Baptiste COLIN
1ère année du cycle Ingénieur Civil *MINES ParisTech*

Projet d'informatique :
Calcul des composantes fortement connexes d'un graphe orienté

Table des matières

La théorie des graphes est un édifice fondamental de l'informatique théorique. Elle a permis de grandes avancées dans de nombreux domaines de par sa généralité, qui repose en fait sur le caractère abstrait de la notion de sommet.

Pour rappel, un graphe est défini par la donnée d'un ensemble de sommets et d'un ensemble d'arêtes, c'est à dire des couples de sommets. Rien n'indique dans cette définition ce qu'est un sommet, et il n'existe aucune contrainte quand aux propriétés que doivent vérifier les sommets. C'est là la puissance des graphes : celles de modéliser des relations entre des éléments, peu importe leur nature.

Les graphes bénéficient de plus de la possibilité d'être représentés intuitivement sur un plan, en assimilant un sommet à un point du plan, et une arête à un segment reliant les sommets de départ et d'arrivée.

Ces deux caractéristiques des graphes justifient la pertinence de ce projet d'informatique, en tant qu'exercice de mise en pratique de la programmation en Java. L'implémentation des graphes en Java nécessite de faire le pont entre la notion abstraite de graphe, et plus fondamentalement de sommet, et les outils fournis par le langage Java. Ce lien se réalise en fait grâce au caractère « orienté objet » du langage Java, et grâce à des structures précises fournies par la bibliothèque java.util. La représentation à l'écran et la manipulation à la souris du graphe est l'occasion d'une mise en application des connaissances en matière d'interface homme-machine. J'ai par ailleurs profité de ce projet pour découvrir et utiliser la bibliothèque graphique Swing, plus complète et pertinente que AWT.

Le problème de théorie des graphes traité par ce programme consiste en la détermination du partitionnement optimal d'un graphe en composantes fortement connexes. L'algorithme de Kosaraju répond à ce problème, et il s'est agit de l'implémenter en Java pour correspondre à l'implémentation des graphes.

Ce projet aura donc été pour moi l'occasion d'approfondir ma connaissance du langage Java et de ses différentes bibliothèques, mais aussi mes connaissances en théorie des graphes. Cela m'a permis de confirmer mon enthousiasme quand à l'étude de l'informatique, si bien pratique que théorique.

Je n'entends pas abandonner ce programme une fois rendu dans le cadre de mes études aux Mines de Paris, mais espère pouvoir continuer à l'enrichir pour le rendre plus ergonomique et lui apporter de nouvelles fonctionnalités (implémentation de l'algorithme de Dijkstra notamment). Mon but serait atteint si il pouvait constituer un véritable outil pédagogique dans un cours de théorie des graphes..

Analyse

Implémentation de la structure de graphe

Le fait que Java soit un langage de programmation orienté objet facilite l'implémentation d'une structure aussi abstraite que celle de graphe. Dans le cadre de ce projet, nous nous intéressons à des graphes orientés « abstraits », en cela que les sommets ne représentent pas des objets précis mais des entités abstraites.

La définition d'un graphe retenue ici est la suivante :

Soit S un ensemble de sommets et A un ensemble d'arêtes de S, c'est-à-dire une partie de S^2. On appelle graphe tout couple G de la forme G=(S,A).

On remarquera que cette définition exclue la possibilité d'introduire des arêtes multiples. La notion sous-jacente qui reste indéfinie est celle de sommet. Cependant la programmation en Java est telle que cette indéfinition n'est pas un obstacle.

Sommet

L'implémentation des sommets est faite par la simple déclaration d'une classe `Sommet`. Le seul attribut de cette classe ayant une origine « théorique » est l'attribut entier *`numéro`*, qui sert à pouvoir différencier deux sommets (cf. la méthode `equals`).

Afin que deux sommets différents aient bien des numéros différents, j'ai mis en place un attribut entier statique *`nombre`*, incrémenté à chaque création de sommet. Lors de l'instanciation d'un sommet, on donne à *`numéro`* la valeur *`nombre`*. *`numéro`* ne sera plus jamais modifié par la suite, de telle sorte que deux sommets sont égaux si et seulement si ils ont le même *`numéro`*. On sera bien amenés à se demander si deux sommets sont bien les mêmes, au moment de colorier les sommets (cf. la méthode `actionPerformed` dans la classe `colorierAction`). Ils auront bien les mêmes `numero` du fait de l'appel, auparavant, à la fonction `clone`.

Les autres différents attributs de la classe `Sommet` ont des vocations graphiques : il s'agit des coordonnées du sommet, de sa couleur, et de son rayon à l'affichage.

Arête

La classe `Arete` ne possède que deux attributs de type `Sommet`, qui correspondent aux sommets de départ et d'arrivée de l'arête. Cette implémentation rudimentaire est en fait une traduction exacte de la définition mathématique d'une arête : c'est un couple de sommets. Toutes les informations nécessaires et sont donc présentes dans cette implémentation.

Trois méthodes sont ensuite utilisées dans la partie théorique : `getDepart` et `getArrivee` , qui renvoient respectivement les sommets de départ et d'arrivée, et `reverse`, qui à une arête associe son « inverse » (elle n'agit pas par effet de bord). C'est à dire que si S et S' sont deux sommets, cette méthode associera l'arête (S',S) à l'arête (S,S'). `reverse` joue un rôle essentiel au moment de déterminer les sommets ascendants d'un sommet donné.

Graphe

Toujours dans une optique de fidélité à la définition mathématique , la classe `Graphe` possède deux attributs : un ensemble de sommets, et un ensemble d'arêtes. La structure d'ensemble mathématique est implémentée grâce à au type paramétré `HashSet`.

Cette structure de donner permet de conserver la rigueur mathématiques de l'opération d'intersection. Un défaut est la nécessité de devoir recourir à un `Iterator` pour en parcourir les éléments, mais aussi et surtout l'impossibilité de trier ces structures.

Algorithme de Kosaraju

L'algorithme de Kosaraju, également appelé méthode de Malgrange, repose sur le principe suivant : étant donné un sommet *S* d'un graphe *G*, la plus grande composante fortement connexe à laquelle appartient *S* est toujours l'intersection des sommets ascendants de *S* et des sommets descendants de *S*. Reste à implémenter cet algorithme, au moyen de plusieurs fonctions intermédiaires que je vais détailler par la suite.

Fonction `voisins`

Elle prend en argument un sommet *S* dont on présuppose qu'il appartient bien au graphe. Cette fonction revoie l'ensemble des sommets *S'* tels qu'il existe une arête (*S,S'*) dans le graphe, sous la forme d'un .

La fonction consiste en un parcours de l'ensemble des arêtes du graphe, où l'on repère celles partant de *S*.

Fonctions `enlever`

Par polymorphisme, cette fonction prend en argument un sommet ou un ensemble de sommets, présupposés appartenir/être inclus dans l'ensemble des sommets du graphe.

La fonction renvoie alors le graphe privé de ces sommets et de toutes les arêtes qui ont un de ces sommets comme extrémité.

Fonction `sommetsDescendantsAux`

Cette fonction prend en argument un sommet *S* et un graphe *G*. Elle renvoie l'ensemble des sommets descendants de *S* dans le graphe *G* .

C'est une fonction récursive qui fonctionne selon le principe suivant : pour obtenir les sommets descendants de *S*, on prend ses voisins, puis les voisins de ses voisins sauf ceux qu'on a déjà visité, et ainsi de suite. C'est pour cela que le graphe de travail est ici passé en argument : il est modifié à chaque appel récursif car on en ôte les sommets déjà visités.

Fonction `sommetsDescendants`

Il s'agit de faire appel à `sommetsDescendantsAux` , en utilisant comme argument le sommet dont on veut les descendants et un clone du graphe étudié.

Il est important de bien utiliser un clone du graphe étudié, car la méthode `sommetsDescendantsAux` va enlever des sommets au graphe sur lequel elle est appelée. Or on ne

veut pas que le graphe étudié soit modifié.

Fonction `reverse`

Cette méthode, lorsqu'elle est appelée sur un graphe *G*, renvoie un nouveau graphe correspondant à celui obtenu en inversant toutes les arêtes de *G*.

Elle consiste en la création d'un nouveau graphe, dont l'ensemble des sommets est le clone de l'ensemble des sommets de *G* (et non pas une référence, pour les mêmes raisons que précédemment), et dont l'ensemble des arêtes est construit on y mettant l'ensemble des inverses des arêtes de *G*.

Fonction `sommetsAscendants`

Grâce aux fonctions `sommetsDescendants` et `reverse`, cette fonction s'obtient astucieusement en observant que les sommets ascendants d'un sommet *S* en tant qu'élément du graphe *G* sont exactement les sommets descendants de *S* en tant qu'élément du graphe *G.* `reverse()`.

