# Décomposition en polygones monotones

Algo2 - Ensimag 1A

## 1 Introduction

On se propose d'implémenter dans ce projet un algorithme de décomposition d'un polygone quelconque en ensemble de polygones monotones.

Un polygone $P$ est dit monotone par rapport à une droite $D$ si il n'existe aucune droite parallèle à $D$ qui intersecte $P$ en plus de deux points.

Dans ce projet, nous travaillerons avec une droite $D$ *verticale*. La figure 1 illustre le concept avec à droite un polygone monotone et à gauche un polygone non-monotone.

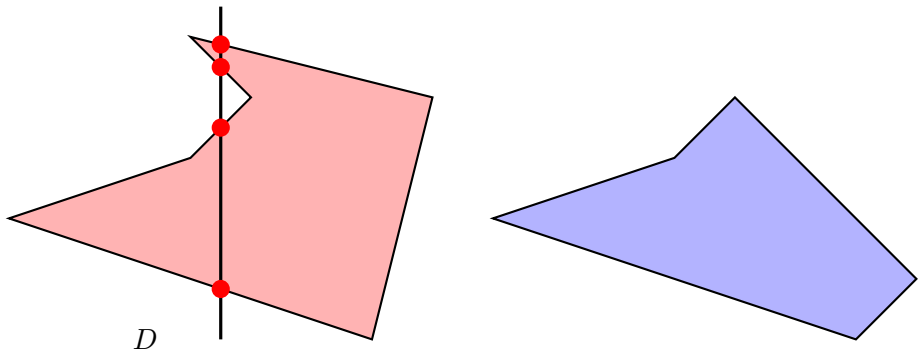


FIGURE 1 – Illustration de la monotonicité

Les polygones monotones présentent la particularité qu'ils sont faciles à découper rapidement en triangles ; or la plupart des systèmes d'affichage 3D (opengl par exemple) fonctionnent à partir de triangles. L'opération consistant à transformer un polygone en un ensemble de triangles est donc particulièrement utilisée en graphisme.

Dans ce projet (qui est déjà assez long) nous nous arrêterons à la première partie de la triangulation en nous contentant de la décomposition en polygones monotones.

On prendra donc en entrée un polygone (défini comme une suite de points) et on cherchera à obtenir une décomposition comme celle de la figure 2 (Notons au passage qu'il existe de nombreuses possibilités de décomposition).

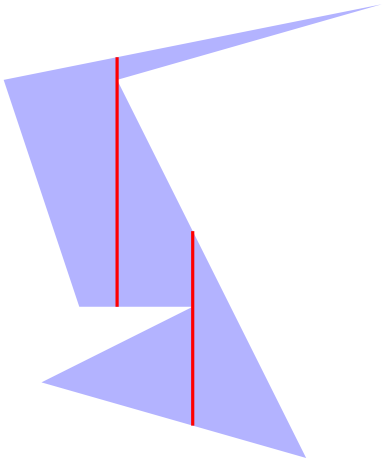

FIGURE 2 – Polygone décomposé en polygones monotones par ajouts de segments

## 2 Arbre binaire de recherche

L'algorithme de décomposition fonctionne en balayant le plan par une droite $D$ de gauche à droite (voir section 3). Il utilise un arbre binaire de recherche qui stocke à tout instant l'ensemble des segments intersectés.

Il vous faut donc implémenter un arbre binaire de recherche classique, tel que vu en cours (avec les fonctions d'insertion, de suppression et de recherche). L'ABR devra être générique et fonctionner avec n'importe quel type de clef (qui dispose d'une fonction de comparaison). On ne demande pas ici d'arbres auto-équilibrés (type AVL, treap, rouge-noir,...) mais il possible d'en utiliser si vous le souhaitez.

On pourra sans doute travailler à plusieurs en parallèle, en laissant un développeur écrire le code de l'ABR et le tester avec de simples entiers (au lieu de segments) pendant que le second membre de l'équipe avance sur l'algorithme principal.

