Complexité algebrique des algorithmes géometrique — le problème d'intersections d'un ensemble de segments

Radu-Lucian Lupşa

January 30, 2004

Abstract

Il est connu qu'un problème difficil dans l'implantation des algorithmes géometriques est le calcul exact des points d'intersection des droites ou des courbes, ainsi que l'évaluation des prédicats géometriques. D'un part, il s'agit du fait que, si les calcules sont inéxactes, il est possible qu'un algorithme se comporte complètement imprevisible. D'autre part, beaucoup d'algorithmes ne traitent pas explicitement les cas particuliers dégénérés.

On étudie dans cet article le cas de l'algorithme de Balaban pour trouver les intersections d'un ensemble de segments, du point de vue de l'evaluation des prédicats géometriques.

1 Introduction

L'article fait un étude sur les problèmes liés à l'evaluation des predicats et au traitement des cas dégénérés. Le cas étudié est celui de l'algorithme de Balaban pour trouver les intersections du'un ensemble de segments.

Le reste de l'introduction presente l'algorithme de Balaban [1], y compris les définition dont on a besoin à la suite. Puis, on présente létude de l'auteur sur les solution au problème des cas dégénérés, au problème de lévaluation des prédicats et finalement une comparaison des performances.

Quelques notations et définitions:

Si la coordoné y des extremités d'un segment ne nous interrèsse pas, le segment sera noté (l, r), où l et r sont les abscises gauche.

On appelle verticale une droite verticale passant par une extrémité de segment. La verticale v est donc la droite $x = v$.

On appelle bande une partie du plan délimitée par deux verticales.

On dit qu'un segment (l, r)

- est interieur pour la bande (b, e) si $l > b$ et $r < e$;
- traverse la bande (b, e) si $l \leq b$ et $r \geq e$.

On note $\mathrm{Int}_{b,e}(S_1, S_2)$, où S_1 et S_2 sont des ensembles quelquonque de segments l'ensemble des pairs $(s_1, s_2) \in S_1 \times S_2$ de segments qui s'intersectent dans la bande (b, e).

Une liste de segments qui intersectent une verticale v est triée par rapport à cette verticale si elle est triée dans l'ordre croissante des ordonée des points d'intersection entre les segments de la liste et la verticale v.

Enfin, on appelle un escalier un triplet (b, e, Q) où (b, e) est une bande et Q est un ensemble de segments qui traversent la bande (b, e) et ne s'intersectent pas à l'interieur de la bande (l'ensemble Q doit être trié d'après la règle ci-dessus par rapport aux verticales b et e (l'ordre est la même par rapport aux deux)).

Pour trouver les intersections à l'interieur d'une bande elementare (sans extrémitées de segments à l'interieur), on utilise l'algorithme suivant:

procedure *SearchInStrip*(b, e, L, R)
    Split(b, e, L, Q, L')
    if $Q = \emptyset$ **then**
        $R := L$
    else
        trouver et ecrire $Int_{b,e}(Q, L')$
        SearchInStrip(b, e, L', R')
        Merge(b, e, R', Q, R)
    endif
end

où

- *Split*(b, e, L, Q, L') divise la liste L triée par rapport à la verticale b dans l'escalier maximal (au sens d'inclusion d'ensembles) (b, e, Q) et la liste restante L'.
- *Merge*(b, e, R', Q, R) fusione la liste R' triée par rapport à la verticale e avec l'escalier (b, e, Q) en sortant la liste R triée par rapport à e.

Ainsi *SearchInStrip* divise la liste initiale L en deux, Q et L'; maintenant

$$\mathrm{Int}_{b,e}(L, L) = \mathrm{Int}_{b,e}(L', L') \cup \mathrm{Int}_{b,e}(L', Q) \cup \mathrm{Int}_{b,e}(Q, Q)$$

où $\mathrm{Int}_{b,e}(Q, Q) = \emptyset$ et $\mathrm{Int}_{b,e}(L', L')$ est trouvé par l'appel récursiv.

Il faut remarquer aussi que *Split* obtient aussi les positions des segments de L' dans l'escalier Q, de maniere que les intersections d'un segment de L' avec les marches de Q puissent être trouvées en balaiant Q à partir de la position donnée par *Split* dans les deux sens jusqu'à trouver une marche qui n'intersecte pas le segment.

Passons maintenant à l'algorithme proprement-dit. Balaban donne au fait deux algorithmes, dont le premier est un peu plus simple, mais asimptotiquement sub-optimal (complexité $\mathcal{O}(n \log^2 n + k)$, où n est le nombre de segments et k le nombre d'intersections), et le deuxième est optimal ($\mathcal{O}(n \log n + k)$) mais plus compliquée.