Fonction `kosaraju`

On va désormais synthétiser les méthodes précédentes.

La partition de l'ensemble sera renvoyée sous la forme d'un `Map<Sommet,HashSet<Sommet>>`, qui à chaque sommet du graphe associe sa composante connexe.

On commence par déterminer une composante fortement connexe. Pour cela, on prend un sommet arbitraire, et on enregistre l'intersection de ses sommets ascendants et descendants dans une variable *E*. Il s'agit d'une composante connexe.

On modifie le map de telle sorte qu'à tous les éléments de *E*, on associe *E*.

On obtient le résultat par la fusion du map provisoire créé précédemment et du map obtenu par l'appel de l'algorithme de Kosaraju sur le graphe privé de *E*.

L'algorithme de Kosaraju est connu pour être de complexité quadratique par rapport au nombre de sommets du graphe.

Interface homme-machine

Description de l'interface

L'interface est divisée en deux parties : un bandeau où figurent côte à côte les boutons, et un « tableau blanc » sur lequel l'utilisateur dessinera le graphe. Ce dessin du graphe est permis en représentant les sommets par des cercles numérotés, et les arêtes par des segments qui relient ces cercles. Les sommets sont déplaçables à la souris par drag'n'drop.

Un des boutons du bandeau permet l'accès aux descriptions des composantes connexes. En cliquant dessus, une nouvelle fenêtre apparaît où un tableau résume les données relatives au graphe tel qu'il était au moment où l'on a cliqué sur le bouton (pas d'actualisation en temps réel). Dans chaque ligne, on a une énumération des nœuds qui appartiennent à une même composante connexe, et une colonne couleur permet de colorier la composante. Pour cela, il faut cliquer sur la case « couleur » de la composante, ce qui fait apparaître une palette sur laquelle on peut choisir la couleur. Une fois les couleurs choisies, le coloriage est appliqué en cliquant sur « recolorier ». Le coloriage est retiré à toute édition du graphe susceptible d'altérer le partitionnement en composantes connexes, et donc la validité du coloriage.

Implémentation de l'interface

La fenêtre principale est une instance de la classe JFrame dont le layout manager est BorderLayout. On y dispose deux JPanel: un destiné aux boutons, en position ***PAGE_START*** , et un destiné au dessin, en position ***CENTER*** . Le second Jpanel est plus exactement une instance de la classe `DessinPanel` , qui hérite de la classe JPanel et qui implémente les interfaces `MouseListener` et `MouseMotionListener`. La fonction de ce JPanel ne fait qu'une seule chose : appeler la fonction de dessin du graphe. Cette fonction fait elle même appel aux fonctions de dessin des sommets et des arêtes, de la manière suivante :

- on peint d'abord toutes les arêtes du graphe.
- on peint ensuite tous les nœuds du graphe.

Un des attributs du JFrame principal est un graphe, qui est en fait le graphe qui sera affiché sur le JPanel et qui sera édité par l'utilisateur.

Le JFrame qui apparaît pour résumer les infos sur les composantes connexes utilise comme layout manager un BorderLayout. En position ***PAGE_START*** est disposé un JScrollPane qui contient le tableau qui résume le partitionnement, et le bouton de recoloriage est disposé en position ***PAGE_END***.

Actions définies

La bibliothèque Swing met à disposition la classe abstraite AbstractAction , qui selon moi permet une architecture plus claire du code. J'ai défini 4 actions dans le cadre de ce projet, chacune correspondant en fait à un des boutons présent dans l'interface.

creerSommetAction

Le constructeur de cette classe prend en argument le DessinPanel sur lequel est dessiné le graphe afin d'avoir accès aux attributs du DessinPanel. Ce sera également le cas pour les autres actions.

L'action creerSommetAction commence par colorer tous les sommets en blanc, car la création d'un nouveau sommet modifie nécessairement le partitionnement en composantes fortement connexes et donc la validité du coloriage. Puis le sommet est effectivement créé, et les booléens qui permettent de savoir à quel étape de la procédure de création d'un sommet en est l'utilisateur sont modifiés en conséquences.

creerAreteAction

La procédure est la même : pour les mêmes raisons, on colorie le graphe en blanc, puis on modifie les variables nécessaires. La création d'une arête est détaillée plus loin.

calculComposantesAction

Il s'agit de mettre au point le Map qui résume le partitionnement en composantes connexes : c'est ici qu'on utilise l'algorithme de Kosaraju.

On fait ensuite apparaître un nouveau JFrame dans lequel sera résumé le partitionnement au travers d'un tableau.

colorierAction

Une fois que l'utilisateur a choisi de quelle couleur il voulait colorer les différentes composantes connexes, cette action applique le coloriage puis fait disparaître la fenêtre.

Gestion des événements

Mis à part ce qui concerne les différents boutons, deux components réagissent à différents évènements : le DessinPanel et le tableau.

Evènements liés au DessinPanel

Clics

Dans toute la suite, « cliquer sur un sommet » désignera le fait de cliquer à une distance inférieur au rayon des sommets du centre d'un sommet (c'est ainsi que cela est implémenté dans le code).

Un clic droit sur un sommet provoque le retrait de ce sommet.

En fonction de l'action qui est en train d'être effectuée, un clic gauche a plusieurs effets :

- si un sommet est en train d'être déplacé, après avoir cliqué sur « Créer Sommet » (`deposeSommet = true`), un clic gauche dépose ce sommet en assignant la valeur `false` à

`deposeSommet`.

- si une arête est en train d'être créée (`creationArete = true`), et que le sommet de départ de cette arête a déjà été sélectionné (`sommetDepartSelectionne = true`), alors lors du clic gauche, on vérifie si l'utilisateur vient de cliquer sur un sommet. Si ce n'est pas la cas, on ne fait rien. Si c'est le cas, alors le sommet sur lequel l'utilisateur vient de cliquer est désigné comme le sommet de l'arête en cours de création.

- si une arête est en train d'être créée (`creationArete = true`), et que le sommet de départ de cette arête a déjà été sélectionné (`sommetDepartSelectionne = true`), alors lors du clic gauche, on vérifie si l'utilisateur vient de cliquer sur un sommet. Si ce n'est pas le cas, on ne fait rien. Si c'est le cas, le sommet sur lequel l'utilisateur vient de cliquer est désigné comme le sommet de départ de l'arête qui est en cours de création.

Provisoirement, on désigne comme sommet d'arrivée de l'arrête en cours de création un sommet fictif nommé « fantôme », qui ne fait pas partie du graphe et qui n'est donc pas affiché à l'écran. Ceci permet de pouvoir dessiner l'arête pendant sa création, sans générer d'erreur due à une non-définition du sommet d'arrivée de l'arête.

- dans les autres cas, si l'utilisateur a cliqué à un sommet, ce sommet devient le sommet déplacé : sa référence est passée à la variable `sommetDeplace`, et ce jusqu'à ce que l'utilisateur relâche le bouton gauche de la souris.

Lorsque le bouton gauche est relâché, on assigne à `sommetDeplace` la valeur `null` : aucun sommet n'est en train d'être déplacé.

Déplacement de la souris

Si un sommet est en train d'être déplacé ou déposé, alors les coordonnées de ce sommet sont constamment actualisées pour qu'il suive le curseur de la souris. Des conditions géométriques font en sorte que le sommet ne puisse pas sortir du cadre du DessinPanel.

Si une arête et en cours de création et que le sommet de départ de l'arête a déjà été désigné, les coordonnées du sommet fantôme sont actualisées pour correspondre au curseur de la souris. Visuellement, l'arête en cours de création part du sommet de départ et se termine sur le curseur.

