Université de Mons
Faculté Des Sciences

Structure de données II

Windowing

Professeur:
Véronique Bruyère
Assistant:
Pierre Hauweele

Auteurs:
Cyril Moreau
Arnaud Moreau

Année académique 2022-2023

Contents

1 Introduction **2**

2 Notre approche **3**
2.1 Composite Number Space 3
2.2 Priority Search Tree 4
2.2.1 Construction 4
2.2.2 Complexité 5
2.3 Windowing 5
2.3.1 Implémentation 7
2.3.2 Complexité 9
2.4 Interface graphique 10
2.4.1 En résumé 10
2.4.2 Détail du package `front` 11

3 Diagramme UML **12**

4 Mode d'emploi **14**

5 Conclusion **15**

1 Introduction

La géométrie informatique (*computational geometry*) est l'étude d'algorithmes appliqués à la géométrie. Certains problèmes tombants dans cette catégorie sont des problèmes que l'on doit résoudre chaque jour dans beaucoup de cas d'utilisation. Un de ces cas, extrêmement courant, est l'affichage d'une carte de grande taille (au sens du nombre de droites qui dessinent les frontières, routes, villes, maisons...). Lorsque le nombre de droites, points... à afficher atteint un ordre de grandeur extrêmement élevé (et c'est très souvent le cas), il faut pouvoir trouver une façon efficace de stocker et récupérer ces données. La technique de *windowing* est répandue dans beaucoup de cas d'utilisation, et consiste à appliquer une requête (*query*) à l'ensemble de données (*dataset*) et de retourner celles qui satisfont certaines conditions. La référence qui a été utilisée est la suivante : De Berg, M., Cheong, O., Van Kreveld, M., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer Science & Business Media.

Dans notre cas, l'ensemble de données est un ensemble de segments et la requête a pour objectif d'obtenir tous les segments contenus (au moins partiellement) dans un certain intervalle en X et en Y. Tous les segments sont soit horizontaux, soit verticaux, et la fenêtre de requête, exprimée comme $[x : x'] \times [y : y']$, forme un rectangle dont le coin inférieur gauche est situé en (x, y) et le coin supérieur droit en (x', y'). Nous considérons qu'un segment est inclus dans une fenêtre de requête (et sera donc *reported* par une query) si un point du segment se trouve dans la fenêtre de requête. Mathématiquement, on veut qu'un point du segment (soit une extrémité, soit un point se situant entre les deux extrémités) de coordonnée (x_1, y_1) satisfasse $x \leq x_1 \leq x'$ et $y \leq y_1 \leq y'$ (en supposant que la fenêtre soit de la forme $[x : x'] \times [y : y']$).

Pour ce projet, nous avons dû construire en Java une application répondant à plusieurs critères :

1. charger un ensemble de segments, c'est-à-dire lire un fichier au format adéquat et stocker les données dans un arbre de recherche à priorité T ;
2. afficher cet ensemble de segments ;
3. saisir une fenêtre (bornée complètement ou partiellement) ;
4. afficher le résultat de l'application du windowing sur base de la fenêtre proposée en utilisant les structures de données proposées par l'ouvrage de référence.

Au travers de ce rapport, nous allons expliquer notre compréhension et notre approche de ce problème, notre réflexion autour de celui-ci ainsi que les techniques que nous avons mises en place afin de résoudre ce défi.

2 Notre approche

2.1 Composite Number Space

Dans la suite de ce rapport, nous ferons l'hypothèse que tous les points sont distincts, car sans cela, plusieurs problèmes apparaitraient par la suite. Et comme il est très peu probable de n'avoir aucun point avec des coordonnées égales dans un ensemble quelconque de segments, il est impératif de gérer ce problème. Le livre de référence nous introduit donc la notion de composite numbers.

Les coordonnées des points de l'ensemble sont remplacées par des éléments du composite number space. Concrètement, nous remplaçons les coordonnées x et y de chaque point par des paires notées $(x \mid y)$ pour avoir un nouveau point $((x \mid y), (y \mid x))$. Un ordre sur ces nouvelles coordonnées est défini tel que pour deux paires $(a \mid b)$ et $(c \mid d)$, nous avons

$$(a \mid b) < (c \mid d) \iff a < c \vee (a = c \wedge b < d)$$

Grâce à cet ordre, nous avons bien que les coordonnées sont distinctes pour un ensemble de points quelconques.

