RAPPORT
PROJET MONDRIAN

2022-2023

ANDY TORRES
STEPHANE RAMAHEFARINAIVO
585k

```
$ cd ./ProjetMondrian
$ cd ./src
$ javac Image.java
$ ./Image.java
```

Enjeux :

Génération d'un tableau de même nature que le peintre Piet Mondrian.

Pour re-créer un tableau similaire à Mondrian nous allons générer un arbre binaire contenant dans chaque nœud les informations nécessaires(coordonnées, couleurs…) pour ensuite avec une fonction ToImage qui créera l'image.
Afin de parvenir à cette fin, implémenter la procédure generateRandomTree() est nécessaire, nous avons construit un 2d-arbre comme indiqué dans le sujet pour permettre la découpe aléatoire de l'arbre selon l'axe des X ou l'axe des Y. La structure d'un AVL fut notre choix a implémenter pour effectuer ce generateRandomTree, chaque nœud généré ce fait placer directement dans l'AVL qui va lui en fonction de la balance total rééquilibrer l'arbre. Cette structure nous permet de récupérer le nœud de poids le plus fort dans l'arbre en un temps logarithmique puisqu'il suffit de trouver le nœud le plus à droite dans l'AVL.
Chaque noeud généré doit avoir se faire en fonction d'une division soit sur l'axe des X soit des Y, la fonction chooseDivision() qui renvoie un réel variant entre une plage délimité par une variable ProportionCoupe qui est une limite à ne pas rentrer pour ne pas a avoir des rectangles trop collés. La variation s'effectue à l'aide du générateur de réels aléatoires : rand.nextDouble(). Ainsi, le réel div obtenu nous permet de calculer plus facilement les coordonnées des points générés par generate Random Tree(). En effet, nous générons les coordonnées des nouveaux nœuds en prenant pour x et pour y les coordonnées du coin supérieur gauche du rectangle que l'on souhaite créer, un choix arbitraire mais logique, cela nous donne beaucoup plus de facilité sur les manipulations de coordonnées.
Cette suite de méthode et de travail pour enfin créer l'image, ToImage va de se fait prendre chaque nœud de l'arbre généré et non l'AVL pour prendre les coordonnées et colorier la zone définie, on a aussi pris le partie de lors du dessinage d'une feuille on dessine aussi les rectangles gris qui séparents les feuilles entre eux en même temps.

Pseudo-code des fonctions principales

Parcours les nœuds à droite de l'AVL pour chercher la feuille ayant le plus gros poids et la retourne.
Paramêtre : Avl
NodeAvl : temp

```
Fonction chooseLeaf(AVL : Avl) → NodeAvl :
        NodeAvl temp ← Avl.searchMax(Avl.getRoot())
        Retourner temp
Fin
```

Complexité temporelle : log(n)

Choisit dans l'axe de division du Node en fonction d'une probabilité donnant la possibilité de pouvoir diviser le même axe plusieurs fois, retourne réel qui est la valeur de la division.
Parametres: Node
réel : prob, div

```
fonction chooseDivision(Node : r) → réel
        prob ← rand.nextDouble()
        div ← 0
        Si(r.getCol() = Color.White et r.getLeft() = NULL et r.getRight() = NULL alors
                div ← (rand.nextDouble() * (r.getH()))
                r.setChoosenDiv_X(faux)
                Tant que((div < (r.getH() * proportionCoupe)) ou (div > (r.getH() *   (1-proportionCoupe))))
faire
                        div ← (rand.nextDouble() * r.getH())
                Fin Tant que
        Sinon
                Si(prob <= r.getW() / (r.getH()+r.getW())) alors
                        div ← (rand.nextDouble() * r.getW())
                        r.setChoosenDiv_X(vrai)
                        Tant que(div < r.getW() * proportionCoupe ou div > r.getW() *
(1-proportionCoupe)) faire
                                div ← (rand.nextDouble() * r.getW())
                        Fin Tant que
                Sinon
                        r.setChoosenDIv_X(faux)
```

```
                        Tant que(div < r.getH() * proportionCoupe ou div > r.getH() *
(1-proportionCoupe)) faire
                                div ← (rand.nextDouble() * (r.getH()))
                        Fin Tant que
                Fin Si
        Fin Si
        retourner div
Fin
```

Complexité temporelle: O(|r.getH() * proportionCoupe - r.getH() * (1-proportion Coupe)|)

En fonction d'une probabilité, va choisir soit la même couleur que son parent ou une couleur au hasard dans un tableau.

```
Parametre: Node
fonction chooseColor(Node: r) → Color
        Si(rand.nextDouble() <= getMemeCouleurProb()) alors
                retourner r.getCol()
        Sinon
                i ← (entier) (rand.nextDouble() * (5))
                retourner tab[i]
        Fin Si
Fin
```

