UNIVERSITÉ D'AVIGNON
ET DES PAYS DE VAUCLUSE

C E N T R E
D'ENSEIGNEMENT
ET DE RECHERCHE
EN INFORMATIQUE

Licence Informatique
Spécialité Ingénierie du Logiciel
UE s5 - projet de programmation

>>>

Tutoriel structure de données

11 mai 2018
G07

CLEMENT Mathieu
FERRER Gregory
HOLSTEIN Kélian
MOLLIER Pablo

CERI - LIA
339 chemin des Meinajariès
BP 1228
84911 AVIGNON Cedex 9
France

Tél. +33 (0)4 90 84 35 00
Fax +33 (0)4 90 84 35 01

http://ceri.univ-avignon.fr

Sommaire

Titre 1
Sommaire 2
1 Introduction 3
2 Créer un graphe 3
3 Convertir un graphe 4
4 Manipuler noeuds et liens 4
4.1 Node 4
4.2 Link 5
5 Créer sa propre mesure 5
5.1 Explications 6
5.2 Conventions de codage 6
6 Utiliser sa propre mesure 8
7 Lire un graphe à partir d'un fichier 9
7.1 Fichier EdgeList 9
7.2 Fichier GEXF 9
7.3 Fichier GraphML 11
8 Réaliser un benchmark 12
8.1 Méthodes de la classe `Benchmark` 12
8.2 Exemple d'utilisation 12

1 Introduction

Ce document est un tutoriel d'utilisation de la structure de données, pour créer ses centralités, manipuler des graphes, etc... N'hésitez pas à regarder le code pour mieux comprendre le fonctionnement. Nous vous conseillons de connaitre les templates (les templates Java) pour bien comprendre

2 Créer un graphe

La nature d'un graphe est définit par le couple noeud/lien. En fonction des types de noeud/-lien, la nature du graphe sera changée. Par exemple un graphe qui possède comme type de noeud `WeightedNode`, sera un graphe pondéré.
La classe qui gère les graphes est `AbstractGraph`. Celle-ci possède toutes les méthodes de base. Ensuite deux autres classes héritent de celle-ci pour définir si les liens sont orientés ou pas :

- Les `AbstractSimpleGraph` sont des graphes simples.
- Les `AbstractDirectedGraph` sont des graphes orientés.

Ainsi la notation "bas-niveau" des graphes est de la forme :

```
// Simple graph with weighted nodes
AbstractSimpleGraph<WeightedNode, Link>

// And so on for other graphs
```

Pour éviter la syntax template (qui peut être lourde par moment) et pour simplifier l'utilisation, une classe est disponible par type de graphe :

- `SimpleGraph`, pour les graphes simple
- `WeightedGraph`, pour les graphes pondérés non orientés
- `SpatialGraph`, pour les graphes spatiaux non orientés
- `SpatialWeightedGraph`, pour les graphes spatiaux et pondérés non orientés

- `DirectedGraph`, pour les graphes orientés
- `DirectedWeightedGraph`, pour les graphes pondérés et orientés
- `DirectedSpatialGraph`, pour les graphes spatiaux et orientés
- `DirectedSpatialWeightedGraph`, pour les graphes pondérés, spatiaux et orientés

Chacune de ces classes propose une méthode `createNode()` qui permet de créer un noeud. Ainsi que la méthode `createLink()` qui permet de créer des liens entre les noeuds. Celle-ci prend en paramètre le noeud source et le noeud de destination. Il est possible de passer deux integers qui représente l'ID des noeuds, ou directement les noeuds concernés.
De plus, pour les graphes pondérés et les graphes spatiaux, il est possible de préciser le poids du lien ou la position du noeud à partir de ces fonctions :

```
WeightedGraph graph = new WeightedGraph();

// Create three nodes
Node node0 = graph.createNode();
Node node1 = graph.createNode();
Node node2 = graph.createNode();

// Link node 0 and 1 by ID
graph.createLink(0, 1);

// Link node 1 and 0
graph.createLink(node1, node0);
```

```
// Link node 1 & 2 and set link weight to 60
graph.createLink(1, 2, 60);

SpatialGraph graph1 = new SpatialGraph();
// Create a spatial node at [5; 10]
graph1.createNode(5, 10);
```

Les graphes se situent dans le package **fr.univavignon.graphcentr.g07.graphs**.

