Quentin Baert

AeA : Coloration

Architecture

```
src
├── main
|   ├── java
|   ├── resources
|   └── scala
|       ├── coloration
|       |   ├── Color.scala
|       |   └── ColorationResult.scala
|       └── graphe
|           ├── Edge.scala
|           ├── Graph.scala
|           ├── GraphBuilder.scala
|           ├── GraphGenerator.scala
|           ├── Main.scala
|           └── Vertex.scala
```

Avec :

- `graphe.Vertex` : représente un sommet d'un graphe
- `graphe.Edge` : représente une arête d'un graphe
- `graphe.Graph` : représente un graphe
- `graphe.GraphBuilder` : permet de construire un graphe depuis un fichier texte
- `graphe.GraphGenerator` : permet de générer un graphe aléatoire selon la méthode Erdos-Renyi
- `coloration.Color` : représente une couleur
- `coloration.ColorationResult` : représente le résultat d'une coloration de graphe

Algorithmes implémentés

Algorithme de coloration de base

L'algorithme naïf et celui de Welsh-Powell repose sur la même base, une méthode `baseColoration()` a donc été codée.

```
  /*
   * coloredVertices : sommets du graphe déjà colorés
   * colors          : couleurs pour l'instant utilisées dans le graphe
   * vertices        : sommets du graphe qu'il reste à colorer
   */
  private def baseColoration(
    coloredVertices: Map[Vertex, Color],
    colors: List[Color],
    vertices: List[Vertex]
  ): ColorationResult =
    // Si tous les sommets ont été colorés, la coloration est retournée
    if (vertices.isEmpty)
      new ColorationResult(coloredVertices, colors.size)
    else {
      // Sommet à colorer
      val vertex = vertices.head
      // Voisins du sommet à colorer
      val neighbours = this getVertexNeighbours vertex
      // Couleurs des voisins du sommet à colorer
      val neighboursColors = neighbours map (n =>
        if (coloredVertices contains n) coloredVertices(n) else Color(-1)
      )
      // Couleurs avec lesquelles le sommet à colorer peut l'être
      val accessibleColors = colors filterNot neighboursColors.contains

      val (vertexColor, newColors) =
        // Si aucune couleur n'est accessible
        if (accessibleColors.isEmpty) {
          // Une nouvelle couleur est crée et ajoutée aux couleurs disponibles
          val newColor = colors.head.next
          (newColor, newColor :: colors)
        } else
          // Sinon la plus petite couleur est choisie
          (accessibleColors.last, colors)

      this.baseColoration(
        coloredVertices + (vertex -> vertexColor),
        newColors,
        vertices.tail
      )
    }
```

Algorithme naïf

L'algorithme naïf consiste donc uniquement à appeler la méthode `baseColoration()` avec des paramètres de base.

```
  /**
   * Donne une coloration du graphe avec un algorithme greedy
   *
   * @return coloration du graphe
   */
  def getGreedyColoration: ColorationResult =
    this.baseColoration(
      Map[Vertex, Color](),
      List(Color(1)),
      this.vertices.toList
    )
```

Algorithme de Welsh-Powell

L'algorithme de Welsh-Powell consiste à appeler la méthode `baseColoration()` mais avec les sommet triés par degrès décroissant.

```
  /**
   * Donne une coloration du graphe avec l'algorithme de Welsh-Powel
   *
   * @return coloration du graphe
   */
  def getWelshPowellColoration: ColorationResult = {
    // Les sommets du graphe sont triés dans l'ordre décroissant de leur degrès
    val orderedVertices = this.vertices.toList sortWith (
      (v1, v2) => this.getVertexDegree(v1) > this.getVertexDegree(v2)
    )

    this.baseColoration(
      Map[Vertex, Color](),
      List(Color(1)),
      orderedVertices
    )
  }
```

Algorithme DSATUR

L'algorithme DSATUR est plutôt similaire aux deux algorithmes précédent sauf que la liste des sommets à colorés et à réordonner à chaque itération. La méthode `baseColoration()` a donc été réécrite et légérement modifiée.

```
    /*
     * coloredVertices : sommets du graphe déjà colorés
     * colors          : couleurs pour l'instant utilisées dans le graphe
     * vertices        : sommets du graphe qu'il reste à colorer
     */
    def dsaturColoration(
      coloredVertices: Map[Vertex, Color],
      colors: List[Color],
      vertices: List[Vertex]
    ): ColorationResult =
      // Si tous les sommets ont été colorés, la coloration est retournée
      if (vertices.isEmpty)
        new ColorationResult(coloredVertices, colors.size)
      else {
        // Sommet à colorer
        val vertex = vertices.head
        // Voisins du sommet à colorer
        val neighbours = this getVertexNeighbours vertex
        // Couleurs des voisins du sommet à colorer
        val neighboursColors = neighbours map (n =>
          if (coloredVertices contains n) coloredVertices(n) else Color(-1)
        )
        // Couleurs avec lesquelles le sommet à colorer peut l'être
        val accessibleColors = colors filterNot neighboursColors.contains

        val (vertexColor, newColors) =
          // Si aucune couleur n'est accessible
          if (accessibleColors.isEmpty) {
            // Une nouvelle couleur est crée et ajoutée aux couleurs disponibles
            val newColor = colors.head.next
            (newColor, newColor :: colors)
          } else
            // Sinon la plus petite couleur est choisie
            (accessibleColors.last, colors)

        // Map mise à jour
        val newMap = coloredVertices + (vertex -> vertexColor)
        // Liste nouvellement ordonnée
        val newList = orderList(vertices.tail, newMap)

        dsaturColoration(newMap, newColors, newList)
      }
```

