Quentin Baert

AeA : Calcul d'arbres recouvrants de coû minimum

Architecture

```
src
├── main
|   ├── java
|   ├── resources
|   └── scala
|       └── graphe
|           ├── Edge.scala
|           ├── Graph.scala
|           ├── GraphBuilder.scala
|           ├── GraphGenerator.scala
|           ├── GraphMST.scala
|           ├── TestPerfMST.scala
|           └── Vertex.scala
```

Avec :

- `Edge` : représente une arête d'un graphe
- `Vertex` : représente un sommet d'un graphe
- `Graph` : représente un graphe
- `GraphBuilder` : permet de construire un graphe depuis un fichier texte
- `GraphGenerator` : permet de générer un graphe aléatoire selon la méthode Erdos-Renyi
- `GraphMST` : main qui génére un graphe depuis un fichier texte et affiche son MST
- `TestPerfMST` : main qui permet de tester les performances des algorithmes implémentés sur plusieurs graphes générés aléatoirement

Algorithmes implémentés

Algorithme de Prim

L'arbre couvrant minimum d'un graphe généré avec l'algorithme de Prim s'obtient grâce à la méthode `Graph.getPrimMST()`. Pour que cette méthode se comporte correctement, il faut que le graphe soit connexe.

```
  /**
   * Donne l'arbre couvrant minimum du graphe grâce à l'algorithme de Prim
   *
   * @return arbre couvrant minimum du graphe
   */
  def getPrimMST: Graph = {
    // L'algorithme ainsi codé nécéssite que le graphe soit connexe
    require(this.isConnex)

    /*
     * v  : Ensemble des points marqués
     * e  : Ensemble des arêtes sortante de l'ensemble de points marqués
     * fe : Ensemble des arêtes à garder pour l'arbre couvrant minimum
     */
    def prim(v: Set[Vertex], e: Set[Edge], fe: Set[Edge]): Graph =
      if (this.vertices forall v.contains)
        new Graph(v, fe)
      else {
        // Arête avec le poids minimum
        val minEdge = e minBy (x => x.weight)
        // Extrémité de l'arête qui est déjà marqué
        val taggedVertex = if (v contains minEdge.v1) minEdge.v1 else minEdge.v2
        // Sommet à ajouter aux sommets marqués
        val vertex = minEdge other taggedVertex
        // Arêtes à ajouter à e
        val edgesToAdd =
          (this getVertexEdges vertex) filterNot (e => v contains (e other vertex)
)
        // Nouvel ensemble de sommets à considérer
        val newV = v + vertex
        // Nouvel ensemble d'arêtes à considérer
        val newE =
          (e ++ edgesToAdd) filterNot (
            e => (newV contains e.v1) && (newV contains e.v2)
          )

        prim(newV, newE, fe + minEdge)
      }

    val vertex = this.vertices.head
    val vertexEdges = this getVertexEdges vertex

    prim(Set(vertex), vertexEdges, Set())
  }
```

Algorithme de Kruskal

L'arbre couvrant minimum d'un graphe généré avec l'algorithme de Kruskal s'obtient grâce à la méthode `Graph.getKruskalMST()`.

```
  /**
   * Donne l'arbre couvrant minimum du graphe grâce à l'algorithme de Kruskal
   *
   * @return arbre couvrant minimum du graphe
   */
  def getKruskalMST: Graph = {
    /*
     * sets        : Ensemble des sommets du graphe, chacun associé à un identifia
nt
     * edges       : arête à ajouter à l'arbre couvrant
     * unusedEdges : arête à potentiellement ajouter à l'arbre couvrant
     */
    def kruskal(sets: List[(Vertex, Int)], edges: Set[Edge], unusedEdges: List[Edg
e]): Graph =
      // Si toutes les arêtes ont été parcourues, le graphe peut être renvoyé
      if (unusedEdges.isEmpty)
        new Graph(this.vertices, edges)
      else {
        // Fonction qui trouve l'identifiant d'un sommet
        def findSetOf(v: Vertex): Int = (sets filter (c => c._1 == v)).head._2
        // Arête à considérer pour cette itération
        val edge = unusedEdges.head
        // Extrémités de l'arête à considérer
        val (v1, v2) = (edge.v1, edge.v2)
        // Identifiants des extrémités
        val (i1, i2) = (findSetOf(v1), findSetOf(v2))

        // Si les identifiants sont différents, on peut ajouter l'arête (pas de cy
cle)
        if (i1 != i2) {
          /*
           * Les identifiants sont mis à jours :
           * Tous les sommets qui avaient l'identifiant i2 ont maintenant
           * l'identifiant i1
           */
          val newSets = sets map (c => if (c._2 == i2) (c._1, i1) else c)
          kruskal(newSets, edges + edge, unusedEdges.tail)
        }
        // Sinon, ajouter l'arête créerait un cycle, elle n'est pas ajoutée
        else
          kruskal(sets, edges, unusedEdges.tail)
      }

    // Chaque sommet possède son propre identifiant
    val initSets = (this.vertices zip (1 to this.vertices.size)).toList
```

```
    val orderedEdges = this.edges.toList sortWith ((e1, e2) => e1.weight < e2.weig
ht)

    kruskal(initSets, Set(), orderedEdges)
  }
```

