Projet de Structures de Données II

Rapport Final

Activité d'Apprentissage S-INFO-820

Groupe numéro: 08

Membres du groupe:

Altruy Alan
Nosal Victor

Année Académique 2022-2023

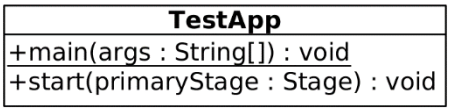
Faculté des Sciences, Université de Mons

Résumé

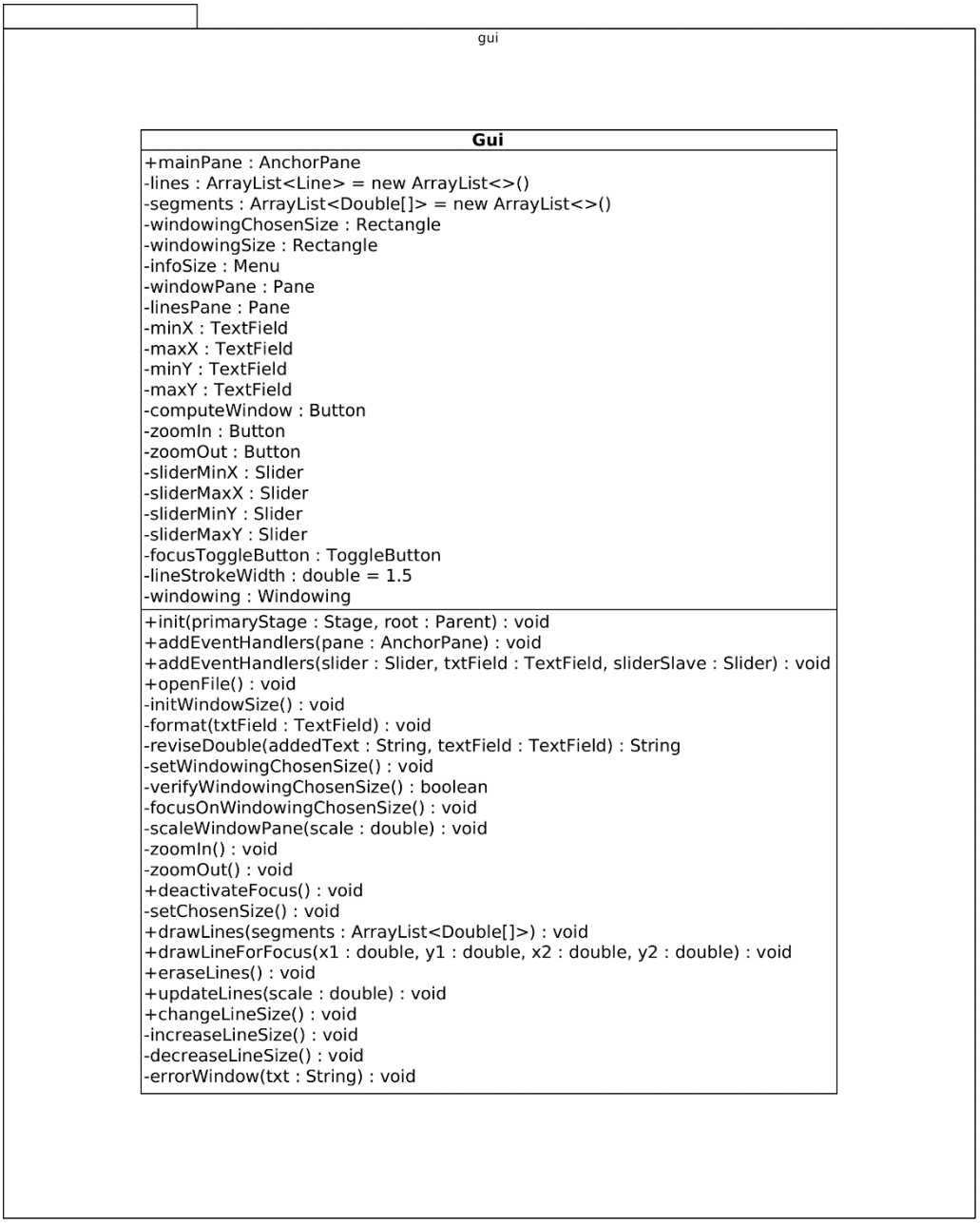
Ce rapport présente l'implémentation en Java d'une technique de 'windowing' sur un ensemble de segments, qui a été réalisée dans le cadre de l'AA S-INFO-820 "Projet de Structures de Données II", dispensé par Mme. BRUYERE Véronique en année académique 2022-2023.

