Structure de données II : Rapport de l'étape préliminaire du projet

Groupe 5 :
HUYLENBROECK Florent
DACHY Corentin

Année Académique 2018-2019
Bachelier en Sciences Informatiques

Faculté des Sciences, Université de Mons

Contents

1 Introduction

Pour ce travail, voici les consignes qui nous on été demandées :

> Pour se familiariser avec les arbres BSP (*Binary space partitions*), il vous est demandé de réaliser l'exercice préliminaire suivant :
>
> Etant donné un arbre BSP représentant une scène dans un plan (ensemble de segments) et deux points x et y dans ce plan, donnez un algorithme recursif en pseudo-code qui indique si le segment d'extrémités x et y appartient à la scène. Veuillez accompagner votre algorithme :
>
> - d'une explication de son fonctionnement; et
> - d'une discussion autour de sa complexité (ne vous limitez pas au pire des cas).
>
> Nous convenons, pour cet exercice, que les segments contenus dans un même nœud de l'arbre BSP sont stockés dans une liste chaînée.
>
> **Remarque** : Cet exercice préliminaire n'est qu'une mise en route du projet. Il ne sera pas nécessaire à la résolution du problème principal.

2 Résolution

2.1 Pseudo-code

2.1.1 Algorithme belongsToScene

Algorithm 1 belongsToScene

Détermine si un segment dont les deux extrémités sont données sous forme de points en entrée appartient exactement à l'arbre BSP donné en entrée.
Sert de phase d'initialisation à l'algorithme *rechercher*.

Entrées : BSP : Partition de recherche binaire
On assume que chaque noeud contient l'équation de la droite qu'il décrit et (facultatif) le segment qui lui est confondu, et chaque feuille contient un segment décrit par une paire de points $S(S.x, S.y)$ et $S'(S'.x, S'.y)$.
A : Point correspondant à une extrémité du segment.
Ce point a pour coordonnées ($A.x$,$A.y$).
B : Point correspondant à l'autre extrémité du segment.
Ce point a pour coordonnées ($B.x$,$B.y$).

Sorties : Boléen, vrai si le segment appartient à la scène, faux sinon.

Effets : /

1: **procedure** BELONGSTOSCENE(BSP, A, B)
2: $d \leftarrow$ COEFFICIENTANGULAIRE(A, B)
3: **retourner** RECHERCHER(BSP, A, B, d)
4: **end procedure**

2.1.2 Algorithme coefficientAngulaire

Algorithm 2 coefficientAngulaire

Calcule le coefficient angulaire d'un segment. Renvoie la valeur sentinelle $+\infty$ si le segment est vertical.

Entrées : A : Point, première extrémité du segment, de coordonnées ($A.x$,$A.y$).
B : Point, deuxième extrémité du segment, de coordonnées ($B.x$,$B.y$).

Sorties : Le coeficient angulaire de la droite passant par A et B.
Une valeur sentinelle $+\infty$ sera retournée si la pente est verticale.

Effets : /

1: **procedure** COEFFICIENTANGULAIRE(A, B)
2: **if** $A.x - B.x == 0$ **then**
3: **retourner** $+\infty$
4: **else**
5: **retourner** $(A.y - B.y)/(A.x - B.x)$
6: **end if**
7: **end procedure**

2.1.3 Algorithme rechercher

Algorithm 3 rechercher

Recherche récursivement un segment dans un arbre BSP.
Pour cela, on localise d'abord tous les segments de l'arbre BSP dont une extrémité est la première extrémité P du segment recherché. On ne garde ensuite (dans une liste S) que les segments qui ont le même coefficient angulaire d que le segment recherché.

Cas de base : Aucun segment de S ne correspond à la recherche, $False$ est retourné.

Récursion : pour chaque segment de S si l'autre extrémité P' de celui-ci correspond à la deuxième extrémité B du segment recherché, on s'arrête et on renvoie $True$. Sinon, on regarde si le segment continue dans une autre feuille ou dans un autre noeud, si oui, on rappelle l'algorithme avec P' comme point de départ et B comme deuxième extrémité.

Entrées : BSP : Partition de recherche binaire.
P : Point, première extrémité du segment recherché dans le BSP
B : Point, deuxième extrémité du segment recherché dans le BSP
d : Entier (ou valeur sentinelle $+\infty$), coefficient angulaire du segment recherché.

Sorties : Boléen, vrai si le segment [PB] appartient à l'arbre BSP.

Effets : /

```
1: procedure RECHERCHER(BSP, P, B, d)
2:     S[] ← nouvelle liste vide
3:     LOCALISER(BSP, P, S[])
4:     REDUIRE(S[], d)
5:     if S[] vide then
6:         retourner False
7:     else
8:         for segment in S[] do
9:             if P ∈ segment then
10:                P' ← extrémité de segment qui n'est pas P
11:                if P' == B then
12:                    retourner True
13:                else if P' sur un bord then
14:                    retourner RECHERCHER(BSP, P', B, d)
15:                end if
16:            end if
17:        end for
18:    end if
19:    retourner False
20: end procedure
```

2.1.4 Algorithme localiser

Algorithm 4 localiser

Recherche récursivement un point P donné dans les segments d'un arbre BSP *root*.

Cas de base : *root* est une feuille. Si le segment dans cette feuille contient P, alors *root* est ajouté à la liste de retour.

