Travaux pratiques de Graphes : introduction

Université du Havre - IUT - Département informatique

1 Introduction

Pour les séances de travaux pratiques de cette année, nous vous proposons d'utiliser une bibliothèque de manipulation de graphes qui permet de visualiser les graphes ainsi que les modifications que vous y apporterez durant leur traitement. Cette première séance a pour principaux objectifs :

1. la mise en place de votre environnement
2. la création et la visualisation de deux graphes simples
3. quelques manipulations de base pour modifier l'aspect du graphe

2 Installation

2.1 Premières vérifications

Avant de commencer, vérifiez que java est installé sur votre poste et que vous êtes en mesure de l'utiliser.

Si vous avez déjà effectué ce type de vérification précédemment pour d'autres modules, vous pouvez passer directement à la section 2.2.

1. ouvrez un terminal
2. tapez la commande `javac`
3. vous devriez obtenir un message du type :

```
Usage : javac <options> <source files>
where possible options include :
  -g              Generate all debugging info
  -g :none        Generate no debugging info
...
  -Werror         Terminate compilation if warnings occur
  @<filename>     Read options and filenames from file
```

A partir de cette étape nous pouvons passer à la vérification suivante.

1. ouvrez un éditeur de texte (`gedit` par exemple)
2. saisissez le code java suivant :

```java
// Test.java

public class Test {
   public static void main(String args[]) {
       System.out.println("ok javac et java fonctionnent");
   }
}
```

3. sauvez-le sous le nom de Test.java
4. compilez-le avec la commande `javac Test.java`
5. exécutez-le avec la commande `java Test`
6. le message *ok javac et java fonctionnent* doit s'afficher à l'écran

Nous pouvons passer à l'installation de la bibliothèque.

2.2 Récupération et installation de la bibliothèque

1. allez sur l'intranet dans l'espace pédagogique et choisissez `M2201 Graphes et langages` dans la rubrique *Approfondissement en culture scientifique, sociale et humaine*.
2. récupérez les trois fichiers jar `gs-core.jar, gs-algo.jar et gs-ui.jar`
3. à partir de maintenant nous supposons que vous avez un répertoire `$HOME/TP/graphes` qui contient les sous-répertoires
   — `lib/`
   — `data/`
   — `tp0/`
   — `tp1/` ...
4. déplacez les trois fichiers jar dans le répertoire `graphes/lib/`
5. modifiez votre variable d'environnement CLASSPATH :

   → éditez le fichier $\sim/.di_shrc_priv$ en ajoutant à la fin les lignes suivantes...

   ⚠ en dehors de l'espace entre export et CLASSPATH chaque ligne ne comporte AUCUN espace

   ⚠ la dernière ligne se termine avec un "." après le ":"
   export CLASSPATH=$CLASSPATH:$HOME/TP/graphes/lib/gs-core.jar
   export CLASSPATH=$CLASSPATH:$HOME/TP/graphes/lib/gs-ui.jar
   export CLASSPATH=$CLASSPATH:$HOME/TP/graphes/lib/gs-algo.jar
   export CLASSPATH=$CLASSPATH:.
6. sauvegardez le fichier ainsi modifié.
7. pour celles et ceux qui auraient oublié, la variable d'environnement CLASSPATH permet à java de savoir dans quels répertoires il doit chercher les .jar et les .class
8. si cette variable est mal positionnée, vous aurez un message d'erreur indiquant que la classe n'a pas été trouvée.

3 Premiers pas

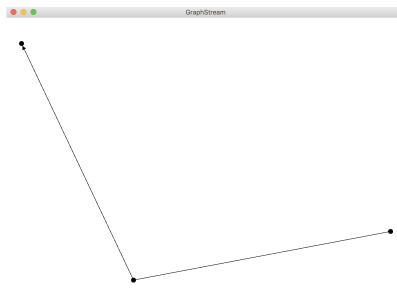
3.1 Premier graphe

Essayez le premier code suivant qui doit vous afficher un graphe très simple formé de trois sommets, d'un arc et d'une arête.

```
// PremierGraphe.java
import org.graphstream.graph.implementations.SingleGraph ;

public class PremierGraphe {
    public static void main(String args[]) {
        SingleGraph graph = new SingleGraph("Premier Graphe") ;
        graph.display() ;
        graph.addNode("a") ;
        graph.addNode("b") ;
        graph.addNode("c") ;
        graph.addEdge("ab","a","b",true) ;
        graph.addEdge("ac", "a","c") ;
    }
}
```

Son exécution doit vous faire apparaître une simple fenêtre avec le graphe.