Table des matières

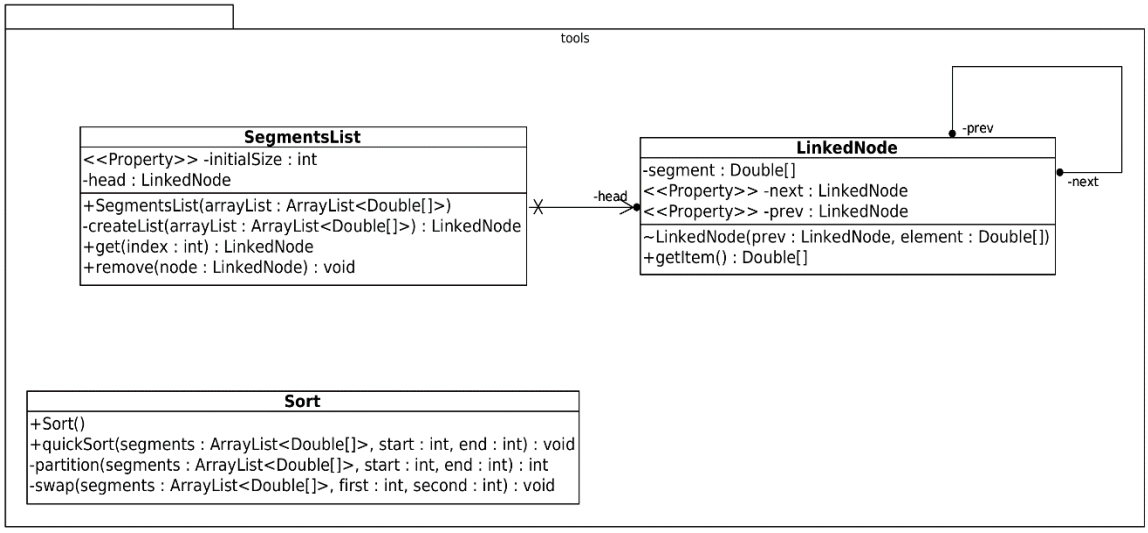
Diagrammes des classes



La classe **TestApp** représente la classe principale de l'application. Elle est responsable de l'initialisation des classes 'Gui' et 'Windowing'.



