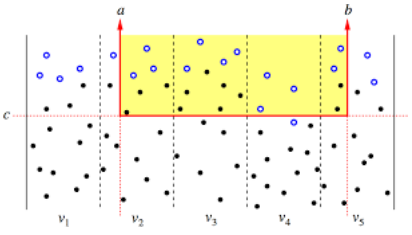
U-MONS

Rapport de projet de Structures de données 2

Priority Search Tree and Windowing



Directeurs :
BRUYÈRE Véronique
DEVILLEZ Gauvain

Groupe :
SALEMI Marco
LECOCQ Alexis

18 avril 2017

Table des matières

1 Introduction

Dans le cadre du cours de structures de données 2, nous avons été amenés à réaliser un projet en Java. Ce projet a pour objectif de créer et manipuler une structure de données non vue au cours. Cette nouvelle structure se base sur un arbre de recherche à priorité (Priority Search Tree en anglais ou PST). De la documentation nous a été fournie afin de nous familiariser avec cette dernière structure qui elle non plus n'a pas été vue au cours.

L'objectif du PST est d'appliquer un "windowing" efficace sur un ensemble de données. Le windowing est une technique qui consiste à extraire tous les points qui sont visibles dans une fenêtre donnée. Un exemple pratique et répandu est l'affichage d'une carte sur un GPS : le GPS n'affichera pas toutes les routes qui se trouvent dans le monde mais seulement celles qui nous entourent.

Un PST partage certaines caractéristiques avec les tas et les Arbres Binaires de Recherche ou ABR. Ces deux dernières structures de données ayant été étudiées en profondeur lors des séances de cours, nous les considérons ici comme connues.

2 Présentation du problème

Il nous a été demandé d'écrire un programme qui permette d'appliquer un windowing à un ensemble de segments de droite verticaux et horizontaux dans $\mathbb{R}^2$. Les différentes fenêtres à implémenter sont les suivantes :

- $[X, X']$ x $[Y, Y']$;
- $[-\infty, X']$ x $[Y, Y']$;
- $[X, +\infty]$ x $[Y, Y']$;
- $[X, X']$ x $[-\infty, Y']$;
- $[X, X']$ x $[Y, +\infty]$.

La structure de données à imaginer devrait donc permettre un windowing efficace pour l'ensemble de ses fenêtres.

3 Priority Search Tree

3.1 Objectif de la structure

Un PST est une structure de données de type arbre binaire (chaque nœud comporte au plus deux fils). Cette structure de données organise des points de l'espace définis par deux coordonnées X et Y. L'organisation des données permet d'effectuer efficacement la recherche des points présents dans une fenêtre de l'espace (sans avoir à parcourir l'ensemble des points).

3.2 Définition

Un PST est une structure de données mixte. Chaque nœud est constitué d'un point et d'un nombre appelé la médiane. Si l'on considère uniquement les coordonnées X, un PST est organisé tel un tas avec le minimum à la racine. Si l'on considère uniquement les coordonnées Y, le PST est organisé tel un ABR avec une petite particularité. Au lieu de trier ses fils par rapport à la donnée du nœud courant, un PST trie ses fils par rapport à la médiane du nœud courant.

Ainsi, tout nœud n d'un PST respecte les contraintes suivantes :
- son fils gauche (s'il existe, ainsi que ses descendants s'ils existent) aura sa coordonnée X plus grande que celle du nœud n et sa coordonnée Y plus petite que la médiane du nœud n ;
- son fils droit (s'il existe, ainsi que ses descendants s'ils existent) aura sa coordonnée X plus grande que celle du nœud n et sa coordonnée Y plus grande que la médiane du nœud n.

3.3 Construction

Bien que la documentation suggérait une construction bottom-up similaire à celle d'un tas, nous avons opté pour une construction top-down qui nous semblait plus facile à implémenter.

3.3.1 Tri des données

La construction d'un PST est plus simple si l'on construit ce dernier à partir d'une liste de points triée selon la coordonnée Y. La première étape est donc de trier les points.

3.3.2 Création de l'arbre

Pour créer un arbre, créons le nœud racine à partir de la liste des points triée.

3.3.3 Création d'un nœud

La coordonnée X d'un nœud devant être plus petite ou égale à celle de ses fils, commençons par rechercher le point avec la plus petite coordonnée X. Attribuons ce point au nœud courant. Séparons le reste des points en deux. Attribuons, comme médiane au nœud courant, la moyenne de la coordonnée Y de 2 points. Le premier est le dernier point de la partie 1 et le second est le premier point de la partie 2. Les points étant triés selon la coordonnée Y, la première partie possède une coordonnée Y plus petite ou égale à la médiane du nœud alors que la deuxième partie possède une coordonnée Y plus grande ou égale à la médiane du nœud. Dans le cas où il ne reste aucun point après le retrait du minimum en X, la valeur de la moyenne n'a pas d'importance. Dans le cas où il ne reste qu'un point, la coordonnée Y du point restant peut être attribuée à la médiane du nœud.

3.4 Windowing

L'application du windowing commence à la racine de l'arbre et se déroule comme suit :

- si la coordonnée X du nœud courant et strictement supérieure à la borne supérieure de la fenêtre X', alors laisser tomber ce nœud ainsi que tous ses descendants (par définition du PST, leur coordonnée X sera aussi supérieure à X') ;
- si le point du nœud courant se trouve dans la fenêtre, le rapporter ;
- si la médiane du nœud courant est inférieure ou égale à la borne supérieure de la fenêtre, parcourir le fils gauche ;
- si la médiane du nœud courant est supérieure ou égale à la borne inférieure de la fenêtre, parcourir le fils droit ;

4 PST Adapté

Étant donné que notre structure utilise des segments au lieu des points, nous ne pouvons utiliser le PST tel que décrit ci-dessus. Nous avons donc adapté cette structure afin de permettre l'utilisation de segments.

Un segment :
- est défini par 2 points, chacun ayant une coordonnée X et une coordonnée Y ;
- possède donc 2 coordonnées X ainsi que 2 coordonnées Y ;
- est soit horizontal soit vertical, c'est-à-dire soit les coordonnées X soit les coordonnées Y sont égales.

