Génération et opération sur les arbres binaires

Groupes 1 et 2

Mercredi 9 juin 2004

Table des matières

I Interface Homme Machine **4**

1 Les structures graphiques **4**

2 Module création **5**

3 Module éxécution **8**
3.1 Diagramme de classe du module 9
3.2 Partie applicative 9
3.3 Exploiter les résultats des algorithmes 10
3.3.1 Panneau d'affichage des arbres 10
3.3.2 Panneau d'affichage de courbe 12

4 Module paramétrage : personnalisation des arbres **16**
4.1 Structure du package module paramétrage 16
4.2 Panel coloration, création 16
4.3 Schéma de la classe paramétrage 17
4.4 Panel thread 18
4.5 Gestionnaire de présentation 18

II Algorithmes du logiciel **19**

5 Structures de données et architecture **19**
5.1 Structure de données 19
5.2 Passage des résultats 19
5.3 Diagramme UML général 20
5.4 Entrées/Sorties 20

6 Génération des arbres **21**
6.1 Génération aléatoire selon une hauteur donnée 21
6.2 Génération aléatoire selon un nombre de noeud donné 21
6.3 Génération de Tous les arbres selon un nombre de noeuds donné 22
6.4 Génération de tous les arbres binaires pour une hauteur donnée sans symétrie 23

7 Ordonnancements **24**
7.1 Ordonnancements optimum local 24
7.2 Ordonnancement Aleatoire et Globaux 25

Introduction

Le but de ce projet était de développer un logiciel complet permettant la création et la manipulation d'arbre binaires parfaits. Ce travail devait être effectué en deux groupes, le premier s'occupant de l'interface humain machine et le deuxiéme groupe s'occupant du travail algorithmique du logiciel.
Plus précisément, l'interface homme machine doit permettre de :

- lire et écrire dans un fichier au format LEDA
- manipuler un arbre binaire parfait et ses caractéristiques
- paramétrer l'affichage d'un arbre

L'interface graphique permet donc la manipulation proprement dite des arbres tandis que la génération et le traitement repose sur la partie algorithmique.

Grâce à cette partie, l'utilisateur doit être capable de :

- générer aléatoirement tous les arbres pour une hauteur ou une taille fixée
- générer tous les arbres pour une hauteur ou une taille fixée en évitant les symétrie
- calculer les ordonnancements de consommation minimale en ressources
- calculer tous les ordonnancements valides pour un arbre

Les résultats calculés par la partie algorithmique sont ensuite exploités par l'interface graphique pour que l'utilisateur puisse les manipuler.

Première partie
Interface Homme Machine

1 Les structures graphiques

Les classes du package structuresgraphique permettent de représenter graphiquement un arbre. Dans ce qui suit, la structure de chaque classe sera détaillée. Seul les méthodes et champs importants figureront. Un arbre est défini récursivement, c'est-à-dire qu'un arbre comporte un sommet, un arbre gauche et un arbre droit.

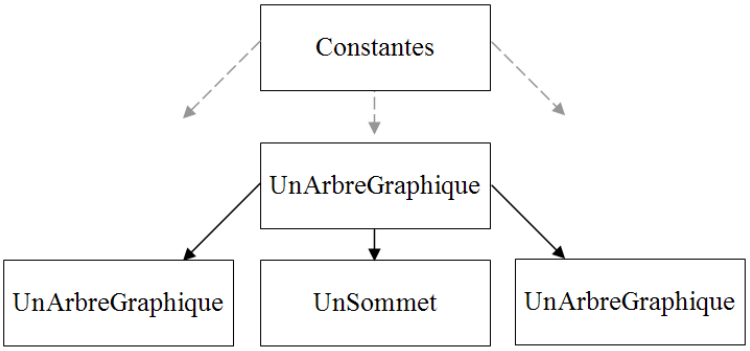
UnArbreGraphique (Structure de données permettant de représenter graphiquement un arbre)	
UnSommet sonSommet	**le sommet racine de cet arbre**
UnArbreGraphique sonFilsGauche	**l'arbre gauche**
UnArbreGraphique sonFilsDroit	**l'arbre droit**
ListeStockage sesDonneesAreteGauche	**les données attachées à l'arête gauche**
ListeStockage sesDonneesAreteDroite	**les données attachées à l'arête droite**
UnArbreGraphique getSonFilsGauche()	**pour récupérer l'arbre gauche**
UnArbreGraphique getSonFilsDroit()	**pour récupérer l'arbre droit**
void addFils()	**pour ajouter les 2 arbres fils**
void supprimeFils()	**pour supprimer les 2 fils**
UnSommet getSonSommet()	**retourne le sommet racine de l'arbre**
UnSommet trouveSommet(int telNb)	**trouve un sommet selon son « id »**
UnArbreGraphique trouveSommetArbre(UnSommet telSommet)	**trouve un sommet et retourne l'arbre dont il est racine**
void supprimerSommet(UnSommet telSommet)	**supprime un sommet et tous ses fils**
dessineArbre(Graphics2D g2d)	**permet de dessiner l'arbre**
void maj_x_y()	**met à jour les positions de chaque sommet**
boolean estVide()	**test si l'arbre est vide**
UnArbreGraphique copyArbre()	**effectue et retourne une copie en profondeur**

UnSommet (Structure permettant de représenter un sommet)	
int sonId	**numéro identifiant le sommet**
ListeStockage sesDonnees	**les données attachées au sommet**
void setSaCouleur(Color telleCouleur)	**pour changer la couleur du sommet**
void dessineSommet(Graphics2D g2d)	**permet de dessiner le sommet**
UnSommet copySommet()	**retourne une copie du sommet**
void setStrahler(int telStrahler)	**change le nombre de Strahler du sommet**

ListeStockage (Liste de données)	
Vector sonContenu	**liste des données**
addStockage(Stockage telS)	**ajoute un nouvel élément à stocker**
Stockage getStockage(int telRang)	**retourne l'élément voulu**
int size()	**retourne la taille de la liste**
boolean removeStockage(Stockage telS)	**supprime l'élement voulu**
removeStockage(int telRang)	**supprime l'élement voulu**
Object clone()	**retourne une copie de la liste**

Stockage (Elément de la liste de données)	
String sonNom	**nom de l'élément**
String sonContenu	**contenu de l'élément**
String getNom()	**Méthodes permettant de**
String getContenu()	**récupérer ou modifier**
void setNom(String telNom)	**les deux champs**
void setContenu(String telContenu)	**de la classe**

Un niveau de l'arbre est donc composé de la façon suivante :



Cette structure sera évidemment utilisée pour tout affichage graphique d'un arbre, aussi bien dans la partie création que dans la partie exécution du programme. Pour passer cette structure avant tout optimisés pour le graphisme à la structure de la partie algorithme, des méthodes de conversion ont été implémentées.

Conversion.donneeToGraphic(ABP telArbreDonnees)
Conversion.graphicToDonnees(UnArbreGraphique telArbreGraphique)

2 Module création

Le module création a pour but de permettra à l'utilisateur de créer graphiquement des arbres ainsi que de les manipuler. Il pourra en effet travailler sur autant d'arbres qu'il le souhaite en même temps, en pouvant passer de l'un à l'autre aisément. Pour afficher les arbres, ce package se servira de la structure vue précédemment.