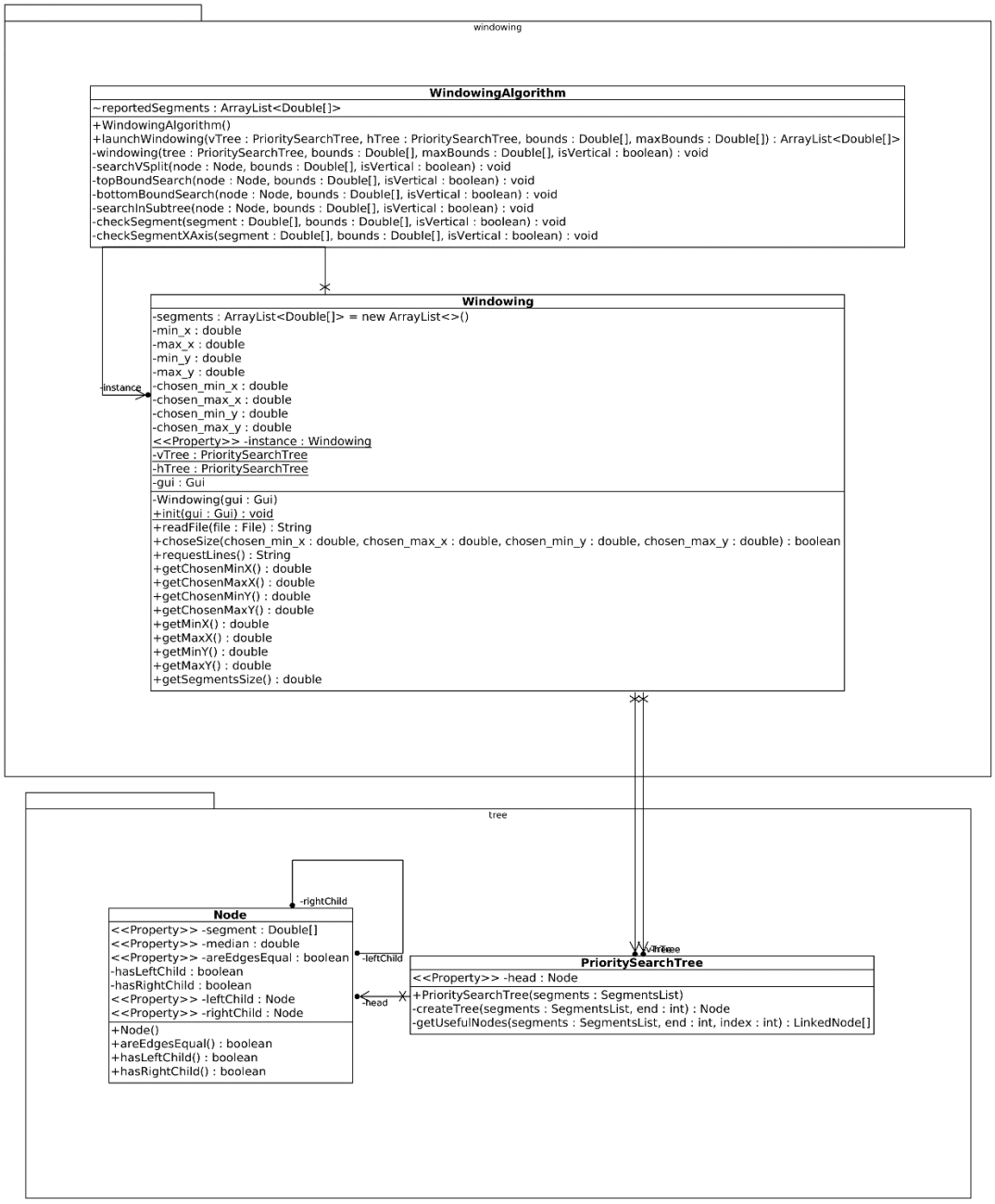
La classe **Gui** représente l'interface graphique de l'application. Elle est responsable de l'affichage de la fenêtre, de la fenêtre de Windowing, des segments, des champs de texte et des boutons.



La classe **SegmentsList** représente une liste chaînée de tableaux Double. Elle est utilisée pour stocker les segments dans les arbres de recherche à priorité verticaux et horizontaux.

La classe **LinkedNode** représente un nœud dans une liste chaînée SegmentsList.

La classe **Sort** contient une méthode quickSort qui trie une ArrayList de tableaux Double en fonction de la coordonnée y de la première extrémité du segment.



La classe **WindowAlgorithm** implémente l'algorithme de windowing pour la recherche de segments dans un arbre de recherche à priorité. Elle lance l'algorithme avec des bornes, des bornes maximales et des arbres de recherche à priorité.

La classe **Windowing** représente la classe principale de l'application. Elle est responsable de l'analyse du fichier d'entrée, du stockage des segments, de la construction des arbres de recherche à priorité, et de l'appel à l'algorithme de windowing pour trouver les segments qui intersectent la fenêtre choisie.

La classe **PrioritySearchTree** représente un arbre de recherche à priorité pour des segments. Elle utilise un algorithme de construction récursif pour construire un arbre de recherche à priorité dont chaque nœud contient un segment et sa médiane en y.

La classe **Node** représente un nœud dans un arbre de recherche à priorité.

Explication de l'arbre de recherche à priorité

L'arbre de recherche à priorité est une structure de données, plus précisément un arbre binaire équilibré. Cet arbre est équilibré car, pour tout nœud, le nombre de nœuds que contiennent ses sous-arbres diffère d'au plus une unité (l'éventuel nœud supplémentaire est placé dans le sous-arbre gauche).

Cette structure de données est utilisée dans ce projet pour stocker des segments, définis par les coordonnées de leurs deux points extrémités.