Main

Affichage d'un graphe et de son MST

Le `jar` éxécutable `GraphMST.jar` permet d'imprimer un graphe et son MST sur la sortie standard. Pour tester, le fichier `graph.txt` correspond au graphe se trouvant à la fin du sujet de TP.

Exemple d'utilisation :

```
$ java -jar GraphMST.jar graph.txt

===== GRAPHE =====
8 -- 9 (1)
5 -- 9 (3)
1 -- 2 (7)
2 -- 3 (2)
4 -- 7 (6)
3 -- 5 (9)
4 -- 5 (19)
5 -- 8 (13)
3 -- 4 (21)
6 -- 10 (20)
5 -- 6 (8)
8 -- 12 (11)
9 -- 10 (12)
4 -- 8 (5)
8 -- 11 (4)
1 -- 3 (10)

===== PRIM =====
8 -- 9 (1)
5 -- 9 (3)
1 -- 2 (7)
2 -- 3 (2)
4 -- 7 (6)
3 -- 5 (9)
5 -- 6 (8)
8 -- 12 (11)
9 -- 10 (12)
4 -- 8 (5)
8 -- 11 (4)

===== KRUSKAL =====
8 -- 9 (1)
5 -- 9 (3)
1 -- 2 (7)
2 -- 3 (2)
4 -- 7 (6)
3 -- 5 (9)
5 -- 6 (8)
8 -- 12 (11)
9 -- 10 (12)
4 -- 8 (5)
8 -- 11 (4)
```

Tests de performances

Le `jar` éxécutable `TestPerfMST.jar` permet de tester les deux algorithmes sur un panel de graphe généré aléatoirement.
Pour s'assurer du bon fonctionnement de l'algorithme de Prim, il est conseillé de demander des graphes composés d'au moins 100 sommets.

Exemple d'utilisation pour 50 graphes de 100 sommets (pour chaque probabilité, 50 graphes à 100 arêtes sont générés) :

```
$java -jar TestPerfMST.jar 50 100

===== PROBA : 0.1 =====
----- PRIM -----
Temps moyen : 6 ms
----- KRUSKAL -----
Temps moyen : 3 ms

===== PROBA : 0.3 =====
----- PRIM -----
Temps moyen : 12 ms
----- KRUSKAL -----
Temps moyen : 2 ms

===== PROBA : 0.5 =====
----- PRIM -----
Temps moyen : 22 ms
----- KRUSKAL -----
Temps moyen : 4 ms

===== PROBA : 0.7 =====
----- PRIM -----
Temps moyen : 37 ms
----- KRUSKAL -----
Temps moyen : 6 ms

===== PROBA : 0.9 =====
----- PRIM -----
Temps moyen : 41 ms
----- KRUSKAL -----
Temps moyen : 7 ms
```

On remarque de l'algorithme de Kruskal est toujours plus efficace. Cela est sûrement dû à l'optimisation apporté à l'implémentation pour la détéction des cycles : chaque sommet est dans un groupe lors de l'éxécution de l'algorithme, si une arête sélectionnée lie deux sommets du même groupe, un cycle est détecté.

Test sur un graphe de taille 1000 :

```
$java -jar TestPerfMST.jar 1 1000

===== PROBA : 0.1 =====
----- PRIM -----
Temps moyen : 7797 ms
----- KRUSKAL -----
Temps moyen : 1450 ms

===== PROBA : 0.3 =====
----- PRIM -----
Temps moyen : 31620 ms
----- KRUSKAL -----
Temps moyen : 3430 ms

===== PROBA : 0.5 =====
----- PRIM -----
Temps moyen : 63924 ms
----- KRUSKAL -----
Temps moyen : 5938 ms

===== PROBA : 0.7 =====
----- PRIM -----
Temps moyen : 90302 ms
----- KRUSKAL -----
Temps moyen : 8493 ms

===== PROBA : 0.9 =====
----- PRIM -----
Temps moyen : 136272 ms
----- KRUSKAL -----
Temps moyen : 11725 ms
```

Même sur un plus gros exemple, l'algorithme de Kruskal reste largement (de l'ordre de 10 fois) plus performant.