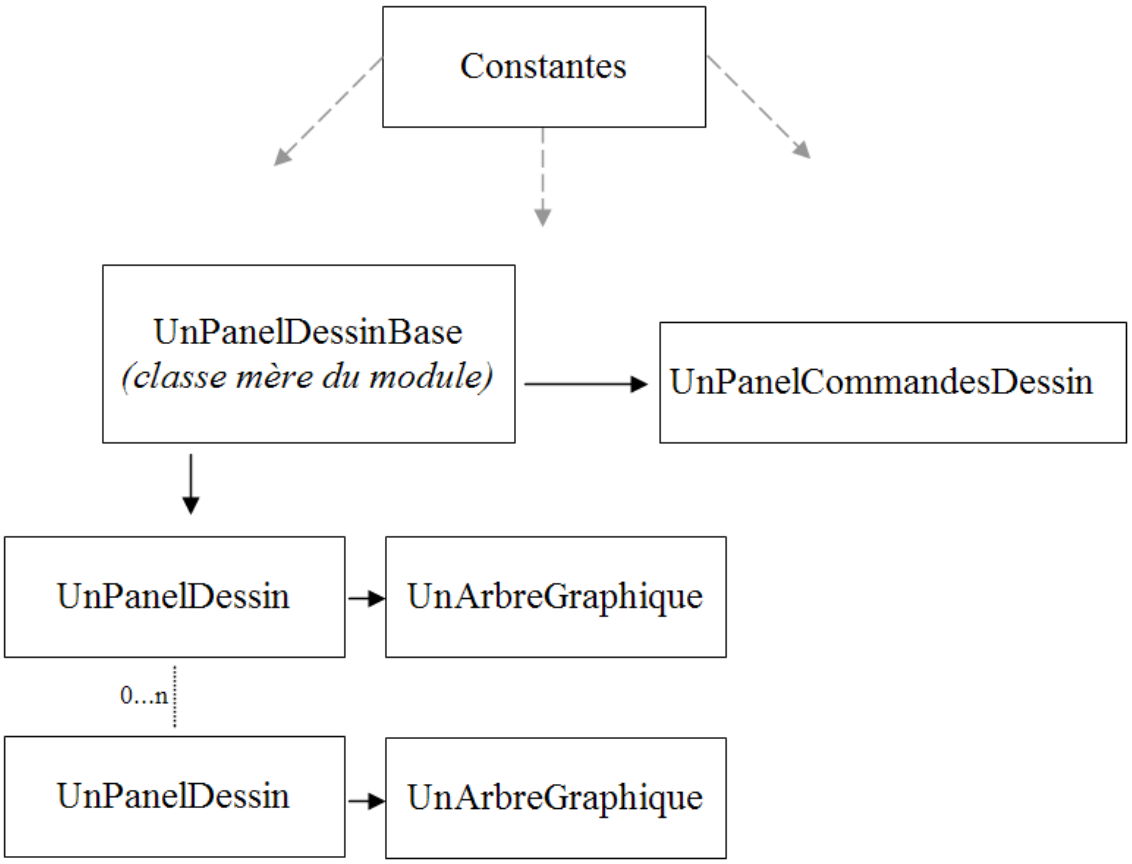
UnPanelDessinBase (la classe de base du package)	
CardLayout gestionnaireDesCartes	**le gestionnaire de repartition du panel cartes**
JPanel sonPanelCartes	**le panel contenant tous les panels d'arbres**
UnPanelCommandesDessin	**le panel avec les boutons**
void addNewPanel(UnArbreGraphique telArbre)	**ajoute un nouveau panel avec l'arbre donné**
void setAntiAlias(boolean telEtat)	**change l'état de l'anti-aliasing**
boolean getAntiAliasState()	**retourne l'état de l'anti-aliasing**
UnArbreGraphique getCurrentArbreGraphique()	**retourne l'arbre courant**
UnPanelDessin getCurrentPanelDessin()	**retourne le panel de dessin courant**
void majColorRacine(Color telleCouleur)	**met à jour la couleur par défaut des racines**
Color getColorRacine()	**retourne la couleur par défaut des racines**
void majColorFeuilles(Color telleCouleur)	**met à jour la couleur par défaut des feuilles**
Color getColorFeuilles()	**retourne la couleur par défaut des feuilles**
void majColorAutresSommets(Color telleCouleur)	**met à jour la couleur par défaut des autres sommets**
Color getColorAutresSommets()	**retourne la couleur par défaut des autres sommets**
void majColorColorAretes(Color telleCouleur)	**met à jour la couleur par défaut des arêtes**
Color getColorAretes()	**retourne la couleur par défaut des arêtes**
void majDiametreSommets(double telDiametre)	**met à jour le diamètre par défaut des sommets**
double getDiametreSommets()	**retourne le diamètre par défaut des sommets**
void sauvegarderCourant(File telFichier)	**sauvegarde l'arbre courant**
void chargerNouveau(File telFichier)	**charge un arbre et l'affiche**
void sauvegarderTout(File telDossier)	**sauvegarde tous les arbres ouverts**
void chargerTout(File telDossier)	**charge tous les arbres d'un dossier**
void fermerTout()	**ferme tous les arbres ouverts**
void genAleaH(int telleHauteur)	**lance**
void genAleaT(int telNbNoeud)	**les différents**
void genTousH(int telleHauteur)	**algorithmes**
void genTousT(int telNbNoeud)	**de génération**

UnPanelDessin (création graphique d'un arbre) hérite de UnPanelArbre	
void affichePopup(MouseEvent e)	**affiche le menu contextuel sur les sommets**
void dessinerFils(UnSommet telSommet)	**dessine les fils du sommet**
UnSommet detecteSommet(double telX, double telY)	**détecte un sommet par rapport à des coordonnées**
bougeSommet(double telX, double telY, UnSommet telSommetTrouve)	**déplace un sommet**

UnPanelArbre (représentation d'un arbre)	
UnArbreGraphique sonArbre	**l'arbre à afficher**
UnArbreGraphique getSonArbre()	**retourne l'arbre affiché**

UnPanelCommandesDessin (commande accessibles à l'utilisateur)	
void nouveauPanel()	**lance la création d'un nouveau panel destiné à afficher un arbre**
void activeBouton()	**active ou désactive les boutons**
void fermerPanel()	**ferme le panel courant**
void premierPanel()	**affiche le 1er panel**
void panelPrecedent()	**affiche le panel précédant le panel courant**
void panelSuivant()	**affiche le panel suivant le panel courant**
void dernierPanel()	**affiche le dernier panel**

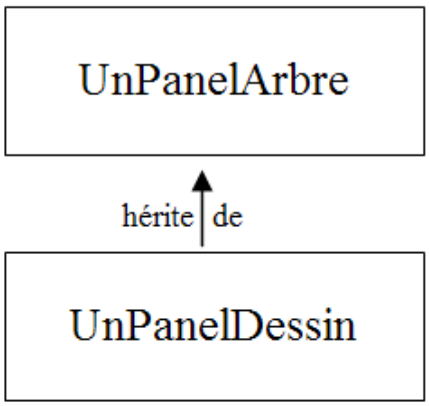
Diagramme de la classe du module :



La classe UnPanelDessinBase est la classe de base du package, qui est ajoutée à l'ihm :

- Elle contient la classe UnPanelCommandesDessin qui regroupe les boutons permettant à l'utilisateur d'intéragir avec les arbres (créer un nouvel arbre, ajouter un fils, passer à l'arbre suivant ect ...).
- Elle contient aussi autant d'objets de la classe UnPanelArbre que l'utilisateur le désire, cette classe permettant de représenter un arbre graphiquement.

En fait la classe UnPanelDessin vue précédemment hérite de la classe UnPanelArbre.

