour régner, la méthode par greffes successives. Nous en présentons deux : le balayage de Graham et le parcours de Jarvis.

Le balayage de Graham (Algorithme 2) construit un ensemble étoilé, puis, en partant d'un sommet qu'on sait être sur l'enveloppe convexe, ne conserve que les points sur les branches de l'étoile. La preuve de son exactitude se fait en montrant qu'à chaque début d'itération, les points sur la pile forment l'enveloppe convexe des points $p_1, \ldots, p_{i-1}$, stockés dans l'ordre contraire. La complexité de cet algorithme est dominée par la complexité du tri, et donc en $\Theta(n \log n)$.

Algorithme 2 : Balayage de Graham pour calculer l'enveloppe convexe.

```
Action BALAYAGE-GRAHAM( E P : tableau[1,n] de Point, S S : pile de Point) ;
Var : i : entier ;
début
    Cherchez P[i] le point de P d'ordonnée minimale, et si égalité, d'abscisse minimale ;
    ECHANGER( P[i], P[1] );
    TRIER( P, 2, n ) ;          // tri selon l'angle polaire mesuré par rapport à P[1]
    CRÉERPILE(S);
    EMPILER(S, P[1]);
    EMPILER(S, P[2]);
    EMPILER(S, P[3]);
    pour i de 4 à n faire
        tant que ORIENTATION(SOUS-SOMMET(S), SOMMET(S), P[i]) ≤ 0 faire
            DÉPILER(S);
        EMPILER(S, P[i]);
```

Le parcours de Jarvis construit l'enveloppe convexe en deux parties, la partie droite puis la partie gauche. L'idée est de partir du sommet le plus bas, puis de chercher le sommet au-dessus faisant le plus petit angle polaire. On continue jusqu'à arriver tout en haut. On fait pareil de l'autre côté. La

complexité est en $O(nh)$ où h est le nombre de sommets de l'enveloppe convexe. On parle d'algorithme *output-sensitive* ou *de complexité dépendante de la sortie*. Dans le cas où $h = o(\log n)$, il vaut mieux utiliser Jarvis que Graham.

Algorithme 3 : Algorithme de Jarvis pour calculer l'enveloppe convexe (côté droit).

```
Action JARVIS-DROIT( E P : tableau[1,n] de Point, S S : pile de Point) ;
début
    Cherchez P[i] le point de P d'ordonnée minimale, et si égalité, d'abscisse minimale ;
    ECHANGER( P[i], P[1] );
    EMPILER( P[1] );
    trouve ← vrai;
    tant que trouve faire
        j ← 2;
        pour i de 3 à n faire
            si P[i] ≠ SOMMET(S) et ORIENTATION(SOMMET(S), P[j], P[i]) ≤ 0 et
              AU-DESSUS(P[i], SOMMET(S)) alors
                j ← i;
        si AU-DESSUS(P[j], SOMMET(S)) alors EMPILER(S, P[j]) ;;
        else trouve ← faux;
// Retourne ''vrai'' si et seulement p est au-dessus de q.
Fonction AU-DESSUS( E p, q : Point ) : booléen ;
début
    Retourner p.y > q.y ou (p.y = q.y et p.x > q.x );
```

A noter, le meilleur algorithme de calcul d'enveloppe convexe de points est en $\Theta(n \log h)$. Si la séquence de points $P[i]$ forme une ligne polygonale sans auto-intersection, alors son enveloppe convexe se calcule en temps $\Theta(n)$ par l'algorithme de Melkman.

Exercices

1. Est-il nécessaire de calculer les angles polaires pour faire le tri dans le balayage de Graham ?
2. Même question pour Jarvis.

1.7 Structures pour découper le plan ou l'espace

Ces structures permettent de représenter des données géométriques du plan ou de l'espace de façon hiérarchisée, afin d'accélérer les requêtes géométriques. Elles permettent par exemple de savoir quels sont les objets géométriques les plus proches d'un point de l'espace.

Les structures stockant des points les plus simples (et sans doute aussi les plus utilisées) sont :

- les arbres 2^k-aires (arbres binaires de recherche en 1D, arbres quaternaires ou *quadtrees* en 2D, *octrees* en 3D) qui sont des extensions des arbres binaires de recherche en toute dimension. Ici chaque point situé à un nœud de l'arbre coupe l'espace en autant de régions qu'il y a d'orthants.
- les arbres kD qui sont des arbres binaires. Chaque point situé à un nœud de l'arbre coupe l'espace en deux régions selon un plan qui dépend de la profondeur dans l'arbre. Ainsi les nœuds de profondeur 0 coupe l'espace selon leur première coordonnée x, les nœuds de profondeur 1 coupe l'espace selon leur deuxième coordonnée y, les nœuds de profondeur 2 coupe l'espace selon leur troisième coordonnée z, etc, en recommançant à la première coordonnée après avoir coupé selon la dernière.

Les seconds ont l'avantage d'être plus facilement équilibrables que les premiers. Il suffit en effet à chaque fois de choisir comme point de découpe le point situé à la médiane des points restants selon la direction de découpe. Une illustration est donnée sur la Figure 1.