Complexité temporelle: O(1)

Génération d'un meilleur arbre, choisit le noeud ayant le poids le plus fort et la divise selon un axe donnée par chooseDivision(), puis ajouts des noeuds dans l'AVL

```
Parametres: Node, AVL
NodeAvl : al
Node : l
réel : div
```

```
procédure generateRandomTree(Node: r, AVL: tree)
        al ← chooseLeaf(tree)
        l ← al.getN()
        tree.setRoot(tree.deleteNode(l,tree.getRoot()))
        Si(nbOfLeaves(r) >= nbFeuille ou minDimensionCoupe > l.getH() * l.getW() alors
```

```
            retourner // ce qui a pour effet de stopper le programme car les conditions sont
            fausses
        Sinon
            div ← chooseDivision(l)
            Si(l.isChoosenDiv_X()) alors
                l.setLeft(Nouveau Node(div, l.getH(), l.getX(), l.getY(), chooseColor(l)))
                tree.setRoot(tree.insertNode(l.getLeft(), tree.getRoot()))
                l.setRight(Nouveau Node(l.getW()-div, l.getH(), l.getX()+div, l.getY(),
                chooseColor(l)))
                tree.setRoot(tree.insertNode(l.getRight(), tree.getRoot()))
            Sinon
                l.setLeft(Nouveau Node(l.getW(), div, l.getX(), l.getY(), chooseColor(l)))
                tree.setRoot(tree.insertNode(l.getLeft(), tree.getRoot()))
                l.setRight(Nouveau Node(l.getW(), l.getH()-div, l.getX(), l.getY()+div,
                chooseColor(l)))
                tree.setRoot(tree.insertNode(l.getRight(), tree.getRoot()))
            Fin Si
            Si(nbOfLeaves(r) > nbFeuille) alors
                l.setRight(NULL)
            Sinon
                generateRandomTree(r, tree)
            Fin Si
        Fin Si
Fin
```

Complexité temporelle: O(|r.getH() * proportionCoupe - r.getH() * (1-proportion Coupe)| * nbFeuille) car generateRandomTree() utilise la fonction chooseDivision() qui est la fonction avec le coût le plus élevé puis on appelle récursivement la procédure tant que le nombre de feuilles n'est pas atteint.

Créer l'image du tableau à partir de l'arbre généré
Paramètres: Node, Tree

```
procédure toImage(Node: r, Tree: a)
        Si(r.getLeft() = NULL et r.getRight() = NULL) alors
            setRectangle((entier)r.getX(), (entier)(r.getX()+r.getW()), (entier)r.getY(),
(entier)(r.getY()+r.getH()), r.getCol())

            // Au-dessus
            setRectangle((entier)r.getX(), (entier)(r.getX()+r.getW()), (entier)r.getY(),
(entier)(r.getY()+(a.getlargeurLigne()/2)), Color.GRAY)
            // A gauche
```

```
                setRectangle((entier)r.getX(), (entier)(r.getX()+((a.getlargeurLigne()/2))),
(entier)r.getY(), (entier)(r.getY()+r.getH()), Color.GRAY)
                // A droite
                setRectangle((entier)(r.getX()+r.getW())-(a.getlargeurLigne()/2)),
(entier)r.getX()+r.getW()), (entier)r.getY(), (entier)(r.getY()+r.getH()), Color.GRAY)
                // Au-dessous
                setRectangle((entier)(r.getX())), (entier)r.getX()+r.getW()),
(entier)r.getY()+r.getH()-(a.getlargeurLigne()/2)), (entier)(r.getY()+r.getH()), Color.GRAY)
        Sinon
                toImage(r.getLeft(), a)
                toImage(r.getRight(),a)
        Fin Si
Fin
```

Complexité temporelle: O(nombre de nœuds) car toImage() coûte O(1) et on l'appelle autant de fois qu'il y a de nœuds dans l'arbre.