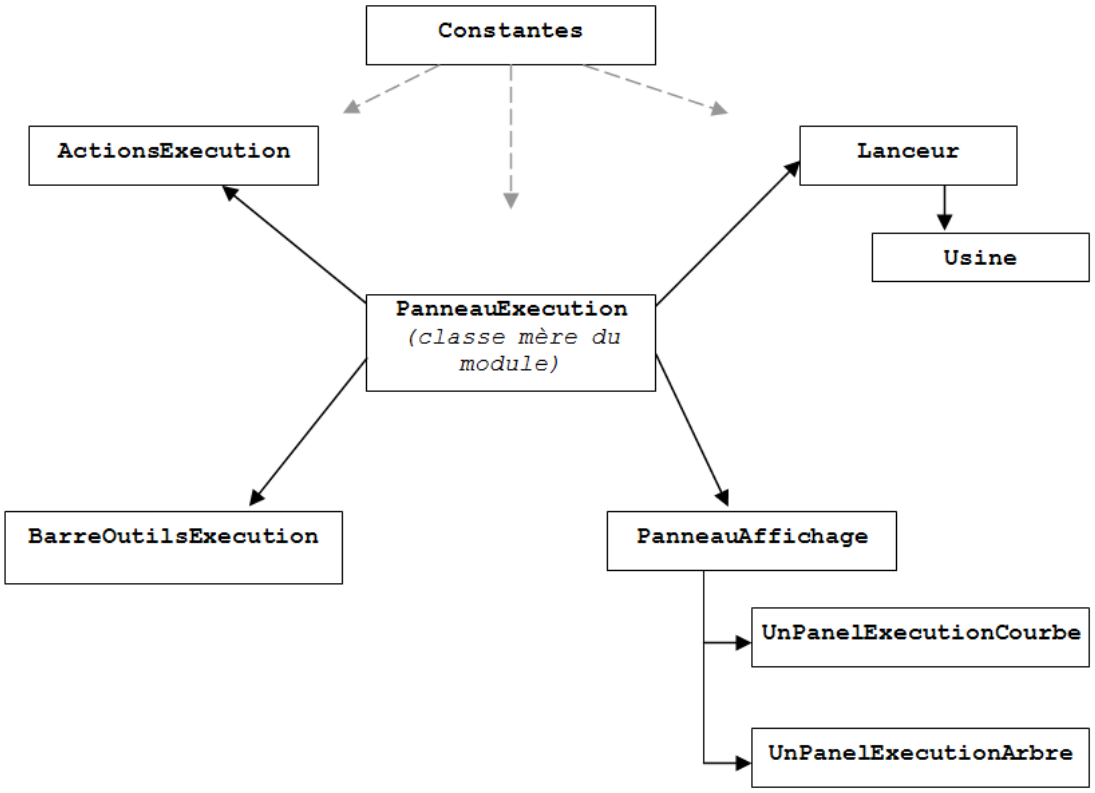


La classe UnPanelArbre est une classe générique qui se contente d'afficher un arbre. Cela s'avère très utile, car cela permet pour le module exécution d'hériter de cette classe pour pouvoir afficher un arbre sans avoir à aucun moment à en connaître l'implémentation.

3 Module éxécution

Le module exécution a pour but d'appliquer des algorithmes sur l'arbre courant de la partie création vue précédemment. La phase d'exécution s'opére en 2 étapes, lancement d'un algorithme sur un arbre, et affichage des résultats. Les moyens mis en oeuvre lors d'un appel sont explicités dans cette partie.
Avant tout, il convient de montrer la structure générale du module afin de mieux cerner les éléments impliqués pour ces 2 étapes.

3.1 Diagramme de classe du module



Le module est instancié par la classe PanneauExecution. Chacun des composants peut communiquer avec les autres en passant par cette classe puisqu'elle est prise comme paramétre de constructeur :

new BarreOutilsExecution(PanneauExecution PE) ;
new PanneauAffichage(PanneauExecution PE) ;

Les champs de la classe Constantes étant statique et destinée à stocker toutes les variables du module, elle peut être accédée par n'importe quelle classe du module.

3.2 Partie applicative

Cette partie décrit les procédés mis en place pour lancer un algorithme sur un arbre. Avant tout appel aux algorithmes, il faut récupérer l'arbre que l'utilisateur souhaite utiliser. Pour cela on a choisi d'afficher constamment dans la vue arbre, l'arbre courant du module " création " (ce qui implique qu'un algorithme peut étre lancé sur un seul arbre à la fois).

La vue arbre étant instancié par un UnPanelExecutionArbre qui est un " clone " de panel utilisé dans le mode " création " permet de récupérer l'objet arbre

proprement dit. A ce stage, l'arbre n'est pas exploitable par les algorithmes car il utilise sa propre structure de données spécifique à l'IHM et doit être converti en une instance ABP, structure de données commune au 2 groupes, les méthodes suivantes sont prévues à cet effet :

Conversion.donneeToGraphic(ABP telArbreDonnees) ;
Conversion.graphicToDonnees(UnArbreGraphique telArbreGraphique)

L'appel aux algorithmes se fait via l'utilisation d'un thread depuis la classe Lanceur qui a accés aux algorithmes. La classe Usine contient tout les algorithmes disponibles, et dans le cadre du module d'exécution, 4 seront utilisés :

Usine. genOrdoT (ABP arbre) ;
Usine. genOrdoA (ABP arbre) ;
Usine. genOrdoOptLocal (ABP arbre) ;
Usine. genOrdoOptGlob (ABP arbre) ;

Le thread a été choisi afin d'éviter d'alourdir l'IHM.

Comme convenu lors de la phase de conception, les algorithmes retournent des Ordonnancements (1 ou plusieurs selon les algorithmes).

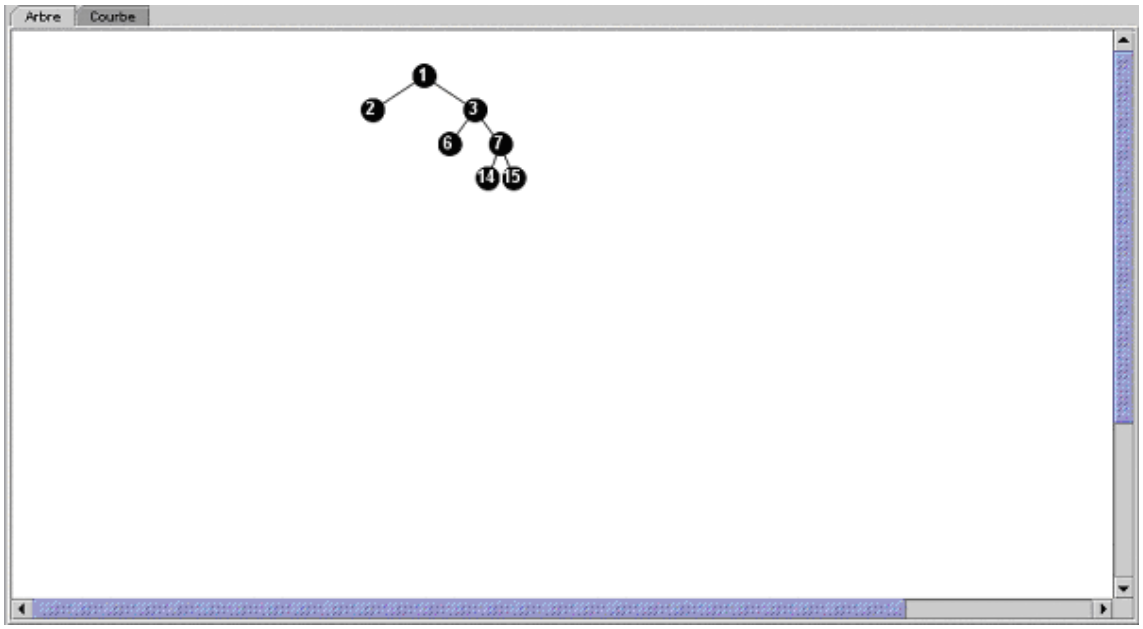
Pour de ce qui est de l'exécution " pas à pas ", il s'agit en fait d'une simulation de pas à pas, puisque en fait, on observe uniquement le résultat en pas à pas, l'algorithme tournant en temps réel. Pour cette visualisation, on a la possibilité de modifier la couleur du sommet de l'étape i, en utilisant les méthodes disponibles pour l'affichage des arbres.

3.3 Exploiter les résultats des algorithmes

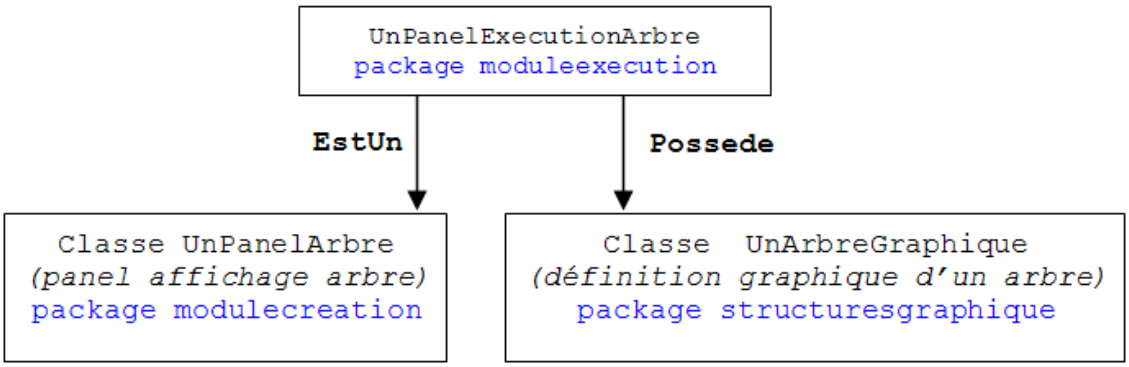
Dans cette partie, nous nous intéresserons à l'affichage de l'arbre courant et des résultats de l'algorithme appliqué.

