UMONS
Université de Mons

Faculté des Sciences

Rapport de projet

Exercice préliminaire

S-INFO-820 Projet de structures de données II

Made by Roméo IBRAIMOVSKI & Maxime NABLI

Faculté
des Sciences

Supervised by Gauvain DEVILLEZ

3e Bachelier en Sciences Informatiques
Année 2021-2022

ACADÉMIE
UNIVERSITAIRE
WALLONIE-
BRUXELLES

Résumé

Ce rapport d'implémentation est rendu dans le cadre de l'AA-S-INFO-820 "Projet de structure de données II", supervisé par l'Assistant Gauvain Devillez en année académique 2021-2022. Ce rapport a pour but d'expliquer et de justifier nos différents choix de conception.

Table des matières

Introduction

Pour le projet du cours de "Structure de données II", il nous est demandé de résoudre un exercice préliminaire afin de nous familiariser avec la structure de données utilisée pour le projet, les arbres BSP.

Cet exercice consiste à concevoir un algorithme récursif en pseudo-code prenant en entrée un arbre BSP représentant une scène dans un plan et deux points a et b dans ce plan, et nous renvoie s'il est vrai ou faux que le segment d'extrémités a et b appartient à la scène.

Dans ce rapport, nous présenterons notre algorithme et ses sous-algorithmes avec leurs explications de fonctionnement ainsi qu'une discussion autour de la complexité de l'algorithme principal.

1. Algorithme principal

1.1 Pseudo-code

Algorithme $BSPRecherche(T, a, b)$

Entrée : Un arbre BSP T représentant une scène dans un plan (donné par la référence à sa racine ou référence vide), et deux points a et b

Sortie : Un booléen (vrai si le segment d'extrémités a et b appartient à la scène, faux sinon)

1: **Si** $estVide(T)$ **Alors**
2: **Retourner Faux**
3: **Sinon**
4: **Si** $listeChaineeRecherche(T_{seg}, a, b)$ **Alors**
5: **Retourner Vrai**
6: **Sinon**
7: $bool_a \leftarrow inOpenNegativeHalfSpace(T_{div}, a)$
8: $bool_b \leftarrow inOpenNegativeHalfSpace(T_{div}, b)$
9: **Si** $bool_a = Vrai$ **and** $bool_b = Vrai$ **Alors**
10: **Retourner** $BSPRecherche(T^-, a, b)$
11: **Sinon**
12: **Si** $bool_a = Faux$ **and** $bool_b = Faux$ **Alors**
13: **Retourner** $BSPRecherche(T^+, a, b)$
14: **Sinon**
15: $D_{ab} \leftarrow segToLine(a, b)$
16: $c \leftarrow intersection(T_{div}, D_{ab})$
17: **Si** $bool_a = Vrai$ **and** $bool_b = Faux$ **Alors**
18: **Retourner** $BSPRecherche(T^-, a, c)$ **and** $BSPRecherche(T^+, c, b)$
19: **Sinon**
20: **Retourner** $BSPRecherche(T^-, b, c)$ **and** $BSPRecherche(T^+, c, a)$
21: **Fin Si**
22: **Fin Si**
23: **Fin Si**
24: **Fin Si**
25: **Fin Si**

1.2 Intuition

Lors de la recherche, en chaque noeud visité, on peut laisser tomber l'un des deux sous-arbre quand le segment se trouve entièrement dans un demi-espace ouvert par rapport à la ligne de subdivision sinon on se retrouve à chercher des segments du segment dans les deux sous-arbres. On suit un chemin de la racine jusqu'à

- un noeud contenant le segment ou les segments du segment quand celui-ci est présent dans T
- ou une référence vide, quand le segment est absent de T

1.3 Raisonnement mathématique

Cas de base : arbre vide T. Alors le booléen est faux car le segment d'extrémités a et b n'est pas présent

Cas général : arbre non vide T de racine v où T_{div} est la droite de subdivision, T_{seg} est une liste chainée des segments inclus dans la ligne de subdivision si v possède au moins un sous arbre non-vide sinon une liste chainée d'un segment seul dans sa partition et T^- et T^+ sont ses sous-arbres

