Algorithmique 2

Ensimag - 1A

Mai 2014

Durée : 3h
Machines électroniques interdites **documents autorisés : notes de cours**

Les différentes parties du sujet sont indépendantes.

Le barème est donné à titre indicatif.

---

1 Intersections d'ensembles de segments (13 points)

On se place pour cet exercice dans le plan. Chaque point P est ici défini par 2 coordonnées flottantes $(P_x, P_y) \in \mathbb{R}^2$. Un segment de droite s est défini comme un couple de points p_1, p_2. Nous considérons ici le problème consistant à calculer à partir d'un ensemble S de n segments différents l'ensemble V de tous les points d'intersection des segments de S.

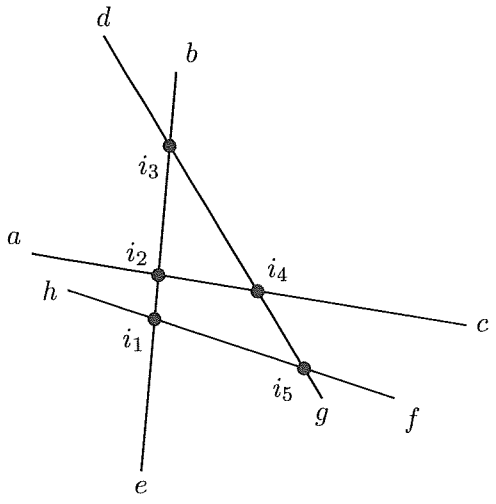

FIGURE 1 – Exemple de calcul d'intersections

La figure 1 illustre ainsi notre problème. Nous avons en entrée les segments $(h, f), (e, b), (a, c), (d, g)$ sous forme d'un *vecteur* et en sortie, un *vecteur* contenant les points i_1, i_2, i_3, i_4, i_5.

Dans un souci de simplification, nous supposerons par la suite que les opérations usuelles sur les vecteurs sont réalisées en temps $O(1)$ en pire cas (au lieu de $O(1)$ en coût amorti).

Nous supposerons également qu'il n'est *pas possible* que plus de deux segments s'intersectent en un même point, que tous les points (d'entrée ou d'intersection) ont une abscisse différente, et enfin qu'aucune extrémité d'un segment ne se trouve sur un autre segment.

Enfin, nous disposons d'une fonction *intersection* renvoyant l'intersection de *deux* segments, si elle existe, en temps $O(1)$.

1.1 Algorithme naïf

1.1.1. (1 point) Donnez en fonction de n un ordre de grandeur asymptotique (Θ) du nombre de points d'intersection au pire cas. Justifiez.

1.1.2. (1 point) Écrire en pseudo-code un algorithme très simple de coût $O(n^2)$ de calcul du vecteur V des points d'intersection.

1.2 Algorithme de Bentley-Ottmann

On se propose d'améliorer l'algorithme précédent. Pour ce faire, nous introduisons une notation supplémentaire : k, le nombre de points d'intersection renvoyés.

L'algorithme de Bentley-Ottmann fonctionne en simulant un balayage du plan par une droite verticale Δ se déplaçant de gauche à droite. L'espace ainsi balayé est continu mais en fait seul un nombre fini d'abscisses nous intéresse : là où "il se passe quelque-chose d'intéressant". On ne considère donc que les "événements" intéressants : début d'un segment, fin d'un segment, intersection.

L'algorithme utilise deux structures de données différentes :

— une file de priorité F contenant les événements à venir (avec comme priorités l'opposé de leurs abscisses) ;

— un dictionnaire D contenant les segments actuellements coupés par la droite Δ (classés par ordonnée croissante d'intersection avec Δ pour son abscisse courante).