3 Convertir un graphe

Un graphe peut être convertis sous forme de matrice d'ajacence ou matrice d'incidence ou liste d'adjacence. Pour cela deux méthodes sont mises à disposition :

- `toAdjacencyMatrix()`
- `toIncidenceMatrix()`

Ces deux fonctions retourne un tableau en deux dimensions :

```
SimpleGraph graph = new SimpleGraph();

graph.createNode();
graph.createNode();
graph.createNode();

graph.createLink(0, 1);
graph.createLink(1, 2);
graph.createLink(2, 0);

double[][] adjacencyMatrix = graph.toAdjacencyMatrix();

// Do your stuff
```

Ces matrices d'adjacences ou d'incidences s'adapte au type de graphe : si le graphe est pondéré, les matrices contiendront le poids des liens.

4 Manipuler noeuds et liens

Lors de la création de vos mesures, vous allez certainement avoir besoin de manipuler des noeuds ou liens. Une petite description des noeuds / liens avec les méthodes associés.

4.1 Node

`Node` est la classe de base de tous les noeuds. Elle contient un indice unique. Le graphe qui contient ce noeud propose plusieurs méthodes (NodeType corespond au type de noeud du graphe) :

- `removeNode(NodeType inNode)` / `removeNode(int inIndex)` : Permet de supprimer un noeud du graphe
- `getNodeAt(int inIndex)` : Retourne le noeud à l'index donné
- `getNodeIndex(NodeType inNode)` : Retourne l'index du noeud
- `getNodes()` : Retourne tous les noeuds du graphe (Non modifiable !)
- `getNodeCount()` : Retourne le nombre de noeud
- `isAdjacentTo(NodeType inSourceNode, NodeType inDestinationNode)` : Retourne vrai si les noeuds sont adjacents

En fonction de la nature du graphe (pondéré ou simple), d'autres méthodes sont disponible. Pour les graphes simples :

- `getNodeDegree(NodeType inNode)` : Retourne le degré d'un noeud
- `getNodesDegree()` : Retourne un tableau des degrés de chaque noeud

Pour les graphes pondérés :

- `getIncomingDegree(NodeType inNode)` : Retourne le degré entrant d'un noeud
- `getOutgoingDegree(NodeType inNode)` : Retourne le degré sortant d'un noeud
- `getNodesIncomingDegree()` : Retourne un tableau des degrés entrant de chaque noeud
- `getNodesOutgoingDegree()` : Retourne un tableau des degrés sortant de chaque noeud

4.2 Link

Link est la classe de base de tous les liens. Elle contient deux indices uniques, pour le noeud source et le noeud de destination.
Comme pour les noeuds (cf. section 4.1), les graphes proposent plusieurs méthodes :

- `removeLink(LinkType inLink)` / `removeLink(int inSourceNodeID, int inDestinationNodeID)` : Supprime le lien du graphe
- `getLinkCount()` : Retourne le nombre de lien
- `getNodeLinks(NodeType inNode)` : Retourne la liste des liens du noeud (Non modifiable !)

Les noeuds/liens se situent dans le package **fr.univavignon.graphcentr.g07.core**.

5 Créer sa propre mesure

Dans la structure de données, une mesure est une classe qui implémente **une** des interfaces avec le suffixe "Centrality". Ces interfaces ont toutes une méthode `evaluate()` qui prend en paramètre un type de graphe en fonction de leur nom :

- `SimpleCentrality` pour les graphes simples, donc `evaluate(SimpleGraph inGraph)`
- `DirectedCentrality` pour les graphes orientés, donc `evaluate(DirectedGraph inGraph)`

Et ainsi de suite pour tous les types de graphe mentionnés (cf. section 2). Il est donc possible pour une mesure de fonctionner sur plusieurs types de graphes.
Un exemple de création d'une mesure sur un graphe simple :

```
class MyAlgorithm implements SimpleCentrality
{
   /**
    * Evaluate given simple-like-graph
    * @param inGraph Graph to evaluate
    * @return AlgorithmResult
    */
    @Override
   CentralityResult evaluate(SimpleGraph inGraph)
   {
      CentralityResult Result = new CentralityResult();

        // Do your stuff

        // Add a result
        Result.add(32.0);

        return Result;
    }
}
```

Si votre mesure à besoin de manipuler plusieurs types de graphes, il faudra créer une classe pour chaque type.

Les interfaces se situent dans le package **fr.univavignon.graphcentr.g07.centrality**.

5.1 Explications

La méthode `evaluate(SimpleGraph inGraph)` vient de l'interface `SimpleCentrality` :

```
class MyAlgorithm implements SimpleCentrality
{
    @Override
   CentralityResult evaluate(SimpleGraph inGraph)
   {
    }
}
```

Les méthodes `evaluate()` retourne un `CentralityResult`. Il s'agit d'une classe de type `ArrayList<Double>` qui permet de stocker les résultats des centralités. Des méthodes seront rajoutées plus tard dans cette classe pour gérer la normalisation des centralités par exemple.