BetterTree

L'enjeux de cette nouvelle version de génération est de pouvoir corriger un/plusieurs problèmes rencontrés sur la génération de l'énoncé. On a pu constater avec plusieurs tests que la première version avait un problème sur les paramètres, en effet l'utilisateur peut spécifier la dimension minimale d'un rectangle tout en donnant un nombre de rectangle à dessiner. Posons la question, si la dimension minimum est trop grande par rapport au nombre de rectangles demandés, que se passe-t-il ? La réponse est qu'on perd obligatoirement des feuilles dans le processus de génération.
Cette nouvelle version prend en compte ce problème et va lui même calculer une dimension minimale pour chaque rectangle en fonction du nombre de feuilles demandé par l'utilisateur, cela donne ainsi un nombre de paramètres moindre et un tableau avec la totalité des feuilles soumises.
Les changements se font principalement sur les méthodes autours de BetterRandomTree : chooseDivision(), Paramètres d'entrées…

```
Parametres: Node, AVL
NodeAvl : al
Node : l
réel : div
procédure generateBetterTree(Node: r, AVL: tree)
        al ← chooseLeaf(tree)
        l ← al.getN()
        tree.setRoot(tree.deleteNode(l,tree.getRoot()))
        Si(nbOfLeaves(node) >= nbFeuille alors
                retourner // ce qui a pour effet de stopper le programme car les conditions sont
fausses
```

```
        Sinon
                div ← chooseDivision(l)
                Si(l.isChoosenDiv_X()) alors
                        l.setLeft(Nouveau Node(div, l.getH(), l.getX(), l.getY(), chooseColor(l)))
                        tree.setRoot(tree.insertNode(l.getLeft(), tree.getRoot()))
                        l.setRight(Nouveau Node(l.getW()-div, l.getH(), l.getX()+div, l.getY(),
chooseColor(l)))
                        tree.setRoot(tree.insertNode(l.getRight(), tree.getRoot()))
                Sinon
                        l.setLeft(Nouveau Node(l.getW(), div, l.getX(), l.getY(), chooseColor(l)))
                        tree.setRoot(tree.insertNode(l.getLeft(), tree.getRoot()))
                        l.setRight(Nouveau Node(l.getW(), l.getH()-div, l.getX(), l.getY()+div,
chooseColor(l)))
                        tree.setRoot(tree.insertNode(l.getRight(), tree.getRoot()))
                Fin Si
                Si(nbOfLeaves(r) > nbFeuille) alors
                        l.setRight(NULL)
                Sinon
                        generateBetterTree(r, tree)
                Fin Si
        Fin Si
Fin
```

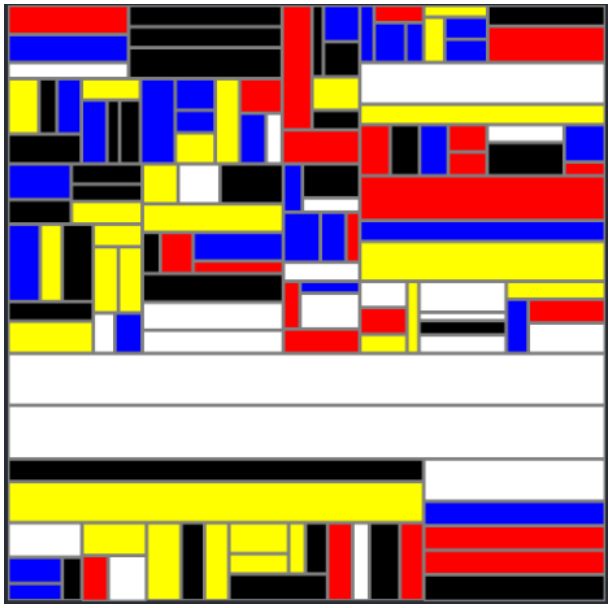
Complexité temporelle: Il s'agit d'une procédure analogue à generateRandomTree() donc la complexité est du même ordre de grandeur.

Résultats

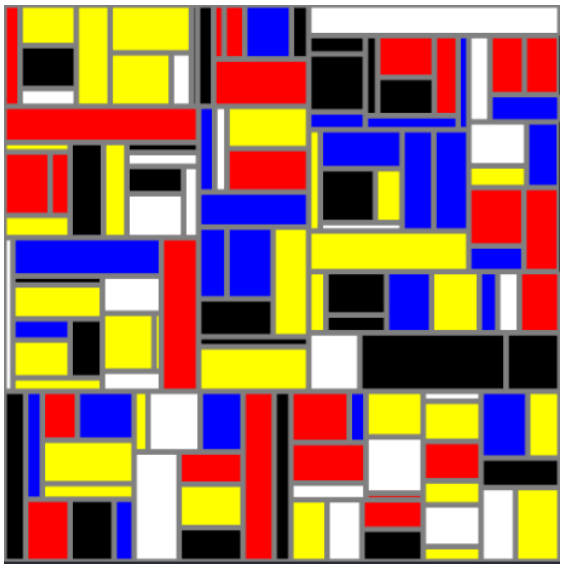
Dimension : 1000x100 | **Seed** : 697

Tree (Version par défault): (nbFeuilles, proportionCoupe, minDimensionCoupe, memeCouleurProb, largeurLigne)