Nous avons considéré deux façons d'adapter le problème :

1. stocker les deux points définissant chaque segment séparément dans la structure initiale, tout en gardant un lien vers le segment initial ;
2. stocker chaque segment dans une nouvelle structure et adapter les différents algorithmes.

Après mûre réflexion, nous avons opté pour la seconde option, plus flexible et qui nous permet donc davantage d'optimisations.

Afin de garder les explications concises et faciles à comprendre, nous allons d'abord considérer uniquement les fenêtres de type $[X, X']$ x $[Y, Y']$ et $[-\infty, X']$ x $[Y, Y']$ et nous expliquerons ensuite comment gérer les autres cas.

4.1 Objectif de la structure

L'objectif de cette structure est d'organiser l'ensemble des segments afin de faciliter la recherche de segments visibles dans les fenêtres de type $[X, X']$ x $[Y, Y']$ et $[-\infty, X']$ x $[Y, Y']$.

4.2 Définition

Notre PST adapté respecte exactement la définition d'un PST excepté 2 adaptation :
- un segment, plutôt qu'un point, sera stocké par nœud ;
- les coordonnées utilisées pour effectuer la comparaison sont le minimum des coordonnées X et le minimum des coordonnées Y du segment, plutôt que les coordonnées X et Y du point.

Le choix des coordonnées de référence est judicieux car il permet, lors de la recherche d'une fenêtre, d'élaguer tout nœud (ainsi que ses descendants par définition du tas) dont la coordonnée minimum en X est strictement supérieure à X' et il permet de ne pas visiter le fils droit (ainsi que ses descendants par définition de l'ABR) de tout nœud dont la médiane est strictement supérieure à Y'.

4.3 Construction

Notre PST adapté se construit à la manière d'un PST excepté les références pour la comparaison des données, qui sont les coordonnées minimales en X ainsi qu'en Y. C'est-à-dire que nous trions les segments selon leur plus petite coordonnée en Y. Lorsque nous recherchons le minimum en X, nous recherchons le segment avec la plus petite coordonnée X.

4.4 Windowing

Étant donné que nous devons rapporter tous les segments visibles dans une certaine fenêtre, nous devons non seulement extraire les segments ayant au moins un point dans la fenêtre, mais aussi les segments commençant avant et finissant après (ceux qui passent à travers la fenêtre). C'est le cas notamment du segment possédant les coordonnées (X', Y-1) (X', Y'+1). La recherche sera donc un peu moins efficace que pour les points et nous allons devoir utiliser une approche différente.

Nouvelle approche : nous concentrer sur le point de départ du segment, c'est-à-dire le point ayant la plus petite coordonnée X pour un segment horizontal ou le point ayant la plus petite coordonnée Y pour un segment vertical. Nous avons ici 3 possibilités pour que le segment soit visible :
- soit le point de départ est dans la fenêtre ;

— soit le point de départ est à gauche de la fenêtre et le segment traverse la limite gauche de la fenêtre ;
— soit le point de départ est en dessous de la fenêtre et le segment traverse la limite basse de la fenêtre.

Il est évident que si le point de départ se situe au-dessus ou à droite de la fenêtre, ce segment ne sera pas visible dans la fenêtre.

Nous allons donc parcourir l'arbre à partir de la racine et le traiter de la sorte :
— si le minimum en X du segment est strictement supérieur à X', laisser tomber ce nœud ainsi que tous ses descendants (le point de départ de leurs segments se situe à droite de la fenêtre) ;
— si ce segment est visible dans la fenêtre (voir conditions ci-dessus), le rapporter ;
— si la médiane du nœud est strictement supérieure à Y', laisser tomber le fils droit ainsi que tous ses descendants (le point de départ de leurs segments se situe au dessus de la fenêtre) et parcourir uniquement le fils gauche, sinon parcourir les deux fils.

L'utilisation de cette structure pour réaliser un windowing de type $[-\infty, X']$ x $[Y, Y']$ est très efficace car aucun nœud inutile n'est parcouru.

4.5 Les autres fenêtres

Nous avons implémenté un algorithme efficace pour les fenêtres de type $[X, X']$ x $[Y, Y']$ et $[-\infty, X']$ x $[Y, Y']$. Voyons comment l'adapter aux autres fenêtres, à savoir $[X, +\infty]$ x $[Y, Y']$, $[X, X']$ x $[-\infty, Y']$ et $[X, X']$ x $[Y, +\infty]$.

Nous remarquons que 4 des 5 fenêtres partagent une particularité : elles sont semi-bornées. Nous remarquons également que la place de la borne infinie dépend du système d'axe utilisé. Deux choix sont alors possibles :

1. implémenter 3 autres arbres supplémentaires qui sont efficaces pour chacune des autres bornes infinies ;
2. effectuer une transformation du système d'axe avant la création de l'arbre afin de réutiliser le PST existant.

Maintenir 4 implémentations différentes du PST ne nous a pas semblé être une bonne idée et, la transformation du système d'axe pouvant se faire en temps constant, nous avons opté pour le second choix.

Pour effectuer un windowing avec une fenêtre de type $[X, +\infty]$ x $[Y, Y']$, $[X, X']$ x $[-\infty, Y']$ ou $[X, X']$ x $[Y, +\infty]$, il faudra construire le PST avec les segments transformés dans le nouveau système d'axe, appliquer le windowing sur la fenêtre transformée et appliquer la transformation inverse sur la réponse du windowing.

4.5.1 L'échange

Pour passer d'une fenêtre de type $[-\infty, X']$ x $[Y, Y']$ à une fenêtre de type $[X, X']$ x $[-\infty, Y']$, nous remarquons qu'il suffit d'échanger les coordonnées X et Y :

1. $X \to Y$
2. $Y \to X$

Nous appellerons cette transformation du système d'axe l'**échange**.

Nous remarquons qu'effectuer une transformation inverse revient à appliquer la transformation une seconde fois.

