TP6 Java Avancé - Trop Graph

Max Ducoudré - INFO2

Exercice 2 - MatrixGraph

1. On souhaite créer la classe paramétrée (par le type des valeurs des arcs) MatrixGraph comme seule implantation possible de l'interface Graph définie par le fichier Graph.java

En Java, il n'est pas possible de créer des tableaux de variables de type car le type n'existe pas à la compilation mais uniquement à l'exécution. Pour régler celà, on peut utiliser un tableau d'object qu'on pourra caster vers le type voulu à l'exécution en ajoutant un `SuppressWarning("unchecked")` sur le constructeur.

Voici le code de la classe MatrixGraph :

```
public final class MatrixGraph<T> implements Graph<T> {
    private final T[] array;
    private final int nodeCount;

    @SuppressWarnings("unchecked")
    public MatrixGraph(int nodeCount) {
        if(nodeCount < 0) {
            throw new IllegalArgumentException("nodeCount must be positive");
        }
        this.nodeCount = nodeCount;
        array = (T[]) new Object[nodeCount*nodeCount];
    }

    public int nodeCount() {
        return nodeCount;
    }
}
```

On rend l'interface Graph `sealed` pour permettre uniquement l'implantation de `MatrixGraph` puis on ajoute une méthode nodeCount :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    int nodeCount();
}
```

2. On peut remarquer que la classe MatrixGraph n'apporte pas de nouvelles méthodes par rapport aux méthodes de l'interface Graph donc il n'est pas nécessaire que la classe MatrixGraph soit publique.

On ajoute la méthode statique à `Graph` qui créer une instance de `MatrixGraph` :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    public static <T> Graph<T> createMatrixGraph(int nodeCount) {
        return new MatrixGraph<T>(nodeCount);
    }
}
```

On retire le mot clé `public` de la classe `MatrixGraph` :

```
final class MatrixGraph<T> implements Graph<T> {
    /* [...] */
}
```

3. Indiquer comment trouver la case (i, j) dans un tableau à une seule dimension de taille nodeCount * nodeCount. Si vous n'y arrivez pas, faites un dessin ! Afin d'implanter correctement la méthode getWeight, rappeler à quoi sert la classe java.util.Optional en Java. Implanter la méthode addEdge en utilisant la javadoc pour savoir quelle est la sémantique exacte. Implanter la méthode getWeight en utilisant la javadoc pour savoir quelle est la sémantique exacte. Vérifier que les tests marqués "Q3" passent.

Pour accéder à un index (i, j) dans un tableau à une seule dimension, il faut y accéder en utilisant `nodeCount * i + j`. La classe optional permet de renvoyer une valeur ou rien.

On ajoute les méthodes `addEdge` et `getWeight` dans l'interface `Graph` :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    /* [...] */
    void addEdge(int src, int dst, T weight);
    Optional<T> getWeight(int src, int dst);
}
```

Puis on les implémentes dans `MatrixGraph` :

```
final class MatrixGraph<T> implements Graph<T> {

    /* [...] */

    public void addEdge(int src, int dst, T weight) {
        Objects.requireNonNull(weight);
        Objects.checkIndex(src, nodeCount);
        Objects.checkIndex(dst, nodeCount);
        array[nodeCount * src + dst] = weight;
    }

    public Optional<T> getWeight(int src, int dst) {
        Objects.checkIndex(src, nodeCount);
        Objects.checkIndex(dst, nodeCount);
```

```
        return (array[nodeCount * src + dst] != null) ?
                Optional.of(array[nodeCount * src + dst]) :
                Optional.empty();
    }
}
```

4. On souhaite maintenant implanter une méthode mergeAll qui permet d'ajouter les valeurs des arcs d'un graphe au graphe courant. Dans le cas où on souhaite ajouter une valeur à un arc qui possède déjà une valeur, on utilise une fonction prise en second paramètre qui prend deux valeurs et renvoie la nouvelle valeur.

La méthode `mergeAll` vas fusionner toutes les valeurs d'un `Graph` vers un autre. Pour cela, il faut que l'utilisateur envoi une fonction de fusion en paramètre. Cette fonction sera l'interface fonctionnelle `BinaryOperator<? extends T>` qui prend deux valeurs et renvoie la nouvelle valeur.

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    /* [...] */
    void mergeAll(Graph<T> graph, BinaryOperator<? extendsT> merge);
}
```

```
final class MatrixGraph<T> implements Graph<T> {
    /* [...] */
    public void mergeAll(Graph<T> graph, BinaryOperator<? extends T> merger) {
        Objects.requireNonNull(graph);
        Objects.requireNonNull(merger);
        if(graph.nodeCount() != nodeCount) {
            throw new IllegalArgumentException("The graph node counts are not
equals !");
        }


        for(int i = 0; i < nodeCount; i++) {
            for(int j = 0; j < nodeCount; j++) {
                var node = getWeight(i, j);
                var otherNode = graph.getWeight(i, j);
                if(node.isEmpty() && otherNode.isPresent()) {
                    addEdge(i, j, otherNode.get());
                } else if(node.isPresent() && otherNode.isPresent()) {
                    addEdge(i, j, merger.apply(node.get(), otherNode.get()));
                }
            }
        }
    }
}
```

5. En fait, on peut remarquer que l'on peut écrire le code de mergeAll pour qu'il soit indépendant de l'implantation et donc écrire l'implantation de mergeAll directement dans l'interface. Déplacer

l'implantation de mergeAll dans l'interface et si nécessaire modifier le code pour qu'il soit indépendant de l'implantation.