3.3.1 Panneau d'affichage des arbres

Il est le panel principal de l'onglet exécution : dés que l'on accéde au module exécution, l'arbre courant du module création est affiché, et prét à recevoir les applications désirées.



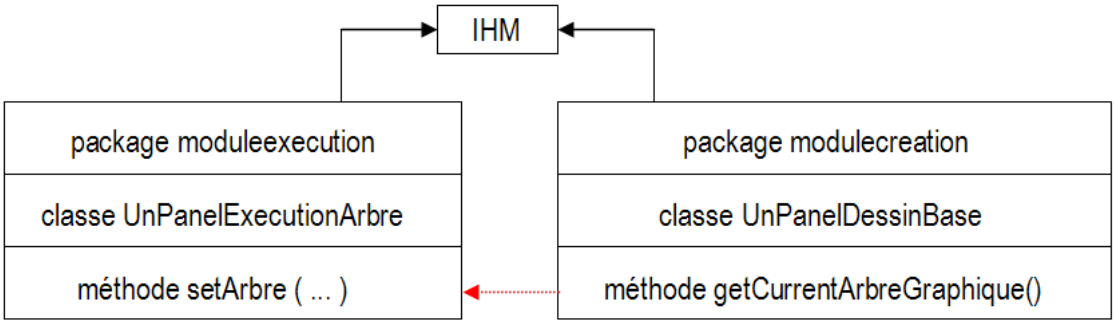
Structure d'implémentation



En effet, comme indiqué dans ce schéma, nous utilisons dans le module exécution l'affichage d'un arbre du module création ; cela nous permet une homogénéité quant au graphisme, et aussi un rendement au niveau du codage.

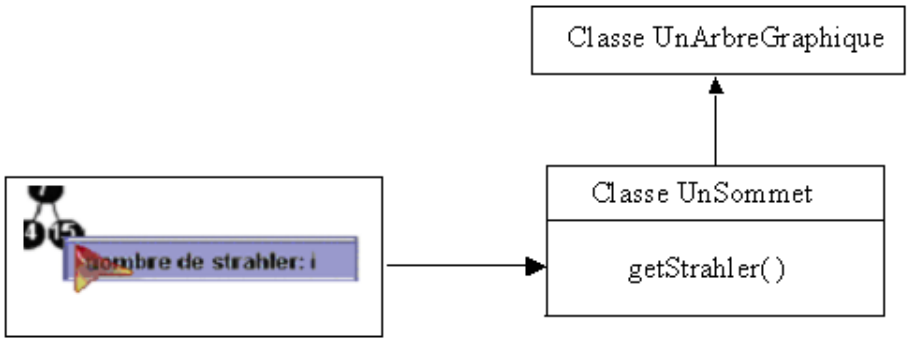
Remarquons juste que pour avoir, à tout moment de l'utilisation, l'arbre courant de la partie création, une mise a jour du panel est lancée chaque fois que l'on accéde a l'onglet exécution.

Schéma de la mise à jour



La définition graphique d'un arbre binaire permet entres autres d'attribuer une couleur pour chaque sommet, ce qui nous permettra de suivre sur ce panel l'ordre de parcours des sommets lors d'un ordonnancement (voir la partie action/exécution).

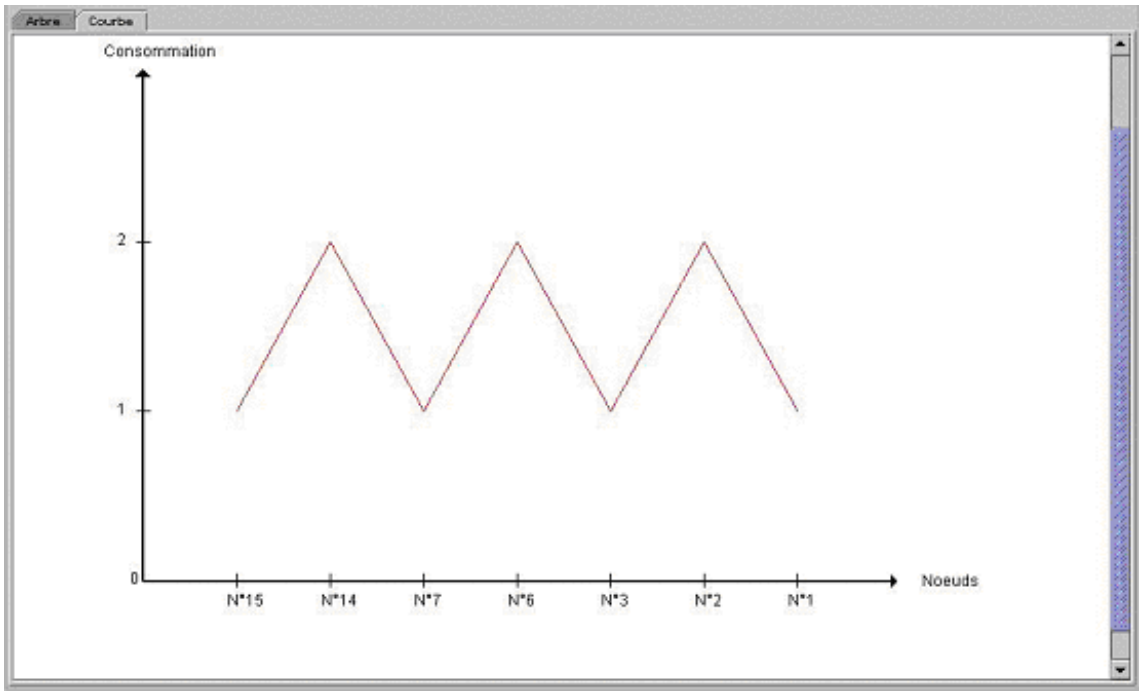
Elle permet de plus d'accéder directement à certaines informations sur les noeuds, que l'on récupére directement dans le corps de l'objet noeud :



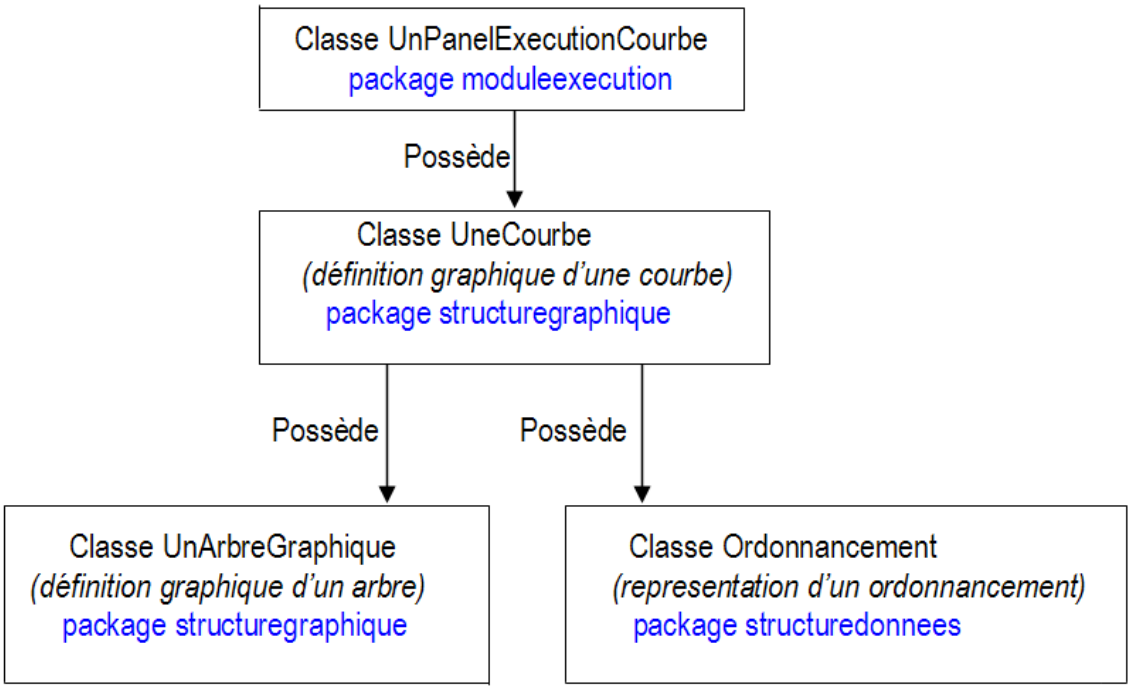
3.3.2 Panneau d'affichage de courbe

Il est le panel secondaire de l'onglet exécution : il est utile pour représenter un ou plusieurs ordonnancements retournés par les algorithmes correspondants.