Evènements liés au tableau

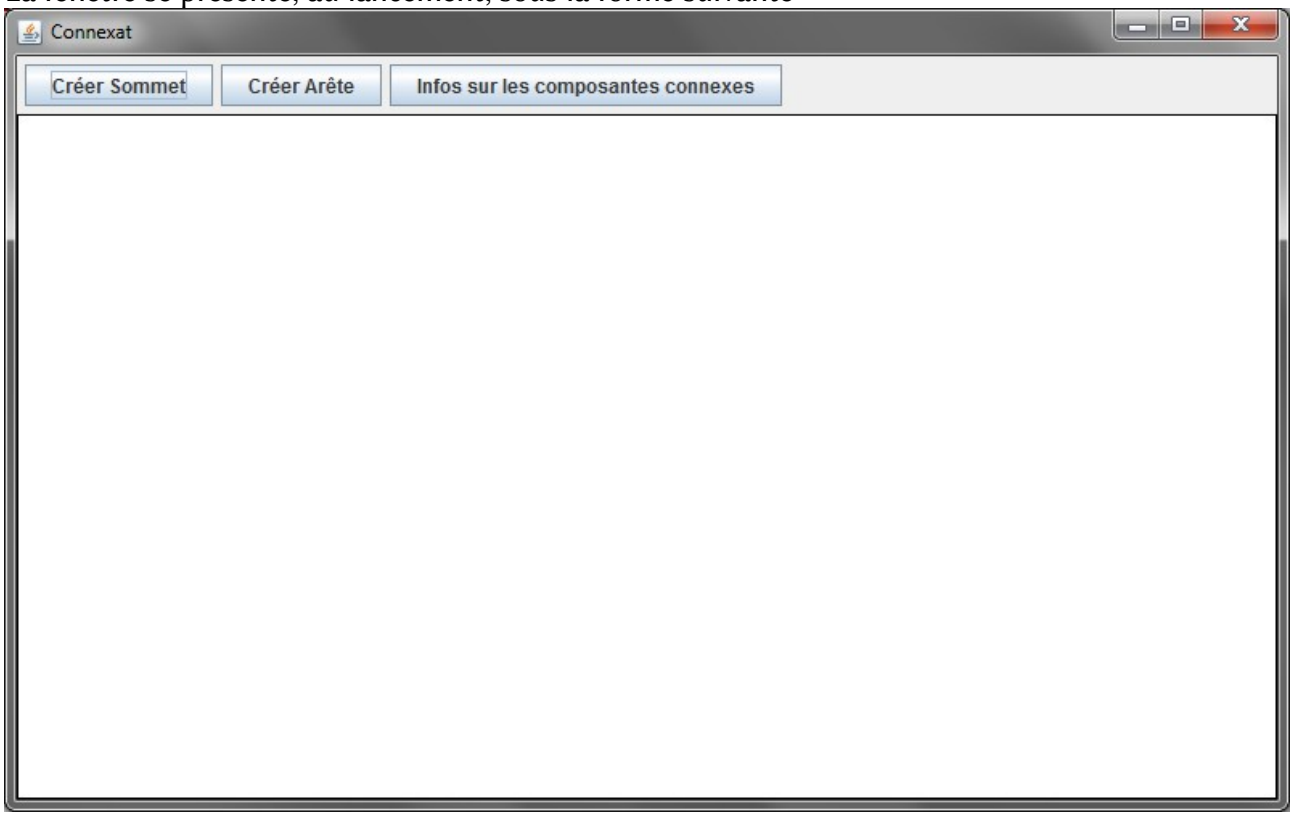
Il s'agit ici de pouvoir afficher une couleur dans le tableau, de pouvoir la modifier naturellement et de pouvoir prendre en compte cette modification.

La bibliothèque Swing implémente la classe ColorChooser, qui correspond tout à fait aux besoins de ce projet. Cependant son implémentation au sein d'un tableau héritant de la classe TablleModel est laborieuse du fait de l'implémentation de la classe TableModel en Java.

Si j'ai pu commencer à comprendre les détails du fonctionnement de la classe Table Model, j'ai été contraint par le temps d'utiliser des classes trouvées sur internet pour mettre en place l'interface que je désirais. J'ai donc utilisées les classes ColorRenderer et ColorEditor, du site http://da2i.univ-lille1.fr/doc/tutorial-java/uiswing/components/table.html .

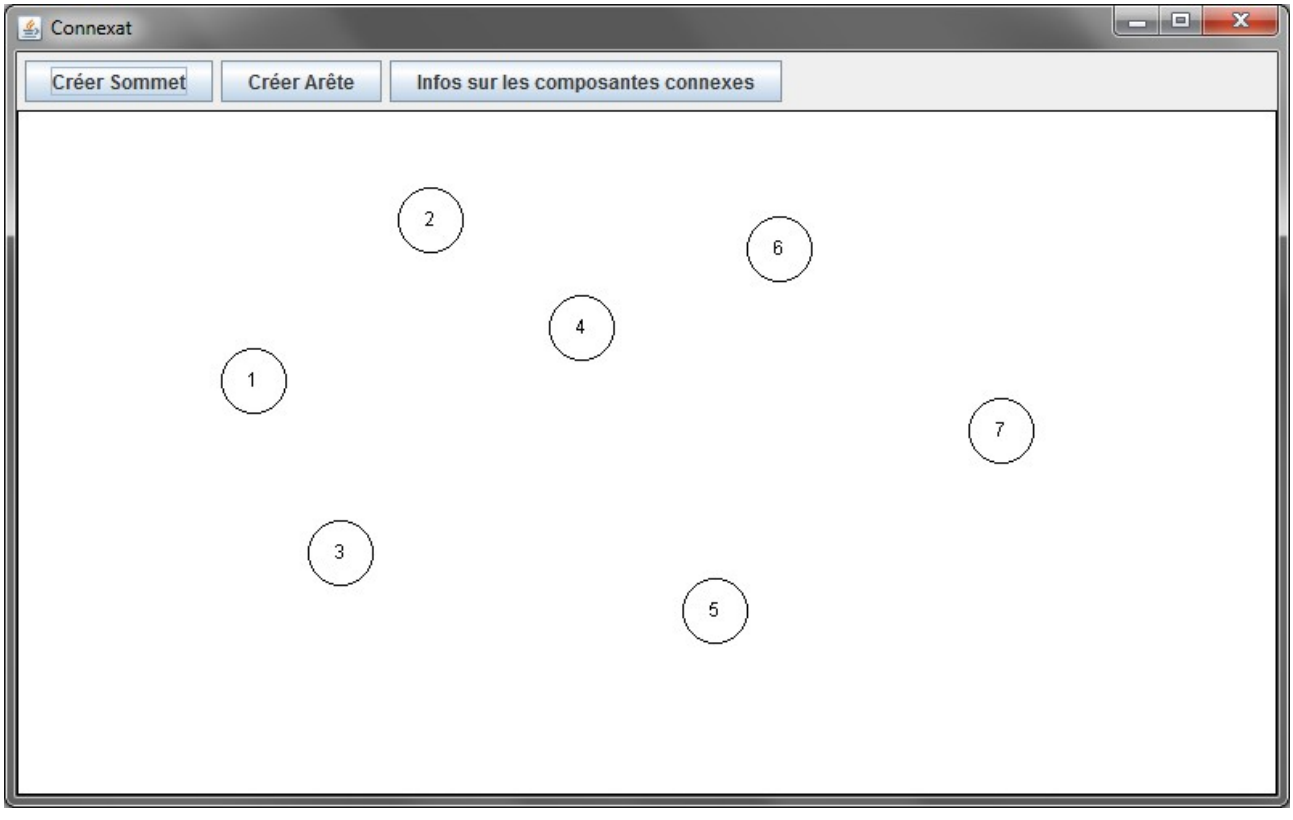
Notice d'utilisation

La fenêtre se présente, au lancement, sous la forme suivante :



On distingue un bandeau supérieur, où se trouvent les boutons, et un tableau blanc, destiné au dessin et à la manipulation du graphe.

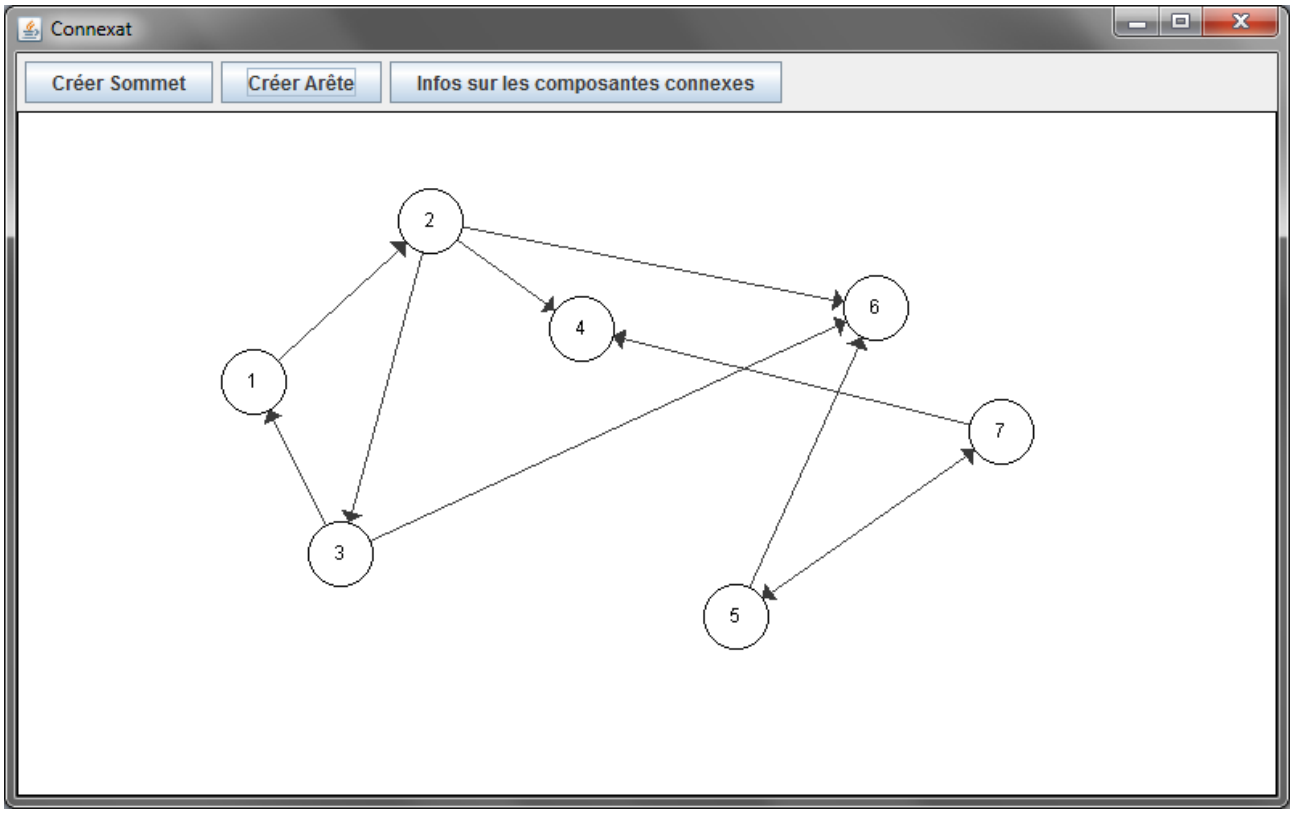
Après avoir cliqué sur le bouton « Créer Sommet » , un nouveau sommet est créé et suis le curseur. En cliquant une nouvelle fois sur le tableau, on dépose le sommet. Il arrête alors de suivre le curseur.
Le sommet, une fois déposé, peut être déplace par « drag'n'drop ».