On vous propose d'utiliser la structure suivante :

```
type Noeud;
type Arbre is access Noeud;
type Direction is (Gauche, Droite);
type Tableau_Fils is array(Direction) of Arbre;
type Noeud is record
  C : Type_Clef;
  Fils : Tableau_Fils;
  Pere : Arbre;
  Compte : Positive; --nombre de noeuds dans le sous-arbre
end record;
```

Vous êtes libres d'utiliser ou non une sentinelle pour la racine de l'arbre.

L'arbre nécessite en plus du code vu en cours deux opérations supplémentaires :

1. une procédure Noeuds_Voisins prenant en entrée un nœud cible et renvoyant les (au plus) deux noeuds voisins du noeud cible : c'est-à dire le nœud de l'arbre de clef juste inférieure à la clef de la cible et celui de clef juste supérieure à la clef de la cible ;
2. une procédure Compte_Position prenant en entrée un nœud et renvoyant le nombre de noeuds de clefs inférieures et le nombre de noeuds de clefs supérieures.

Ces deux algorithmes devront impérativement avoir un coût au pire cas de $O(h)$.

On utilisera l'arbre de la figure 3 pour les illustrer.

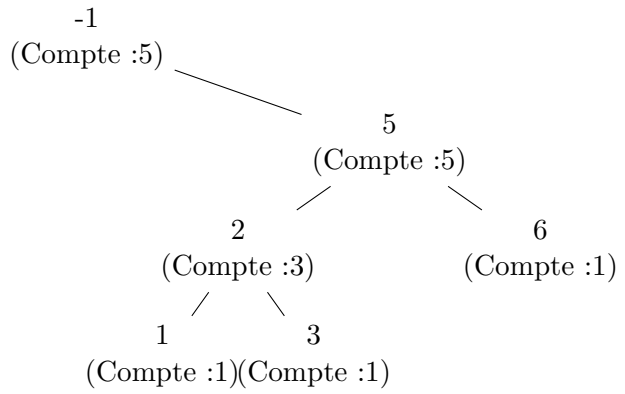


FIGURE 3 – Arbre exemple, avec sentinelle

## 2.1 Nœuds_voisins

On utilise le prototype suivant :

```
procedure Noeuds_Voisins(Cible : Arbre ; Petit_Voisin, Grand_Voisin : out Arbre)
```

Sur le graphe de la figure 3 on obtient ainsi par exemple :

— pour 5 : 3 et 6 ;
— pour 2 : 1 et 3 ;
— pour 1 : **null** et 2 ;
— pour 6 : 5 et **null**.

## 2.2 Compte_position

On utilise le prototype suivant :

```
procedure Compte_Position(Cible : Arbre ; Nb_Petits, Nb_Grands : out Natural)
```

Sur le graphe de la figure 3 on obtient ainsi par exemple :
— pour 5 : 3 et 1 ;
— pour 2 : 1 et 3 ;
— pour 1 : 0 et 4 ;
— pour 6 : 4 et 0.

# 3 Algorithme de décomposition

L'algorithme de décomposition est relativement simple, à condition de prendre les choses sur des bases solides (en testant au fur et à mesure les différentes parties du code). Nous vous conseillons de vérifier au fur et à mesure que tout marche comme prévu et donc d'utiliser des outils de visualisation (on pourra par exemple disposer d'un export *svg* pour dessiner les polygones et les segments ou bien encore d'un export *dot* pour dessiner les arbres).

On va parcourir le plan par une suite de droites $D$ verticales, de gauche à droite. À tout instant on dispose d'un arbre binaire de recherche stockant l'ensemble des segments intersectés par la droite $D$ courante.

La fonction de comparaison utilisée pour la comparaison de deux segments dans l'arbre est la fonction *"au dessus de"* qui appliquée aux segments $(s_1, s_2)$ renvoie 'vrai' si $s_1$ est au dessus de $s_2$ et faux sinon.

On ne compare jamais deux segments non comparables, ce qui signifie en particulier que les segments qui terminent sur un point doivent être retirés de l'arbre avant que les segments qui commencent sur ce même point y soient entrés.

La figure 4 montre un polygone, avec l'évolution de l'ABR pour deux droites $D_1$ et $D_2$.

Dans toute la suite on utilise également une fonction de comparaison lexicographique sur les *points* pour définir si un point $A$ est *avant* ou *après* un point $B$.