Les segments stockés dans cette structure sont soit tous verticaux, soit tous horizontaux, et sont considérés selon leur extrémité (appelée "point déterminant") basse ou gauche, respectivement.

L'arbre se construit récursivement. Le nœud racine contient le segment dont le point déterminant a le plus petit x. Ensuite, les segments sont répartis en deux groupes selon la coordonnée y de leur point déterminant, l'arbre gauche regroupe les y les plus petits, et le droit les plus grands. Une médiane séparant les deux groupes de segments est alors calculée.

Chaque nœud est composé d'un segment, de la médiane séparant ses deux sous-arbres, d'une référence vers chacun de ses deux sous-arbres, de booléens signalant l'existence ou non des sous-arbres gauche et droit, ainsi que d'un booléen permettant de savoir si la plus grande coordonnée y du sous-arbre gauche est égale à la plus petite du sous-arbre droit.

Explication des principaux algorithmes

1. createTree:

Cet algorithme construit de manière récursive un arbre à partir d'une liste de segments soit tous horizontaux, soit tous verticaux. Les segments sont définis par les coordonnées de leurs deux extrémités. L'algorithme est prévu pour travailler avec des segments horizontaux. On adapte la situation pour les segments verticaux en inversant les coordonnées x et y de leurs extrémités, ainsi qu'en considérant plus tard des bornes également modifiées. Les segments présents dans la liste en entrée sont triés selon la coordonnée y de leur point gauche. L'indice end définit la sous-liste à traiter.

L'algorithme createTree commence par récupérer deux nœuds, dans la sous-liste à traiter, via l'algorithme getUsefulNodes. L'un des nœuds contient le segment dont le point gauche a la plus petite coordonnée x et l'autre contient le segment qui permettra de calculer la médiane séparant les coordonnées y des segments appartenant aux deux sous-arbres. Ensuite, le segment le plus à gauche est supprimé de la liste et placé dans le nœud que l'on construit. S'il ne reste plus qu'un nœud dans la sous-liste, on crée récursivement le sous-arbre gauche de l'arbre. Si nous avons plus d'un nœud, ce sont alors les deux sous-arbres qui doivent être construits récursivement. Nous calculons aussi la médiane et le booléen permettant de savoir si la plus grande coordonnée y du sous-arbre gauche est égale à la plus petite du sous-arbre droit.

```
Algorithme createTree(segments, end)
Require: Liste de segments (représentée par une liste de tableaux de quatre nombres à virgule),
         triés par coordonnée y (du point gauche) croissante. Indice de fin (end) de la sous-liste
         à traiter.
Ensure: Noeud racine (target) de l'arbre de recherche à priorité créé.

    target ← Node()
    index ← end/2
    usefulNodes ← getUsefulNodes(segments, end, index + end%2)
    target.segment ← usefulNodes[0].item
    Supprime le segment le plus à gauche de la sous-liste à traiter
    if end = 1 then
        target.leftChild ← createTree(segments, 0)
    else if end > 1 then
        firstY ← usefulNodes[1].item[1]
        secondY ← usefulNodes[1].prev.item[1]
        target.median ← (firstY + secondY) / 2
        if firstY = secondY then
            target.AreEdgesEqual ← true
        end if
        target.leftChild ← createTree(segments, index-1 + end%2)
        target.rightChild ← createTree(segments, index-1)
    end if
    Retourner target
```

```
Algorithme getUsefulNodes(segments, end, index)
Require: Liste de segments (représentée par une liste de tableaux de quatre nombres
         à virgule). Indice de fin (end) de la sous-liste à traiter et Indice du
noeud    permettant de calculer la médiane.
Ensure: Tableau de noeuds contenant, pour la sous-liste à traiter, le noeud du
segment dont le point gauche à la plus petite coordonnée x et le noeud du segment
        permettant de calculer la médiane.

    leftmostSegmentIndex ← 0
    node ← segments[0]
    leftmostNode ← node
    indexNode ← node
    for i from 1 to end do
        nextNode ← node.next
        if nextNode.item[0] < leftmostNode.item[0] then
            leftmostSegmentIndex ← i
            leftmostNode ← nextNode
        end if
        if i = index then
            indexNode ← nextNode
        end if
        node ← nextNode
    end for
    if leftmostSegmentIndex ⩽ index then
        indexNode ← indexNode.next
    Retourner [leftmostNode, indexNode]
```