Nous devons également adapter la façon de faire des requêtes de windowing pour utiliser cette technique. Imaginons que l'on souhaite récupérer les points d'un ensemble P qui se trouve dans la fenêtre $R = [x : x'] \times [y : y']$ en utilisant la technique vue ci-dessus. On va donc construire $\hat{\text{P}}$ l'ensemble P où chaque point $p = (p_x, p_y)$ est remplacé par $((p_x \mid p_y), (p_y \mid p_x))$.

Nous devons également modifier la fenêtre R pour qu'elle soit compatible avec la nouvelle représentation des points. on a donc :

$$\hat{\text{R}} = [(x \mid -\infty) : (x' \mid +\infty)] \times [(y \mid -\infty) : (y' - +\infty)]$$

Nous n'avons pas réellement besoin de stocker directement ces nouvelles coordonnées. En effet, il suffit de comparer également l'autre coordonnée lorsque la première est identique aux deux points. Voici l'algorithme que nous avons utilisé pour la comparaison des coordonnées y.

Algorithm 1 compareTo(p1, p2)

Input : p1, p2 $\in \mathbb{R}^2$
Output : -1 if $p1 < p2$, 1 if $p1 < p2$ and 0 if $p1 = p2$

1: **if** $p1_y < p2_y$ **then return** -1
2: **else if** $p1_y > p2_y$ **then return** 1
3: **else if** $p1_x > p2_x$ **then return** 1
4: **else if** $p1_x < p2_x$ **then return** -1
5: **else return** 0
6: **end if**

2.2 Priority Search Tree

La structure d'arbre de recherche à priorité est une structure permettant de stocker des points de $\mathbb{R}^2$ et de rechercher efficacement ceux qui sont dans une fenêtre de type $(-\infty : q_x] \times [q_y : q'_y]$. Pour permettre cela, cette structure mélange le concept d'arbre binaire de recherche sur les coordonnées y et de file à priorité sur les coordonnées x. En effet, pour tout nœud interne de l'arbre, la coordonnée x du point qui y est stockée est toujours inférieure à la coordonnée x de ses fils. De plus, à chaque nœud une valeur y_{mid} est stockée. Cette valeur correspond à la médiane des coordonnées y de tous les points dans les sous-arbres du nœud. Ces sous-arbres sont organisés tels que

$$P_{below} := \{p \in P \backslash \{p_{min}\} \mid p_y < y_{mid}\}$$
$$P_{above} := \{p \in P \backslash \{p_{min}\} \mid p_y > y_{mid}\}$$

La structure est construite telle que P_{below} correspond au sous-arbre gauche et P_{above} correspond au sous-arbre droit du nœud courant.

2.2.1 Construction

Pour construire une telle structure, un algorithme récursif est rapidement trouvé.

Algorithm 2 buildPSTHelper(p)

Input : P a set of n points $\in \mathbb{R}^2$ sorted on the y coordinates (of the left point)
Output : The root of the PST

```
1:  if length(data) = 1 then
2:      return node(data)
3:  else if length(data) = 0 then
4:      return null
5:  else
6:      p_min ← findMinX(data)
7:      P* ← removeRootFromData(data,p_min)
8:      p_mid ← P*[length(P*) // 2]
9:      res ← node(p_min, y_mid)
10:     P_below = {p ∈ P* | p_y ≤ p_mid}
11:     P_above = {p ∈ P* | p_y > p_mid}
12:     leftChild(res) ← createPST(P_below)
13:     rightChild(res) ← createPST(P_above)
14:     return res
15: end if
```

Pour y_{mid}, au lieu de garder la coordonnée y médiane, nous stockons le point médian de la liste permettant ensuite de séparer correctement les points même si tous les points ont la même coordonnée y grâce aux composite numbers.

L'algorithme *buildPSTHelper(p)* a comme hypothèse que les coordonnées sont triées selon les y ce qui permet de trouver rapidement la médiane. Pour

trier les points, nous faisons d'abord appel à un algorithme *buildPST(p)* qui va trier les données grâce à un algorithme *sort(p,var)* qui va trier la liste de points p selon la composante var. Ce tri est réalisé avec un tri par tas en $O(n \log n)$.

Algorithm 3 buildPST(p)

Input : P a set of n points $\in \mathbb{R}^2$
Output : The root of the PST

```
1: if length(data) = 1 then
2:     return node(data)
3: else if length(data) = 0 then
4:     return null
5: else
6:     sortedData ← sort(p, y)
7:     return buildPSTHelper(sortedData)
8: end if
```

2.2.2 Complexité

Proposition Un priority search tree est construit en $O(n \log_2 n)$ avec un prétraitement en $O(n \log n)$.

Prétraitement Sans prendre en compte la complexité de *buildPSTHelper(p)*, l'algorithme *buildPST(p)* est en $O(n \log n)$.