On commence par associer à chaque point $P$ un ensemble de segments entrants (provenant d'un point avant $P$) ainsi qu'un autre ensemble de segments sortants (menant à un point après $P$).

On trie ensuite l'ensemble des points.

L'algorithme parcourt les points en mettant à jour l'arbre au fur et à mesure (attention, on met à jour à chaque changement de point, y compris pour des points alignés verticalement).

En fait, seul un petit ensemble de points va nécessiter l'ajout de segments supplémentaires. Tout d'abord, il faut qu'il s'agisse d'un *point de rebroussement*, c'est-à dire que les deux segments associés au point proviennent d'un même côté.

Pour chaque point de rebroussement il est potentiellement nécessaire de rajouter des segments (nous nous contenterons d'un affichage en svg). On

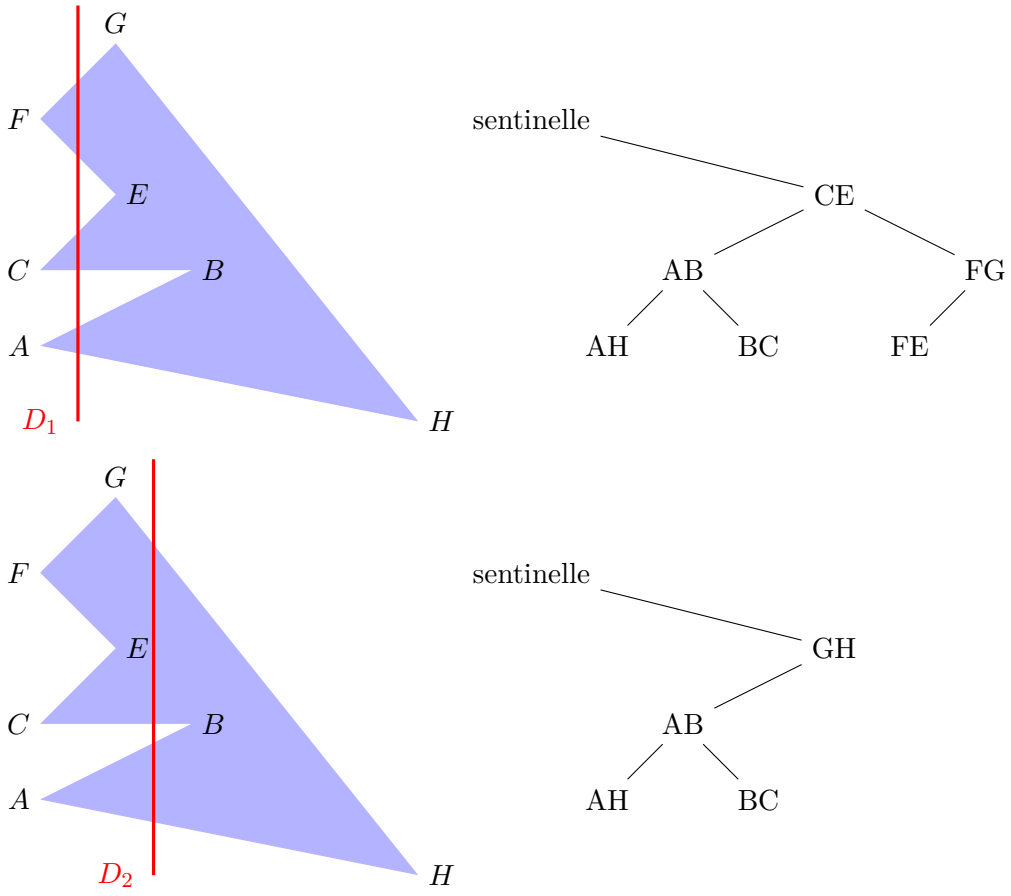


FIGURE 4 – Stockage de segments

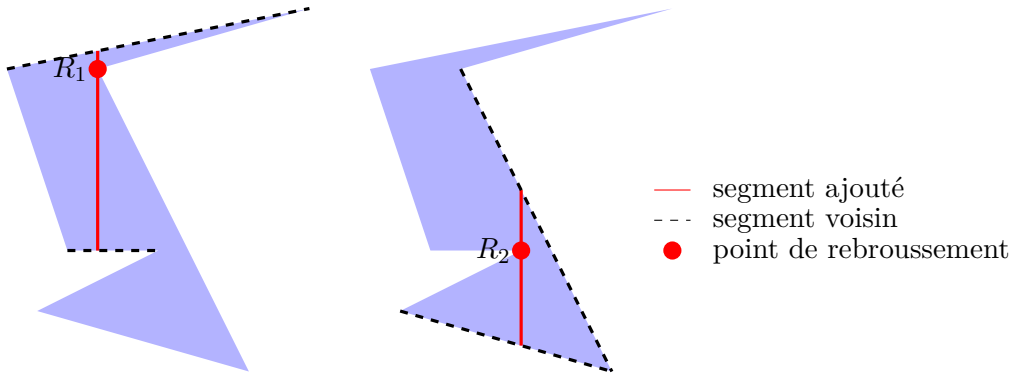


FIGURE 5 – Ajouts de segments sur les points de rebroussement

commence par extraire de l'ABR le nombre de segments au dessus et au dessous du point. Si il y a un nombre impair de segments situés au dessus ou au dessous alors, on extrait les segments voisins. On crée ensuite deux nouveaux segments verticaux, reliant le point courant aux segments voisins. La figure 5 illustre l'opération permettant de découper le polygone de la figure 2.

L'ensemble des opérations correspondant au traitement d'un point (quelconque) est résumé par l'algorithme 1. Il suffit donc de parcourir l'ensemble des points dans l'ordre et d'appeller l'algorithme de traitement sur chacun d'entre eux.

À noter que dans le cas où des points de rebroussement sont alignés verticalement, l'algorithme peut créer plusieurs segments qui se recouvrent. Nous considérerons ici que ce petit défaut n'est pas très important.

# 4 Code

Votre code doit :

— lire des fichiers de polygones au format fourni (voir les fichiers d'exemple) ;
— imprimer le code svg du polygone initial ainsi que des segments ajoutés sur stdout (pour permettre une visualisation dans inkscape par exemple) ;