Nous avons une propriété importante : si deux segments s_1 et s_2 s'intersectent, alors il existe une abscisse pour Δ pour laquelle s_1 et s_2 sont côte à côte dans D. Nous en déduisons l'algorithme 1 (avec l'aimable participation de Wikipedia).

```
1  intialiser F, D et V (vecteur résultat) à des structures vides;
2  pour chaque segment s ∈ S faire
3  |  ajouter les deux événements "début de s" et "fin de s" à F;
4  fin
5  tant que F n'est pas vide faire
6  |  récupérer e, événement le plus prioritaire de F, et l'enlever de F;
7  |  si e n'a pas encore été vu alors
8  |  |  si e correspond au début d'un segment s alors
9  |  |  |  ajouter s à D;
10 |  |  |  soient r et t les segments immédiatement au-dessous et au-dessus de s dans D;
11 |  |  |  si r existe et s intersecte r alors
12 |  |  |  |  ajouter l'événement "intersection(s,r)" dans F;
13 |  |  |  fin
14 |  |  |  si t existe et s intersecte t alors
15 |  |  |  |  ajouter l'événement "intersection(s,t)" dans F;
16 |  |  |  fin
17 |  |  sinon si e correspond à la fin d'un segment s alors
18 |  |  |  soient r et t les segments immédiatement au-dessous et au-dessus de s dans D;
19 |  |  |  enlever s de D;
20 |  |  |  si r et t existent et r intersecte t alors
21 |  |  |  |  ajouter l'événement "intersection(r,t)" dans F;
22 |  |  |  fin
23 |  |  sinon
24 |  |  |  nous sommes sur une intersection, l'ajouter à V;
25 |  |  |  on récupère s1 et s2 les deux segments intersectés;
26 |  |  |  échanger les positions de s1 et s2 dans D;
27 |  |  |  trouver le segment r juste au dessous de s1 dans D;
28 |  |  |  trouver le segment t juste au dessus de s2 dans D;
29 |  |  |  si r existe et r intersecte s2 alors
30 |  |  |  |  ajouter l'événement "intersection(r,s2)" dans F;
31 |  |  |  fin
32 |  |  |  si t existe et t intersecte s1 alors
33 |  |  |  |  ajouter l'événement "intersection(t,s1)" dans F;
34 |  |  |  fin
35 |  |  fin
36 |  fin
37 fin
38 retourner V;
```

Algorithme 1 : Bentley-Ottmann

1.2.1. (0.5 points) Exécutez l'algorithme de Bentley-Ottmann sur l'entrée de la figure 1. Vous afficherez l'ensemble des événements rencontrés (dans l'ordre où ils sont rencontrés) ainsi que le changement du contenu de D lors de l'événement "intersection de (d, g) et (b, e)".

1.2.2. (1 point) Expliquez comment réaliser l'opération de la ligne 7 vérifiant si un évènement a déjà été traité en temps $O(1)$. Vous préciserez les structures de données utilisées.

1.2.3. (1 point) Donnez une borne supérieure sur le nombre d'éléments de F. (Justifiez)

1.2.4. (1 point) Proposez une structure de donnée efficace pour stocker F. Quel est le coût total de toutes les insertions et toutes les suppressions dans F ?

1.2.5. (1 point) On s'intéresse maintenant au choix de la structure utilisée pour le dictionnaire

contenant les segments. On propose d'utiliser un arbre binaire de recherche. Nous supposerons ici que les choses se passent bien et que la hauteur de l'arbre est logarithmique. Nous supposons également disposer dans chaque nœud d'un pointeur vers le nœud père (en plus des pointeurs vers les fils). Chaque clef est un segment. Les arbres binaires de recherche fonctionnent à l'aide d'une fonction de comparaison. Écrivez en pseudo-Ada le code de la fonction de comparaison *(Indication: vous utiliserez une variable globale X contenant l'abscisse courante de la droite de balayage).* L'objectif est qu'un parcours infixe en profondeur voit tous les segments de bas en haut pour l'abscisse X.

1.2.6. (2 points) Étant donné un pointeur sur un nœud de l'arbre (contenant un segment donc), comment retrouver le segment situé immédiatement au-dessous et celui situé immédiatement au-dessus ? Quel est le coût au pire cas d'une telle opération ?

1.2.7. (1 point) Écrivez en pseudo-Ada une procédure réalisant l'échange de deux segments dans l'arbre. Vous ferez très attention à bien expliquer le prototype (le profil, l'en-tête, la signature) de votre procédure et à en préciser les éventuelles pré-conditions.

1.2.8. (1 point) Quel est le coût total (au pire cas), en fonction de n et k de tous les accès à D ?

