Des graphes sur lesquels on peut compter

POnGL, TP2, héritage, surcharge, classes abstraites, redéfinition, exponentiation rapide.

1 Présentation

Dans ce TP, on va créer des graphes orientés qui sont spécialement conçus pour faire des calculs mathématiques. Chaque nœud du graphe sera en charge de faire une opération, et les arêtes indiqueront comment les résultats intermédiaires sont passés d'un nœud à l'autre.

Pour ce TP, vous compléterez le fichier `Graphes.java` à télécharger à l'adresse suivante :
`http://www.lri.fr/~blsk/POnGL/Graphes.java`
Ce fichier est essentiellement constitué d'une fonction `main`, qui contient une série d'exemples avec lesquels vous pourrez tester votre code. Pour l'instant, ces exemples sont placés en commentaires pour éviter les erreurs de compilation. Vous pourrez décommenter chaque exemple dès que les opérations dont il a besoin auront été définies.

2 Mise en jambes

Tous les nœuds que nous allons manipuler hériteront de la classe abstraite suivante :

```
abstract class Noeud {
    public Noeud() { }
    abstract public double compute();
}
```

Un appel à la méthode `compute()` doit renvoyer le résultat calculé par ce nœud du graphe, et ce résultat sera un `double`.

Les nœuds pourront de plus posséder un certain nombre d'attributs représentant les les arguments des opérations effectuées par le nœud. Ces attributs privés seront appelés `source1`, `source2`, ... et seront toujours de la classe `Noeud`.

Créer quatre classes pour des nœuds effectuant les opérations suivantes :

- Une classe `NSource`, dont les objets possèdent un attribut `cst` qui contient une valeur `double` constante. La méthode `compute()` doit renvoyer la valeur de cet attribut.
- Une classe `NMoins`, dont les objets possèdent un attribut `source1`. La méthode `compute()` doit renvoyer l'opposé de la valeur de la source.
- Une classe `NRacine`, dont les objets possèdent un attribut `source1`. La méthode `compute()` doit renvoyer la racine carrée de la valeur de la source.
- Une classe `NAdd`, dont les objets possèdent deux attributs `source1` et `source2`. La méthode `compute()` doit renvoyer la seomme des valeurs des deux sources.

Pour tester ces classes, décommenter l'exemple (a), qui calcule $2-\sqrt{2}$. Le résultat attendu est environ 0.5858. En suivant le modèle de ce premier exemple, créer un autre graphe, qui calcule $\sqrt{1+\sqrt{2}}-1$. Il faut obtenir environ 0.5538.

3 Une hiérarchie de nœuds

Nous allons réorganiser nos classes de nœuds en une hiérarchie plus propre, qui nous permettra ensuite plus facilement d'ajouter de nouvelles sortes de nœuds.

Créer d'abord une classe abstraite `NoeudUnaire`, qui hérite de `Noeud` et y ajoute les ingrédients suivants (cette liste est un minimum, vous pouvez y ajouter d'autres attributs ou méthodes si cela vous semble utile, maintenant ou plus tard dans le TP) :

— Un attribut `source1` de classe `Noeud`.
— Une méthode `setSource1(Noeud n)` qui définit l'attribut `source1` comme étant le nœud `n`.

Redéfinir les classes `NMoins` et `NRacine` pour qu'elles héritent de `NoeudUnaire`.

Créer selon le même modèle une nouvelle classe abstraite `NoeudBinaire`, qui hérite de `NoeudUnaire` et y ajoute un deuxième attribut `source2` ainsi que la méthode `setSource2` associée, puis modifier la classe `NAdd` pour qu'elle hérite de `NoeudBinaire`. Ajouter une classe `NMult` pour représenter un nœud binaire qui calcule le produit ses deux arguments.

Avant de passer à la partie suivante, ne pas oublier de vérifier que les exemples fonctionnent toujours.

4 Quelques nœuds plus compliqués

4.1 Un nœud de choix

Créer un nœud `NChoix` ayant les caractéristiques suivantes :
— Ce nœud possède trois sources.
— Ce nœud possède un attribut `seuil` de type `double`.
— Le résultat renvoyé lors d'un appel à `compute()` est le suivant : si la valeur de la troisième source est plus petite que le seuil, alors on retourne la valeur de la première source. Sinon, on retourne la valeur de la deuxième source.

Tester avec l'exemple (b), qui calcule le minimum entre les deux valeurs 1 et 2. Le résultat attendu est environ 1.

4.2 Un compteur

Créer un nœud `NCompteur` ayant les caractéristiques suivantes :
— Ce nœud ne possède aucune source.
— Le premier appel à `compute()` doit renvoyer 1.
— Le deuxième appel à `compute()` doit renvoyer 2.
— Le n-ème appel à `compute()` doit renvoyer n.

Tester avec l'exemple (c), qui calcule la partie entière du nombre 4.2. Le résultat attendu est environ 4.

4.3 Calcul de puissances

Créer un nœud `NPuissance` ayant les caractéristiques suivantes :
— Ce nœud possède une source.
— Ce nœud possède un attribut `n`.
— Ce nœud possède une méthode privée de signature `double expRapide(double x, int n);` qui calcule x^n.
— Un appel à `compute()` retourne la valeur de la source mise à la puissance `n` en utilisant la méthode `expRapide`.

Indication : pour la méthode `expRapide`, faire en sorte que le nombre du multiplications effectuées soit proportionnel au logarithme de `n`. Voici trois équations qui peuvent vous aider pour mettre au point cette méthode :

$$\begin{aligned} x^0 &= 1 \\ x^{2n} &= (x^n) * (x^n) \\ x^{2n+1} &= (x^n) * (x^n) * x \end{aligned}$$

Tester avec l'exemple (d), qui calcule le carré de la racine de 2. Le résultat attendu est environ 2. Tester ensuite avec l'exemple (e), qui calcule la 100-ème puissance de 0.999999999. Le résultat attendu est environ 0.0000454.