— être le plus propre et clair possible ;
— être testé au fur et à mesure du développement : n'hésitez donc pas à inclure vos fichiers et procédures de test dans votre archive ;
— implémenter les opérations décrites section 2 pour l'ABR.

Vous avez le droit d'utiliser toutes les structures de données et procédures fournies par la bibliothèque standard Ada.

## 5 Rapport

Le rapport du projet devra avoir une taille de quelques pages (en géneral de 2 à 3). Il vous permettra d'expliquer quels choix vous avez réalisés et pourquoi (quelles structures de données ? quelle organisation du code ?...). Vous pourrez également présenter les tests réalisés ainsi que d'éventuelles mesures de performance.

Le projet est à terminer pour le vendredi 10 avril 2015.

**Entrées** : Point Courant, ABR
$R \leftarrow$ Faux;
`/* on regarde si on est sur un point de rebroussement */`
**si** *Le nombre de segments commençant au point courant est 2* **alors**
    $R \leftarrow$ Vrai;
    $S \leftarrow$ *Nouveau_Segment*(Point Courant, Point Courant);
    $N \leftarrow$ *Insérer*(ABR, S);
    *Noeuds_Voisins*(N, V_petit, V_Grand);
    *Compte_Position*(N, C_petits, C_Grands);
    *Enlever*($N$);
**fin**
Enlever les segments qui terminent sur le point courant de l'ABR;
Ajouter les segments qui commencent sur le point courant dans l'ABR;
**si** *Le nombre de segments terminant au point courant est 2* **alors**
    $R \leftarrow$ Vrai;
    $S \leftarrow$ *Nouveau_Segment*(Point Courant, Point Courant);
    $N \leftarrow$ *Insérer*(ABR, S);
    *Noeuds_Voisins*(N, V_petit, V_Grand);
    *Compte_Position*(N, C_petits, C_Grands);
    *Enlever*($N$);
**fin**
`/* on traite l'éventuel point de rebroussement */`
**si** $R$ **alors**
    **si** *C_petits est impair ou C_Grands est impair* **alors**
        reconnecter le Point Courant verticalement aux segments voisins;
        Afficher les segments ajoutés sur STDOUT;
    **fin**
**fin**

**Algorithme 1 :** Traitement d'un point