130, 0.2, 120, 0.3, 6

 Nombre de feuilles visibles : 130

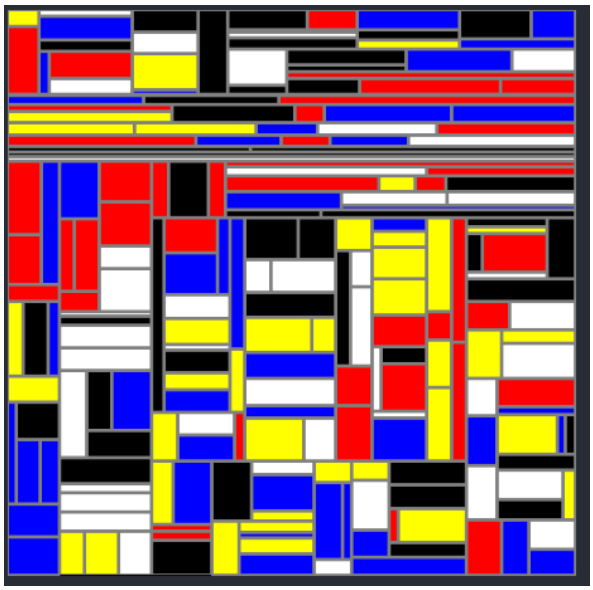
BetterTree: (NbFeuille, memeProbCouleur)
130 , 0.3

 Nombre de feuilles visibles : 130

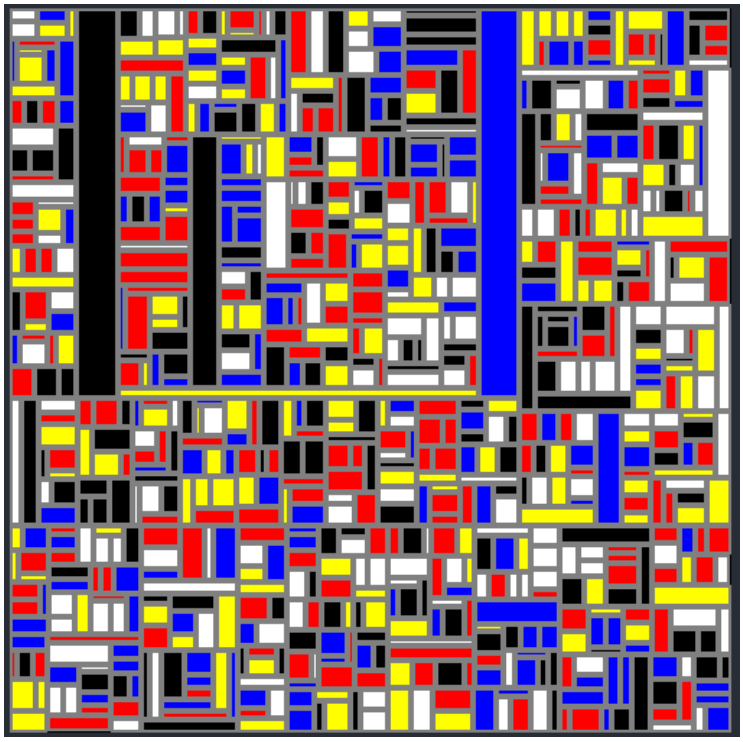
On note une différence, plus homogène sur les tailles de rectangles dans le better

Dimension : 500x500| **Seed** : 987

Tree (Version par défault): (nbFeuilles, proportionCoupe, minDimensionCoupe, memeCouleurProb, largeurLigne)
800, 0.1, 2000, 0.3, 3


Nombre de feuilles visibles : 217

BetterTree: (NbFeuille, memeProbCouleur)
800, 0.3


Nombre de feuilles visibles : 800

Conclusion

Dans sa globalité du projet nous avons rendu un programme fonctionnel et plus ou moins dans la vision qu'on avait, un projet non difficile en soit mais qui prenait du temps sur certaines choses comme par exemple: l'implémentation de l'AVL où dans notre cas on devait refaire la structure de nos méthodes étant donné qu'on avait commencé le projet sans utiliser d'AVL. Nous sommes conscients que notre betterRandomTree n'est pas aussi poussée que ce qu'on pouvait attendre et nous avons évidemment quelques regrets qu'on peut citer, on a un peu manqué de temps sur la fin pour faire un vrai changement majeur sur la stratégie de base, dans les changements qu'on voulait apporter : l'implémentation d'un QuadTree pour déjà une réduction de temps de création, et une homogénéisation des rectangles pour ne pas se retrouver dans le cas (cf. Resultats) de gros rectangles dans la peinture et donner un autre visuelle géométrique des tableaux de Mondrian.