Il représente plus précisément la consommation en registres pour chaque nœud de l'arbre.



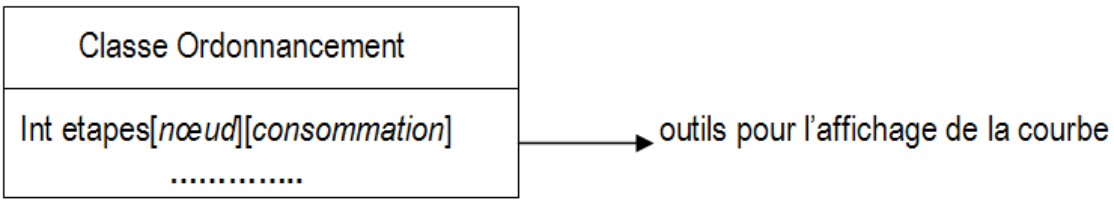
Structure implémentation



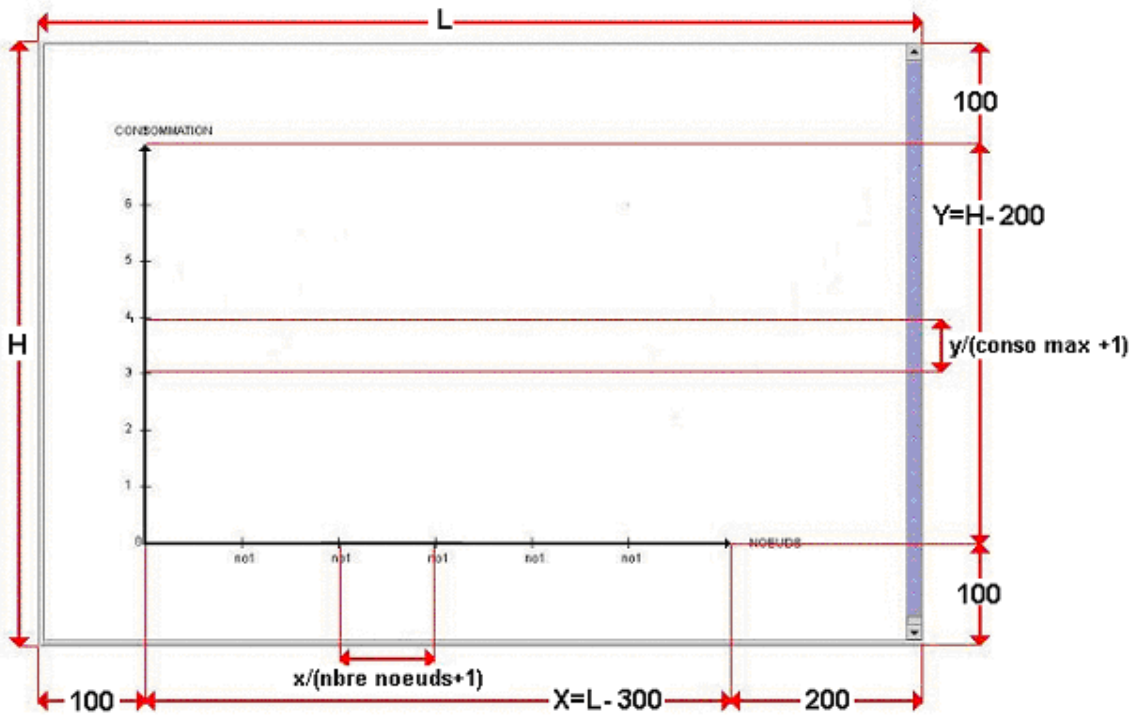
Selon le même principe que l'arbre graphique, la création d'un objet de type UneCourbe assure son affichage automatiquement.

Affichage d'une courbe

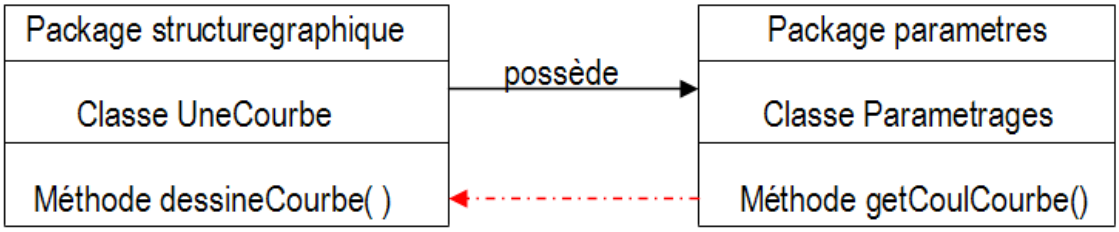
Avant toute chose, rappelons brièvement la structure d'un ordonnancement :



Regardons maintenant le dessin de la courbe (méthode paint dans l'objet Courbe). On utilise pour cela la gestion des graphiques dans AWT .
Dessin des axes :



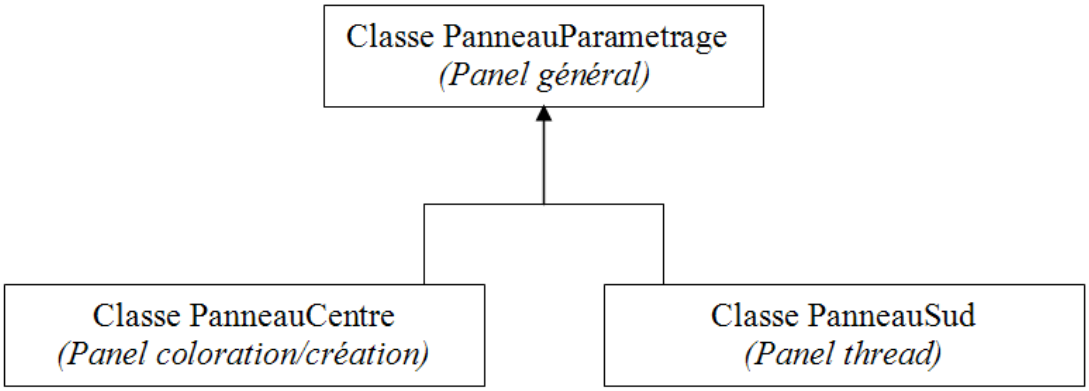
Dessin de la courbe :
Rappelons tout d'abord que la courbe a afficher est du type f(x)=y, avec x ensemble des nœuds, et y = consommation en registres du nœud x.
Nous avons choisi une représentation en dents de scie (entres les noeuds), de façon a bien voir les variations de consommation.
Enfin, au moment de l'affichage de la courbe, la couleur de celle-ci est paramétrable.
Avant de la dessiner on récupère donc sa couleur :