4.5.2 L'opposition

Pour passer d'une fenêtre de type $[-\infty, X']$ x $[Y, Y']$ à une fenêtre de type $[X, +\infty]$ x $[Y, Y']$, nous remarquons qu'il suffit d'opposer les coordonnées :

1. $X \to -X$
2. $Y \to -Y$

Nous appellerons cette transformation du système d'axe l'**opposition**.

Nous remarquons que, comme pour l'échange, effectuer une transformation inverse revient à appliquer la transformation une seconde fois.

4.5.3 La combinaison des deux

Pour passer d'une fenêtre de type $[-\infty, X']$ x $[Y, Y']$ à une fenêtre de type $[X, X']$ x $[Y, +\infty]$, nous remarquons qu'il suffit d'appliquer les deux transformations vues précédemment :

1. $X \rightarrow -Y$
2. $Y \rightarrow -X$

Nous appellerons cette transformation du système d'axe la **combinaison**.

Encore une fois, effectuer une transformation inverse revient à appliquer la transformation une seconde fois.

4.5.4 La fenêtre fermée

Lors de la recherche dans une fenêtre fermée avec le PST original, nous ne pouvons jamais éliminer le fils gauche de nos recherches. En conséquence, les nœuds que nous parcourons, ayant un minimum en X strictement inférieur au X de la fenêtre et ayant un minimum en Y strictement inférieur au Y de la fenêtre, sont parcourus inutilement. Ces segment sont ceux qui commencent en bas à gauche de la fenêtre et il est évident qu'ils n'y sont pas visibles. Nous avons donc adapté notre windowing afin qu'il prenne deux paramètres :

— center : définit si les segments commençant au centre de la fenêtre doivent être reportés ;
— down : définit si les segments commençant en dessous de la fenêtre doivent être reportés.

Si center est faux, alors nous pouvons laisser tomber les nœuds qui ont un minimum en X plus grand que le X de la fenêtre. Si down est faux, alors nous pouvons laisser tomber les fils gauches des nœuds dont la médiane est strictement inférieure au Y de la fenêtre.

Afin d'effectuer la recherche de la fenêtre fermée efficacement, nous appliquerons le windowing deux fois. Une première fois avec down à faux afin de rapporter les segments ayant le point de départ à gauche et dans la fenêtre. Une seconde fois sur la fenêtre échangée avec la même technique que pour la fenêtre $[X, X']$ x $[-\infty, Y']$ avec center et down à faux afin de rapporter tous les segments ayant leur point de départ en dessous.

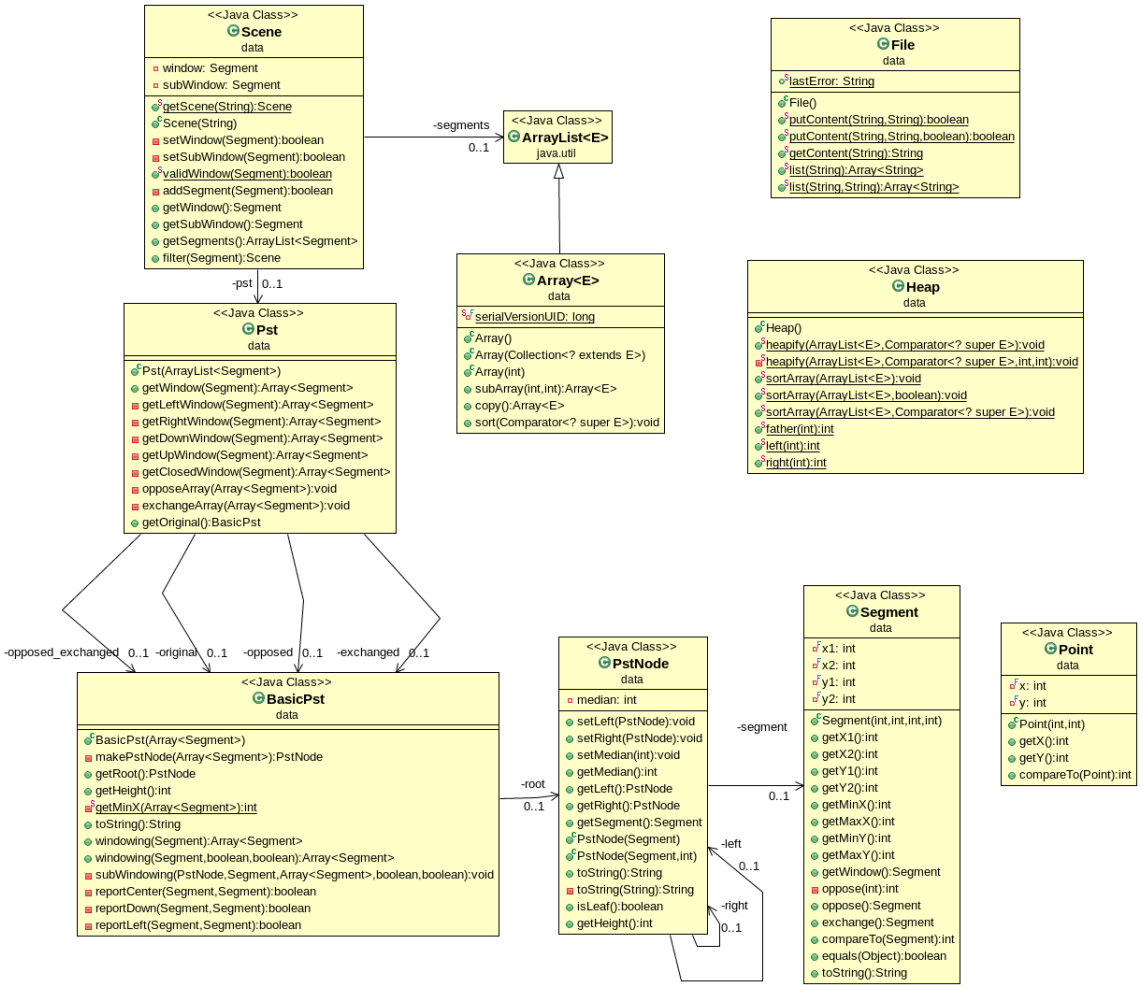
5 Diagrammes de classes

Cette section comporte 2 diagrammes de classes :