3.2 Méthodes de base

— **new SingleGraph("id")** permet de créer une instance d'un graphe simple.
sur ce graphe il est possible d'invoquer de multiple méthodes permettant d'agir sur son affichage, sur son ensemble de sommets et d'arêtes (ou d'arcs si le graphe est orienté).

— **graph.display()** permet de visualiser le graphe, la disposition des sommets se fait automatiquement de manière à rendre le graphe le plus lisible possible nous verrons plus tard qu'il est possible de débrayer cette disposition automatique.

— **addNode(identifiant)** permet d'ajouter un nouveau sommet au graphe.
⚠ deux sommets ne peuvent pas avoir le même identifiant.

— **addEdge(identifiant, id sommet origine, id sommet extrémité, orienté)** permet de créer une arête si le paramètre `orienté` est absent ou est positionné à `false`, et permet de créer un arc si le paramètre `orienté` est positionné à `true`.

3.3 Ajouter des attributs

Lorsque nous manipulons des graphes il est souvent intéressant d'afficher davantage d'information que sa simple structure (on parle de topologie). Dans le code suivant nous allons afficher l'identifiant

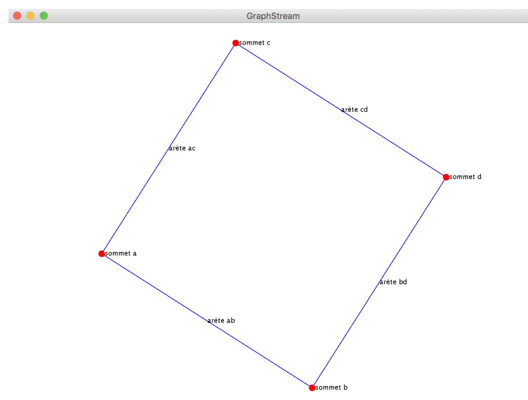
des sommets et des arêtes et nous allons changer la couleur des sommets et des arêtes. Mais, pour cela, nous avons besoin de parcourir l'ensemble des sommets et des arêtes du graphe ce qui peut se faire en invoquant sur l'instance du graphe créé les méthodes `getNodeSet()` et `getEdgeSet()`. Tous les sommets vont devenir rouges et toutes les arêtes vont devenir bleues.

```java
// SecondGraphe.java
import org.graphstream.graph.implementations.SingleGraph ;
import org.graphstream.graph.Node ;
import org.graphstream.graph.Edge ;

public class SecondGraphe {
  public static void main(String args[]) {
    SingleGraph graph = new SingleGraph("Second Graphe") ;
    graph.display() ;
    graph.addNode("a") ;
    graph.addNode("b") ;
    graph.addNode("c") ;
    graph.addNode("d") ;
    graph.addEdge("ab","a","b") ;
    graph.addEdge("ac","a","c") ;
    graph.addEdge("cd","c","d") ;
    graph.addEdge("bd","b","d") ;

    // iteration sur l'ensemble des sommets
    for(Node n :graph.getNodeSet()) {
      n.setAttribute("ui.style","fill-color :red ;") ;
      n.setAttribute("label","sommet "+n.getId()) ;
    }

    // iteration sur l'ensemble des aretes
    for(Edge e :graph.getEdgeSet()) {
      e.setAttribute("ui.style","fill-color :blue ;") ;
      e.setAttribute("label","arete "+e.getId()) ;
    }
  }
}
```