- Hypothèse : on connait la valeur des booléens retournés par le ou les appels récursifs
- Si le segment d'extrémités a et b se trouve dans T_{seg}, alors le booléen est vrai car on a trouvé le segment
- Si le segment d'extrémités a et b se trouve entièrement dans un demi-espace négatif ouvert, alors il faut rechercher le segment dans T^-, et donc le booléen vaudra la valeur retourné par cet appel
- Si le segment d'extrémités a et b se trouve entièrement dans un demi-espace positif ouvert, alors il faut rechercher le segment dans T^+, et donc le booléen vaudra la valeur retourné par cet appel
- Si le segment d'extrémités a et b ne se trouve pas entièrement dans un des demi-espace ouvert, alors il faut trouver le point d'intersection entre le segment et la droite de subdivision et rechercher les segments du segment dans T^- et T^+

2. Sous-algorithmes

2.1 listeChaineeRecherche(L, a, b)

Cet algorithme recherche un segment dans une liste chainée de segment.

Algorithme $listeChaineeRecherche(L, a, b)$

Entrée : Une Liste chaînée L de segments de droites dans R^2 (donné par la référence de sa tête de liste ou référence vide), et deux points a et b

Sortie : Un booléen (vrai si le segment d'extrémités a et b se trouve dans la liste, faux sinon)

1: **Si** $estVide(L)$ **Alors**
2: **Retourner Faux**
3: **Sinon**
4: **Tant que** $L \neq vide$ **Faire**
5: **Si** $inSegment(L_{data}, a, b) = Vrai$ **Alors**
6: **Retourner Vrai**
7: **Fin Si**
8: $L \leftarrow L_{next}$
9: **Fin Tant que**
10: **Retourner Faux**
11: **Fin Si**

2.2 inSegment(s, a, b)

Cet algorithme donne l'appartenance d'un segment à un autre segment.

Algorithme $inSegment(s, a, b)$

Entrée : Un segment s et deux points a et b de coordonnées (x_a, y_a) et (x_b, y_b)

Sortie : Un booléen (vrai si le segment d'extrémités a et b est dans le segment s, faux sinon)

1: *Récupération des points du segments et des coordonnées des points*
2: $D_s \leftarrow segToLine(a_s, b_s)$
3: **Si** $a_{D_s} * x_a + b_{D_s} * y_a + c_{D_s} = 0$ **and** $a_{D_s} * x_b + b_{D_s} * y_b + c_{D_s} = 0$ **Alors**
4: $x_{min} \leftarrow min(x_{a_s}, x_{b_s})$
5: $x_{max} \leftarrow max(x_{a_s}, x_{b_s})$
6: $y_{min} \leftarrow min(y_{a_s}, y_{b_s})$
7: $y_{max} \leftarrow max(y_{a_s}, y_{b_s})$
8: **Si** $x_{min} \leq x_a$ **and** $x_a \leq x_{max}$ **and** $x_{min} \leq x_b$ **and** $x_b \leq x_{max}$ **and** $y_{min} \leq y_a$ **and** $y_a \leq y_{max}$ **and** $y_{min} \leq y_b$ **and** $y_b \leq y_{max}$ **Alors**
9: **Retourner Vrai**
10: **Sinon**
11: **Retourner Faux**
12: **Fin Si**
13: **Fin Si**
14: **Retourner Faux**

2.3 inOpenNegativeHalfSpace(D, p)

Cet algorithme donne l'appartenance d'un point à un demi-espace négatif ouvert par rapport à une droite.

Algorithme $inOpenNegativeHalfSpace(D, p)$

Entrée : Une droite D dans R^2 représentée par un triplet (a_D, b_D, c_D) tel que $D \equiv a_D * x + b_D * y + c_D = 0$, un points p de coordonnée (x_p, y_p)

Sortie : Un booléen (vrai si le point p se trouve dans le demi-espace négatif ouvert par rapport à D, faux sinon)

1: *Récupération du triplet de la droite D et des composants du point p*
2: **Si** $a_D * x_p + b_D * y_p + c_D < 0$ **Alors**
3: **Retourner Vrai**
4: **Fin Si**
5: **Retourner Faux**

2.4 segToLine(a, b)

Cet algorithme donne les coefficients de l'équation cartésienne de la droite passant par les deux points.