5.2 Conventions de codage

Quelques conventions à respecter quand vous réalisez votre mesure :

- Votre mesure aura certainement des paramètres, initialiez-les dans le constructeur de votre mesure (ou avec une valeur par défaut lors de la déclaration). Cela permettra de réaliser des tests à la chaîne de différentes mesures
- Définissez des méthodes pour configurer ces paramètres
- Essayez d'être clair sur le nom des variables / méthodes, et n'hésitez pas à mettre des commentaires pour expliquer
- Vous allez devoir réaliser des tests pour vérifier le bon fonctionnement de votre mesure. Tout se passe dans le package **fr.univavignon.graphcentr.tests**. Vous devrez créer un package du nom de votre groupe avec une classe contenant une méthode statique `execute()`. Celle-ci réalisera tous les tests de votre mesure. Une fois votre classe finie, vous devrez l'appeller dans la méthode `execute()` de la classe Test (qui se situe dans **fr.univavignon.graphcentr.tests**)
- Respectez les conventions Java !

Un exemple de centralité :

```
package fr.univavignon.graphcentr.g07

import fr.univavignon.graphcentr.g07.core.centrality.SimpleCentrality;

import fr.univavignon.graphcentr.g07.core.graphs.SimpleGraph;

/**
 * @author Holstein Kelian
 *
 * @brief My centrality description
 */
class MyCentrality implements SimpleCentrality
{
   /** Parameter1 description */
   private int parameter1 = 0;

    /**
     * Default constructor
     */
    MyAlgorithm()
```

```
    {
    }

    @Override
   CentralityResult evaluate(SimpleGraph inGraph)
   {
      CentralityResult result = new CentralityResult();

        // Do your stuff
        result.add(1.0);

        return result;
    }

    /**
    * Sets parameter1's value
    * @param inValue
    */
    public void setParamater1(int inValue)
    {
      parameter1 = inValue;
    }

    /**
    * Returns parameter1's value
    */
    public int getParamater1()
    {
      return parameter1;
    }
}
```

Pour les tests de votre centralité, seule la méthode statique `execute()` est imposé. Libre à vous d'implémenter la gestion des erreurs, nous vous conseillons d'utiliser des exceptions pour gérer ça (vous pouvez jeter un oeil à notre code dans les classes `Graph` et `Reader`). Un exemple de test pour une centralité, ainsi que son intégration dans les tests de la librairie :

```
/**
 * @author Holstein Kelian
 *
 * @brief Tests on my centrality
 */
class MyCentralityTest
{
    /**
    * Tests on MyCentrality
    */
   public static void execute()
   {
      SimpleGraph graph = new SimpleGraph();
        // Add nodes & links

        MyCentrality centrality = new MyCentrality();
        // Test with default parameters
        centrality.evaluate(graph);
        // Check if results are ok

        // Then test with other parameters
```

```
        centrality.setParameter1(10);

        // And so on
    }
}
```

Et l'intégration de votre test, il s'agit de juste rajouter votre test après le commentaire '// Tests on "centralitites"' :

```
package fr.univavignon.graphcentr.tests;

import fr.univavignon.graphcentr.tests.g07.CoreTest;

/**
 *
 * @author Holstein Kelian
 *
 * @brief Class used to execute all library's test.
 */
public class Test
{
   /**
    * Execute all library's tests
    */
   public static void execute()
   {
      // Core tests (G07)
      CoreTest.execute();

      // Tests on "centralitites"
        MyCentralityTest.execute();
   }
}
```

6 Utiliser sa propre mesure

Pour utiliser votre mesure vous n'avez qu'à créer un graphe compatible avec votre mesure, ajouter des noeuds, des liens et simplement appeller votre mesure :

```
public class Main
{
   public static void main(String[] args)
   {
        SimpleGraph graph = new SimpleGraph();

        // Create 3 nodes
        graph.createNode();
        graph.createNode();
        graph.createNode();

        // Link node 0 with 1, etc...
        graph.createLink(0, 1);
        graph.createLink(0, 2);
        graph.createLink(1, 2);

        MyCentrality myCentrality = new MyCentrality();
```

```
        // Configure your centrality
        // Evaluate your algorithm
        CentralityResult result = myCentrality.evaluate(graph);

        // And print result
    }
}
```

7 Lire un graphe à partir d'un fichier

Il existe trois types de fichier gérés par la librairie :

- EdgeList
- GEXF
- GraphML

À chaque type fichier est associé un `Reader`. Pour les fichiers EdgeList il s'agit de `EdgeListReader`, pour les fichiers GEXF il s'agit de `GEXFReader` et `GraphMLReader` pour les fichiers GraphML.
Chacune de ces classes possède une méthode `updateFromFile` qui prend en paramètre le fichier à lire et le graphe à mettre à jour. Cette méthode permet d'extraire les noeuds/liens des fichiers et de les rajouter au graphe. La nature du graphe sera prise en compte **automatiquement**, si le graphe est orienté, les liens lu le seront, si le graphe est pondéré, les liens auront un poid etc...