3.4 Méthodes de base (suite)

- **graph.getEdgeSet()** retourne l'ensemble des liens du graphe (arcs et arêtes).
- **graph.getNodeSet()** retourne l'ensemble des sommets du graphe.
- **x.setAttribute("nom",valeur)** permet d'associer un attribut à un élément du graphe qu'il s'agisse d'un sommet ou d'une arête (ou d'un arc).
- le nom de l'attribut est une chaîne de caractères (un String), la valeur peut être un Objet, une chaîne de caractères (String), un entier...
- certains noms d'attributs sont prédéfinis : *ui.style* ou *label* par exemple (on vous les donnera systématiquement lorsque ce sera le cas) :

  $\rightarrow$ **e.setAttribute("label",chaîne)** où `chaîne` est une instance de String qui sera affichée à côté de l'élément e.

  $\rightarrow$ **e.setAttribute("ui.style","size :5 ;")** permet de définir la taille d'un sommet ou l'épaisseur d'un arc/une arête.

  $\rightarrow$ **e.setAttribute("ui.style","fill-color:red ;")**

  - pour ce dernier cas il est également possible de coder la couleur en donnant les proportions du rouge du vert et du bleu (RGB) sous la forme : **#rrggbb** où `rr` varie entre 00 et FF. C'est de l'hexadécimal chaque chiffre peut prendre sa valeur parmi $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$.
- **x.getAttribute("nom")** permet de récupérer la valeur d'un attribut du graphe qu'il s'agisse d'un sommet ou d'une arête, à partir du nom de l'attribut.

4 Quelques opérations simples

4.1 Degré moyen d'un graphe

En vous inspirant de ce qui a été fait dans la section 3.3, et des méthodes présentées dans la section 5.3.2, proposez un code qui permette d'afficher le degré moyen de votre graphe.

4.2 Voisins

De nombreux traitements nécessitent de lister l'ensemble des voisins d'un sommet particulier. En utilisant les primitives de la section 5.3.2 et en vous inspirant de ce qui a été fait précédemment, choisissez un sommet aléatoire dans votre graphe, colorez-le en rouge (ou une autre couleur de votre choix) et colorez l'ensemble de ses voisins en bleu (ou une autre couleur de votre choix différente).

4.3 Méthodes de base (suite)

- **n.getDegree()** retourne un entier qui représente le degré du sommet n
- **Node noeudChoisiAleatoirement = randomNode(graphe)**
  méthode permettant de choisir un sommet aléatoire du graphe, nécessite d'ajouter
  import org.graphstream.algorithm.Toolkit ;
- **Iterator<Node> lesVoisins = sommet.getNeighborNodeIterator()**
  méthode permettant de lister les voisins d'un sommet à l'aide d'un Iterator, nécessite d'ajouter
  import java.util.Iterator ;

Exemple d'utilisation d'un `Iterator` :

```
import java.util.Iterator;
```

```java
import java.util.Arrays;

public class ExempleIterator {

   public static void main(String args[]) {
     Iterator<String> lesChaines = Arrays.asList(args).iterator();
     while(lesChaines.hasNext()) {
        String laChaine = lesChaines.next();
        System.out.println("--->"+laChaine);
     }
   }
}
```

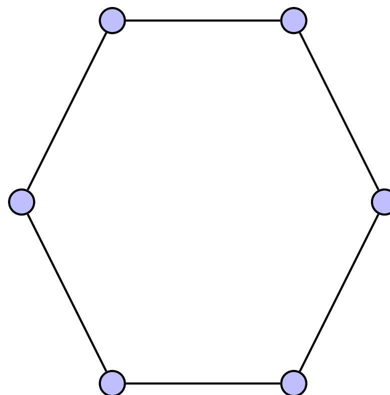
5 Génération de graphes

Il peut être assez fastidieux de construire un graphe en codant chaque ajout de sommet et d'arête. Il est donc intéressant de générer des graphes en utilisant certaines de leurs propriétés.

5.1 Génération d'anneaux

5.1.1 Rappel de définition

définition : un **anneau** est un graphe connexe dont tous les sommets sont de degré 2.