Note. Dans le pire des cas, les lignes 1 à 4 sont en $O(1)$ tandis que le tri par tas en ligne 6 est en $O(n \log n)$. Nous avons donc un prétraitement en $O(n \log n)$.

Construction Dans le pire des cas, *buildPSTHelper(p)* est également en $O(n \log n)$.

Note. Les lignes 1 à 4, 8, 9 et 14 sont en $O(1)$ tandis que *findMinX(data)*, *removeRootFromData(data, p_{min})* et la séparation en 2 listes P_{below} et P_{above} sont en $O(n)$ car, dans le pire des cas, il faut parcourir toute la liste p.

Finalement, on peut montrer grâce au master theorem que l'algorithme est en $O(n \log_2 n)$. Pour simplifier les calculs, prenons que n est un nombre impair. De cette façon, lorsque la racine sera enlevée, on pourra séparer l'ensemble en 2 parties de même taille. Nous avons donc

$$T(n) = 2 * T(n/2) + O(n)$$

En effet, avons 2 appels récursif séparant l'ensemble en 2 et si on ne considère pas les lignes 12 et 13, l'algortihme est en O(n).
Par le master theorem, nous avons bien que *buildPSTHelper(p)* est en $O(n \log_2 n)$

2.3 Windowing

Le but de ce travail était de retourner les segments contenus dans une fenêtre rectangulaire. La référence nous explique comment récupérer les points contenus

dans une fenêtre non bornée vers la gauche grâce à un arbre de recherche à priorité. Nous avons donc dû adapter la méthode pour pouvoir retourner des segments et pouvoir également gérer les autres types de fenêtres.

Lien entre segment et point Nous nous occupons de segments verticaux et horizontaux ce qui simplifie grandement le problème. Les segments peuvent soit être complètement contenu dans la fenêtre, soit avoir un seul point à l'intérieur, soit n'avoir aucun point dans la fenêtre.

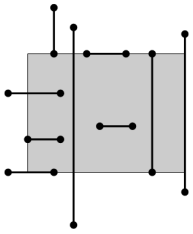


Figure 1: Exemple de segments

L'algorithme proposé par la référence fonctionnerait bien pour les 2 premiers types de segments, car il retournerait les points situés dans la fenêtre. Cependant, il ne détecterait pas les segments qui traversent la fenêtre avec aucun point à l'intérieur de celle-ci.
Nous avons donc choisi une méthode qui consiste à stocker les segments horizontaux et verticaux dans 2 arbres de recherche à priorité différents. Dans la suite de l'explication, nous nous focaliserons sur les segments horizontaux. L'idée sera la même pour les segments verticaux, mais avec le rôle des coordonnées x et y inversé.

Nous stockons les segments directement dans l'arbre, mais nous ne considérons que les points de gauche dans un premier temps. Lors d'une requête de windowing, nous effectuons une requête non bornée sur la gauche qui retournera tous les points qui sont à gauche de la borne q'_x. Pour vérifier si le segment retourné est bien dans la fenêtre, il suffit de vérifier si la coordonnée x du deuxième point est plus grande que q_x.

Exemple. Sur la figure 2, nous avons 4 segments différents et une fenêtre de la forme $W = [2, 10] \times [0, 15]$. Si l'on applique la technique expliquée ci-dessus, nous récupérons les points rouges. On remarque que seul le segment noir à gauche de la figure ne doit pas être retourné car la coordonnée x de son deuxième point vaut 0 ce qui est plus petit que la coordonnée minimale de la fenêtre (= 2). Les segments rouges ont leur 2e point à droite de la borne gauche de la fenêtre, ils sont donc dans la fenêtre ou traverse celle-ci et sont donc rapportés.

Différentes fenêtres Comme montré ci-dessus, les fenêtres complètement bornées et non bornées sur la gauche fonctionnent dans notre implémentation.

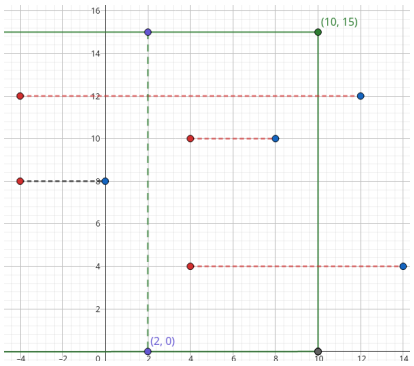


Figure 2: Exemple de requête avec une fenêtre $W = [2, 10] \times [0, 15]$

Cependant, il reste encore les fenêtres non bornées selon la coordonnée y et non bornée sur la droite. Ces différentes fenêtres sont directement gérées par le fait qu'un Double en java peut valoir $\pm\infty$ et que l'on puisse faire des comparaisons entre Double.