7.1 Fichier EdgeList

Les fichiers EdgeList (comme le nom l'indique) est un fichier qui contient des listes de liens.

Attention : Pour les fichiers EdgeList, un autre argument est à rajouter il s'agit du séparateur entre les différents noeuds. De plus il ne possède pas d'information concernant le poid des liens ou la position des noeuds. Si vous travaillez sur des graphes pondérés/spatiaux regardez les parties 7.2 et 7.3.
Un exemple de lecture d'un graphe à partir d'un fichier EdgeList :

```
EdgeList.txt:

0, 1, 2
1, 2
```

```
public class Main
{
    public static void main(String[] args)
    {
        SimpleGraph graph = new SimpleGraph();
          EdgeListReader reader = new EdgeListReader();
        reader.updateFromFile("EdgeList.txt", ',', graph);

          // Do stuff
    }
}
```

7.2 Fichier GEXF

Les fichiers GEXF permettent de rajouter des informations en plus sur les noeuds/liens. Le *GEXFReader* va rechercher ces informations à travers le fichier (si le graphe en a besoin bien sûr). Vous

devrez nommer, pour gérer le poid, un attribut 'Weight' et pour la position deux attributs 'X' et 'Y' (non sensible à la casse).
Attention : Les fichiers GEXF contiennent beacoup d'informations qui seront ignorés par la librairie.
Un exemple d'un fichier GEXF avec la gestion du poid sur les liens :

```
<gexf>
  <graph>

    <!-- Create an attribute for edges named "Weight". Its default value is 5.0 -->
    <attributes class="edge">
      <attribute id="0" title="Weight">
       <default>5.0</default>
      </attribute>
    </attributes>

    <!-- Create 3 nodes -->
    <nodes>
      <node id="0"/>
      <node id="1"/>
      <node id="2"/>
    </nodes>

    <!-- Create two links between 0->1 and 1->2 -->
    <edges>
      <edge id="0" source="0" target="1">
        <!-- Set link weight to 15.0 -->
        <attvalues>
          <attvalue for="0" value="15.0" />
        </attvalues>
      </edge>
      <edge id="1" source="1" target="2">
        <!-- Set link weight to 20.0 -->
        <attvalues>
          <attvalue for="0" value="20.0" />
        </attvalues>
      </edge>
    </edges>
  </graph>
</gexf>
```

Il est important de mettre des IDs différents pour les attributs afin de les différencier (Ici l'id 0 pour le poid des noeuds). Cette valeur est un chaîne de caractères, vous pouvez donc mettre n'importe quoi. Cet ID sera réutilisé lors de l'affectation des valeurs :

```
<attvalue for="0" value="15.0" />
```

for="0" indique qu'on affecte la valeur 15 à l'attribut 0, donc le poid.
Pour gérer la position il s'agit du même code mais avec des attributs sur les noeuds :

```
<attributes class="node">
  <attribute id="0" title="X">
   <default>0.0</default>
  </attribute>
  <attribute id="0" title="Y">
   <default>0.0</default>
  </attribute>
</attributes>
```

Un exemple de lecture d'un fichier GEXF :

```
public class Main
{
   public static void main(String[] args)
   {
      WeightedGraph graph = new WeightedGraph();
      GEXFReader.updateFromFile("GEXFFile.txt", graph);

        // Do stuff
   }
}
```

7.3 Fichier GraphML

Les fichiers GraphML ont les mêmes fonctionnalités que les fichiers GEXF, seulement la syntax change.
Attention : Lors de la déclaration des attributs dans les fichiers GraphML, un type de valeur doit être mis (double, string etc...). Ceci sera ignoré par la librairie, actuellement nous avons que des types double (pour le poid et la position).
Pour définir des attributs avec leurs valeurs par défaut :

```
<!-- Edge attribut -->
<key id="d0" for="edge" attr.name="weight"> 0.0 </key>
<!-- Node attribut -->
<key id="d0" for="node" attr.name="X"> 0.0 </key>
```