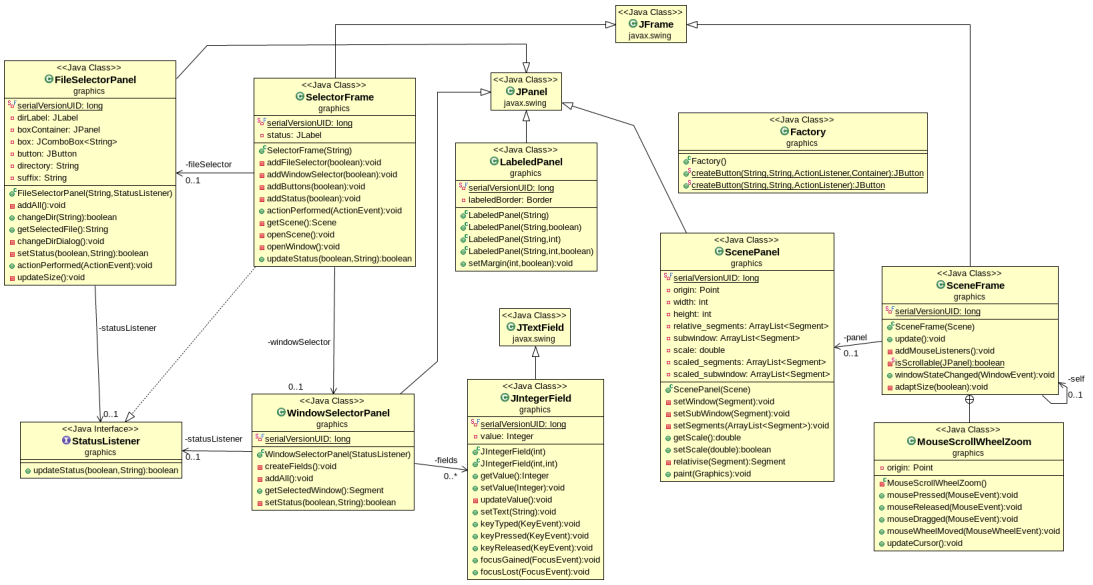
1. le premier concerne l'organisation et le traitement des données ;
2. le second concerne l'interface graphique.

Ces diagrammes sont suivis de quelques explications sur les classes implémentées. Si vous désirez de plus amples d'informations, n'hésitez pas à consulter la Javadoc.

5.1 Diagramme de classe des données



5.2 Diagramme de classe de l'interface graphique



5.3 Pst et BasicPst

BasicPst correspond au PST gérant efficacement une fenêtre de type $[-\infty, X']$ x $[Y, Y']$. Afin de gérer toutes les transformations, nous avons créé une autre classe Pst qui comporte 4 BasicPst avec :

1. les segments originaux pour les fenêtres $[-\infty, X']$ x $[Y, Y']$;
2. les segments échangés pour les fenêtres $[X, X']$ x $[-\infty, Y']$;
3. les segments opposés pour les fenêtres $[X, +\infty]$ x $[Y, Y']$;
4. les segments échangés et opposés pour les fenêtres $[X, X']$ x $[Y, +\infty]$;

Cette classe s'occupe de choisir l'arbre sur lequel effectuer le windowing afin d'obtenir la meilleure efficacité. Plusieurs arbres sont utilisés pour les fenêtres $[X, X']$ x $[Y, Y']$.

6 Algorithmes et explications

Dans cette section, nous exposons nos différents algorithmes importants en pseudo-code ainsi que leur complexité en temps. Une courte explication en français y est aussi ajoutée afin de favoriser la compréhension de ceux-ci et leurs utilisations.

6.1 Hauteur de l'arbre

Afin de compléter les différentes explications, nous allons en premier lieu vous expliquer et prouver que la hauteur de l'arbre est en $\mathcal{O}(log_2(n))$ où n est le nombre de segments présents dans le plan. En effet, la structure du Pst adapté est un arbre complètement équilibré, nous pouvons donc comparer celui-ci à un AVL dont la balance de chaque nœud en valeur absolue ne dépasse pas 1. Nous avons effectué un test dans notre classe PstTests dans notre projet visant à vérifier ceci. L'intuition de cet équilibre s'explique par le fait que lors de la construction de l'arbre, pour chaque nœud de l'arbre, nous répartissons le nombre de segments restants en deux parties égales à un segment près, et chaque partie constitue toutes les données du sous-arbre gauche ou droit de ce nœud.

6.2 Construction de l'arbre

6.2.1 Algorithmes

Algorithme 1 Construction de l'arbre

Entrée: une liste de Segment A, un PstNode $Root$ qui est la racine de l'arbre
Sortie: /

1: $A \leftarrow heapSort(A)$
2: $root \leftarrow makePstNode(A)$

Algorithme 2 makePstNode

Entrée: une liste de Segment triée en y A
Sortie: un PstNode qui représente la racine de l'arbre

1: **si** A est vide **alors**
2: **return** $null$
3: **fin si**
4: $node \leftarrow A[getMinX()]$
5: **si** A est non vide **alors**
6: $i \leftarrow longueur(A)/2$
7: $node.median \leftarrow A[i].y1$
8: $node.left \leftarrow construct(A.sousListe(0, i))$
9: $node.right \leftarrow construct(A.sousListe(i, longueur(A)))$
10: **fin si**
11: **return** node

Algorithme 3 getMinX

Entrée: une liste de Segment L
Sortie: l'indice entier de l'élément minimum en X

```
1:  si L est vide alors
2:      return -1
3:  fin si
4:  min ← 0
5:  for i ← 1 à longueur[A] do
6:      si A[i].minX() < A[min].minX() alors
7:          min ← i
8:      fin si
9:  end for
10: return min
```

6.2.2 Explication

Nous construisons ici l'arbre de haut en bas, ayant initialement la racine qui est le minimum en x des segments, nous effectuons ensuite une séparation des données en deux vis-à-vis de la médiane, qui est la valeur en y1 (minimum en y du segment) se trouvant au milieu de la liste, et ceci récursivement.