L'algorithme sub-optimal est le suivant

procedure *TreeSearch*(b, e, L, I, R)
  if b et e consecutives **then**
    SearchInStrip(b, e, L, R)
  else
    c :=la verticale qui divise en deux l'ensemble
      des verticales comprises entre b et e
    Split(b, e, L, Q, L_1)
    trouver $\mathrm{Int}_{b,e}(Q, L_1)$
    $I_1 := \{s \in I | s$ interieur pour $(b, c)\}$
    TreeSearch(b, c, L_1, I_1, R_1)
    s :=le segment dont une extremit
    if s commence sur c **then**
      $L_2 := R_1 \cup \{s\}$
    else
      $L_2 := R_1 \setminus \{s\}$
    endif
    $I_2 := \{s \in I | s$ interieur pour (c, e)
    TreeSearch(c, e, L_2, I_2, R_2)
    trouver $\mathrm{Int}_{b,e}(Q, I)$
    Merge(b, e, R_2, Q, R)
    trouver $\mathrm{Int}_{b,c}(Q, R_2)$
  endif
end

begin
  $L := \{$le segment qui commence sur la première verticale$\}$
  I :=l'ensemble initial moins L moins le segment qui
    finit sur la derniere verticale
  TreeSearch$($*première vert., dernière vert.*$, L, I, R)$
end

Dans cet algorithme, la seule operation qui demande un temps plus grand que $\mathcal{O}(n \log n + k)$ est *trouver* $\mathrm{Int}_{b,e}(Q, I)$, car elle nécesite une recherche binnaire de la position de chaque segment de I dans Q. Pour éviter ceci, la solution consiste à garder les positions des segmentes de I trovée dans les appels récursives de *TreeSearch*; en ce but il faut garder des rélations entre les escaliers générés au differents niveaux d'appel de *TreeSearch*.

2 Cas particuliers et dégénérées

Le premier probleme avec cet algorithme est le fait qu'il ne traite pas éxplicitement les cas particuliers:

- si l'intersection de deux segmentes se trouve exactement sur une verticale;

- si le point d'intersection se trouve sur une extremité de segment;
- si deux extremitées de segment se trouvent sur la même verticale;
- si un segment est vertical;
- si deux segments se superposent (ont la même droite-support).

A ce point il est utile de regarder un peu la preuve de correction de l'algorithme. On s'apperçoit alors que les démandes sur les prédicats sont les suivantes:

1. Si deux segments ne s'intersectent pas dans une bande, alors leur ordre relative sur les verticale gauche et droite doit être la même;
2. Si un segment est localisé entre deux marches m_i et m_{i+1} d'un éscalier $Q_{b,e}$ et il n'intersecte ni m_i, ni m_{i+1}, alors il ne doit intersecter aucune autre marche de $Q_{b,e}$;
3. Un segment ne peut intersecter que des marches consecutives d'un même escalier.

Prenons les cas un à la fois.

Si le point d'intersection se trouve exactement sur une verticale, il suffit de considerer q'il se trouve "un peu à droite" ou "un peu à gauche", donc dans une des deux bandes délimitées par la verticale en question. Mettons-le donc à gauche. Maintenant pour garder la cohérence, si deux degmentes s'intersectent sur une verticale, il faut mettre en premier sur la liste triée celui qui se trouvera en premier sur la verticale suivante (car ils ne s'intersecteront plus dans la bande de droite), c'est-à-dire le premier en ordre trigonométrique dans le demi-plan droit.

Maintenant si le point d'intersection coïncide avec une extrémité, il se trouvera donc sur une verticale, et conformement au paragraphe precedent il va être consideré dans la bande de gauche. S'il coïncide avec l'extremité droite du segment, il va être traité comme s'il était à l'intereur; par contre s'il coïncide avec l'extremité gauche il ne sera pas vu car le segment n'existe pas dans la bande gauche. Ce cas d'intersection doit donc être tarité lors de l'insertion du segment sur la liste de segments correspondante à cette verticale. *Remarque:* si au moment de l'insertion l'autre segment est une marche, l'intersection sera vue normalement par l'algorithme.

S'il existe plusieurs extrémitées de segment sur la même verticale, il y a deux solutions: soit on considere des bandes de longueur zero, soit on prépare la liste triée des extrémitées (avec les segments correspondants) et on la fusione avec la liste des segments associés à la verticale. J'ai pris la deuxiéme approche. En faisant ainsi, le traitement d'une verticale ne consiste plus seulement à ajouter ou effacer un segment (obtenir la liste L_2 à partir de la liste R_1), mais à fusioner la liste L_1 avec la liste des segments qui commencent sur la verticale traitée et a effacer par la même occasion les segments qui finissent sur la verticale traitée.

En même temps, on garde une liste de segments verticaux, qu'on met au jour chaque fois qu'on rencontre une extrémité d'un segment vertical.