1.2.9. (1.5 points) Donnez le coût total de l'algorithme (au pire cas).

1.3 Application-extension

On dispose de deux ensembles de segments, R contenant une abstraction du réseau routier et S contenant une abstraction de l'ensemble des sentiers de randonnée sur le département de l'Isère. Un sentier ou une route est dans la réalité courbe, mais cette courbe est approchée par un ensemble de segments.

1.3.1. (1 point) On cherche à trouver les croisements entre les routes et les sentiers. Quel est le meilleur choix d'algorithme ? Justifiez-vous en analysant les caractéristiques des données en entrée.

2 Algorithmes de sélection

Dans ce second exercice, on cherche à résoudre le problème de la *sélection* du k$^{\text{ème}}$ élément. On dispose en entrée d'un tableau T de n éléments $T(1), T(2), \ldots, T(n)$. T n'est pas nécessairement trié. On dispose également d'un entier k tel que $1 \leq k \leq n$. On cherche à déterminer en sortie la valeur du k$^{\text{ème}}$ plus petit élément de T. Par exemple pour $k = 1$ on doit renvoyer la valeur du plus petit élément de T. Nous supposerons pour faire simple que tous les éléments de T sont différents.

Une solution naïve au problème consiste à trier T par ordre croissant puis à renvoyer $T(k)$. Malheureusement cette solution coûte $O(n \log(n))$ ce qui est un coût élevé pour un problème aussi simple.

2.1 Quickselect (7 points)

On propose un premier algorithme utilisant une procédure de *Segmentation* similaire à celle du *Quicksort*. On tire *au hasard* un indice pivot p du tableau puis la segmentation place en temps linéaire (et en n comparaisons) tous les éléments inférieurs à p devant p et tous les éléments supérieurs à p derrière p. La procédure retourne également la nouvelle place du pivot. Nous supposerons disposer d'une procédure de segmentation et son code n'est donc pas à écrire.

2.1.1. (1 point) Utilisez la procédure de segmentation pour écrire un algorithme récursif (en pseudo-Ada) résolvant *sélection*.

2.1.2. (1 point) L'algorithme étant probabiliste, on cherche à calculer l'espérance (sur les tirages réalisés) du coût. En considérant un tirage uniforme du pivot, proposez une formule récursive calculant $C(n)$, l'espérance du coût pour la sélection d'un élément dans un tableau de taille n. On ne comptera dans ce coût que les comparaisons d'éléments du tableau.

2.1.3. (2 points) Montrez par récurrence que $C(n) = r \times n$ où r est une constante. On s'autorisera à borner en remplaçant k par $n/2$ dans nos formules (ce qui correspond au pire cas).

2.2 Soft Heaps

On cherche maintenant à obtenir un algorithme *déterministe* de performance équivalente. Pour ce faire, nous gardons le même principe de récursion, mais modifions la manière de choisir le pivot.

Nous utilisons une variante des tas appellée *soft heap* et introduite par Chazelle en 2000. Il s'agit comme un tas d'une file de priorité. On insère donc des éléments avec une certaine priorité puis on dispose d'une opération permettant d'extraire l'élément le plus prioritaire du tas.

Le soft heap à la particularité de *corrompre* certaines priorités ce qui signifie qu'elles peuvent alors prendre une valeur *erronée*, inférieure à la valeur réelle. On dispose d'un paramètre ϵ, compris entre 0 et 1 que l'utilisateur choisit pour obtenir les garanties suivantes :

— si n éléments ont été insérés dans le tas, au plus $n \times \epsilon$ priorités ont été corrompues ;
— les insertions se font en temps $O(1)$;
— les suppressions se font en temps $O(\log(1/\epsilon))$.

Nous prenons ici $\epsilon = 1/3$.

2.2.1. (1.5 points) Proposez un algorithme de choix de pivot garantissant que le choix n'est jamais trop mauvais. *(Indication: essayez d'abord avec un tas normal qui ne corrompt pas les clefs pour utiliser comme pivot la médiane)*

2.2.2. (1.5 points) Montrez à l'aide du *master theorem* que le coût au pire cas de l'algorithme est $O(n)$.