Nous séparons ainsi à chaque étape la liste des segments en deux parties égales en taille à une valeur près, que nous répartissons entre les deux fils (première moitié au fils gauche et deuxième moitié au fils droit). Bien sûr pour que ceci soit possible, nous avons trié les segments par ordre croissant suivant leur composante en y1 avant d'utiliser la liste pour la création de l'arbre.

6.2.3 Complexité dans le pire des cas

Nous commençons d'abord par calculer la complexité dans le pire des cas en temps de l'algorithme 3. Les 4 premières lignes sont en $\mathcal{O}(1)$, ainsi que les lignes 7 et 10, car le retour est constant, la condition du if est constante et une affectation de valeur est constante en temps. La condition du if de la ligne 6 est elle aussi constante, car nous faisons une comparaison de valeurs (temps constant) et qu'avoir accès à la valeur du minimum x d'un segment se fait en temps constant aussi (minX()).

Nous aurons donc un for en ligne 5 qui sera en $\mathcal{O}(n)$ où n est la taille de la liste. L'algorithme possède donc une complexité de $\mathcal{O}(n)$ au total.
Pour ce qui concerne l'algorithme 2, les lignes s'exécutent toutes en temps constant pour les mêmes raisons que celles énoncées pour le 3^e algorithme mise à part pour les lignes 4,7 et 8. En effet, la ligne 4 s'exécute en $\mathcal{O}(n)$ où n est la taille de la liste donnée en paramètre, par la preuve ci-dessus. Nous effectuons ensuite deux appels récursifs aux lignes 7 et 8, ce qui nous donnera une complexité finale de l'algorithme en temps de $\mathcal{O}(n.log(n))$. Voici le raisonnement pour trouver cette complexité finale :

Notre algorithme s'effectue à chaque étape de la récursivité en $\mathcal{O}(n)$ dût à la recherche du minimum dans la liste. Nous effectuons ensuite un appel récursif sur une moitié de liste, et un autre appel sur l'autre moitié.
Ce qui nous donne donc une complexité de $\mathcal{O}(n)$ + $\mathcal{O}(n/2)$ + $\mathcal{O}(n/2)$ + $\mathcal{O}$(de la prochaine étape de la récursivité) + ...
Ce qui nous donne donc une complexité de $\mathcal{O}(n)$ + $\mathcal{O}(n)$ + $\mathcal{O}$(de la prochaine étape) + ...
On voit donc qu'à chaque niveau de l'arbre, nous effectuons des calculs en $\mathcal{O}(n)$, nous les effectuons log(n) fois qui est la hauteur de l'arbre. Nous aurons donc une complexité finale de l'algorithme 2 en $\mathcal{O}(n.log(n))$ où n est la taille de la liste initialement donnée en paramètre à la fonction.

Finalement, pour ce qui concerne l'algorithme 1, celui-ci possède une première ligne en $\mathcal{O}(n)$ (complexité en temps du heapSort, notion étudiée au cours), et une seconde en $\mathcal{O}(n.log(n))$. Nous aurons donc une complexité finale de $\mathcal{O}(n.log(n))$

6.3 Windowing

6.3.1 Algorithmes

Algorithme 4 Windowing

Entrée: un Segment $window$ représentant la fenêtre à appliquer
Sortie: Une liste contenant tous les Segments dans la fenêtre
1: **return** window2($window$,$true$,$true$)

Algorithme 5 Windowing2

Entrée: un Segment $window$(objet) représentant la fenêtre à appliquer, un booléen $down$ et un booléen $center$ représentant les type de report à appliquer
Sortie: Une liste contenant tous les Segments dans la fenêtre
1: $window \leftarrow window.getWindow()$
2: $reported \leftarrow$ nouvelle liste vide
3: subwindow($root$,$window$,$reported$,$true$,$true$)
4: **return** $reported$

Algorithme 6 Subwindowing

Entrée: un Node $node$, un Segment $window$, une liste de Segment $reported$, un booléen $down$ et un booléen $center$

Sortie: / Mais effet de bord : $reported$ contient tous les Segments dans la fenêtre

si $node$ est vide **alors**
    fin de l'algorithme
fin si
$S \leftarrow node.getSegment()$
si $center$ est faux et $S.minX() > window.x1$ **alors**
    fin de l'algorithme
fin si
si $S.minX() > window.x2$ **alors**
    fin de l'algorithme
fin si
si $center$ est vrai et $reportCenter(S, window)$ est vrai **alors**
    Ajout de S dans $reported$
fin si
si $down$ est vrai et $reportDown(S, window)$ est vrai **alors**
    Ajout de S dans $reported$
fin si
si $reportLeft(S, window)$ est vrai **alors**
    Ajout de S dans $reported$
fin si
si $down$ est vrai ou $node.median >= window.y1$ **alors**
    subWindowing($node.left, window, reported, down, center$)
fin si
si $node.median <= window.y2$ **alors**
    subWindowing($node.right, window, reported, down, center$)
fin si

6.3.2 Explication

Le windowing appliqué dans les algorithmes ci-dessus sont basés sur un windowing d'une fenêtre de type :

- Soit : $[-\infty; x2]X[y1; y2]$
- Soit : $[x1; x2]X[y1; y2]$

Selon cette hypothèse, nous avons considéré tous les différents cas de segments devant être considérés dans la fenêtre (au moins un point dans la fenêtre ou passant au travers) et ceux qui ne devaient pas s'y trouver. Afin de favoriser la compréhension de notre algorithme, nous expliquons les deux algorithmes l'un après l'autre.

En ce qui concerne le windowing (algorithme 4), il ne nécessite pas d'explication supplémentaire, le pseudo-code est très clair.

Pour l'algorithme 5 (windowing2), l'algorithme effectue dans cet ordre ces différentes actions : rendre la structure représentant la fenêtre ordonnée(x1<x2 et y1<y2) afin de respecter la notion mathématique d'une fenêtre et simplifier le code, et ensuite calculer à l'aide de l'algorithme 6 les segments dans la fenêtre pour pouvoir les retourner dans une liste.

