

# 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Résolution</b>	<b>3</b>
2.1	Pseudo-code . . . . .	3
2.1.1	Algorithme belongsToScene . . . . .	3
2.1.2	Algorithme coefficientAngulaire . . . . .	3
2.1.3	Algorithme rechercher . . . . .	4
2.1.4	Algorithme localiser . . . . .	5
2.1.5	Algorithme reduire . . . . .	6
2.2	Discussion de la complexité . . . . .	7
2.2.1	Algorithme coefficientAngulaire . . . . .	7
2.2.2	Algorithme reduire . . . . .	7
2.2.3	Algorithme localiser . . . . .	7
2.2.4	Algorithme rechercher . . . . .	7
2.2.5	Algorithme belongsToScene . . . . .	7

# 1 Introduction

Pour ce travail, voici les consignes qui nous ont 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 récursif 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 données sous formes de points en entrée appartient à l'arbre BSP donné en entrée.

**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 :** Booléen, vrai si le segment appartient à la scene, 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.

**Entrées :**  $A$  : Racine de la sous-partition de recherche binaire que l'on doit chercher  
 $B$  : Point que l'on recherche.

**Sorties :** Le coefficient 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.

**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 :**                Booléen, vrai si le segment PB appartient au BSP  
**Effets :**                /

```
1: procedure RECHERCHER(BSP, P, B, d)
2:   S[]  $\leftarrow$  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  $\in$  segment then
10:        P'  $\leftarrow$  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 donné dans les segments d'un arbre BSP

**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 celui-ci, sous la forme de deux points par segments (s'il y en a plusieurs, ils seront contenus dans une liste chaînée).  
*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.  
      *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  $P \in \text{root}.S$  then
4:       ajouter root dans return[]
5:     end if
6:   else
7:     res  $\leftarrow$  résultat de la résolution de root.d par le point P.x et P.y
8:     if  $res \geq 0$  then
9:       LOCALISER(root+, P, return[])
10:    else if  $res \leq 0$  then
11:      LOCALISER(root-, P, return[])
12:    else
13:      for segment in root.S do
14:        if  $P \in \text{segment}$  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 :**  $S[]$  : Ensemble 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 elements  $s$  de  $S[]$  do
3:      $sd \leftarrow \text{COEFICIENTANGULAIRE}(s.x, s.y)$ 
4:     if  $sd \neq d$  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  $O(\text{nombre de segments dans } S[])$ .

Pour la suite de la discussion, nous considérerons une complexité de  $O(s)$  pour cet algorithme.

### 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 fonctions 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 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'appel 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 reste  $O(h + s + s \cdot l)$  par les considérations précédentes.

### 2.2.5 Algorithme belongsToScene

Nous avons donc au final, un algorithme ayant comme complexité  $O(1 + h + s + s \cdot l)$  donc  $O(h + s + s \cdot l)$ .