Le score DSAT d'un sommet est calculé de la manière suivante.

```
    // Donne le score DSAT d'un sommet
    def vertexDSAT(vertex: Vertex, coloredVertices: Map[Vertex, Color]) = {
      val neighbours = this getVertexNeighbours vertex
      val coloredNeighbours = neighbours filter coloredVertices.contains

      // Si le sommet ne possède pas de voisins colorés
      if (coloredNeighbours.isEmpty)
        // Son degrès est retourné
        this getVertexDegree vertex
      else
        /*
         * Sinon c'est le nombre de couleurs différentes utilisées dans son
         * voisinnage
         */
        coloredVertices.
        filter(c => coloredNeighbours contains c._1).
        map(c => c._2).
        toList.
        distinct.
        size
    }
```

Et les sommets sont réorganisés à chaque itération à l'aide de cette méthode.

```
    // Fonction d'ordre utilisé dans l'algorithme
    def orderFunction(
      v1: Vertex,
      v2: Vertex,
      coloredVertices: Map[Vertex, Color]
    ): Boolean = {
      val dsat1 = vertexDSAT(v1, coloredVertices)
      val dsat2 = vertexDSAT(v2, coloredVertices)

      // Si le score DSAT des deux sommets est différent
      if (dsat1 != dsat2)
        // Le sommet à placé en premier est celui qui a le score le plus élevé
        dsat1 > dsat2
      else
        /*
         * Si le score est le même, c'est le sommet de plus haut degrès qui est
         * placé en premier
         */
        (this getVertexDegree v1) > (this getVertexDegree v2)
    }
```

Tests sur de petits graphes

Pour s'assurer que les algorithmes fonctionnent correctement, ils ont été appliqués sur de petits graphes.

Exemple du graphe se trouvant à la fin du sujet du TP 2 :

```
scala> val g = GraphBuilder.buildGraph("graph.txt")
g: Graph =
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)

scala> g.getGreedyColoration
res0: ColorationResult =
4 colors :
12 -> Color 1
8 -> Color 2
4 -> Color 1
11 -> Color 1
9 -> Color 1
5 -> Color 3
10 -> Color 2
6 -> Color 1
1 -> Color 1
2 -> Color 2
7 -> Color 2
3 -> Color 4

scala> g.getWelshPowellColoration
res1: ColorationResult =
3 colors :
12 -> Color 2
8 -> Color 1
4 -> Color 3
11 -> Color 2
9 -> Color 3
5 -> Color 2
```

```
10 -> Color 1
6 -> Color 3
1 -> Color 2
2 -> Color 3
7 -> Color 1
3 -> Color 1

scala> g.getDSATURColoration
res2: ColorationResult =
3 colors :
12 -> Color 2
8 -> Color 1
4 -> Color 3
11 -> Color 2
9 -> Color 3
5 -> Color 2
10 -> Color 1
6 -> Color 3
1 -> Color 2
2 -> Color 3
7 -> Color 1
3 -> Color 1
```

Main

Tests de performances

Le `jar` éxécutable `TestPerfColo.jar` permet de tester les trois algorithmes sur un panel de graphe généré aléatoirement.

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

```
$ java -jar TestPerfColo 50 100

===== PROBA : 0.1 =====
----- GREEDY -----
Nombre de couleurs en moyenne : 7
Temps moyen                   : 53 ms
----- WELSH-POWELL -----
Nombre de couleurs en moyenne : 6
Temps moyen                   : 34 ms
----- DSATUR -----
Nombre de couleurs en moyenne : 5
Temps moyen                   : 946 ms
```

```
===== PROBA : 0.3 =====
----- GREEDY -----
Nombre de couleurs en moyenne : 14
Temps moyen                   : 9 ms
----- WELSH-POWELL -----
Nombre de couleurs en moyenne : 12
Temps moyen                   : 74 ms
----- DSATUR -----
Nombre de couleurs en moyenne : 11
Temps moyen                   : 3120 ms

===== PROBA : 0.5 =====
----- GREEDY -----
Nombre de couleurs en moyenne : 21
Temps moyen                   : 11 ms
----- WELSH-POWELL -----
Nombre de couleurs en moyenne : 19
Temps moyen                   : 93 ms
----- DSATUR -----
Nombre de couleurs en moyenne : 18
Temps moyen                   : 5926 ms

===== PROBA : 0.7 =====
----- GREEDY -----
Nombre de couleurs en moyenne : 29
Temps moyen                   : 14 ms
----- WELSH-POWELL -----
Nombre de couleurs en moyenne : 28
Temps moyen                   : 162 ms
----- DSATUR -----
Nombre de couleurs en moyenne : 26
Temps moyen                   : 8582 ms

===== PROBA : 0.9 =====
----- GREEDY -----
Nombre de couleurs en moyenne : 46
Temps moyen                   : 19 ms
----- WELSH-POWELL -----
Nombre de couleurs en moyenne : 44
Temps moyen                   : 198 ms
----- DSATUR -----
Nombre de couleurs en moyenne : 41
Temps moyen                   : 9658 ms
```

On remarque que plus un algorithme est efficace quand à la coloration, moins il est rapide.