2. launchWindowing:

L'algorithme launchWindowing récupère les bornes de la fenêtre demandée et lance l'algorithme windowing sur les deux arbres contenant des segments soit tous horizontaux, soit tous verticaux. Comme pour l'algorithme de création de l'arbre, cet algorithme de windowing est prévu pour des segments horizontaux. Pour les segments verticaux, ceux-ci sont gérés de la même manière que pour la création de l'arbre, on inverse les coordonnées x et y de leurs extrémités. De plus, les bornes basse et haute deviennent respectivement les bornes gauche et droite. C'est grâce au booléen isVertical que la vérification des segments pourra se faire différemment selon si le segment est horizontal ou vertical.

```
Algorithme launchWindowing(vTree, hTree, bounds, maxBounds)
Require: Arbres de recherche à priorité (vTree et hTree) contenant des segments respectivement
         verticaux ou horizontaux. Bornes de la fenêtre (bounds) et de l'ensemble des segments
         (maxBounds), représentées par un tableau de quatre nombres à virgule (bornes basse,haute,
         gauche et droite).
Ensure: Les segments à afficher (reportedSegments) appartenant à la fenêtre.

    hMaxBounds ← [maxBounds[0], maxBounds[1]]
    vMaxBounds ← [maxBounds[2], maxBounds[3]]
    vBounds ← [bounds[2], bounds[3], bounds[0], bounds[1]]
    windowing(hTree, bounds, hMaxBounds, false)
    windowing(vTree, vBounds, vMaxBounds, true)
    Retourner reportedSegments
```

3. windowing:

L'algorithme windowing lance, selon les bornes haute et basse de la fenêtre, un des quatre algorithmes permettant de parcourir l'arbre : searchInSubtree si la fenêtre est non-bornée selon y, topBoundSearch si la borne basse est inexistante, bottomBoundSearch si la borne haute est inexistante et searchVSplit si la fenêtre est bornée selon y.

```
Algorithme windowing(tree, bounds, maxBounds, isVertical)
Require: Arbre de recherche à priorité (tree) contenant des segments. Bornes de la fenêtre
         (bounds) et de l'ensemble des segments (maxBounds), représentées par un tableau de
         quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical)
         vrai si l'arbre tree contient des segments verticaux, faux (donc horizontaux) sinon.
Ensure: / (Alimente la liste des segments à reporter).

    head ← tree.head
    if bounds[0] = maxBounds[0] then
        if bounds[1] = maxBounds[1] then
            searchInSubtree(head, bounds, isVertical)
        else
            topBoundSearch(head, bounds, isVertical)
        end if
    else
        if bounds[1] = maxBounds[1] then
            bottomBoundSearch(head, bounds, isVertical)
        else
            searchVSplit(head, bounds, isVertical)
        end if
    end if
```

4. searchVSplit

SearchVSplit est un algorithme qui va parcourir l'arbre à la recherche du nœud (dit VSplit) dont le sous-arbre gauche (respectivement droit) contiendra la borne inférieure (respectivement supérieure) en y. Les nœuds parcourus par l'algorithme sont vérifiés et SearchVSplit, une fois arrivé au nœud VSplit, lance bottomBoundSearch (respectivement topBoundSearch) sur le sous-arbre gauche (respectivement droit).

```
Algorithme searchVSplit(node, bounds, isVertical)
Require: Noeud de recherche courant (node). Bornes de la fenêtre (bounds), représentées par un tableau
         de quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical) vrai
         si l'arbre de racine node contient des segments verticaux, faux (donc horizontaux) sinon.
Ensure: / (Parcourt l'arbre de racine node et alimente la liste des segments à reporter).

    checkSegment(node.segment, bounds, isVertical)
    if node.leftChild existe then
        if node.rightChild existe then
            if node.median < bounds[0] ou (node.median = bounds[0] et node.areEdgesEqual = false) then
                searchVSplit(node.rightChild, bounds, isVertical)
            else if node.median > bounds[1] ou (node.median = bounds[1] et node.areEdgesEqual = false) then
                searchVSplit(node.leftChild, bounds, isVertical)
            else
                topBoundSearch(node.rightChild, bounds, isVertical)
                bottomBoundSearch(node.leftChild, bounds, isVertical)
            end if
        else
            checkSegment(node.leftChild.segment, bounds, isVertical)
        end if
    end if
```