Un exemple d'un fichier GraphML avec la gestion de la position sur les noeuds :

```
<graphml>
  <key id="d1" for="node" attr.name="X" attr.type="double"> 0.0 </key>
  <key id="d2" for="node" attr.name="Y" attr.type="double"> 0.0</key>
  <graph id="G">
    <node id="n0">
      <data key="d1">10.0</data>
    </node>
    <node id="n1"/>
    <node id="n2">
      <data key="d2">20.0</data>
    </node>

    <edge id="e0" source="n0" target="n2"/>
    <edge id="e1" source="n0" target="n1"/>
  </graph>
</graphml>
```

Un exemple de lecture d'un fichier GraphML :

```
public class Main
{
   public static void main(String[] args)
   {
      SpatialGraph graph = new SpatialGraph();
      GraphMLReader.updateFromFile("GraphMLFile.txt", graph);

        // Do stuff
   }
```

```
}
```

Les readers se situent dans le package **fr.univavignon.graphcentr.g07.readers**.

8 Réaliser un benchmark

La classe `Benchmark` permet de récuperer le nombre d'itérations et le temps que met une fonction. Elle propose un système de snapshots qui permet de définir des points d'extractions. Un snapshot contiendra le nombre d'itérations et le temps écoulé par rapport au précédent snapshot (donc un autre snapshot ou simplement le point de départ).
Cette classe pourra vous être utile lorsque vous réaliserez vos tests de performances, vérifier que la compléxité est bien celle que vous avez indiqué etc...

8.1 Méthodes de la classe `Benchmark`

Important : Il s'agit de méthode statiques (pour permettre une façon très souple de faire un benchmark).
Liste des méthodes :

- `Benchmark.start()` : Initialize le benchmark, cette méthode doit être appelée **avant tout autre appel d'autre méthodes**. Cette méthode sert aussi à reset le précédent benchmark.
- `Benchmark.stop()` : Arrête le benchmark et sauvegarde le temps d'éxécution totale ainsi que le nombre d'itérations total.
- `Benchmark.addSnapshot(String inSnapshotName)` : Rajoute un snapshot en associant un nom à celui-ci. Comme dis précédemment les snapshots sont relatifs, si un nouveau snapshot est créé, le prochain aura son nombre d'itérations à 0, de même pour son temps.
- `Benchmark.addIteration()` : Rajoute une itération au snapshot courant (donc le dernier créé).
- `Benchmark.printSnapshots()` : Affiche dans la console l'ensemble du benchmark : Tous les snapshots avec leurs temps / nombre d'itérations, le temps total d'éxécution et le nombre total d'itérations. Il faut appeller cette méthode **après** `Benchmark.stop()`, sinon vous n'aurez pas les temps totaux ainsi que les informations du snapshot courant.
- `Benchmark.saveToFile(String inFileName)` : Réalise la même opération que `Benchmark.printSnapshots()` mais l'enregistre dans le fichier donné. Cette méthode peut être utile pour réaliser des graphiques par exemple (Les différents champs sont séparés par un espace).

8.2 Exemple d'utilisation

```
import fr.univavignon.graphcentr.g07.core.utility.Benchmark;

public class Main
{
   /**
     * External function that does a loop
     */
   public static void foo()
    {
      for(int i = 0; i < 100; i++)
         Benchmark.addIteration();
    }

   public static void main(String[] args)
```

```
  {
     // Start benchmark
     Benchmark.start();
       // Create first snapshot point, called "First loop"
       Benchmark.addSnapshot("First loop");

       // Create an O(n^2) loop
       for(int i = 0; i < 100; i++)
       {
        for(int j = 0; j < 100; j++)
           {
              Benchmark.addIteration();
           }
       }

       // Create a second snapshot. Doing this will finish "First loop" snapshot
       Benchmark.addSnapshot("foo()");
       // Call foo function
       foo();

       // Finishes benchmark
       Benchmark.stop();
       // And print snapshots
       Benchmark.printSnapshots();
  }
}
```

Ce qui donne lorsque l'on exécute ce code :

```
--------------- Benchmark ---------------
Snapshots :
Name       |Ms          |Iteration count
First loop |0.893697ms |10000
foo()      |0.02304ms   |100

Full execution time : 1.198337ms
Total iteration count : 10100
------------ End of Benchmark -----------
```

Note : Il est normal que le temps total ne correspond pas à la somme des temps des snapshots, il s'agit du temps écoulé entre `Benchmark.start()` et `Benchmark.stop()`.
La classe `Benchmark` se situent dans le package **fr.univavignon.graphcentr.g07.core.utility.Benchmark**.