Exemple. Si une fenêtre est de la forme $W = [q_x : q'_x] \times (q_y : q'_y]$ avec $q_y = -\infty$, les comparaisons entre un point et q_y renverra toujours 1, c'est-à-dire que le point sera toujours plus grand que la borne q_y car on ne peut pas avoir de point avec des coordonnées infinies.

2.3.1 Implémentation

Pour récupérer les segments horizontaux traversant ou contenu dans une fenêtre de type $W = [q_x : q'_x] \times [q_y : q'_y]$. Nous parcourons tout d'abord l'arbre de recherche à priorité à la recherche de q_y et q'_y afin de récupérer tous les segments horizontaux qui ont la coordonnée y de leur point de gauche dans la fenêtre.

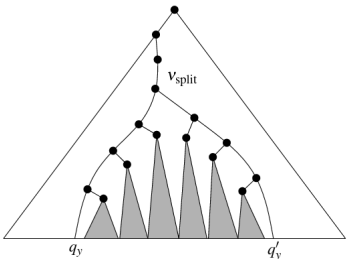


Figure 3: Requête sur un arbre de recherche à priorité

Les structures en gris sur la figure 3 représentent un ensemble de segments de l'arbre qui ont la coordonnée y de leur point de gauche dans la fenêtre. Sur ceux-ci, il suffit de vérifier que la coordonnée x de leur point de gauche est plus petite que q'_x et que la coordonnée x de leur point de droite soit plus grande

que q_x (voir figure 2). Nous appelons donc l'algorithme *reportInSubtree(v,w)* sur chaque sous-arbre en gris.

Algorithm 4 reportInSubtree(v, W)

Input : W a window $[x_{min} : x_{max}] \times [y_{min} : y_{max}]$, v a priority search tree containing horizontal segments (P1,P2) where P1,P2 $\in \mathbb{R}^2$ and $P1_y \in [y_{min} : y_{max}]$

Output : The segments crossing or lying in the window

```
1: if x_max > v(P1_x) then
2:     if x_min < v(P2_x) then
3:         report(v)
4:     end if
5:     if hasLeftSubTree(v) then
6:         report(reportInSubtree(v_below))
7:     end if
8:     if hasRightSubtree(v) then
9:         report(reportInSubtree(v_above))
10:    end if
11: end if
```

De plus, les nœuds rencontrés lors de la recherche de q_y et q'_y peuvent également être des segments qui croisent ou sont contenus dans la fenêtre, il faut donc vérifier cela également. Nous arrivons donc à l'algorithme suivant :

Algorithm 5 query(T, W)

Input : T a horizontal priority search tree and W a window
Output : The segments crossing or lying in the window

```
1: Search q_y and q'_y in T.
2: V_split ← the root node where the two search paths split.
3: for each node V on the search path of q_y or q'_y do
4:     if p(v) ∈ W then report p(v).
5:     end if
6: end for
7: for each node v on the path of q_y in the left subtree of V_split do
8:     if q_y < p(v) then reportInSubTree(rc(v), W)
9:     end if
10: end for
11: for each node v on the path of q'_y in the right subtree of V_split do
12:     if q'_y > p(v) then reportInSubTree(lc(v), W)
13:     end if
14: end for
```

2.3.2 Complexité

ReportInSubTree L'algorithme *reportInSubTree(V)* est en $O(k)$ où k est le nombre de noeuds du sous-arbre v. Pour être plus précis, l'algorithme est en $O(q)$ où q est le nombre de segments S tel que $left(S)_x \leq q'_x$

Note Soit un point p stocké dans un nœud n du sous-arbre de racine v. Par définition de l'arbre de recherche à priorité, on a que p_x est plus grand ou égal la coordonnée x de son père. On utilise donc cette propriété pour visiter chaque nœud du sous-arbre jusqu'au moment où l'on tombe sur un nœud n tel que $p(n)_x > q'_x$. On s'arrête ici parce que tous les points dans les sous-arbres de n ont leur coordonnée x plus grande que la borne supérieure de la fenêtre. En utilisant cette technique on aura visité tous les nœuds n tels que $p(n)_x \leq q'_x$. De plus, chaque visite de nœud est en O(1). Nous avons bien que *reportInSubTree()* est en $O(q)$ où q est le nombre de segments dont le point de gauche a sa coordonnée x plus petite que la borne q_x de la fenêtre.