4 Module paramétrage : personnalisation des arbres

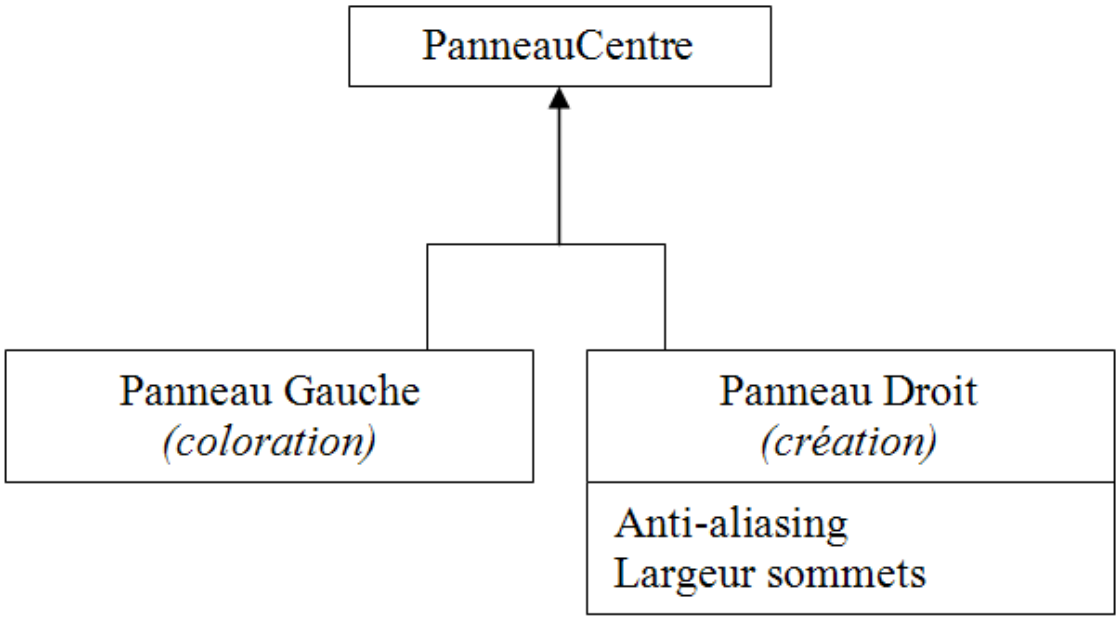
Le module paramétrage est dédié à l'amélioration visuelle et au choix de la vitesse des threads pour les arbres traités dans les parties création et exécution.

4.1 Structure du package module paramétrage



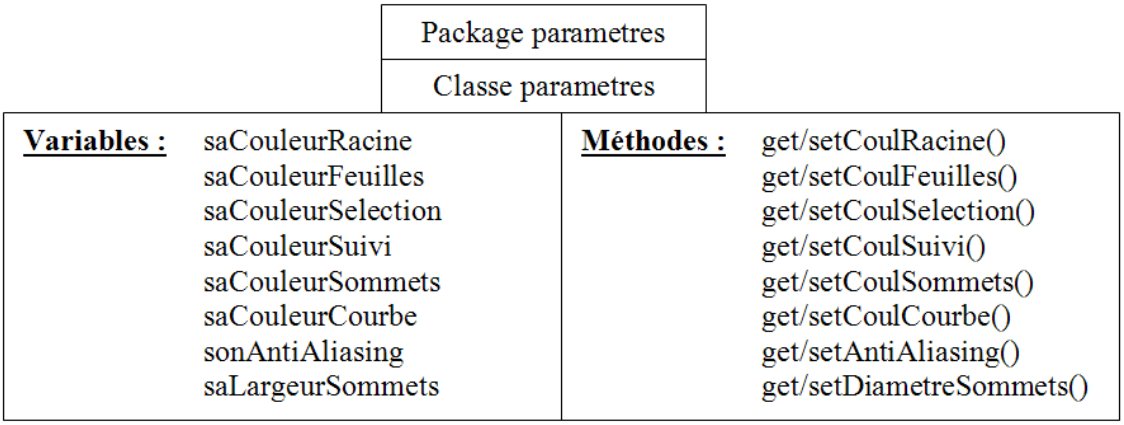
4.2 Panel coloration, création

C'est le panel qui va servir à changer les couleurs des différents éléments d'un arbre, la largeur des sommets et l'utilisation de l'anti-aliasing. Il est décomposé en deux partie : un panneau à gauche pour la coloration, un panneau à droite pour la création.



Pour pouvoir changer ces valeurs nous utilisons la classe Parametrages du package paramètres. En effet cette classe contient toute les variables ainsi que les méthodes qui permettent de les récupérées et de les modifier.

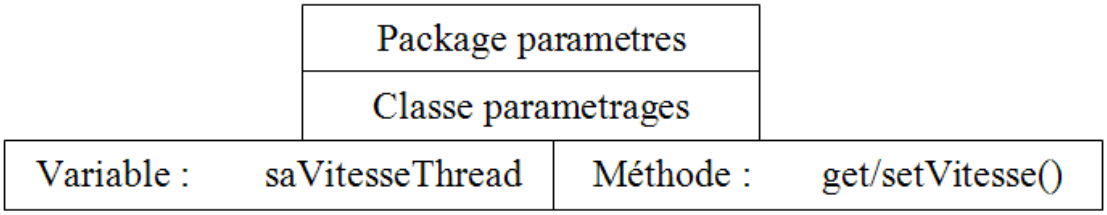
4.3 Schéma de la classe paramétrage



Il faut donc importer le package parametres dans la classe PanneauCentre. Les variables intervenant aussi pour la partie création doivent être changer à la fois dans la classe Paramétrages et dans le package modulecréation. Donc lors de la mise à jour d'une telle variable il faut donc aussi récupérer le panelBase de l'ihm pour pouvoir changer les différentes valeurs aux deux endroits.

4.4 Panel thread

C'est le panel qui se situe en bas du panel général. Il sert uniquement à choisir la vitesse des threads pour le déroulement en pas à pas des algorithmes. Il utilise lui aussi la classe Parametrages du package parametres pour changer la vitesse.



4.5 Gestionnaire de présentation

Le gestionnaire de présentation utilisé est de type GridBagLayout. De plus, un objet de type GridBagConstraints permet de configurer un composant par rapport aux autres pour que le layout puisse être plus flexible pour avoir un meilleur affichage. Ainsi pour chaque panel nous avons définit un layout ainsi qu'une constrainte générale qui, par exemple, peut être modifiée en fonction du nombre d'éléments que l'on veut placer sur une ligne. Pour éviter d'avoir un code trop long et redondant, une méthode add(int colonne, int ligne, int largeur, int longueur, Component éltAjouté, GridBagConstraints contrainte, Jpanel panel) a été rajouté afin de modifier la contrainte et d'ajouter un composant en utilisant une seule méthode. Cette méthode static est situé dans la classe Parametrage-Constante avec toutes les chaînes de caractères servant au différents affichages de texte dans les panneaux.

Deuxième partie

Algorithmes du logiciel

Cette partie du logiciel doit donc permettre à l'utilisateur de générer des arbres aléatoirement ou suivant différents paramétres ou bien de pouvoir calculer les ordonnancements.

5 Structures de données et architecture

5.1 Structure de données

Deux options se présentent pour la structure de donnée :

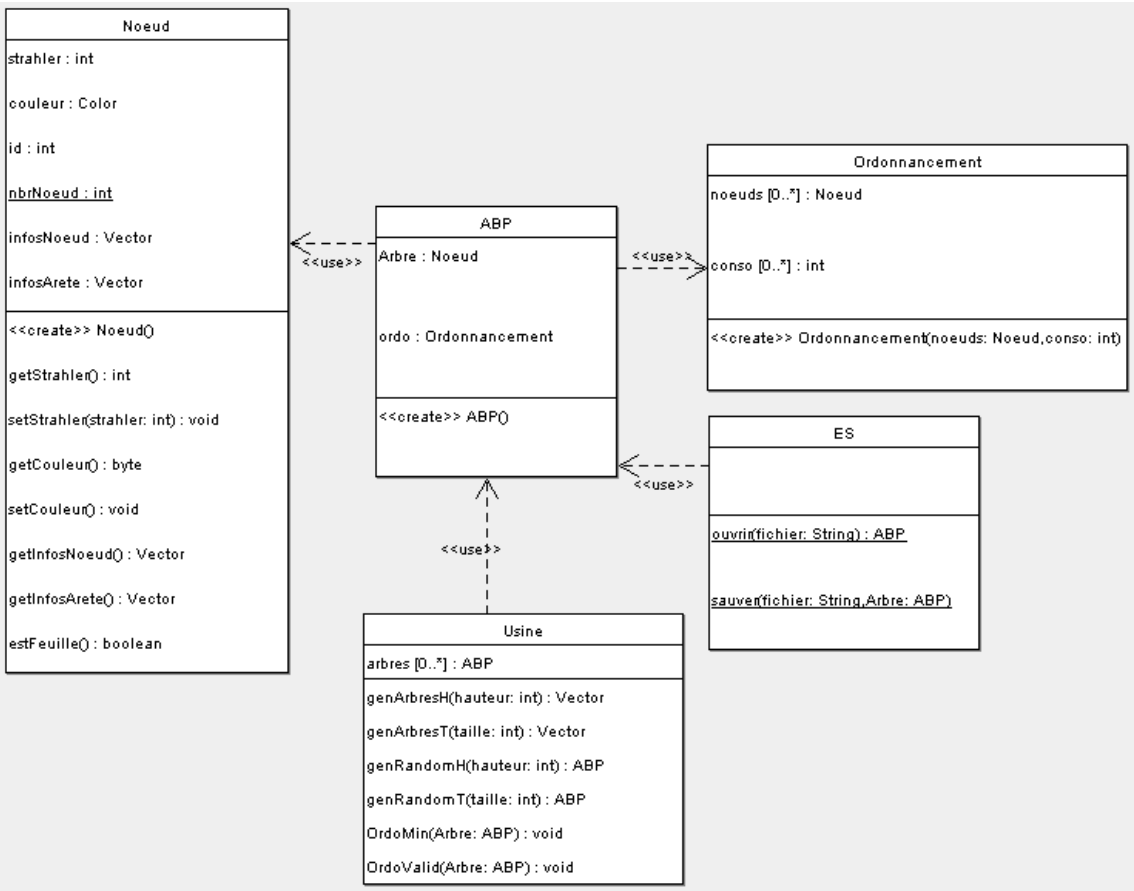
- Utiliser une structure de pointeur en prenant la définition de l'arbre binaire parfait (une racine ou une racine suivi de deux sous-arbres), qui présente l'avantage de permettre facilement un parcours, et donc l'ordonnancement.
- Utiliser un tableau d'objet où l'objet i aura pour fils les objets $2i$ et $2i+1$. Cette structure permet une génération d'arbre simple à mettre en oeuvre, notamment pour une hauteur fixée.