L'algorithme 6 est la partie plus délicate de l'algorithme. En effet, c'est ici que nous avons mis en œuvre l'exploration de l'arbre pour prendre tous les segments qui nous intéressent. Nous avons donc procédé par toute une série de conditions.

Tout d'abord nous vérifions que le nœud n'est pas vide(cas se produisant après l'exploration d'une feuille).

Pour les deux conditions suivantes, nous cherchons les conditions d'arrêt de l'algorithme qui pourraient se produire avant d'arriver à la fin de l'arbre.

La première condition vérifie que nous ne devons plus prendre de segments se trouvant à l'intérieur de la fenêtre et que le nœud courant dans lequel nous nous trouvons se trouve soit dans la fenêtre, soit à

droite de la fenêtre pour ses coordonnées en x. Par définition de la structure, nous savons donc que tous les nœuds fils se trouvent soit dans la fenêtre, soit à droite de la fenêtre en x (car le nœud courant est le minimum en x). Nous pouvons donc arrêter l'algorithme et ignorer ses descendants.

La deuxième condition d'arrêt est le cas où le segment du nœud courant se trouve complètement à droite de la fenêtre en x, nous pouvons donc l'ignorer et ignorer les sous-arbres qui suivent car ils seront tous à droite de la fenêtre (donc pas dedans) par la définition de la structure comme ci-dessus. Dis autrement, les coordonnées en x du segment se trouvant dans le nœud courant nous indiquent qu'elles sont strictement plus grandes que la fenêtre, or ce segment possède la coordonnée en x minimum du sous-arbre dans lequel elle se trouve, il faut donc ignorer ses descendants.

En ce qui concerne les 3 conditions suivantes dans l'algorithme, celles-ci sont utilisées pour détecter quels segments se trouvent réellement dans la fenêtre, cette détection est faite durant toute l'exploration de l'algorithme dans l'arbre. Les conditions booléenne $down$ et $center$ sont utilisées afin de savoir quel type de report nous devons effectuer. En effet nous devons pouvoir détecter les différents types de segments énoncés lors de l'explication du windowing dans le point précédent "Pst adapté".

Nous appelons donc trois méthodes qui testent ces différentes conditions. Nous n'avons pas fourni les pseudo-codes correspondant à ces méthodes qui effectuent simplement une condition et un retour booléen décrivant si le segment est à prendre en compte ou pas. Nous avons favorisé une explication écrite plutôt qu'algorithmique ce qui est plus compréhensible pour la compréhension du lecteur et moins fastidieux à lire. Bien sur pour plus de détails, le code source est disponible vous donnant un exemple pratique d'appliquer les différents reports.

Les différentes conditions de reports sont donc celles -ci :

reportCenter() Pour détecter les segments se trouvant dans la fenêtre totalement ou avec le premier point en y dans la fenêtre(y1 est la plus petite coordonnée en y), nous testons si le premier point du segment en y se trouve dans la fenêtre.

reportDown() Pour détecter les segments verticaux passants à travers la borne inférieure de la fenêtre, donc avec le premier point en y commençant avant la fenêtre et le deuxième point se terminant dans ou après la fenêtre (cas d'un segment qui traverse). Nous testons si le premier point en y se trouve avant la fenêtre, et le second après, ainsi que les coordonnées en x devant se trouver toutes les deux dans la fenêtre (puisqu'elles sont égales car segment vertical, un seul test sur un seul x est nécessaire)

reportLeft() Pour détecter les segments horizontaux qui traversent complètement la fenêtre ou qui commencent avant pour finir dans celle-ci. Nous testons si les coordonnées y se trouvent dans la fenêtre (puisqu'elles sont égales car segment horizontal, un seul test sur un seul y est nécessaire), et ensuite vérifions que le segment commence en ses coordonnées en x avant la fenêtre pour se terminer après ou dedans.

Nous avons ainsi géré tous les différents segments pouvant se produire dans nos deux types de fenêtre. En effet, ces différents cas s'appliquent tous pour une fenêtre complètement bornée et le reportleft() n'aura pas lieu dans une fenêtre non bornée en x1 ($-\infty$)

Finalement nous regardons par rapport à la médiane du nœud courant où nous devons nous déplacer dans l'arbre. Nous considérons donc les deux limites en y de la fenêtre pour savoir quand se déplacer à gauche dans l'arbre (éléments <= à la médiane en y1) ou à droite (éléments >= à la médiane en y1). Nous faisons donc des appels récursifs pour avancer toujours dans la bonne direction dans l'arbre et éviter des reports inutiles.

6.3.3 Complexité dans le pire des cas

Les différentes lignes dans les algorithmes 4 et 5 sont tous en $\mathcal{O}(1)$ car nous ne faisons que des appels à des fonctions , des retour de valeurs et un appel à getWindow() dans l'algorithme 5-ligne 1 , qui s'effectue aussi en $\mathcal{O}(1)$ aussi car il ne fait que donner un nouveau segment en prenant les données du premier. Le coût du windowing sera donc dû à l'exécution à la fonction subwindowing() qui va effectuer le réel travail d'exploration dans le Pst.

Nous calculons donc la complexité en temps de l'algorithme 6. Pour les lignes 1 à 10, elles s'exécutent toutes en $\mathcal{O}(1)$ car nous faisons de simples tests booléen sur des valeurs, des accès à des variables d'objets,

une affectation à une variable, et que le corps de toutes les conditions sont en $\mathcal{O}(1)$ aussi. Pour les lignes 11 à 19, les différents tests booléens s'effectuent en $\mathcal{O}(1)$, ainsi que l'ajout à une liste et l'appel aux différents report. En effet ceux-ci testent juste des conditions sur les coordonnées d'après les cas et retournent un booléen, elles s'exécutent donc en un temps constant également.