5. topBoundSearch & bottomBoundSearch:

topBoundSearch est un algorithme qui consiste à chercher la borne supérieure dans un sous-arbre dont tous les segments sont supérieurs à la borne inférieure. Si la médiane du nœud courant est inférieure à la borne supérieure, cela signifie que la borne supérieure se situe dans le sous-arbre droit et que tous les nœuds du sous-arbre gauche sont entre les bornes basse et haute. Dans ce cas, l'algorithme searchInSubtree est lancé sur le sous-arbre gauche et topBoundSearch est lancé sur le sous-arbre droit, à la recherche de la borne supérieure. Si par contre la médiane est supérieure à la borne supérieure, cette dernière se trouve dans le sous-arbre gauche et tous les nœuds du sous-arbre droit contiennent des segments hors-bornes (supérieurs à la borne supérieure), topBoundSearch est alors lancé sur le sous-arbre gauche, à la recherche de la borne supérieure. Le nœud courant ayant été choisi selon une composante x, la médiane ne nous donne aucune information sur le segment stocké dans ce nœud, et son appartenance à la fenêtre est alors vérifiée selon x et y. L'algorithme bottomBoundSearch fonctionne de la même manière mais s'occupe de la borne inférieure.

```
Algorithme topBoundSearch(node, bounds, isVertical)
Require: Noeud de recherche courant (node). Bornes de la fenêtre (bounds), représentées par un tableau
         de quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical) vrai
         si l'arbre de racine node contient des segments verticaux, faux (donc horizontaux) sinon.
Ensure: / (Parcourt l'arbre de racine node et alimente la liste des segments à reporter).

    checkSegment(node.segment, bounds, isVertical)
    if node.leftChild existe then
        if node.rightChild existe then
            if node.median > bounds[1] ou (node.median = bounds[1] et node.areEdgesEqual = false) then
                topBoundSearch(node.leftChild, bounds, isVertical)
            else
                topBoundSearch(node.rightChild, bounds, isVertical)
                searchInSubtree(node.leftChild, bounds, isVertical)
            end if
        else
            checkSegment(node.leftChild.segment, bounds, isVertical)
        end if
    end if
```

```
Algorithme bottomBoundSearch(node, bounds, isVertical)
Require: Noeud de recherche courant (node). Bornes de la fenêtre (bounds), représentées par un tableau
         de quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical) vrai
         si l'arbre de racine node contient des segments verticaux, faux (donc horizontaux) sinon.
Ensure: / (Parcourt l'arbre de racine node et alimente la liste des segments à reporter).

    checkSegment(node.segment, bounds, isVertical)
    if node.leftChild existe then
        if node.rightChild existe then
            if node.median < bounds[0] ou (node.median = bounds[0] et node.areEdgesEqual = false) then
                bottomBoundSearch(node.rightChild, bounds, isVertical)
            else
                bottomBoundSearch(node.leftChild, bounds, isVertical)
                searchInSubtree(node.rightChild, bounds, isVertical)
            end if
        else
            checkSegment(node.leftChild.segment, bounds, isVertical)
        end if
    end if
```

6. searchInSubtree:

En ce qui concerne searchInSubtree, nous savons que l'arbre donné en paramètre ne contient que des segments qui respectent les bornes inférieure et supérieure. Ces segments sont tous vérifiés, mais seulement par rapport aux bornes gauche et droite.

```
Algorithme searchInSubtree(node, bounds, isVertical)
Require: Noeud de recherche courant (node). Bornes de la fenêtre (bounds), représentées par un tableau
         de quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical) vrai
         si l'arbre de racine node contient des segments verticaux, faux (donc horizontaux) sinon.
Ensure: / (Parcourt l'arbre de racine node et alimente la liste des segments à reporter).

    checkSegmentXAxis(node.segment, bounds, isVertical)
    if node.leftChild existe then
        if node.rightChild existe then
            searchInSubtree(node.rightChild, bounds, isVertical)
        end if
        searchInSubtree(node.leftChild, bounds, isVertical)
    end if
```