Algorithme $segToLine(a, b)$

Entrée : Deux points a et b de coordonnées (x_a, y_a) et (x_b, y_b)
Sortie : La droite D dans R^2 représentée par le triplet (a_D, b_D, c_D)
tel que $D \equiv a_D * x + b_D * y + c_D = 0$, passant par les points a et b

1: *Récupération des coordonnées des points*
2: $vecNorm \leftarrow vecNorm(a, b)$
3: **Si** $a_n = 0$ **and** $b_n = 0$ **Alors**
4: **Retourner** $Null$
5: **Sinon**
6: $c \leftarrow -a_n * x_a - b_n * y_a$
7: $D \leftarrow [a_n, b_n, c]$
8: **Retourner** D
9: **Fin Si**

2.5 vecNorm(a, b)

Cet algorithme donne un vecteur normal d'une droite passant par les deux points.

Algorithme $vecNorm(a, b)$

Entrée : Deux points a et b de coordonnées (x_a, y_a) et (x_b, y_b)
Sortie : Un vecteur normal d'une droite passant par les points a et b

1: *Récupération des coordonnées des points*
2: $vecDir \leftarrow vecDir(a, b)$
3: $a_n \leftarrow -b_d$
4: $b_n \leftarrow a_d$
5: $vecNorm \leftarrow [a_n, b_n]$
6: **Retourner** $vecNorm$

2.6 vecDir(a, b)

Cet algorithme donne un vecteur directeur d'une droite passant par les deux points.

Algorithme $vecDir(a, b)$

Entrée : Deux points a et b de coordonnées (x_a, y_a) et (x_b, y_b)
Sortie : Un vecteur directeur d'une droite d'équation inconnue passant par les points a et b

1: *Récupération des coordonnées des points*
2: $a_d \leftarrow x_a - x_b$
3: $b_d \leftarrow y_a - y_b$
4: $vecDir \leftarrow [a_d, b_d]$
5: **Retourner** $vecDir$

2.7 intersection(D1, D2)

Cet algorithme donne le point d'intersection des deux droites sécantes.

Algorithme $intersection(D1, D2)$

Entrée : Deux droites D1 et D2 sécantes dans R^2 chacune représentée par un triplet (a_D, b_D, c_D) tel que $D \equiv a_D * x + b_D * y + c_D = 0$
Sortie : Le point p d'intersection des deux droites

1: *Récupération des triplets des droite.*
2: $n_1 \leftarrow b_{D1} * c_{D2} - b_{D2} * c_{D1}$
3: $n_2 \leftarrow a_{D2} * c_{D1} - a_{D1} * c_{D2}$
4: $d \leftarrow a_{D2} * b_{D1} - a_{D1} * b_{D2}$
5: $x_p \leftarrow n_1/d$
6: $y_p \leftarrow n_2/d$
7: $p \leftarrow [x_p, y_p]$
8: **Retourner** p

Remarque : On ne prends pas en compte le parallélisme car

- Si la droite associée au segment est parallèle jointe à la droite de subdivision, alors nous sommes dans le cas de où le segment se trouve dans T_{seg} à la racine
- Si la droite associée au segment est parallèle disjointe à la droite de subdivision, alors nous sommes dans le cas de où le segment est entièrement compris dans un demi-espace ouvert et n'a donc pas besoin d'être segmenté

3. Complexité

Le pire cas se produit quand on suit le chemin le plus long possible pour un segment d'extrémités a et b absent de T. Il y a un cas un peu plus pire, c'est si l'aglorithme se met à chercher les segments du segment en plus absent.

- Nombre de noeuds visités : n où n est le nombre de noeud de T
- Coût local par noeud : en O(l) car
  - Les affectation, tests et appels de fonction effectuant du calcul en temps constant. Tout est en O(1)
  - La recherche dans une liste chainée est en O(l) où l est le nombre d'élément de la liste (relativement petit en pratique)
  - Retour d'un booléen en O(1)
- Nombre de références vides visitées : n + 1
- Coût local par référence vide : en O(1) car
  - Le test estVide(T) en O(1)
  - Retour de faux en O(1)

Au total algorithme linéaire en n : n*O(l) + (n + 1)*O(1) = O(l*n) + O(n + 1) = O(l*n)

Conclusion

Cet exercice préliminaire nous a permis de nous familiariser avec les arbres BSP, la structure de données principalement utilisée pour le projet. Et donc de nous préparer à la suite du projet.