Il nous reste donc les lignes 20 à 24. Pour ce qui concerne ces lignes, l'étude de leur complexité est plus particulière. En effet elles effectuent un appel récursif selon certaines conditions qui un coût en temps de $\mathcal{O}(1)$. Considérant que l'algorithme est jusque là en $\mathcal{O}(1)$ par ce que nous avons expliqué ci-dessus, l'algorithme se réalisera en $\mathcal{O}(1)$ x le nombre d'appels récursifs.

Si nous analysons les appels nous constatons que les appels récursifs peuvent se réaliser soit sur le sous-arbre gauche du nœud courant (initialement la racine) si la fenêtre se trouve totalement à gauche de la médiane, soit le sous-arbre droit si la fenêtre se trouve totalement à droite de la médiane, soit les deux sous-arbres si la médiane se trouve dans la fenêtre. Nous considérons bien sur que les segments sont répartis plus ou moins uniformément dans l'espace (tous les segments ne se trouvent pas dans un même petit espace). Nous considérons aussi qu'à chaque avancement dans l'arbre, la médiane du nouveau nœud courant diminue(fils gauche) ou augmente (fils droit) par rapport à son père

Lors de la recherche de notre fenêtre en y, donc lorsque la médiane n'est pas encore dans la fenêtre, nous explorons les différents nœuds et testons les différents report le long du chemin jusqu'à ce que la médiane soit dans la fenêtre. Nous effectuons donc un appel récursif soit sur le fils gauche, soit sur le fils droit. Dans le pire des cas nous explorons donc la hauteur de l'arbre pour arriver au cas où la médiane se trouve dans la fenêtre. Celle-ci étant en log(n) comme vu précédemment, nous ferons log(n) appels dans le pire des cas pour arriver à la fenêtre que nous cherchons. Nous ferons ensuite k appels récursifs correspondants au nombre de Segments se trouvant dans la fenêtre. Nous aurons donc log(n)+k appels récursifs dans le pire des cas.

Nous avons donc une complexité en temps dans le pire des cas pour cet algorithme en $\mathcal{O}(log(n) + k)$ où n est le nombre de segments et k le nombre de segments dans la fenêtre.

6.4 Modification des fenêtres

6.4.1 Explication

Le but de cette classe est de créer différents Pst en changeant les coordonnées des segments afin de transformer n'importe quel type de fenêtre lors du windowing en notre fenêtre idéale. Nous ne réexpliquons pas tout le principe en détail, car il a déjà été expliquer précédemment dans le rapport.

Nous créons donc 4 listes avec les segments changés afin de construire 4 Pst différents sur lesquels nous appliquerons nos différents windowing. Nous réutiliserons la même méthode à la fin d'un windowing pour avoir les segments d'origine.

6.4.2 Complexité dans le pire des cas

Deux calculs de complexité en temps doivent être pris en compte :

- Les transformations de segments, qui se font en $\mathcal{O}(n)$ chacune. Nous additionnons donc 4 fois cette complexité, ce qui nous donne du $\mathcal{O}(n)$
- Les constructions des 4 arbres , qui se font chacune en $\mathcal{O}(n.log(n))$, ce qui nous donne aussi un total de $\mathcal{O}(n.log(n))$.

Notre complexité finale pour la construction généralisée sera donc en $\mathcal{O}(n.log(n))$ dans le pire des cas.

7 Mode d'emploi

Pour lancer une commande, rendez-vous dans le dossier racine du projet à l'aide du terminal.

7.1 Javadoc

Pour compiler la Javadoc, entrez la commande `ant javadoc`. La documentation compilée se trouve alors dans le répertoire **build/doc/**.

7.2 Tests unitaires

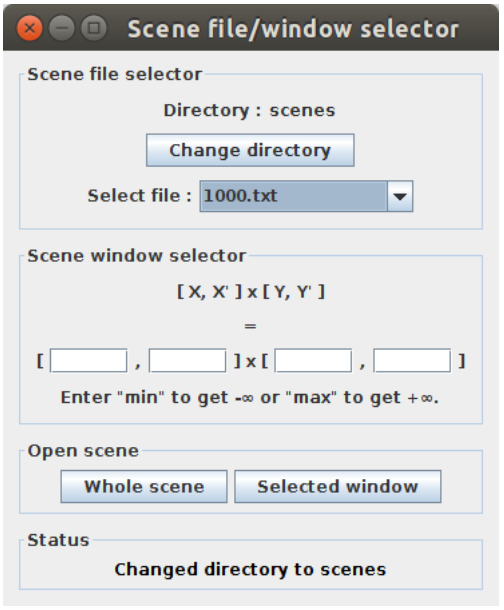
Pour lancer les tests unitaires, entrez la commande `ant test`.

7.3 Lancement du programme

Pour compiler et lancer le programme à partir des fichiers sources, entrez la commande `ant run`.

7.4 Interface utilisateur

L'interface utilisateur est particulièrement intuitive et ressemble à ceci :



Comme vous pouvez le voir, l'interface est divisée en 4 parties :

7.4.1 Sélecteur de fichier

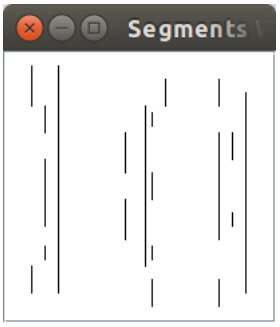
La première partie contient un sélecteur de fichier qui vous proposera par défaut de choisir un fichier .txt parmi ceux présents dans le dossier **scenes** (relatif à l'application). Vous pouvez changer de répertoire si vous souhaitez choisir des fichiers qui se trouvent à un autre emplacement.

7.4.2 Sélecteur de fenêtre

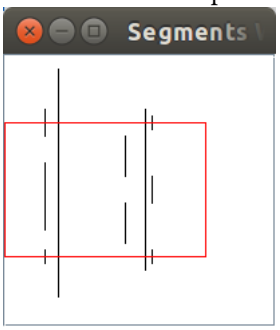
La seconde partie vous permet d'indiquer au programme la fenêtre à utiliser pour effectuer le windowing. Entrez "min" et "max" pour indiquer respectivement $-\infty$ et $+\infty$.