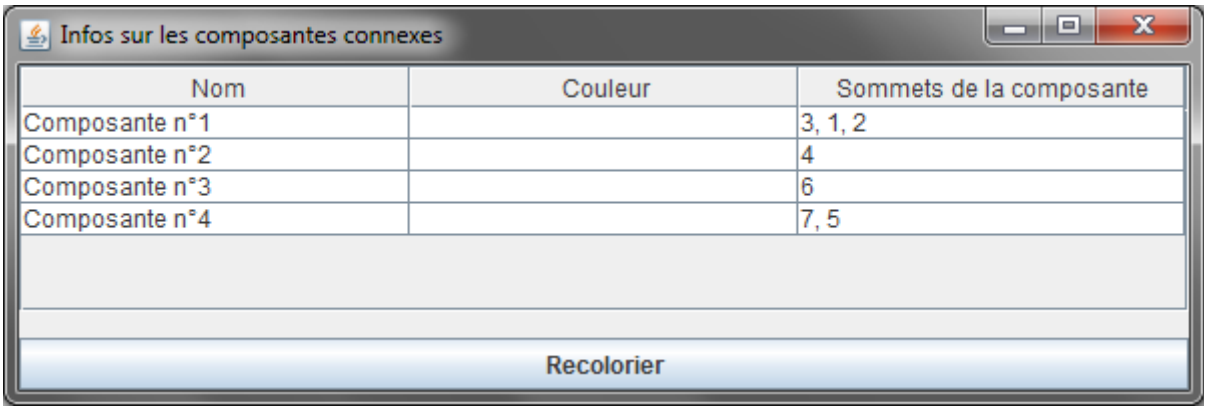
Il n'y a pas de limite du nombre de sommets.

Une fois que 2 sommets ou plus sont disposés, on peut créer des arêtes enre eux.
Pour cela, cliquer sur le bouton « Créer Arête ».
Cliquer sur le sommet de déparrt de l'arête.
Cliquer ensuite sur le sommet d'arrivée de l'arête.
L'arête est ainsi créée.

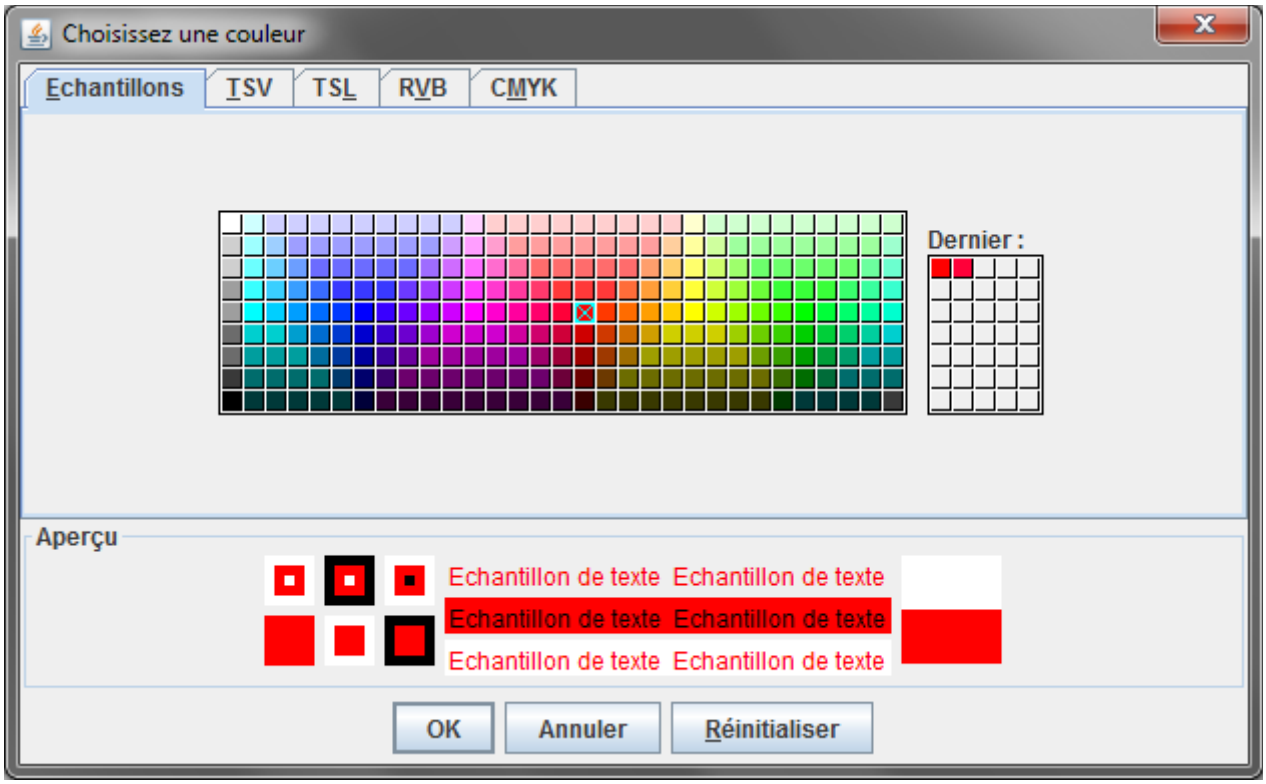
Il n'y a pas non plus de limitation au nombre d'arêtes.



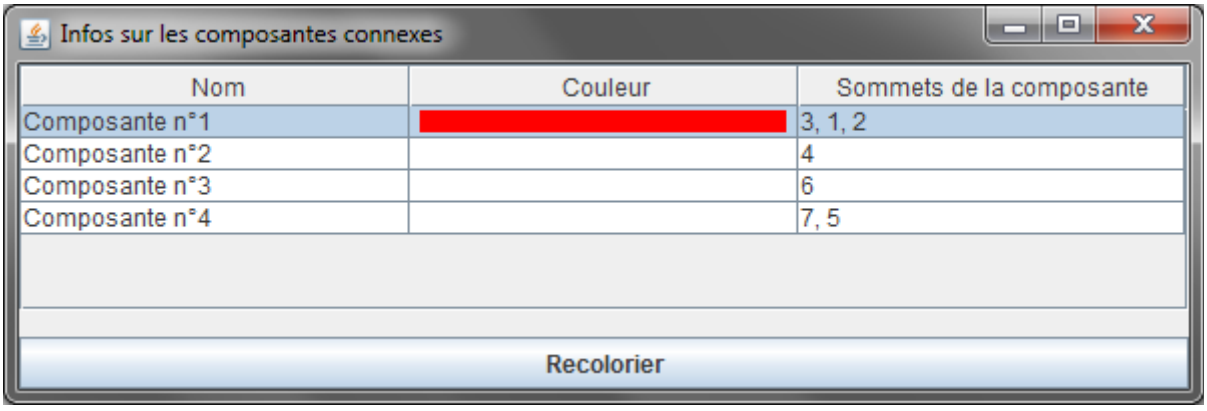
Pour avoir des informations sur les composantes fortement connexes du graphe ainsi créé, cliquer sur « Infos sur les composantes connexes » . Une fenêtre s'ouvre.



