

Polytechnique Montréal

TP3-Log3430

Travail de :

Xavier Brazeau (1854911)

Amine Kamal (1718831)

Maxime Bernier (1893115)

Remis à Mme Hiba Bagane

Jeudi 28 février 2019

Mise en contexte

Pour ce laboratoire, nous nous intéresserons aux tests orientés objets. Nous utiliserons la stratégie MaDUM. Le fichier à tester est Digraph provenant de algs4-master du laboratoire précédent. Le Digraph sert à créer un graph orienté. Nous avons ainsi une seule classe à tester.

Stratégie

Établir la matrice MaDUM

La première étape consiste à constituer la matrice nécessaire à la stratégie. Celle-ci est constituée des attributs de la classe ainsi que des méthodes de celle-ci. Nous avons ignoré la constante statique NEWLINE puisque celle-ci est seulement la séquence de caractères qui correspond à un changement de ligne. Les attributs retenus sont donc :

- final int V: nombres de sommets dans le graphe
- int E: nombre d'arcs dans le graphe
- Bag<integer>[] adj: liste des voisins pour chaque sommet
- int[] indegree: Nombre d'arcs pointant sur un sommet, et ce, pour chaque sommet

La classe Digraph possède trois constructeurs :

- Digraph(int V) : Constitue le graphe en fonction d'un nombre de sommets désirés
- Digraph(In in) : Constitue un graphe en fonction d'une entrée du type de la classe In fournie par algs4-master.
- Digraph(Digraph G) : Constructeur par copie

Les méthodes sont les suivantes :

- Reporters
 - V() : retourne le nombre de sommets
 - E() : retourne le nombre d'arcs
 - adj(int V) : retourne un itérable des voisins du sommet V
 - indegree(int V) : retourne le nombre d'arcs entrant dans le sommet V
- Transformers
 - addEdge(int v, int w) : rajoute un arc allant de v à w dans le graph
 -
- Others
 - reverse() : retourne un graphe avec les arcs inversés de celui actuel
 - validateVertex(int v): lance une erreur si le sommet v n'est pas valide
 - outdegree(int v) : retourne le nombre d'arcs sortant d'un sommet v
 - toString() : Traduit un graphe en string

Ensuite, nous nous sommes questionnés sur les attributs que les méthodes utilisent ou modifient. Pour la méthode adj(v), il est à noter qu'elle retourne uniquement un élément du tableau adj. La méthode addEdge(v,w) incrémente E, ajoute un arc à adj et modifie le indegree d'un sommet. La matrice résultante se trouve en annexe.

Écrire les tests

L'écriture des tests se fait par tranche, soit par attribut. Dans Il faut aussi tester chaque ordre de méthodes pour celles qui transforment un de nos attributs. Il faut donc $c * n!$ tests par attribut où n est le nombre de méthode modifiant l'attribut et c , le nombre de constructeur. Dans notre cas, nous avons une seule méthode transformante, soit `addEdge(v,w)` et trois constructeurs. Nous aurons ainsi toujours trois tests à faire par attribut.

Pour l'attribut `V`, il suffit de trois tests pour tester les trois constructeurs. La méthode `V()` est utilisé par valider notre état. Les méthodes autres sont : `reverse()`, `toString()` et `validateVertex()`. Un problème survient pour `validateVertex` : il s'agit d'une méthode privée. Nous la testerons à travers la fonction `addEdge`. On s'attend à ce que la méthode vertex retourne une erreur lorsque le sommet n'est pas valide. Ainsi, nous appelons `addEdge` avec des paramètres égal ou supérieur à `V` et inférieur à 0. L'ordre des trois fonctions est sans importance et ne doivent pas modifier le valeur de `V`.































Pour l'attribut `E`, la méthode `E()` est le getter de `E`. Les méthodes qui nous intéressent sont : `addEdge`, qui modifie la valeur, et `toString`. Nous devons donc construire un digraph et vérifier qu'il s'est construit convenablement. Ensuite, on exécute `addEdge` et on vérifie que l'attribut `E` a été incrémenté de 1. On s'assure que `toString` utilise bien la valeur de `E` sans la modifier.

Pour l'attribut `adj`, il est important de souligner que le getter `adj()` retourne seulement un élément du tableau. Ainsi, lors de nos vérifications de l'état, nous devons nous assurer que l'ensemble des éléments du tableau correspond à nos attentes. Nous avons donc utilisé une boucle `for`. Les méthodes d'intérêt sont `addEdge`, qui ajoute un élément pour le sommet `v`, et `toString`.