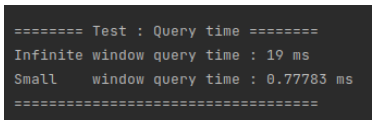
Query L'algorithme *Query()* est en $O(\log n+q)$ où q est le nombre de segments S tel que $left(S)_x \leq q'_x$

Note La visite de chaque nœud lors de la recherche de q_x et q'_x est en temps constant plus le temps des appels à *reportInSubTree()*. L'arbre est un arbre binaire tel que chaque "étage" est rempli, sauf éventuellement le dernier, sa hauteur est donc en $O(\log n)$. Dans le pire des cas, $\log n$ nœuds sont donc visités lors de la recherche de q_x et q'_x et les *reportInSubTree()* est en $O(q)$. On a bien que *Query()* est en $O(\log n + q)$.

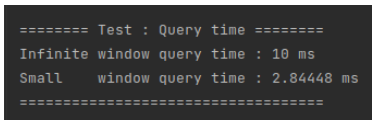
Comparaison avec l'implémentation en Java Les complexités annoncées ci-dessus restent théoriques, lors de l'implémentation, certaines lignes de codes sont rajoutées afin de respecter la syntaxe du langage choisi. Dans notre cas, nous ajoutons majoritairement des appels à des fonctions et des comparaisons élémentaires qui sont en temps constant et donc ne modifient pas la complexité de notre algorithme. Cependant, nous utilisons des *ArrayList* pour stocker le résultat des query. Lors d'un appel à add(), la complexité dans le pire des cas est en $O(n)$[1]. En effet, si l'on a atteint la taille maximale du tableau, il faut en recréer un et copier toutes les données, ce qui est en $O(n)$ tandis que si la taille maximale du tableau n'est pas atteinte, la complexité est en $O(1)$. Cette différence pose un problème, car cela modifie les complexités annoncées ci-dessus en faisant passer la complexité de Query() à $O(n \log n)$ comme chaque visite de nœud ne serait plus en temps constant, mais en $O(n)$. Cependant, à chaque fois que la taille du tableau doit augmenter, sa taille double au lieu de simplement être incrémentée, cela limite donc le nombre de fois que cette situation arrive.

Nous avons donc mesuré le temps d'exécution de la fonction *query*() sur un ensemble de 100 000 segments et réalisé une moyenne sur 100 itérations.

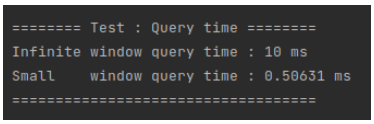
[1]Source : `https://www.baeldung.com/java-collections-complexity`. (12/04/2023)



(a) Première version



(b) Avec capacité initiale



(c) ArrayList générale

Figure 4: Comparaison du temps d'exécution de l'algorithme *Query*()

Une première amélioration a été d'utiliser une seule ArrayList dans la classe *Windowing* qui va récupérer les différents segments reportés par les *query*(). Bien que la complexité dans le pire des cas reste inchangée, on peut voir sur la figure 4c une amélioration sur le temps d'exécution qui est expliqué par la réduction du nombre d'ArrayList créée et du nombre de copies.

Une solution naïve pour avoir la complexité attendue aurait été de fixer la taille initiale de l'ArrayList au nombre maximum de segments qui peuvent y être ajouté. Chaque appel à *add*() serait en $O(1)$ et permettrait donc de respecter la complexité. On remarque sur la figure 4b qu'à cause de la grande taille du tableau, le temps n'est pas amélioré pour une grande window, donc une grande liste, mais que le temps pour une petite window est clairement plus grand.

Finalement, nous avons choisi de rester sur la situation de la figure 4c. Bien que la complexité théorique ne soit pas complètement respectée, elle apportait les meilleures performances grâce au fait que le nombre de copies de tableau soit très réduit.

2.4 Interface graphique

2.4.1 En résumé

Afin d'afficher le résultat d'une requête de façon intuitive et simple d'utilisation, nous devons suivre plusieurs étapes. Premièrement, il faut évidemment effectuer la requête et récupérer dans une liste les segments qui la satisfont et qui seront donc affichés à l'écran. Une fois cette liste récupérée, nous les plaçons dans un Group (objet JavaFX) avec les quatre (ou moins) segments rouges qui dessinent la fenêtre de requête. Tout cela est ajouté à l'affichage comme un seul objet (grâce au Group).