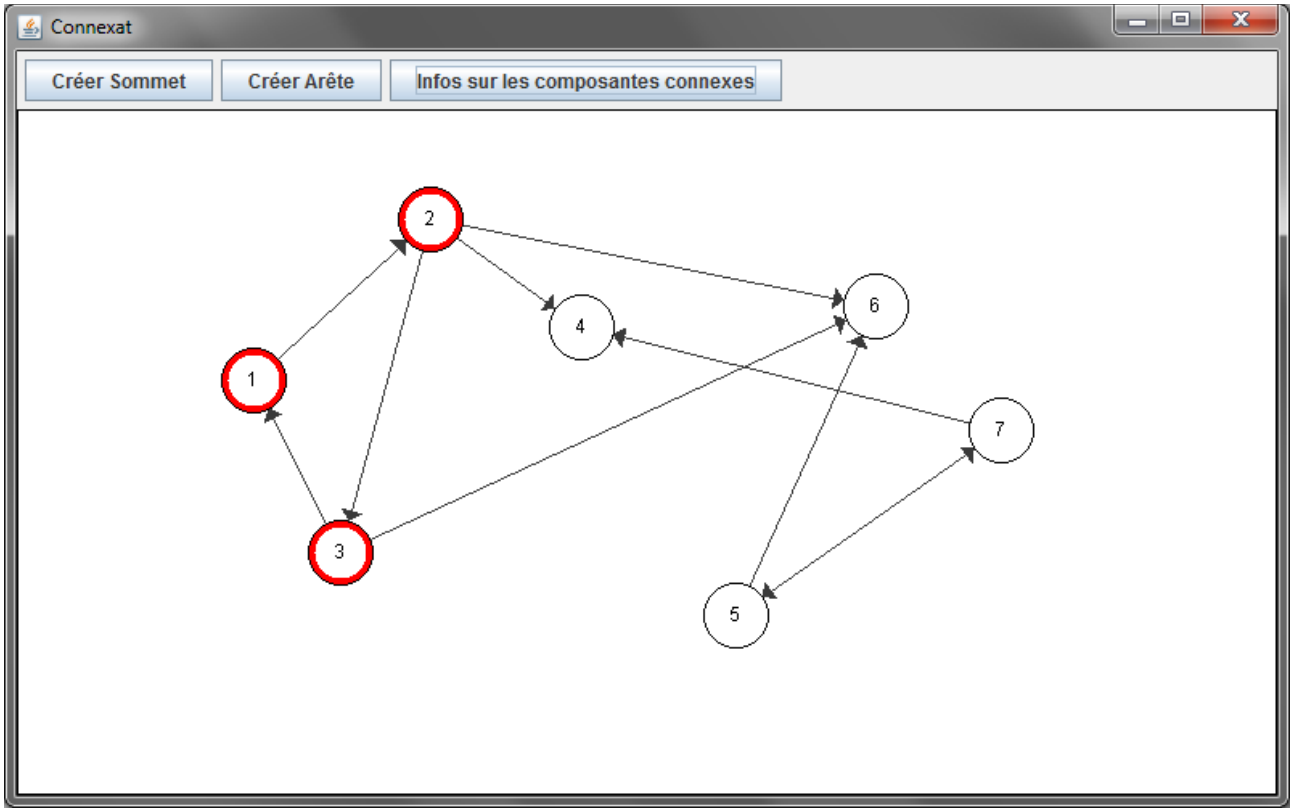
On y lit les informations relatives aux différentes composantes connexes : chaque ligne du tableau correspond à une composante.
La troisième colonne indique les sommets qui appartiennent à une même composante.
La deuxième colonne indique la couleur choisie pour colorier la composante. Par défaut, cette couleur est blanche. Pour la modifier, cliquer sur la case. Une palette de couleur s'affiche :



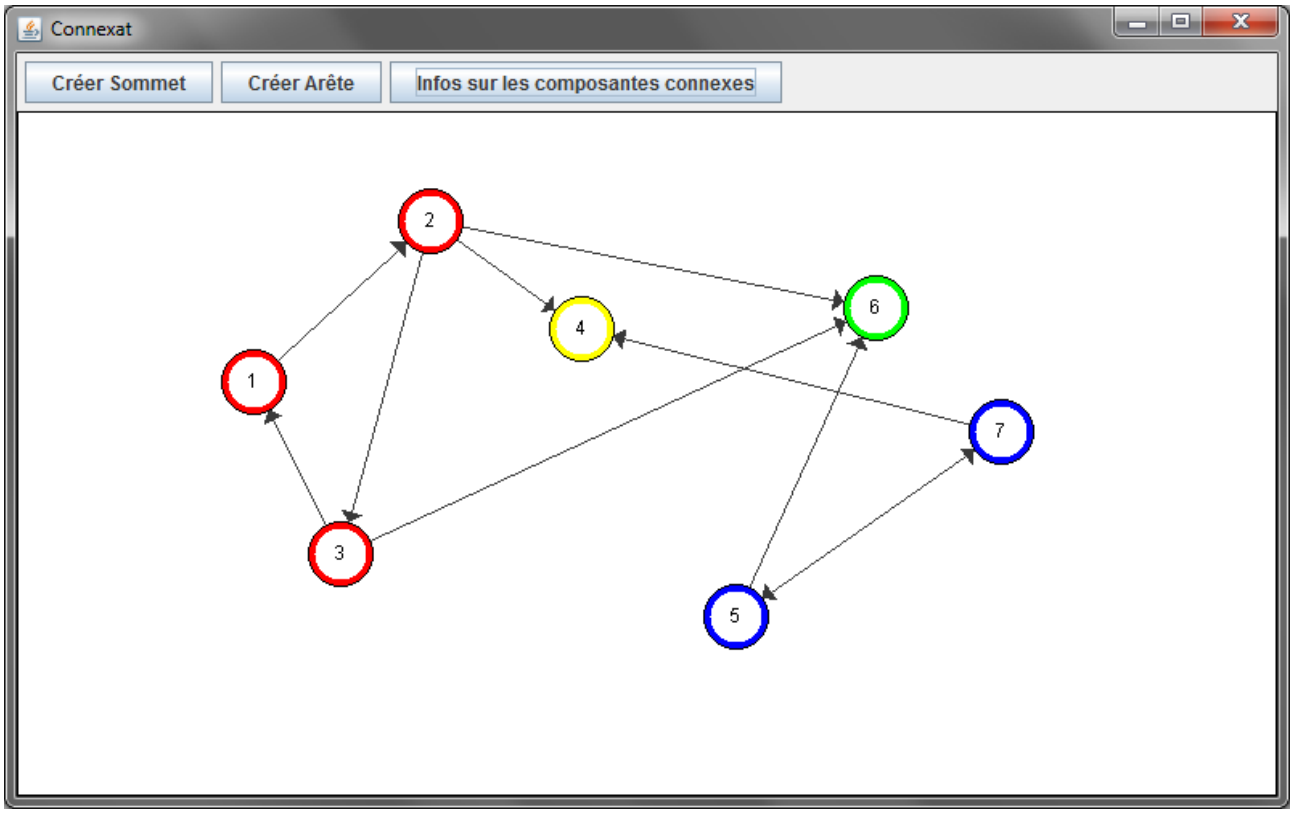
Sélectionner la couleur par laquelle vous voulez colorier la composante, puis cliquer sur « OK ».
La couleur associée à la composante est modifiée dans le tableau :