7. checkSegment:

L'algorithme checkSegment vérifie si le segment respecte les bornes supérieure et inférieure. Si c'est le cas, il appelle l'algorithme checkSegmentXAxis et lui passe le segment approuvé en paramètre.

```
Algorithme checkSegment(segment, bounds, isVertical)
Require: Segment (segment) à analyser. Bornes de la fenêtre (bounds), représentées par un tableau
         de quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical)
         vrai si le segment est vertical, faux (donc horizontal) sinon.
Ensure: / (Reporte le segment si il appartient à la fenêtre).

    if segment[1] ⩾ bounds[0] et segment[1] ⩽ bounds[1] then
        checkSegmentXAxis(segment, bounds, isVertical)
    end if
```

8. checkSegmentXAxis

L'algorithme checkSegmentXAxis vérifie si le segment respecte les bornes gauche et droite. Si c'est le cas, le segment est reporté. Pour les segments verticaux, le booléen isVertical permet de les identifier pour qu'ils puissent être modifiés de nouveau avant d'être reportés. Ils avaient été transformés en segments "horizontaux" avant même la création des arbres et reprennent enfin leur caractéristique verticale.

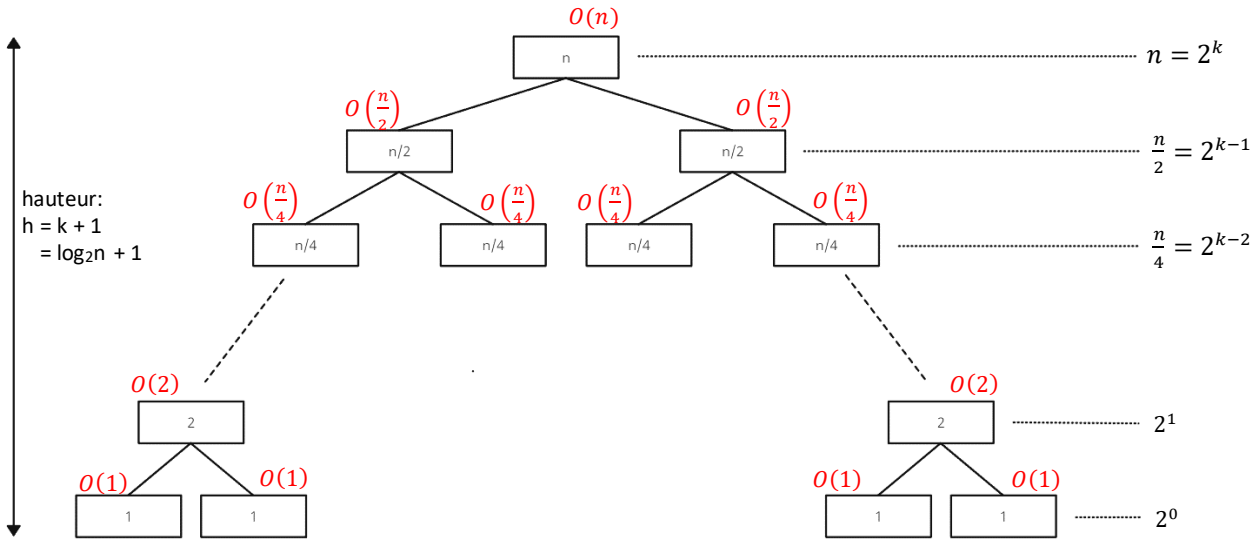
```
Algorithme checkSegmentXAxis(segment, bounds, isVertical)
Require: Segment (segment) à analyser. Bornes de la fenêtre (bounds), représentées par un tableau
         de quatre nombres à virgule (bornes basse, haute, gauche et droite). Booléen (isVertical)
         vrai si le segment est vertical, faux (donc horizontal) sinon.
Ensure: / (Reporte le segment si il appartient à la fenêtre).

    if segment[0] ⩽ bounds[3] et segment[2] ⩾ bounds[2] then
        if isVertical = true then
            ajoute [segment[1], segment[0], segment[3], segment[2]] aux segments à reporter
        else
            ajoute le segment aux segments à reporter
        end if
    end if
```

Complexité

1. <u>createTree:</u>
   Toutes les instructions dans createTree ont une complexité constante excepté l'appel à getUsefulNodes. Il y a n (nombre de segments) appels à createTree, donc la complexité de l'algorithme est égale à n fois la complexité de l'algorithme getUsefulNodes. Etant donné que l'algorithme getUsefulNodes a une complexité logarithmique, on sait que la complexité de createTree est en O(n $\log_2$n). A noter que n correspond au nombre de nœuds dans l'arbre.