Ensuite, il faut adapter la taille des segments affichés à l'interface. On ne veut pas qu'un segment allant de -1000 à 1000 en X soit beaucoup trop grand pour l'écran ; on veut que, s'il est seul par exemple, il soit simplement affiché dans son entièreté. De même pour un segment allant de -10 à 10 en X : on ne veut pas que celui-ci soit minuscule s'il n'est pas affiché aux côtés d'un segment allant de -1000 à 1000. Pour que tous les segments soient initialement affichés

entièrement, on va *scale up* ou *scale down* la taille du Group. Cela permet de changer la taille de tous les segments d'une seule fois. Pour les déplacer grâce à la souris, on suit la même logique, mais en déplaçant le Group en X ou en Y.

2.4.2 Détail du package `front`

Package `controllers` Le package `controllers` regroupe chaque *controller* utilisé dans l'application. Un controller permet de définir des méthodes qui seront appelées suite à divers évènements, ainsi que des attributs faisant références vers des objets contenus dans une scène. Dans un fichier `.fxml`, on retrouvera chacun des éléments qui structure la scène. Ces éléments ont un attribut `fx:controller` (qui ne peut en réalité être défini que pour l'élément racine) qui pointe vers un *controller* Java et peuvent posséder des enfants. Ces enfants (ainsi que l'élément mère) seront capables d'appeler les méthodes définies dans ce *controller* et de définir leur attribut `fx:id` permettant de les identifier de façon unique. Pour chaque `fx:id` définit, on peut créer en attribut un objet du même type dans le *controller* et lui appliquer des modifications qui se répercuteront directement sur l'interface graphique (changement de position, taille, attributs de style...). Attention ! On ne peut définir l'attribut `fx:controller` d'un élément si et seulement si cet élément est l'élément racine du fichier `.fxml`.

Package `events` Le package `events` contient les classes permettant de gérer des évènements personnalisés. Dans notre cas, nous utilisons un évènement *InvalidInput* permettant de signaler à certains composants qu'une entrée est incorrecte. Cela sert à, par exemple, dévoiler les points d'exclamation rouges notifiants à l'utilisateur que son entrée n'est pas valide. En terme d'implémentation, un léger détail pourrait être amélioré : au lieu d'envoyer l'évènement au champ de texte et que ce dernier s'occupe d'afficher/masquer les points d'exclamation, on aurait dû envoyer l'évènement directement aux points d'exclamation afin que ceux-ci s'affichent. En conséquence, il nous faudrait un second évènement permettant de les masquer de nouveau lorsque l'entrée de l'utilisateur est correcte. Par soucis de temps, nous n'avons pas pu modifier notre implémentation.

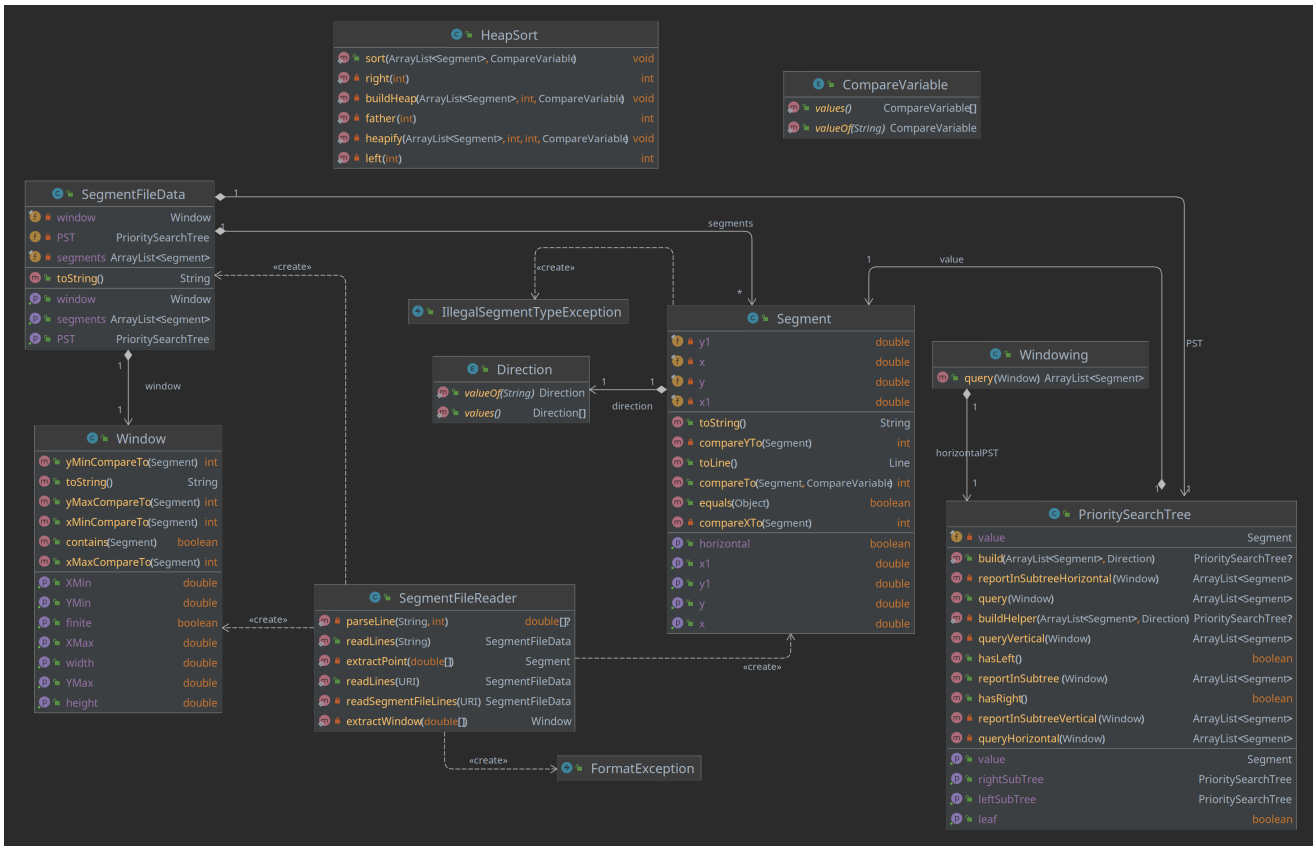
Package `scenes` Le package `scenes` contient deux classes. La première, `Scenes`, contient en attribut une référence vers chaque scène utilisée. La seconde, `SceneLoader` contient la méthode `load` à laquelle on peut passer un nom de scène afin de facilement charger le fichier `.fxml` situé dans les *resources* correspondant à ce nom.