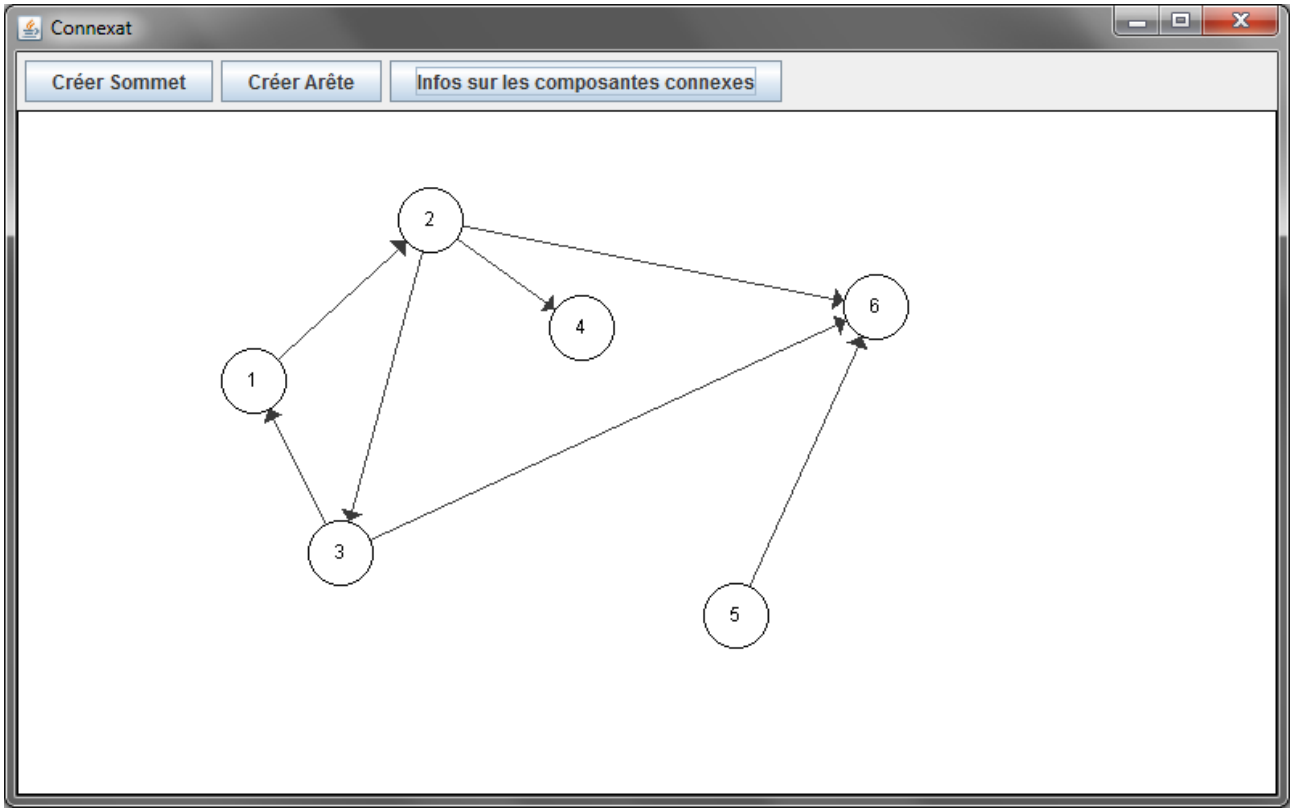
Cliquer sur « Recolorier » pour appliquer le coloriage ainsi défini.



Colorier ainsi toutes les composantes connexes permet une visualisation plus facile des composantes connexes du graphe.



Pour supprimer un sommet, cliquer avec le bouton droit de la souris sur ce sommet. Toutes les arêtes ayant ce sommet pour extrémité sont supprimées par la même occasion.
Il faut noter qu'à la création ou suppression de tout sommet, les sommets sont re-coloriés en blanc afin que le coloriage corresponde toujours à une partition en composantes connexes.



Code source commenté

Classe Sommet

```
import java.awt.*;


public class Sommet {
	//attributs
	private static int nombre = 0;
	private int numero; // à chaque noeud est attribué un numéro, c'est ce qui permettra de les différencier.
	private int x;
	private int y;		// x et y sont les coordonnées du sommet
	private static int rayon = 20; //rayon d'un sommet, en pixels, lorsqu'on le dessine.
	private Color couleur = Color.white;

	//constructeurs
	public Sommet() {
		numero = nombre++; //à la création de chaque sommet, on incrémente nombre pour que le prochain sommet créé ait un numero différent.
	}

	public Sommet(int xs, int ys) {
		numero = nombre++;
		x=xs;
		y=ys;
	}

	//méthodes
	public int num() {
		return numero;
	}

	public boolean equals (Object o) {
		if (o instanceof Sommet) {
			return (((Sommet) o).num() == num());  //deux sommets sont égaux ssi ils ont le même numero.
		}
		else {
			return false;
		}
	}

	//méthodes pour obtenir les coordonnées
	public int getAbs() {
		return x;
	}

	public int getOrd() {
		return y;
	}

	public Point getLocation() {
		return new Point(x,y);
```

```
	}

	//méthodes pour changer les coordonnées
	public void setPosition(Point p) {
		x = (int) p.getX();
		y = (int) p.getY();
	}

	//méthode pour dessiner un noeud
	public void draw(Graphics g) {
		g.setColor(couleur);
		g.fillOval(x-rayon,y-rayon,2*rayon,2*rayon); //on remplit d'abord le
cercle de couleur
		g.setColor(Color.white);
		g.fillOval(x-rayon+4, y-rayon+4, 2*(rayon-4), 2*(rayon-4)); //puis on
remplit un cercle blanc à l'intérieur du cercle de couleur pour obtenir une couronne de
couleur
		g.setColor(Color.black);
		g.drawOval(x-rayon,y-rayon,2*rayon,2*rayon); //enfin on entoure d'un cerle
noir
		g.drawString(""+numero, x-4, y+4); //et on affiche le numéro du noeud
	}

	public int getRayon() {
		return rayon;
	}

	public void setColor(Color c) {
		couleur = c;
	}

	public Color getCouleur() {
		return couleur;
	}
}
```

Classe Arete

```
import java.awt.Graphics;

public class Arete {
	//attributs
	private Sommet depart;
	private Sommet arrivee;

	//constructeur
	public Arete() {

	}
	public Arete (Sommet dep , Sommet arr) {
		depart = dep;
		arrivee = arr;
	}

	//méthodes
```

```
	public Sommet getDepart() {
		return depart;
	}

	public Sommet getArrivee() {
		return arrivee;
	}

	public Arete reverse() {
		return (new Arete (arrivee , depart));
	}

	public void draw(Graphics g) {
		try{
			int xb = arrivee.getAbs();
			int xa = depart.getAbs();
			int yb = arrivee.getOrd();
			int ya = depart.getOrd();
			int r = arrivee.getRayon()+2;
			double n = Math.sqrt((xb-xa)*(xb-xa) + (yb-ya)*(yb-ya));
			int xm = (int) (xa+((n-r)/n)*(xb-xa));
			int ym = (int) (ya+((n-r)/n)*(yb-ya));
			int xmm = (int) (xa+((n-arrivee.getRayon())/n)*(xb-xa));
			int ymm = (int) (ya+((n-arrivee.getRayon())/n)*(yb-ya));
			int l = 7;


			if (arrivee.num()!=0) {
				g.drawLine(xa, ya, xb, yb);
			}
			else {
				g.drawLine(xa, ya, xm, ym);
			} //distinction de cas nécessaire à un affichage propre de l'arête
lorsque le sommet d'arrivée n'a pas encore été sélectionné par l'utiliateur


			double xd = (xb-xa)/n;
			double yd = (yb-ya)/n; //ce sont les coordonnées du vecteur
directeur unitaire de l'arête, orienté vers le sommet d'arrivée
			double xn = -yd;
			double yn = xd; //ce sont les coordonnées d'un vecteur normal à
l'arête


			int[] x = new int[3];
			int[] y = new int[3];
				//on dessine la pointe de la flèche à l'aide des coordonnées
des 3 points qui le définissent.

			x[0] = (int) Math.round(xmm);
			x[1] = (int) Math.round((xb-(l+arrivee.getRayon())*xd+l*xn));
			x[2] = (int) Math.round((xb-(l+arrivee.getRayon())*xd-l*xn));

			y[0] = (int) Math.round(ymm);
			y[1] = (int) Math.round((yb-(l+arrivee.getRayon())*yd+l*yn));
			y[2] = (int) Math.round((yb-(l+arrivee.getRayon())*yd-l*yn));
```

```
                g.fillPolygon(x,y,3);
            }
            catch (NullPointerException e) {}
        }

        public void setDepart(Sommet som) {
            depart=som;
        }

        public void setArrivee(Sommet som) {
            arrivee=som;
        }
}
```