5.1.2 Algorithme

Pour les algorithmes demandés on pourra utliser les méthodes :

— `creerGraphe()` qui retourne g une instance de graphe vide.

— `g.ajouterSommet(s)` où s représente l'identifiant du sommet

— `g.ajouterArete(`s_i`,`s_j`)` qui ajoute une arête entre les sommets s_i et s_j

Proposez un algorithme qui permette de générer un anneau d'ordre n (n est une donnée d'entrée de votre algorithme).

Lorsque vous aurez terminé, implémentez l'algorithme sous GraphStream.

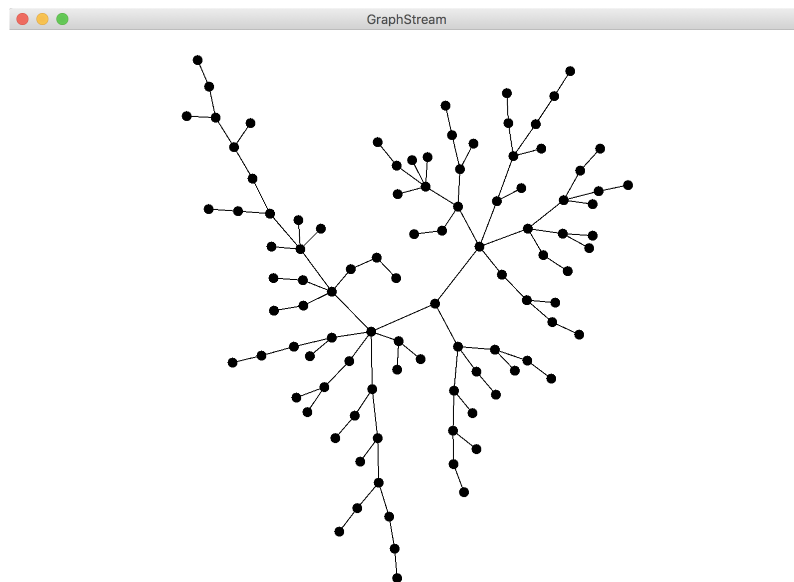
5.2 Génération d'arbres

Nous nous intéressons à la génération d'un arbre. Un arbre est un graphe connexe dont le nombre d'arêtes est égal au nombre de sommets moins un.

Notre générateur ne prendra en entrée qu'un seul paramètre : le nombre de sommets.
Pour l'algorithme demandé on pourra utliser les méthodes :

— `creerGraphe()` qui retourne g une instance de graphe vide.
— `g.ajouterSommet(s)` où s représente l'identifiant du sommet
— `g.ajouterArete(`s_i`,`s_j`)` qui ajoute une arête entre les sommets s_i et s_j
— $s_i \leftarrow$ `g.tirerSommetAleatoire()` où s représente un sommet tiré aléatoirement dans le graphe g

Proposez un algorithme qui permette de générer un arbre d'ordre n (n est une donnée d'entrée de votre algorithme).
Lorsque vous aurez terminé, implémentez votre algorithme en utilisant GraphStream.



5.3 Génération de graphes complets

5.3.1 Algorithme

Un graphe complet est un graphe connexe dans lequel chaque sommet est lié à tous les autres sommets.
Notre générateur ne prendra en entrée qu'un seul paramètre : le nombre de sommets.

Pour l'algorithme demandé on pourra utliser les méthodes :

— `creerGraphe()` qui retourne g une instance de graphe vide.
— `g.ajouterSommet(s)` où s représente l'identifiant du sommet
— `g.ajouterArete(`s_i`,`s_j`)` qui ajoute une arête entre les sommets s_i et s_j
— `s.possedeUneAreteAvec(t)` qui renvoie vrai si une arête existe entre les sommets s et t et faux sinon.

Proposez un algorithme qui permette de générer un graphe complet d'ordre n (n est une donnée d'entrée de votre algorithme).

Lorsque vous aurez terminé, implémentez votre algorithme en utilisant GraphStream.

5.3.2 Méthodes de base (suite)

— **n.hasEdgeBetween(t)** retourne true si il existe une arête entre n et t.