Fonctionnement de l'architecture Tout d'abord, on crée un fichier `.fxml` dans les ressources, dans le sous-répertoire `scenes`. Ensuite, on définit un attribut de la classe `Scenes` et on l'initialise à la valeur de retour de la méthode `load` de la classe `SceneLoader` à laquelle on passe le nom du fichier `.fxml` que l'on vient de créer. Une fois que cela est fait, on crée un `Stage` et on définit

sa scène comme étant celle que l'on vient de créer. Pour définir le comportement des composantes de notre scène, on crée un *controller* dans le package `controllers` et on l'assigne à l'élément racine du fichier `.fxml`.

3 Diagramme UML

Diagramme de classes du package `back` qui contient les structures de données, l'algorithme de tri, les segments, la fenêtre...



HeapSort Cette classe statique contient notre implémentation du tri par tas en $O(n \log n)$. La méthode *sort(ArrayList, CompareVariable)* prend la liste de segment à trier et selon quelle composante le tri doit être effectué.

CompareVariable Énumération ayant 2 valeurs **X** et **Y** permettant de choisir la variable à considérer lors de comparaisons.

Segment La classe segment implémente la notion de segment que l'on va insérer dans l'arbre de recherche à priorité. Dans notre cas, un segment contient 4 coordonnées qui représentent les 2 points d'un segment que nous appelons point gauche et point droit. Lors des opérations sur les segments, nous nous focalisons sur le point gauche. Cependant, le point droit est utilisé pour vérifier si le segment croise bien la fenêtre. (Voir section 2.3)

Direction Énumération permettant de décrire le type de segment qu'un arbre de recherche à piorité va stocker. L'énumération peut avoir 2 valeurs : *Horizontal* et *Vertical*.

Windowing Classe permettant d'interagir avec les différentes méthodes de windowing. Le constructeur de la classe reçoit une liste de segment et construit 2 priority search tree, un pour les segments verticaux et un pour les segments horizontaux. Ensuite, la méthode *query(Window)* retourne les segments qui croisent ou son inclut dans la fenêtre passée en paramètre. L'intérêt de cette classe était de pouvoir séparer la partie sur le windowing et la partie interface graphique.

Window Classe implémentant le concept de fenêtre. Contient le point inférieur gauche et le point supérieur droit de la fenêtre. Chaque composante peut être infinie tant qu'elle respecte la contrainte que les composantes du coin supérieur droit soient supérieures aux composantes des points inférieur gauche.

PrioritySearchTree Implémente les méthodes vues dans les sections 2.2 et 2.3 sur les arbres de recherche à priorité en ajoutant des méthodes pour gérer les arbres horizontaux et verticaux.