Classe Graphe

```
import java.awt.*;
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;

public class Graphe implements Cloneable {
        //attributs
        private HashSet<Sommet> sommets;
        private HashSet<Arete> aretes;

        //constructeurs
        public Graphe(HashSet<Sommet> N, HashSet<Arete> A) {
            sommets = N;
            aretes = A;
        }

        public Graphe() {
            this(new HashSet<Sommet>(), new HashSet<Arete>());
        }

        //méthodes

        //pour dessiner un graphe, on dessine toutes les arêtes puis tous les sommets.
        //en dessinant les sommets après les arêtes, on s'assure qu'à l'écran les arêtes
ne "rentrent" pas dans les sommets.
        public void draw(Graphics g) {
            Iterator<Arete> itAretes = aretes.iterator();
            while (itAretes.hasNext()) {
                itAretes.next().draw(g);
            }

            Iterator<Sommet> itSommets = sommets.iterator();
            while (itSommets.hasNext()) {
                itSommets.next().draw(g);
            }
        }
```

```
	public Graphe clone() {
		Graphe res = new Graphe((HashSet<Sommet>) sommets.clone(),
(HashSet<Arete>) aretes.clone());
		return res;
	}

	public void ajouteSommet(Sommet N) {
		this.sommets.add(N);
	}

	public void ajouteSommet(Sommet N, HashSet<Sommet> dep, HashSet<Sommet> arr) {
//ajoute un sommet N au graphe, et créé des arêtes allant de tous les
		//ajout de N au graphe
					//sommets de dep vers N, et de N vers tous les sommets
de arr
		this.sommets.add(N);


		//construction des arrêtes allant vers N
		Iterator<Sommet> it = dep.iterator();
		while (it.hasNext()) {
			ajouteArete(new Arete(it.next(), N));
		}

		//construction des arrêtes partant de N
		it = arr.iterator();
		while(it.hasNext()) {
			ajouteArete(new Arete(N, it.next()));
		}

	}

	public void ajouteArete(Arete A) {
		this.aretes.add(A);
	}


	public HashSet<Sommet> voisins(Sommet N) {//cette méthode renvoie tous les
sommets qui sont la terminaison d'une arete partant de N
		HashSet<Sommet> res = new HashSet<Sommet>();

		Iterator<Arete> it = aretes.iterator();
		while (it.hasNext()) {
			Arete A = it.next();
			if (A.getDepart().equals(N)) {
				res.add(A.getArrivee());
			}
		}
		return res;
	}

	public Graphe enlever(Sommet N) {//renvoie le graphe obtenu en enlevant toutes
les arrêtes arrivant sur N ainsi que le sommet N
		Graphe g = this.clone();
		Iterator<Arete> it = this.aretes.iterator();
		while (it.hasNext()) { //on examine toutes les arêtes; si elles
aboutissent sur N ou partent de N on les enlève.
```

```
                        Arete A = it.next();
                        if (A.getArrivee().equals(N) || A.getDepart().equals(N)) {
                                g.aretes.remove(A);
                        }
                }
                g.sommets.remove(N); //on enlève le sommet N

                return g;
        }

        public void remove(Sommet N) { //même fonction mais par effet de bord cette fois
ci
                Graphe g = this.clone();
                Iterator<Arete> it = g.aretes.iterator();
                while (it.hasNext()) { //on examine toutes les arêtes; si elles
aboutissent sur N ou partent de N on les enlève.
                        Arete A = it.next();
                        if (A.getArrivee().equals(N) || A.getDepart().equals(N)) {
                                this.aretes.remove(A);
                        }
                }
                this.sommets.remove(N); //on enlève le sommet N
        }

        public Graphe enlever(HashSet<Sommet> E) {// même chose, mais en enlevant un
ensemble de noeuds
                Graphe g = this.clone();
                Iterator<Sommet> it = E.iterator();
                while (it.hasNext()) {
                        g = g.enlever(it.next());
                }
                return g;
        }

        public HashSet<Sommet> sommetsDescendantsAux(Sommet N, Graphe g) { //cette
méthode  récursive renvoie l'ensemble des sommets du graphe qui peuvent être atteints
en partant de N
                HashSet<Sommet> res = new HashSet<Sommet>();
                HashSet<Sommet> voisins = g.voisins(N);

                res.add(N);
                g = g.enlever(N);

                Iterator<Sommet> it = voisins.iterator();
                while (it.hasNext()) {
                        res.addAll(this.sommetsDescendantsAux(it.next(),g));
                }

                return res;
        }

        public HashSet<Sommet> sommetsDescendants(Sommet N) {
                Graphe g = this.clone();
                return sommetsDescendantsAux(N,g);
        }

        public HashSet<Sommet> sommetsAscendants(Sommet N) {
```

```
            return (this.reverse()).sommetsDescendants(N); //on utilise ici le fait
que si A est descendant de B, alors B est ascendant de A
      }

      public Graphe reverse() { //fonction qui à un graphe g associe le graphe g'
comportant les mêmes sommets et les arrêtes de g inversées (g comprend une arête de A
vers B sssi g' comprend une arête de B vers A)
            HashSet<Sommet> resNoeuds = (HashSet<Sommet>) this.sommets.clone();
            HashSet<Arete> resAretes = new HashSet<Arete>();

            Iterator<Arete> it = this.aretes.iterator();
            while (it.hasNext()) {
                  resAretes.add(it.next().reverse());
            }

            return (new Graphe(resNoeuds, resAretes));
      }

      public Map<Sommet,HashSet<Sommet>> kosaraju() { //on va renvoyer un Map qui à un
sommet associe sa composante connexe.
            if (this.sommets.isEmpty()) {
                  return(new HashMap<Sommet,HashSet<Sommet>>()); //si le graphe est
vide, on renvoie un Map vide
            }
            else {
                  Iterator<Sommet> it = this.sommets.iterator();
                  Sommet N = it.next(); //on part d'un sommet quelconque

                  HashSet<Sommet> composante = sommetsDescendants(N);
                  composante.retainAll(sommetsAscendants(N));

                  Map<Sommet,HashSet<Sommet>> res = new
HashMap<Sommet,HashSet<Sommet>>(); //on a construit une composante connexe en prenant
l'intersection des sommet ascendants et descendants d'un certain sommet
                  Iterator<Sommet> iter = composante.iterator();

                  while(iter.hasNext()) {
                        res.put(iter.next(), composante);
                  }

                  res.putAll((this.enlever(composante)).kosaraju()); //on réitère
l'opération en enlevant les noeuds pours lesquels on a déjà trouvé la composante
connexe
                  return res;
            }
      }

      public HashSet<Sommet> getSommets() {
            return sommets;
      }

      public HashSet<Arete> getAretes() {
            return aretes;
      }
}
```

Classe LanceFenetre

```
import javax.swing.*;

public class LanceFenetre {

	public static void main(String[] args) {
		SwingUtilities.invokeLater(new Runnable(){
			public void run(){
				Fenetre fenetre = new Fenetre();
				fenetre.setVisible(true);
			}
		});

	}

}
```

Classe Fenetre

```
import javax.swing.*;

import java.awt.*;
import java.util.*;

public class Fenetre extends JFrame {
	private DessinPanel dessinPanel;
	private JPanel boutonsPanel;
	private JPanel composantesPanel;
	private JButton boutonSommet;
	private JButton boutonArete;
	private JButton boutonColoriage;
	private JList<String> liste;
	private String[] tab = new String[10];
	private DefaultListModel<HashSet<Sommet>> composantes = new
DefaultListModel<HashSet<Sommet>>();

	private Graphe graphe;

	public Fenetre() {
		super();

		build();
	}

	private void build(){
		setTitle("Connexat");
		setSize(800,500);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setContentPane(buildContentPane());
	}

	private JPanel buildContentPane() {
		JPanel panel = new JPanel();
		panel.setLayout(new BorderLayout());
```

```
		dessinPanel = new DessinPanel(this);
		dessinPanel.setBackground(Color.white);
		dessinPanel.setVisible(true);
		panel.add(dessinPanel, BorderLayout.CENTER);

		boutonsPanel = buildBoutonsPanel();
		panel.add(boutonsPanel, BorderLayout.PAGE_START);


		HashSet<Sommet> sommetsSet = new HashSet<Sommet>();
		HashSet<Arete> aretesSet = new HashSet<Arete>();

		graphe = new Graphe(sommetsSet,aretesSet);
		composantes = collectionToListModel(graphe.kosaraju().values());
		return panel;
	}

	private JPanel buildBoutonsPanel() {
		JPanel result = new JPanel();
		result.setLayout(new BorderLayout());

		JPanel boutons = new JPanel();
		boutons.setLayout(new FlowLayout());

		boutonSommet = new JButton(new creerSommetAction(this.dessinPanel,"Créer
Sommet"));
		boutons.add(boutonSommet);
		boutonArete = new JButton(new creerAreteAction(this.dessinPanel,"Créer
Arête"));
		boutons.add(boutonArete);
		boutonColoriage = new JButton(new calculComposantesAction(this, "Infos sur
les composantes connexes"));
		boutons.add(boutonColoriage);
		result.add(boutons, BorderLayout.LINE_START);

		return result;
	}


	public Graphe getGraphe() {
		return graphe;
	}

	private DefaultListModel<HashSet<Sommet>>
collectionToListModel(Collection<HashSet<Sommet>> collec) {
		Iterator<HashSet<Sommet>> iter = collec.iterator();
		DefaultListModel<HashSet<Sommet>> res = new
DefaultListModel<HashSet<Sommet>>();

		while(iter.hasNext()) {
			HashSet<Sommet> e = iter.next();
			if(!res.contains(e)) {
				res.addElement(e);
			}
		}
		return res;
	}
```

```
	public JList<String> getListe() {
		return liste;
	}

	public String[] getTab() {
		return tab;
	}
}
```