Récursion : On calcule *res* le résultat de la résolution de l'équation de droite décrite par *root* par les coordonnées de P pour trouver où P se situe par rapport à celle-ci.
Si P est à droite (*resp.* gauche) de *root i.e.* $res > 0$ (*resp.* $res < 0$), on rappelle la fonction sur l'arbre de droite (*resp.* gauche).
Si P fait partie d'un segment contenu dans *root i.e.* $res = 0$, alors *root* est ajouté à la liste de retour et on rappelle la fonction sur les deux fils de *root*.

Entrées :

- *root* : Racine de la sous-partition de recherche binaire où l'on doit chercher. On assume que *root* possède un attribut d étant l'équation de la droite décrite par *root* et S qui contient le segment (*root* est une feuille) ou les segments (*root* est un noeud) contenus dans *root*, sous la forme de deux points par segments (s'il y en a plusieurs, ils seront contenus dans une liste chainée). *root*+ représente le sous-arbre au-dessus de *root*.d, et *root*− représente le sous-arbre en dessous de *root*.d .
- P : Point que l'on recherche, de coordonnées ($P.x$,$P.y$).
- *return*[] : Liste des segments (paires de points) contenant le point recherché.

Sorties : /

Effets : Les segments contenant P ont été ajoutés à *return*[] .

```
1: procedure LOCALISER(root, P, return[])
2:     if root est une feuille then
3:         if P ∈ root.S then
4:             ajouter root dans return[]
5:         end if
6:     else
7:         res ← résultat de la résolution de root.d par le point P.x et P.y
8:         if res ≥ 0 then
9:             LOCALISER(root+, P, return[])
10:        else if res ≤ 0 then
11:            LOCALISER(root−, P, return[])
12:        else
13:            for segment in root.S do
14:                if P ∈ segment then
15:                    ajouter segment dans return[]
16:                end if
17:            end for
18:            LOCALISER(root+, P, return[])
19:            LOCALISER(root−, P, return[])
20:        end if
21:    end if
22: end procedure
```

2.1.5 Algorithme reduire

Algorithm 5 reduire

Réduit un ensemble de segments pour ne garder que ceux qui ont un coefficient angulaire donné.

Entrées : $S[]$: Liste de segments à réduire.
d : Entier (ou valeur sentinelle $+\infty$), coefficient angulaire du segment recherché.

Sorties : /

Effets : La liste $S[]$ ne contient plus que les segments qui ont un coefficient angulaire d.

```
1: procedure REDUIRE(S[], d)
2:     for all éléments s de S[] do
3:         sd ←COEFFICIENTANGULAIRE(s.x, s.y)
4:         if sd ≠ d then
5:             retirer s de S[]
6:         end if
7:     end for
8: end procedure
```

2.2 Discussion de la complexité

Dans l'étude de la complexité de nos algorithmes, nous allons nommer :

- s le nombre de segments dans la scène.
- h la hauteur de l'arbre BSP.
- l la largeur de l'arbre BSP.

2.2.1 Algorithme coefficientAngulaire

Dans le pire des cas comme dans un cas moyen, l'appel à cet algorithme se fait en $O(1)$.

2.2.2 Algorithme reduire

Dans le pire des cas, $S[]$ contient tous les segments de la scène. Ce cas n'est envisageable dans notre procédure que lorsque tous les segments de la scène ont une extrémité en commun. Il s'agira alors d'une complexité en $O(s)$ (on suppose que l'on retire les segments de la liste par index, donc en $O(1)$).

Dans le cas moyen par contre, $S[]$ ne contiendra que quelques segments. Le corps de la boucle ne contenant que des instructions en $O(1)$, l'algorithme aura pour complexité O(nombre de segments dans $S[]$).

Pour la suite de la discussion, nous considérerons une complexité de $O(s)$ pour cet algorithme. Nous considérerons aussi que, dans le pire des cas, la liste $S[]$ contiendra, après application de la fonction, s segments, contre 1 dans un cas moyen.

2.2.3 Algorithme localiser

Chaque appel récursif de cet algorithme "descend" d'un noeud dans l'arbre. La condition d'arrêt est que l'on atteigne une ou plusieurs feuilles. Les instructions autres que les appels à la fonction étant en $O(1)$ (affectation, ajout d'un élément à une liste) et que l'on considère (lignes 13:17) que $root.S$ contient dans le pire des cas tous les segments de la scène, et dans un cas moyen un seul segment. On aura, dans le pire des cas une complexité en $O(h + s)$, et dans un cas moyen $O(h)$.

Note : il est possible qu'une récursion appelle deux fois la fonction (*cf* lignes 18:19, mais cela est négligeable car $O(2h) \in O(h)$.

2.2.4 Algorithme rechercher

Dans le pire des cas, le segment recherché traverse tout l'arbre BSP. On aura donc un nombre d'appels récursifs égal au nombre l de feuilles de l'arbre. On a donc une complexité de $O(h + s + s$ (lignes 3:4) $+l \cdot s$ (lignes 7:18 si $S[]$ contient tous les segments de la scène)). Donc $O(h + s + l \cdot s)$.

Dans un cas moyen la complexité sera $O(h + s + l)$ par les considérations précédentes.

2.2.5 Algorithme belongsToScene

Nous avons donc, au final, un algorithme ayant comme complexité dans le pire des cas $O(1 + h + s + s \cdot l)$ donc $O(h + s + s \cdot l)$ et dans un cas moyen $O(1 + h + s + l)$ donc $O(h + s + l)$.