SegmentFileData Représente les données lues dans un fichier. Contient la fenêtre initiale (première ligne du fichier) dans un objet `Window` ainsi qu'une liste de segments (contenu des autres lignes).

SegmentFileReader Sert à lire un fichier `.seg` et extraire son contenu sous forme d'objets utilisables. Les méthodes `readLines` servent de wrapper et sont les méthodes publiques qui permettent d'interagir avec la classe. La classe contient deux méthodes pour créer un point ou une window à partir de quatre `double`s, une méthode qui utilise une expression régulière pour vérifier qu'une ligne est correctement formattée et une méthode `readSegmentFileLines` qui utilise les trois méthodes précédentes. Cette dernière lit le contenu du fichier et retourne un objet `SegmentFileData` décrit plus haut.

Tests Les méthodes importantes utilisées pour récupérer les segments et construire l'abre de recherche à priorité ont été testées. La plupart des tests ont une image correspondant à l'ensemble de segments testé. Ces images se trouvent dans le dossier *misc*.

4 Mode d'emploi

Grâce au code source, l'application se lance via Gradle. On peut lancer l'application en se plaçant à la racine (fichier contenant `app`, `misc`, ...) via `gradle run` ou via `./gradlew run`. Pour lancer les tests, il suffit de remplacer `run` par `test`.

Attention ! La version minimale de Java requise est la version 16.

Lorsque l'application se lance, on se trouve sur l'écran principal. La surface rouge contiendra les segments chargés.

Charger des segments Le bouton `Load` permet de charger un fichier de segments parmi les trois exemples fournis, ou bien de choisir un fichier stocké localement sur la machine. Attention : un fichier de segments doit être au format `.seg` ou `.txt` et contenir du texte formaté comme présenté dans l'énoncé.

Appliquer une fenêtre Les quatres champs de texte `xMin`, `xMax`, `yMin` et `yMax` permettent d'entrer des coordonnées afin de définir une fenêtre au format [xMin : xMax] $\times$ [yMin : yMax], qui pourra être appliquée grâce au bouton `Apply` une fois qu'un fichier de segments est chargé. Pour définir une fenêtre non bornée, les champs acceptent les valeurs `-inf` et `inf`. Par exemple, entrer `-inf` dans le champ `xMin` donnera lieu à une fenêtre non bornée en X, dans la direction des X négatifs (vers la gauche donc).

Zoom & déplacement de l'affichage Il est possible de zoomer *in* et *out* grâce à la molette, ainsi que de déplacer les segments par *drag and drop*.

Lorsqu'un fichier de segments est chargé, on observe les segments en noir et un rectangle rouge (dont certains côtés sont manquants si la fenêtre est non bornée, afin de marquer l'absence de borne dans la direction correspondante) permettant de visualiser la position de la fenêtre appliquée. Le rectangle rouge est légèrement décalé vers l'extérieur des segments afin de pas les chevaucher.

Le bouton `"?"` peut être survolé afin d'afficher un message d'aide dans l'application.

5 Conclusion

Grâce à ce projet, nous avons découvert l'aspect théorique se cachant derrière la technique du *windowing*. Nous avons pu implémenter une structure de donnée complexe et la lier à une interface graphique afin de former un produit fini fonctionnel ayant une réelle utilité.

La plus grande difficulté rencontrée était de trouver une technique pour retourner des segments et non des points. En effet, la référence nous explique comment récupérer les points dans une fenêtre grâce au priority search tree et non des segments, un problème se posait lorsqu'un segment traverse la fenêtre, mais n'a aucun point à l'intérieur de celle-ci. Heureusement, les commentaires de P.Hauweele sur notre exercice préliminaire nous ont permis de nous diriger vers une solution fonctionnelle permettant de retourner les segments qui croisent la fenêtre ou ont un point à l'intérieur.

Nous avons réussi à nous rapprocher des complexités annoncées dans la référence pour la construction du priority search tree et pour les requêtes de windowing. Cependant, l'auteur annonce que si les données sont triées sur les y, la construction peut se faire en $O(n)$. Nous ne sommes malheureusement pas arrivés à un résultat permettant de garder l'équilibre de l'arbre, malgré nos nombreuses recherches à ce sujet.

Comme annoncé ci-dessus, nous avons réussi respecter la plupart des complexités annoncées dans la référence, ce qui rend notre application plutôt rapide. Le chargement d'un grand nombre de segments peut prendre un certain temps de chargement, mais après mesures de notre part, nous avons conclu que ce chargement était dû à l'affichage des segments et non aux calculs liés à la création du priority search tree et à la requête de windowing.