Enfin, le dernier problème concerne le cas où deux segments partagent la même droite support et leur intersection est un segment. Pour les segments verticaux, ceci n'est pas dérangeant, leur intersection etant trouvée lors du traitement de la verticale sur laquelle ils se trouvent. Si, au contraire, les segments sont obliques ou horizontals, ils s'intersectent dans plusieurs bandes en agrandisant le nombre d'intersections et donc le temps de calcule. (n segmentes superposés peuvent donner ainsi jusqu'à $\frac{1}{6}(2n^3 + n)$ "points" d'intersection). Dans ce cas on peut considerer qu'il s'intersectent toujours en un seul point; soit ce point le point le plus à droite de l'intersection. L'ordre de ces segments sur une liste associée à une verticale n'a pas alors d'importance.

3 Evaluation des prédicats

Il est connu que les algorithmes géométriques sont très sensibles aux erreurs numériques; d'ici la nécessité de faire les calcules exactes, ou au moins que les valeurs de vérité des divers predicats évalués pendant l'execution de l'algorithm soit cohérentes; sinon il y a le risque que la réponse de l'algorithme ne soit ni même une approximation du résultat reel, ou même que l'algorithme ne s'arrête pas.

Les calcules exactes d'autre part sont chères car non supportés directement par le materiel de l'ordinateur. Et dans tous les cas, plus longue soit la réprésentation, plus long est le temps de calcule. L'opération la plus chère en terme de démande de chiffres significatives sur la réprésentation etant la multiplication, on voit l'interrêt de réduire le plus possible le dégrée des polinômes qui apparaissent dans l'evaluation des prédicates.

Dans le cas de l'algorithme de Balaban, les prédicates décrits dans la section precedante sont basée sur les trois suivants ($<_x$ signifie "l'abscise du premier point est plus petite que celle du deuxième et $<_y$ signifie la même chose sur l'ordonée):

1. étant donée 2 points, p_1 et p_2, est-ce que $p_1 <_x p_2$?
2. étant donée 3 points, est-ce que p_3 se trouve à gauche, sur ou à droite de la droite p_1p_2 (orientée de p_1 vers p_2)
3. étand donée 5 points $p_0 \ldots p_4$ tels que $\{q\} = p_1p_2 \cap p_3p_4$, est-ce que $p_0 <_x q$ (q est inconnu et il n'est pas nécessaire de le connaître éxplicitement)?

En supposant une réprésentation cartesienne des points, l'évaluation des trois prédicats ci-desus est equivalente á trouver le signe d'un polynôme de degrée 1,2 respectivement 3, irreductible. Pour évaluer ledit signe, il faut en principe travailler avec des nombres ayant 3 fois plus de chiffres que les coordonées des points. Mais dans [2] il est montré que l'evaluation "presque exacte" du predicate 3 est possible en travaillant en réels (IEEE 754) double-précision, les coordonées des points étant simple-précision. Ce "presque exacte" veut dire

que si le signe calculé du polynôme est -1 alors le signe réel est aussi -1; même chose pour $+1$; mais si le signe calculée est 0 alors on ne sait rien sur le signe réel.

4 Les performances

Dans le tableau ci-dessous sont marqués les temps de calcul utilisés (sur le même ordinateur) par l'algorithme naïf et l'algorithme sub-optimal et l'algorithme optimal de Balaban en utilisant chacun les predicates géométriques de CGAL et des predicates écrits en utilisant les réels double-précision.

Les exemples de test ont été génerés soit aléatoirement, soit (pour les dernières 4) sous la forme d'une grille carrée.

Méthode génération	Nr. seg.	Nr. pairs	Temps alg. naïf	Temps alg. sub-opt CGAL	Temps alg. optimal CGAL	Temps alg. sub-opt double-prec	Temps alg. optimal double-prec
random	100	215	1	1	4	1	1
random	200	886	3	3	11	1	2
random	800	14113	45	29	97	8	24
random	2000	6869	—	28	129	7	29
random	5000	43184	—	122	550	28	125
random	2000	89352	—			47	99
random	5000	555640	—			261	382
grille	800	2281	15	1	1	1	1
grille	1800	5221	84	1	1	1	1
grille	5000	14701	—	2	2	4	2
grille	20000	59401	—	9	8	16	8

On voit facilement que pour des exemples de taille raisonnable, l'algorithme sous-optimal asimptotiquement se comporte nettement mieux, le gain d'un facteur $\log n$ (7–13 dans les exemples ci-dessus) étant contrebalancé par la perte due aux complications de l'algorithme. Il faut remarquer aussi le gain de vitesse obtenu en remplaceant les rationnels en précision illimité (bibliothéque LEDA) par les réels double-precision de la machine.

References

[1] I. Balaban, An Optimal Algorithm for Finding Segment Intersections, Proceedings of the Eleventh Annual Symposium on Computational Geometry, Vancouver, Canada, June 5-7, 1995, pg. 211-219

[2] Jean-Daniel Boissonnat, Franco P. Preparata, Robust Plane Sweep for Intersecting Segments, Rapport de recherche no. 3270 septembre 1997, Institut National de Recherche en Informatique et en Automatique