Cependant, ces structures possèdent aussi leurs inconvénients : Atteindre un noeud particulier dans un arbre avec la première structure se fait au pire en $O(n^2)$ alors qu'un tableau permet une identification immédiate. De même, la génération d'un arbre avec une taille donnée en partant de la structure tableau est relativement ardue, la hauteur n'étant pas connue à l'avance. Pour finir, la structure de donnée qui a été décidé était un tableau car cela convenait mieux aux algorithmes, particuliérement dans le cas de la génération des arbres, et que dans le cas de l'IHM cela revenait au meme.

5.2 Passage des résultats

Afin de pouvoir communiquer les résultats à l'IHM, nous avons décidé d'utiliser un Vector d'arbre binaires ou d'ordonnancements. De cette maniére, l'interface graphique est capable de récupérer tous les résultats et même d'afficher le pas à pas et le déroulement d'un algorithme.

5.3 Diagramme UML général



5.4 Entrées/Sorties

Les Arbres peuvent êtres sauvegardés au format LEDA, à la base une librairie d'algorithmique de graphe mais qui a aussi défini plusieurs formats de sauvegarde, helas, ces normes étant tres vastes ils nous a fallu procéder à resserrer les possiblités d'entrée/sortie afin de ne conserver que les arbres binaires parfaits.
Exemple :

```
LEDA.GRAPH
string
string
7 //nombre total de noeuds
|{-65536,}|
|{-16777216,}|
|{-16777216,}|
|{-16776961,}|
|{-16776961,}|
|{-16776961,}|
```

```
|{-16776961,}|
6 //nombre total d'aretes
1 2 0 |{}|  // signifie : il existe une arete du sommet 1 a 2
1 3 0 |{}|
2 4 0 |{}|
2 5 0 |{}|
3 6 0 |{}|
3 7 0 |{}|}
```

6 Génération des arbres

La premiére partie des algorithmes permet de générer des arbres binaires selon divers critéres.

6.1 Génération aléatoire selon une hauteur donnée

Dans cette méthode (genAleaH), nous créons un tableau de nombres binaires, de taille 2^{h+1} (remplis de 1 pour un sommet, 0 pour rien du tout).
Pour remplir ce tableau, nous appelons la fonction genTab, avec en argument le tableau en question (puisque c'est une fonction récursive) et le champ hauteur. La signature de la méthode genTab : int genTab(tab, int hauteur, int nbrNoeud).

genTab pour une génération en hauteur fonctionne comme suit :

- c'est une méthode récursive qui renvoie un tableau d'entiers, de taille le nombre de noeuds(ici ce sera 2^{h+1}) ;
- Pour générer un tableau aléatoire de hauteur h(cad de 2^{h+1} noeuds au plus),on génère un tab de hauteur h-1, On a alors un tab de 2^h cases à 1 ;
- Parmis ces cases remplies de 1, on en tire une aléatoirement, cad qu'on tire un entier compris entre 1 et 2n+1, C'est à cet endroit qu'on va insérer un nouveau sommet, notons S la case ou le sommet ainsi choisi.
- On tire au hasard entre droite et gauche (un tirage aléatoire entre 256 valeurs ramenées modulo 2) S devient alors le fils droit, ou le fils gauche, du sommet inséré (en décalant dans le tableau les fils existants),l'autre fils du sommet inséré est alors une feuille.
- A cet instant, plutôt que de toujours conserver l'autre fils comme une feuille, on fait appel à une méthode qui va décider aléatoirement si on ajoute des fils ou non à cette feuille, cela en prenant garde de ne pas dépasser la hauteur souhaitée.On a ainsi un arbre aléatoirement construit, de la hauteur voulue.
- L'ABP est alors créé en ajoutant un noeud à chaque fois qu'on a un 1 dans une case du tableau.

6.2 Génération aléatoire selon un nombre de noeud donné

Pour ce qui est de la génération aléatoire en fonction d'un nombre de sommets : nbrNoeud.

- On teste tout d'abord que nbrNoeud est impair, car seuls les nb de noeuds impairs sont possibles dans un ABP.
- Ensuite, on crée un ABP de hauteur (nbrNoeud-1)/2 . C'est la hauteur max qu'on peut avoir pour nbrNoeud.
- On créé également un tableau d'entiers, de taille 2^{h+1}, comme précédemment.
- Puis on fait appel à la méthode genTab, la même que ci-dessus, à la différence près que cette fois c'est le champs hauteur qui est à -1, et dont on ne va pas se servir.
- Dans genTab, on va donc remplir récursivement le tab jusqu'à l'obtention de nbrNoeud sommets, cad nbrNoeud cases 1.
- Pour cela : on utilise un compteur static de sommets, qui s'incrémente à chaque fois qu'on ajoute un sommet.
- Pour générer un tab avec nbrNoeud sommets, on génère récursivement un tab de nbrNoeud-2 (c'est un arbre binaire ne l'oublions pas).
- puis comme précédemment, on choisit aléatoirement un sommet où en insérer un nouveau, en décalant d'éventuels fils existants.
- A chaque fois qu'on créé un nouveau sommet, on incrémente la variable static et on quitte la méthode genTab en renvoyant le tableaudés qu'on atteint le nombre de sommets souhaités.
- On récupère le tableau d'entiers, et on construit l'ABP en lui ajoutant un noeud à chaque fois qu'on a une case 1 dans le tableau.

6.3 Génération de Tous les arbres selon un nombre de noeuds donné

Cet algorithme s'inspire en grande partie de celui de Joe Sawada permettant de générer des permutations dans l'ordre lexicographique (i.e., dans l'ordre numérique). Ici on va générer tous les parenthésages possibles pour un nombre de noeud internes (i.e. de parenthèses)donné.
On représentera le parenthèsage sous forme binaire, avec 1 si on ouvre une parenthèse et 0 si on la referme.On peut remarquer que cette représentation correspond aussi au parcours préfixe d'un arbre binaire, en effet, 1 signifie que c'est un noeud, et 0 que c'est une feuille (excepté la dernière feuille qui est ignorée).
Ainsi 11110000 représentera l'arbre binaire parfait avec 4 noeuds internes où ceux ci sont tous situés à gauche de le racine.Comme on peut le remarquer cette répresentation possède une valeur numérique équivalente et est la plus grande valeur binaire que l'on puisse obtenir avec quatre 1 et quatre zeros. Ainsi, si l'on part de 01010101 et que l'on incrémente de 2 où de 4 (afin d'obtenir un parenthèsage cohérent) jusqu'à 11110000 on obtiendra tous les parenthèsages possibles et ainsi tous les arbres binaires parfaits contenant 4 noeuds internes.
Pour construire l'arbre coresspondant, il suffit ensuite d'effectuer un parcours en profondeur et de construire les fils lorsque l'on tombe sur un 1.
L'algorithme étant itératif, il se révèle très léger en consommation de ressources et peut ainsi générer jusqu'à plus de 50 000 arbres binaires (pour 11 noeuds internes).