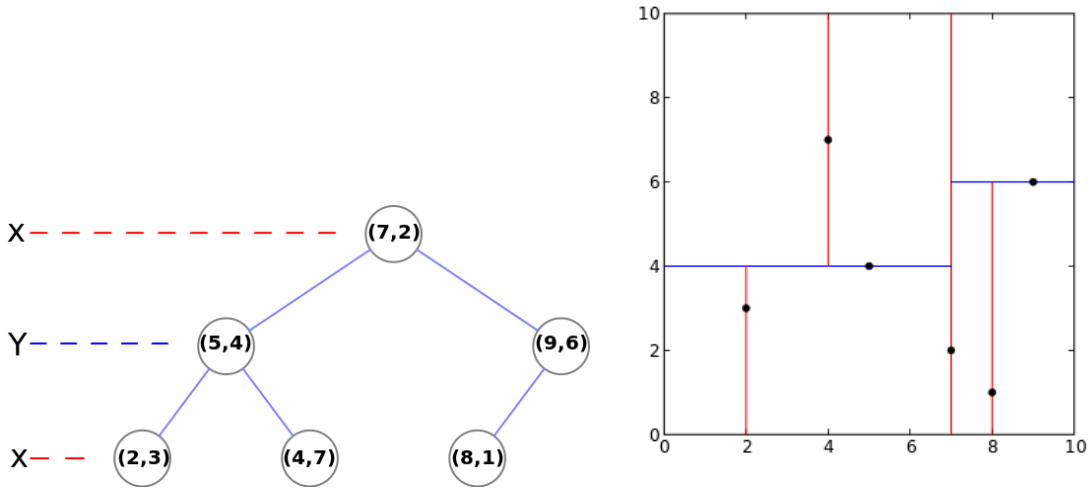

FIGURE 1 – Décomposition en arbre kD des points (2,3), (5,4), (9,6), (4,7), (8,1), (7,2). A gauche, la vision arbre binaire. A droite, le découpage du plan correspondant.

L'algorithme de construction d'un kD-tree bien équilibré est donné par Algorithme 4. En fait, un tel arbre défini un ordre total entre les points (mais qui n'est pas calculable en temps constant).

Algorithme 4 : Créer un arbre kD à partir des points d'un tableau de points T entre les indices i et j. L'entier a est l'axe initial utilisé entre 0 et $k-1$, où k est une constante égale à la dimension de l'espace. Attention, le tableau T voit l'ordre de ses points modifié dans la construction.

Fonction CRÉER-ARBRE-KD*(ES T : Tableau[0..MAX] de Point, E i, j, a : entier) : Arbre ;*
Var : A : Arbre ; p : Point ; m : entier ;
début
  si $i > j$ **alors**
    Retourner *Null ;*
  si $i = j$ **alors**
    CRÉER-ARBRE$(A, T[i])$;
    Retourner A ;
  $p \leftarrow$ MÉDIANE-SELON-AXE(T, i, j, a) ;
  PARTITION(T, i, j, p);
  $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$;
  CRÉER-ARBRE$(A, T[m])$;
  MODIFIER-GAUCHE$(A,$ RACINE$(A),$ CRÉER-ARBRE-KD$(T, i, m-1, (a+1)\%k))$;
  MODIFIER-DROITE$(A,$ RACINE$(A),$ CRÉER-ARBRE-KD$(T, m+1, j, (a+1)\%k))$;
  Retourner *A;*

Ce type de structure accélère en général les requêtes de proximité géométrique en changeant un facteur linéaire n par un facteur logarithmique h, où h est la hauteur de l'arbre ($h = O(\log n)$ si on utilise la consruction de l'Algorithme 4). On peut citer par exemple :

- Déterminer si un point p appartient à la structure prend un temps $O(h)$.
- Déterminer s'il existe un point dans la structure à distance ϵ d'un point p prend un temps $O(h)$, si ϵ est de l'ordre de la plus petite distance entre deux points (voir Algorithme 5).
- Déterminer la liste des m points dans une zone (pour une zone de géométrie simple) prend un temps de l'ordre de $O(mh)$.
- Déterminer les deux points les plus proches prend un temps $O(nh)$.

Algorithme 5 : Sort dans la file F les points de A qui sont dans la boule de centre p et de rayon r. L'entier a désigne l'axe courant dans le kD-tree.

Action POINTS-DANS-BOULE*(S F : File de Point, E A : kD-tree, E N : Noeud, E p : Point, E r : réel, E a : entier);*
Var : q : Point;
début
  si $N \neq Null$ **alors**
    $q \leftarrow$ VALEUR(A, N);
    si DISTANCE$(p, q) \leq r$ **alors** ENFILER(F, q);
    si $p[a] \leq q[a] + r$ **alors**
      POINTS-DANS-BOULE$(F, A,$ GAUCHE$(A, N), p, r, (a+1)\%k)$;
    si $p[a] \geq q[a] - r$ **alors**
      POINTS-DANS-BOULE$(F, A,$ DROITE$(A, N), p, r, (a+1)\%k)$;

En résumé : Géométrie algorithmique

— Les imprécisions numériques rendent certains calculs géométriques délicats sur ordinateur. On préferera donc utiliser des opérations qui minimisent les erreurs numériques : plus, moins, fois, puissance, mais on évitera divisions et racines carrées. Le produit en croix est l'outil essentiel pour positionner les points dans le plan.

— Les ensembles/polygones convexes sont très importants en géométrie algorithmique, et servent à rendre robuste ou accélerer les calculs, ou avoir de premières approximations de résultats.

— Les structures de proximité sont très souvent des arbres, qui hiérarchisent les points et leurs zones d'influence. En général ils accélèrent les temps de calcul avec des complexités en pire cas proportionnelles à leur profondeur (en général $\mathcal{O}(\log n)$ si n est le nombre de points).

illustrations

illustrations