2. <u>getUsefulNodes :</u>
   Toutes les instructions dans getUsefulNodes ont une complexité constante. Cependant, la présence d'une boucle for va dicter la complexité de l'algorithme. La complexité de la boucle for est déterminée par la longueur de la sous-liste à traiter. Chaque appel à createTree après le premier va être effectué sur une sous-liste deux fois plus petite que celle de l'appel précédent. On peut donc dire que la complexité de l'algorithme getUsefulNodes est logarithmique.



Au total, si on additionne les différents niveaux, on a:

$$O(n) + 2\,O\left(\frac{n}{2}\right) + 4\,O\left(\frac{n}{4}\right) + 8\,O\left(\frac{n}{8}\right) + \cdots + \frac{n}{2}\,O(2) + n\,O(1)$$

$$= O(n) + O(n) + O(n) + O(n) + \cdots + O(n) + O(n)$$

$$= h * O(n) = (log_2 n + 1) * O(n) = O(n\,log_2 n)$$

3. <u>windowing:</u>

   La complexité du windowing dépend du nombre de nœuds visités. Dans le pire (respectivement meilleur) des cas, l'algorithme aura une complexité en O(n) (respectivement O($\log_2$n), la hauteur de l'arbre). Plus vSplit est haut (respectivement bas) dans l'arbre, plus on se rapproche d'une complexité en O(n) (respectivement O($\log_2$n)). A noter que n correspond au nombre de nœuds dans l'arbre.

Mode d'emploi et fonctionnalités

1. Exécution du programme:

   Nous avons décidé d'utiliser Gradle pour construire et exécuter notre projet. Il est donc nécessaire d'utiliser la commande 'gradlew run' dans le dossier "Projet-SDD-II-Code" pour lancer le programme.

2. Fonctionnalités de base

   Pour ouvrir et lire un fichier de segments, il est nécessaire de cliquer sur 'File' puis sur 'Open', une fenêtre permettant la sélection de fichier s'ouvrira alors.

   Après avoir ouvert le fichier, il faut utiliser les différentes commandes situées sur la gauche de la fenêtre pour permettre les manipulations de windowing.

   Il existe deux manières différentes pour définir les bornes de la fenêtre de windowing, des sliders ou des zones de texte (la fenêtre de windowing est celle en rouge, tandis que la fenêtre noire correspond aux bornes maximales définies par le fichier de segments).

   Une fois la fenêtre de windowing choisie, il faut appuyer sur le bouton 'Compute' pour calculer et afficher les segments reportés par l'algorithme de windowing.

3. Fonctionnalités supplémentaires

   Les boutons '-' et '+' à droite de "Line size" permettent respectivement de rétrécir et de grossir l'épaisseur du trait des segments.

   Le bouton 'Focus' permet d'afficher la fenêtre de windowing en plein écran. Lorsque le focus est activé, un bouton '-' à gauche du bouton 'Focus' et un bouton '+' à droite de celui-ci permettent respectivement de dézoomer et zoomer sur la fenêtre de windowing.

Conclusion

Ce projet nous a permis de nous confronter à un challenge reposant sur les concepts théoriques vus en cours, notamment la difficulté liée aux contraintes de complexité. Nous avons également appris à utiliser au mieux Github en réduisant la taille des commits et en augmentant la fréquence de ceux-ci.

En ce qui concerne les difficultés rencontrées, la lenteur de l'affichage des lignes et la complexité de la création d'un arbre de recherche à priorité se sont avérées être les plus gros obstacles lors de notre implémentation.

Finalement, les logs ajoutés au code ont été d'une grande utilité pour détecter les éventuelles erreurs. Ils nous ont aussi permis de vérifier que notre implémentation était cohérente avec les algorithmes étudiés en théorie.