6.4 Génération de tous les arbres binaires pour une hauteur donnée sans symétrie

Le but de cette partie du projet était de trouver un algorithme permettant de générer tous les arbres binaires pour une hauteur donnée tout en évitant les symétries.

Pour cela, nous avons essayé de trouver un algorithme qui soit capable de générer tous les arbres binaires en évitant les symétries en meme temps que l'arbre était calculé en mémoire. Malheureusement, nous ne sommes pas parvenus à concevoir un tel algorithme et nous avons finalement décidé d'utiliser la méthode de rejet, c'est à dire que lorsqu'on s'apprête à ajouter un arbre nouvellement calculé à la la liste de sortie, on vérifie d'abord qu'il ne soit pas déja le symétrique d'un arbre déja existant dans cette liste. On évite ainsi toute symétrie à la sortie.

Afin de générer tous les arbres, nous avons décidé de tirer partie de la strucutre et tableau dans laquelle était rangée l'arbre. L'algorithme repose sur un autre tableau, qui fonctionne suivant cette régle : si en i il y'a un 1 alors cela signifie que les noeuds qu'il y'a des noeuds en position 2i et 2i+1 mais pas nécessairement qu'il y'a un 1 dans ces positions, en revanche si en i il y'a un 0 cela signifie qu'en 2i et 2i+1 il ne peut y'avoir de noeuds donc il ne peut y avoir de 1 car si il n'y a pas de noeuds Ã un endroit il ne peut pas y'avoir de fils. En utilisant ce résonnement, on est sur de garder l'arbre de maniére binaire c'est à dire qu'un noeud ne peut avoir que 0 ou 2 fils. Avec une traditionnelle structure en tableau, un noeud aurait pû posséder un seul fils ce qui aurait rompu la symétrie.

Une fois cette structure fixée, nous avons utilisé un algorithme qui compte en binaire de maniére récursive à chaque étage. En effet, puisqu'on veut parcourir tous les arbres binaires possibles, cela revient à essayer toutes les possibilités sur chaque hauteur de l'arbre, c'est à dire en quelque sorte compter en binaire sur chaque hauteur de l'arbre. En effet si pour deux bits on compte 00, 01, 10 et 11, on a parcouru toutes les possiblités et en quelque sorte tous les arbres binaires pour cet étage.

Il restait alors le probléme de propager les descendances de l'arbre qui ne doivent pas avoir de fils, en effet comme expliqué si dessus, si il y'a un 0 sur un noeud il ne doit y avoir aucun arbre qui descend de ce noeud. L'idée fut donc qu'il ne fallait pas propager des 0 lors de l'appel récursifs aux hauteurs plus basses de l'arbre mais remplacer ces 0 par des 2 et lorsqu'à l'étage suivant on incrémentera la valeur binaire de l'étage, si on essaye d'ajoute un 1 à un 2 on considérera qu'il s'agit d'une zone noire et on se déplacera à gauche du 2 et on recommencera cette opération jusqu'a retrouver un 1 ou un 0 que l'on sait incrémenter normalement. De cette maniére, chaque étage peut incrémenter sa valeur binaire de maniére à parcourir toutes ses possiblités tout en étant sur de ne pas incrémenter aux endroits qui correspondent à des fils inexistants des hauteurs supérieurs.

On appelera récursivement l'algo sur la premiére hauteur, c'est à dire sur un noeud 1 qui lui même s'appellera sur la hauteur suivante qui peut être soit 01, soit 10 soit 11 et chacune de ses hauteurs appellera la hauteur suivante avant de s'incrémenter, etc.. On renvoie un arbre complet quand on se trouve à une

hauteur égale à 2 car cela signifie qu'il n'y a plus qu'un seul étage de noeud en dessous. A chaque étage, quand à la fin d'une incrémentation il reste une retenue on remonte à l'étage précédent car cela signfie qu'on a parcouru toutes les possiblités pour l'étage en cours.

De cette maniére, on a généré tous les arbres binaires existants en évitant les symétries.

7 Ordonnancements

Un ordonnancement est une liste de noeuds qui respecte la règle suivante : *chacun des noeuds de la liste s'y trouve avant son père.* Les différents algorithmes mis en place possède deux visions d'un ABP.

- un des algorithme voie l'ABP de manière récursive
- Les autres le voient comme un anti-arbre.

7.1 Ordonnancements optimum local

Cet algorithme est l'unique parmis ceux mis en place à possèder une vision récursive de l'arbre binaire parfait (i.e. un ABP est formé d'une racine qui est un noeud et de deux fils qui sont eux-même des ABP).

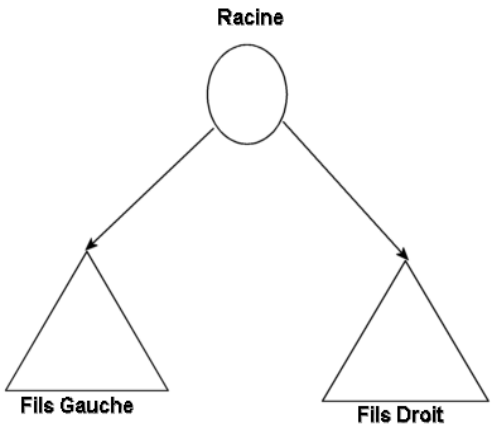


Fig. 1 – U
n arbre ABP

L'algorithme récursif effectue un parcours en profondeur suffixe.

- Ce parcours se fait en descendant en premier dans le fils qui possède le nombre Strahler le plus grand afin d'obtenir des ordonnancements optimaux.

- Une fois remonter sur le noeuds on agence les ordonnancements de la manière suivante :
- Si les fils ont des nb de Strahler différent alors les ordonnancements du fils ayant le nb le plus elevé seront placés avant ceux de l'autre fils.
- Sinon on créera des ordonnancements qui commenceront dans un premier temp par ceux du fils droit puis dans un second par ceux du fils gauche.

Cet vision de l'arbre a ses limites car elle ne permet pas de melanger les ordonnancements du fils droit et du fils gauche. Pour pouvoir générer un ordonnancement aléatoire parmis tous ceux qui existent, nous avons donc décidé de voir les ABP comme des anti-arbres.

7.2 Ordonnancement Aleatoire et Globaux

On entend par globaux des ordonnancements qui mélange les noeuds du fils droit et du fils gauche contrairement aux ordonnancement locaux obtenus par un parcour en profondeur. Ces deux algorithmes fonctionnent sur la même vison d'un ABP qui est celle d'un anti-arbre.

L'algo pour les ordonnancements aleatoire n'est pas recursif contrairement a l'algo pour les ordonnancements globaux (optimum ou non).

Au commencement cette algo possède une liste contenant l'ensemble des feuilles de l ABP. Pour l'ordonnancement aléatoire il va remonter des noeuds en piochant un noeud se trouvant dans la liste, tandis que pour les ordonnancements globaux et optimum globaux il va commencer par remonter le premier élement de la liste et trouver tous les ordonnancements commencant par ce noeud en remontant systematiquement le premier de la liste puis le second et ainsi de suite jusqu'a l'obtention d'un ordonnancement valide puis va revenir en arriére pour ajouter à la place du premier le second et ainsi de suite pour chaque appel recursif jusqu'à ce qu'on ai obtenu tout les ordonnancements valides de l'ABP. Un noeud est ajoute si et seulement si ses deux fils sont deja présents dans l'ordonnancement.
Remarque : à cause du nombre d'ordonnancement qui grandit trés vite en fonction du nombre de noeuds et de se complétude de celui-ci la memoire de l'ordinateur est vite saturée.

Conclusion

Ce projet nous a permis de développer un logiciel complet de manipulation et de génération d'arbres binaires. Nous avons donc du planifier et se mettre d'accord sur les différentes parties du logiciel. Nous avons aussi pu travailler sur la conception et l'implémentation d'une ihm compléte ainsi que sur toute la partie algorithmique pour générer et manipuler des arbres binaires parfait.
Une extension possible au projet serait l'execution des algorithmes en meme temps que s'affiche l'arbre sur l'ihm.