Rapport du projet
"*Décomposition en polygones monotones*"

Equipe 82

GOUTTEFARDE Léo
PIERUCCI Dimitri

Mardi 7 Avril 2015

Introduction

Nous avons réalisé l'intégralité du sujet, dont pour commencer la conception d'un package d'arbre binaire de recherche générique, que nous avons décidé d'implémenter à l'aider d'un AVL pour optimiser au maximum l'efficacité, ce qui permet ainsi à toutes les opérations demandées (insertion, suppression, recherche) de s'effectuer en $O(log_2(n))$.

L'exécutable décompose sans problème les polygones au format .in en polygones monotones, et effectue également des mises à l'échelle et translations lors de l'export en code *svg* afin d'obtenir des polygones qui ne débordent pas des dimensions de la figure.

De plus nous avons bien respecté la contrainte de coût au pire cas de $O(h)$ pour l'écriture des fonctions `Noeuds_Voisins` et `Compte_Position`, en intégrant notamment l'actualisation du champ `Compte` des noeuds de l'AVL au sein des opérations de rotation et autres.

Finalement les libérations mémoire sont bien effectuées et il n'y a pas de memory leak.

1 Choix d'implémentation

1.1 Structures de données

L'essentiel de nos structures de données est défini dans le header du package `Common` :

```
type SimplePoint is record
        X : Float := 0.0;
        Y : Float := 0.0;
end record;

type Segment is array (Positive range 1 .. 2) of SimplePoint;
package Segment_Lists is new Ada.Containers.Doubly_Linked_Lists ( Segment );


type Point is record
        Pt : SimplePoint;
        InSegs : Segment_Lists.List;
        OutSegs : Segment_Lists.List;
end record;

package Point_Lists is new Ada.Containers.Doubly_Linked_Lists ( Point, "=" );
package Point_Sorting is new Point_Lists.Generic_Sorting( "<" );
```

Nous avons choisi d'utiliser une première structure `SimplePoint` de représentation des points simples, et une seconde `Point` composée d'un `SimplePoint` ainsi que des deux listes de segments respectivement entrants et sortants.

Les segments sont quant à eux représentés à l'aide d'un tableau de deux `SimplePoint`.

Pour finir nous avons utilisé les listes chaînées fournies par la bibliothèque standard Ada afin de parcourir les points et segments simplement. Aussi, le tri des points de gauche à droite est ainsi réalisé grâce au package de tri générique des listes.

1.2 Organisation du code

Nous avons créé un package `Parseur` pour implémenter le parseur des fichiers .in, un package `SVG` chargé des affichages en *svg*, un package `Decompose` implémentant la majorité des fonctions de décomposition en polygones monotones, et pour finir un package `AVL` chargé de la gestion des arbres binaires de recherche équilibrés.

Le programme principal est implémenté au sein du fichier main.adb, qui est relativement simple.

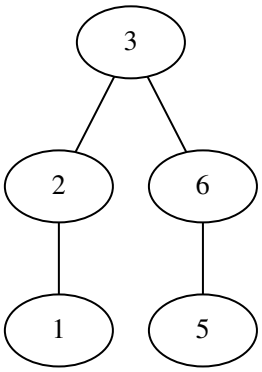
Pour finir, nous avons également réimplémenté la majorité des comparaisons entre flottants afin de les réaliser correctement à $\epsilon = 10^{-4}$ près, en effet des erreurs de comparaisons entre flottants égaux survenaient sans cela.

2 Tests

2.1 Tests unitaires

Nous avons bien évidemment testé le code au fur et à mesure tout au long du projet, tout particulièrement le package `AVL` ainsi que la fonction de comparaison des segments essentielle à son bon fonctionnement.

Pour ces tests nous avons principalement utilisé les fichiers test_avl.adb et test_comp.adb.



Le fichier test_avl.adb implémente l'exemple d'arbre d'entiers du sujet afin de vérifier les valeurs attendues et en exporte un fichier *test_ avl.dot*, ce qui donne la figure ci-dessus.

En effet nous avons réalisé une fonction d'export générique *dot*, `AVL.Generic_Display.Export`, afin de vérifier le bon équilibrage des arbres et visualiser les problèmes de comparaison de segments. Cette fonction d'export permet notamment de visualiser la valeur des clefs (entiers, segments) facilement.

Le fichier test_comp quant à lui, parse certains fichiers de test afin d'en extraire les segments problématiques et en vérifier la bonne comparaison, assurant ainsi la qualité de comparaison des segments.

2.2 Mesures de performance

L'exécutable est relativement rapide et décompose les polygones de test 1 à 8 en 3 à 5 millièmes de seconde.