Etant donné que la méthode `mergeAll` ne dépent pas de l'implantation, on peut directement écrire le code de la méthode dans l'interface `Graph` en la mettant en `default` :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    /* [...] */

    default void mergeAll(Graph<? extends T> graph, BinaryOperator<T> merger) {
      Objects.requireNonNull(graph);
        Objects.requireNonNull(merger);
        if(graph.nodeCount() != this.nodeCount()) {
            throw new IllegalArgumentException("The graph node counts are not
equals !");
        }

        for(int i = 0; i < this.nodeCount(); i++) {
            for(int j = 0; j < this.nodeCount(); j++) {
                var node = getWeight(i, j);
                var otherNode = graph.getWeight(i, j);
                if(node.isEmpty() && otherNode.isPresent()) {
                    addEdge(i, j, otherNode.get());
                } else if(node.isPresent() && otherNode.isPresent()) {
                    addEdge(i, j, merger.apply(node.get(), otherNode.get()));
                }
            }
        }
   }
}
```

6. Rappeler le fonctionnement d'un itérateur et de ses méthodes hasNext et next. Que renvoie next si hasNext retourne false ? Expliquer pourquoi il n'est pas nécessaire, dans un premier temps, d'implanter la méthode remove qui fait pourtant partie de l'interface. Implanter la méthode neighborsIterator(src) qui renvoie un itérateur sur tous les nœuds ayant un arc dont la source est src. Vérifier que les tests marqués "Q6" passent. Note : ça pourrait être une bonne idée de calculer quel est le prochain arc valide AVANT que l'on vous demande s'il existe.

Un itérateur permet d'itérer les éléments d'une collection. Il est composé d'une méthode `hasNext` pour savoir s'il y a un prochain élément et d'une méthode `next` pour accéder à l'élément suivant.
Pour générer un itérateur, on ajoute la méthode `neighborsIterator` dans `MatrixGraph`. Cette méthode vas créer un itérateur permettant de récupérer tous les index "destination" à partir d'un index "source":

```
public final MatrixGraph<T> {
    /* [...] */

    public Iterator<Integer> neighborIterator(int src) {
        return new NeighborIterator(src);
    }
```

```
    private class NeighborIterator implements Iterator<Integer> {
        private int next = -1;
        private int current = -1;
        private final int src;

        public NeighborIterator(int src) {
            Objects.checkIndex(src, nodeCount);
            this.src = src;
            findNext();
        }

        @Override
        public boolean hasNext() {
            return next != -1;
        }

        @Override
        public Integer next() {


            if(!hasNext()) {
                throw new NoSuchElementException();
            }
            current = next;
            findNext();
            return current;
        }

        private void findNext() {
            next = -1;
            for(int i = current + 1; i < nodeCount(); i++) {
                if(getWeight(src, i).isPresent()) {
                    next = i;
                    return;
                }
            }
        }
    }
}
```

7. Expliquer le fonctionnement précis de la méthode remove de l'interface Iterator. Implanter la méthode remove de l'itérateur.

La méthode `remove` retire l'élément renvoyé par next() de la collection.

On implémente la méthode `remove` dans `NeighborIterator`. Pour cela, il suffit de placer l'élément à l'index de `current` à `null`:

```
public final MatrixGraph<T> {
    /* [...] */
```

```
    private class NeighborIterator implements Iterator<Integer> {
        /* [...] */
        @Override
        public void remove() {
            if(current == -1) {
                throw new IllegalStateException();
            }
            array[nodeCount * src + current] = null;
            current = -1;
        }
    }
}
```

8. On souhaite ajouter une méthode forEachEdge qui prend en paramètre un index d'un nœud et une fonction qui est appel cette fonction avec chaque arc sortant de ce nœud. Pour cela, nous allons, dans un premier temps, définir le type Graph.Edge à l'intérieur de l'interface Graph. Un Graph.Edge est définie par un entier src, un entier dst et un poids weight.

On ajoute le record `Edge` dans l'interface `Graph` :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    /* [...] */
   record Edge<T>(int src, int dst, T weight) {
       public Edge {
           Objects.requireNonNull(weight);
       }
   }
}
```

Puis on implémente la méthode `forEachEdge` dans directement dans l'interface `Graph` :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    /* [...] */
    default void forEachEdge(int src, Consumer<? super Edge<T>> consumer) {
        Objects.requireNonNull(consumer);
        Objects.checkIndex(src, nodeCount());

        for(int i = 0; i < nodeCount(); i++) {
            var weight = getWeight(src, i);
            if(weight.isPresent()) {
                consumer.accept(new Edge<T>(src, i, weight.get()));
            }
        }
    }
}
```

9. Enfin, on souhaite écrire une méthode edges qui renvoie tous les arcs du graphe sous forme d'un stream. L'idée ici n'est pas de réimplanter son propre stream (c'est prévu dans la suite du cours) mais

de créer un stream sur tous les nœuds (sous forme d'entier) puis pour chaque nœud de renvoyer tous les arcs en réutilisant la méthode forEachEdge que l'on vient d'écrire.

On implémente la méthode `edges` dans directement dans l'interface `Graph` :

```
public sealed interface Graph<T> permits MatrixGraph<T> {
    /* [...] */
    default Stream<Edge<T>> edges() {
       var edges = new ArrayList<Edge<T>>();
       for(int i = 0; i < nodeCount(); i++) {
           forEachEdge(i, edges::add);
       }
       return edges.stream();
    }
}
```