7.4.3 Ouvreur de scène

La troisième partie vous propose deux boutons. L'un ouvrira, dans une nouvelle fenêtre, toute la scène contenue dans le fichier sélectionné par le sélecteur de fichier. Exemple :



L'autre bouton ouvrira, également dans une nouvelle fenêtre, la scène contenue dans le fichier sélectionné, après y avoir appliqué le windowing en utilisant la fenêtre fournie par le sélecteur de fenêtre. La fenêtre sélectionnée s'affichera en rouge par dessus la scène. Exemple :



Si la scène est trop grande pour entrer dans la fenêtre de visualisation nouvellement ouverte, des barres de défilement apparaîtront et il vous sera possible de naviguer dans la scène en maintenant le pointeur de la souris enfoncé.

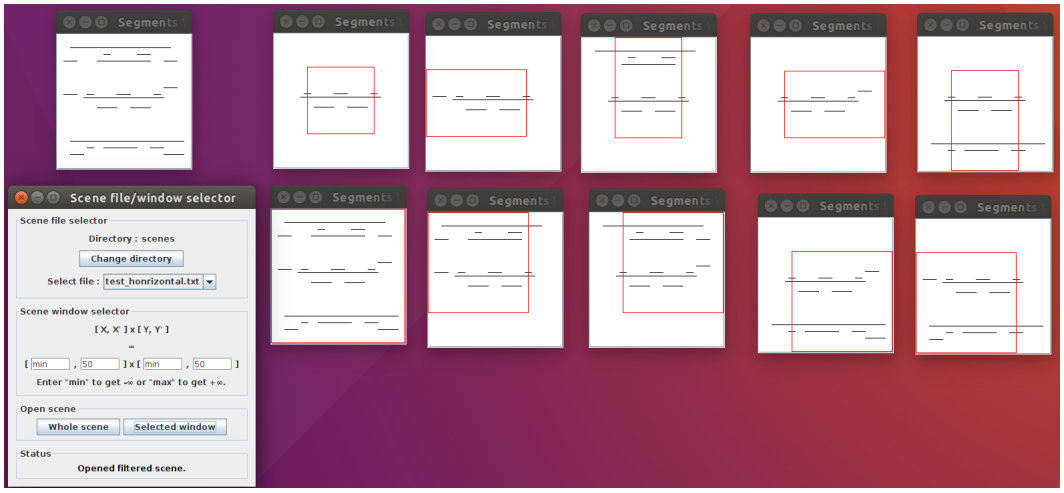
Si vous le désirez, il est possible de changer la taille de la visualisation à l'aide de la molette de votre souris.

7.4.4 Information sur l'état du programme

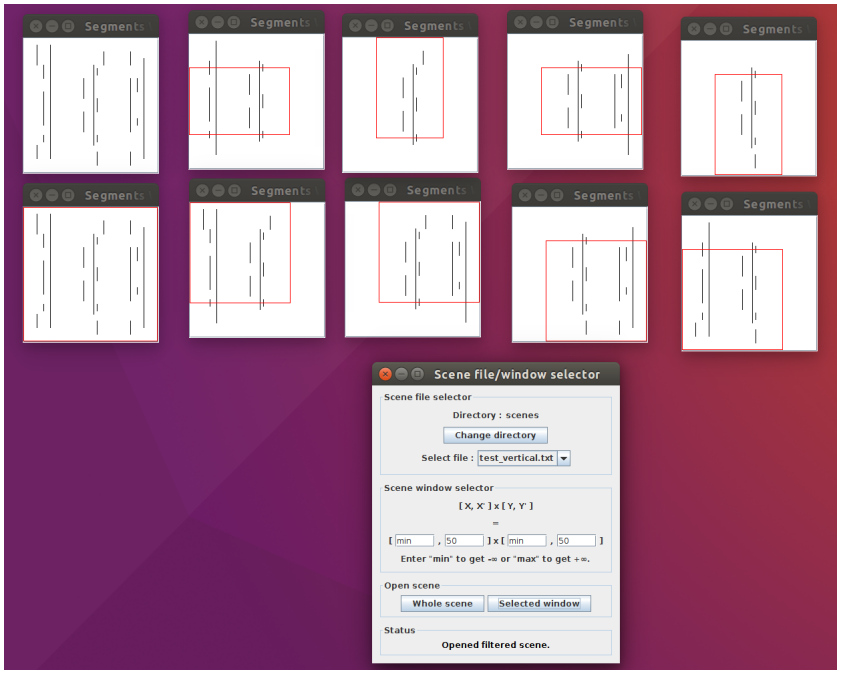
La quatrième et dernière partie de l'interface affiche l'état du programme. Les erreurs y sont affichées en rouge, alors que les informations y sont en noir.

8 Illustrations

Cette section sert d'illustration, afin de vous permettre d'observer différents cas d'application du windowing via l'application que nous avons créée. Cette image concerne des applications du windowing sur des segments horizontaux :



Cette image concerne des applications du windowing sur des segments verticaux :



9 Améliorations possibles

Plusieurs améliorations auraient encore été possibles :

- calculer le point moyen (parmi tous les points définissant les segments) et décider des transformations à utiliser en fonction de la situation de la fenêtre fermée par rapport au point moyen ;
- bien qu'il ne nous ait pas été demandé de traiter les fenêtres ayant 2 bornes infinies, décider des transformations à utiliser en fonction de l'orientation des 2 bornes.

10 Conclusion

Nous avons bien réalisé les objectifs fixés dans l'introduction, à savoir créer et manipuler une structure de données non vue au cours. Pour ce faire, nous avons utilisé différents concepts vus au cours "Structure de données 2", comme les tas et les ABR.

Ce travail nous a permis de faire un peu de "recherche" en groupe, dans l'optique de résoudre un problème nouveau. Lors de nos réunions de groupe, nous avons pu partager nos points de vue individuels et ainsi considérer le problème sous différents aspects. Nous gardons un souvenir très positif de ce travail. Nous tenons à remercier les professeur et assistant qui nous ont soutenues pour mener à bien ce projet, à savoir BRUYÈRE Véronique et DEVILLEZ Gauvain.