Pour l'attribut `indegree`, il faut s'assurer que tous les éléments du tableau correspondent aux attentes. La méthode getter est `indegree(v)`. La seule méthode qui nous intéresse est `addEdge` qui augmente le `indegree` du sommet `v`.

Vérifier la couverture

Voici le résultat de la couverture :

 Digraph.java		90,8 %	364	37	401
▼  Digraph		90,8 %	364	37	401
 Digraph(In)		77,3 %	58	17	75
 ^S main(String[])		0,0 %	0	15	15
 Digraph(int)		86,8 %	33	5	38
 Digraph(Digraph)		100,0 %	74	0	74
 addEdge(int, int)		100,0 %	28	0	28
 adj(int)		100,0 %	8	0	8
 E()		100,0 %	3	0	3
 indegree(int)		100,0 %	8	0	8
 outdegree(int)		100,0 %	9	0	9
 reverse()		100,0 %	34	0	34
 toString()		100,0 %	77	0	77
 V()		100,0 %	3	0	3
 validateVertex(int)		100,0 %	25	0	25

La couverture actuelle est de 90.8%. Nous constatons que deux constructeurs font défaut. L'analyse du code nous permet de constater que nous n'avons pas traité les cas où le constructeur échoue. Pour le constructeur qui prend en paramètre un nombre de sommets, il fallait tester un nombre négatif. Pour le constructeur avec le paramètre de type `ln`, il y a trois cas d'erreurs à traiter : un nombre négatif de sommets, un nombre négatif d'arcs et un fichier vide de sorte que la fonction `readInt()` retourne une erreur. Pour accomplir ceci, nous avons créé trois fichiers : `badGraph`, `badGraph2` et `badGraph3` qui sont dans le même répertoire que le fichier de test. `badGraph` contient un nombre négatif de sommet. `badGraph2` contient un nombre d'arcs négatifs. `badGraph3` est vide.

De plus, il y a la méthode `main` qui n'a pas été testée. Celle-ci crée un graphe à partir d'un paramètre `ln` qui constitué à partir du paramètre de la méthode. Ensuite, il affiche le résultat sur la sortie standard. Pour la tester, nous avons du remplacer la sortie standard par un `PrintStream`. Le paramètre passé est le fichier `customGraph` qui est un graphe valide sans arc. On génère le résultat de la méthode dans le `PrintWriter` qui se vide dans une variable `baos`. Ensuite, on exécute normalement la méthode `main` en rétablissant la sortie standard et on compare les deux résultats. Voici la couverture résultante :

Digraph.java	100.0 %	401	0	401
Digraph	100.0 %	401	0	401
main(String[])	100.0 %	15	0	15
Digraph(Digraph)	100.0 %	74	0	74
Digraph(ln)	100.0 %	75	0	75
Digraph(int)	100.0 %	38	0	38
addEdge(int, int)	100.0 %	28	0	28
adj(int)	100.0 %	8	0	8
E()	100.0 %	3	0	3
indegree(int)	100.0 %	8	0	8
outdegree(int)	100.0 %	9	0	9
reverse()	100.0 %	34	0	34
toString()	100.0 %	77	0	77
V()	100.0 %	3	0	3
validateVertex(int)	100.0 %	25	0	25

Nous obtenons bien une couverture de 100%.

Conclusion

Ce laboratoire nous a permis de nous familiariser avec la stratégie MaDUM. Toutefois, étant donné qu'une seule fonction transformante est présente. Nous n'avons pas constaté que le nombre de tests requis est de $c * n!$. De plus, la présence d'une fonction privée a soulevé des questionnements quant à son traitement : une seule fois ou à travers toutes les méthodes l'utilisant. Le temps requis pour ce laboratoire est adéquat et la matière d'un intérêt certain.

Annexe

Matrice MaDUM

Attribut	Cstr V	Cstr In	Cstr copie	V()	E()	validateVertex(V)	addEdge(V, W)	adj(V)	outDegree(V)	indegree(V)	reverse()	toString()
NEWLINE	c	c	c									o
V	c	c	c	r		o					o	o
E	c	c	c		r		T					o
adj	c	c	c				T	r	o		o	o
indegree	c	c	c				T			r		

Cstr = constructeur