Classe DessinPanel

```
import javax.swing.*;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class DessinPanel extends JPanel implements MouseListener, MouseMotionListener {
	private Fenetre fenetre;
	private Sommet sommetDeplace;

	//variables relatives à la création d'une arete
	private boolean creationArete = false;
	private boolean sommetDepartSelectionne = false;
	private Arete areteCreee;
	private Sommet fantome = new Sommet();

	//variables relatives à la création d'un sommet
	private boolean deposeSommet = false;
	private Sommet sommetDepose;

	public DessinPanel(Fenetre fen) {
		super();
		fenetre = fen;
		setBorder(BorderFactory.createLineBorder(Color.black));
		addMouseListener(this);
		addMouseMotionListener(this);
	}

	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		fenetre.getGraphe().draw(g);
	}

	@Override
	public void mouseClicked(MouseEvent e) {

	}

	@Override
	public void mouseEntered(MouseEvent e) {

	}
```

```
	@Override
	public void mouseExited(MouseEvent e) {

	}

	@Override
	public void mousePressed(MouseEvent e) {
		if(SwingUtilities.isLeftMouseButton(e)) {
			if (deposeSommet) {
				deposeSommet = false;
			}
			else if (creationArete) {

				if (sommetDepartSelectionne) {
					Iterator<Sommet> it =
fenetre.getGraphe().getSommets().iterator();

					while (it.hasNext()) {
						Sommet s = it.next();
						if
(e.getPoint().distance(s.getLocation())<s.getRayon()) {
							areteCreee.setArrivee(s);
							creationArete = false;
							repaint();
						}
					}
				}
				else {
					Iterator<Sommet> it =
fenetre.getGraphe().getSommets().iterator();

					while (it.hasNext()) {
						Sommet s = it.next();
						if
(e.getPoint().distance(s.getLocation())<s.getRayon()) {
							areteCreee.setDepart(s);
							sommetDepartSelectionne = true;
							areteCreee.setArrivee(fantome);
						}
					}
				}
			}
			else {
				Iterator<Sommet> it =
fenetre.getGraphe().getSommets().iterator();

				while (it.hasNext()) {
					Sommet s = it.next();
					if
(e.getPoint().distance(s.getLocation())<s.getRayon()) {
						sommetDeplace = s;
					}
				}
			}
		}
		else if (SwingUtilities.isRightMouseButton(e)) {
			Iterator<Sommet> it = ((HashSet<Sommet>)
```

```
fenetre.getGraphe().getSommets().clone()).iterator();
                    while (it.hasNext()) {
                          Sommet s = it.next();
                          if (e.getPoint().distance(s.getLocation())<s.getRayon()) {
                                fenetre.getGraphe().remove(s);
                          }
                    }

                    Iterator<Sommet> iter =
fenetre.getGraphe().getSommets().iterator();
                    while(iter.hasNext()) {
                          iter.next().setColor(Color.white);
                    }

                    repaint();
              }
        }

        @Override
        public void mouseReleased(MouseEvent e) {
              sommetDeplace = null;
        }

        @Override
        public void mouseDragged(MouseEvent e) {
              if (sommetDeplace != null) {
                    if(sommetDeplace.getRayon()<e.getPoint().getX() &&
e.getPoint().getX()<getWidth()-sommetDeplace.getRayon() &&
sommetDeplace.getRayon()<e.getPoint().getY() && e.getPoint().getY()<getHeight()-
sommetDeplace.getRayon())
                    sommetDeplace.setPosition(e.getPoint());
                    repaint();
              }
        }

        @Override
        public void mouseMoved(MouseEvent e) {
              if (deposeSommet) {
                    sommetDepose.setPosition(e.getPoint());
                    repaint();
              }
              if (creationArete && sommetDepartSelectionne) {
                    fantome.setPosition(e.getPoint());
                    repaint();
              }
        }

        public Graphe getGraphe() {
              return fenetre.getGraphe();
        }

        public void setDeposeSommet(boolean b) {
              deposeSommet = b;
        }

        public void setSommetDepose(Sommet s) {
              sommetDepose = s;
```

```
	}

	public void setAreteCreee(Arete a) {
		areteCreee = a;
	}

	public void setSommetDepartSelectionne(boolean b) {
		sommetDepartSelectionne = b;
	}

	public void setCreationArete(boolean b) {
		creationArete = b;
	}

	public Fenetre getFenetre() {
		return fenetre;
	}
}
```

Classe creerAreteAction

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.util.Iterator;

import javax.swing.*;

public class creerAreteAction extends AbstractAction {
	private DessinPanel panel;

	public creerAreteAction(DessinPanel fen, String nom) {
		super(nom);
		panel = fen;
	}

	@Override
	public void actionPerformed(ActionEvent e) {
		Iterator<Sommet> it = panel.getGraphe().getSommets().iterator();
		while(it.hasNext()) {
			it.next().setColor(Color.white);
		}
		panel.repaint();

		Arete a = new Arete();
		panel.getGraphe().ajouteArete(a);

		panel.setCreationArete(true);
		panel.setSommetDepartSelectionne(false);
		panel.setAreteCreee(a);
	}

}
```

Classe `creerSommetAction`

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.util.Iterator;

import javax.swing.*;

public class creerSommetAction extends AbstractAction {
	private DessinPanel panel;

	public creerSommetAction(DessinPanel fen, String nom) {
		super(nom);
		panel = fen;
	}

	public void actionPerformed(ActionEvent arg0) {

		Iterator<Sommet> it = panel.getGraphe().getSommets().iterator();
		while(it.hasNext()) {
			it.next().setColor(Color.white);
		}
		panel.repaint();

		Sommet s = new Sommet();
		panel.getGraphe().ajouteSommet(s);

		panel.setDeposeSommet(true);
		panel.setSommetDepose(s);
	}

}
```

Classe `calculComposanteAction`

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.util.*;

import javax.swing.*;

public class calculComposantesAction extends AbstractAction {
	private Fenetre fenetre;

	public calculComposantesAction(Fenetre fen, String nom) {
		super(nom);
		fenetre = fen;
	}

	@Override
	public void actionPerformed(ActionEvent e) {
			Map<Sommet,HashSet<Sommet>> partition =
fenetre.getGraphe().kosaraju();
```

```
                    HashSet<HashSet<Sommet>> composantes =
collectionToSet(partition.values());

                    TableFenetre tableFenetre = new TableFenetre(composantes,fenetre);
                    tableFenetre.setVisible(true);

              fenetre.repaint();
       }

       private HashSet<HashSet<Sommet>> collectionToSet(Collection<HashSet<Sommet>> c) {
              HashSet<HashSet<Sommet>> res = new HashSet<HashSet<Sommet>>();

              Iterator<HashSet<Sommet>> it = c.iterator();
              while(it.hasNext()) {
                    res.add(it.next());
              }

              return res ;
       }

}
```

Classe TableFenetre

```
import javax.swing.*;
import javax.swing.table.AbstractTableModel;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Window;
import java.util.*;

public class TableFenetre extends JFrame {
       //atributs
       HashSet<HashSet<Sommet>> composantes;
       Fenetre fenetre;
       HashSet<Sommet>[] tableauUtile;

       //constructeurs
       public TableFenetre(HashSet<HashSet<Sommet>> c, Fenetre f) {
              super();
              composantes = c;
              fenetre = f;
              build();
       }

       //méthodes
       private void build() {
              setTitle("Infos sur les composantes connexes");
              setSize(600,200);
              setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
              setContentPane(buildContentPane());
```

```
	}

	private JPanel buildContentPane() {
		JPanel panel = new JPanel();
		JTable table;
		JButton coloriageBouton;

		panel.setLayout(new BorderLayout());

		table = new JTable(new MyTableModel(composantes));

		JScrollPane scrollPane = new JScrollPane(table);

		table.setDefaultRenderer(Color.class, new ColorRenderer(true));
		table.setDefaultEditor(Color.class, new ColorEditor());
		table.setPreferredScrollableViewportSize(new Dimension(600, 100));
		panel.add(scrollPane, BorderLayout.PAGE_START);

		coloriageBouton = new JButton(new colorierAction(this, fenetre,
"Recolorier", table.getModel(), tableauUtile, fenetre.getGraphe()));
		panel.add(coloriageBouton, BorderLayout.PAGE_END);

		return panel;
	}

	class MyTableModel extends AbstractTableModel {
	  private String[] columnNames = {"Nom",
	                                  "Couleur",
	                                  "Sommets de la composante"};

	  private Object[][] resultat  = new Object[composantes.size()][3];

	  public MyTableModel(HashSet<HashSet<Sommet>> composantes) {
		Iterator<HashSet<Sommet>> itSet = composantes.iterator();

		tableauUtile = new HashSet[composantes.size()];

		int i = 1;
		for(Object[] array : resultat) {
			HashSet<Sommet> set = itSet.next();

			tableauUtile[i-1] = set;

			array[0] = "Composante n°"+i++;
			array[1] = couleur(set);
			array[2] = "";

			Iterator<Sommet> itSom = set.iterator();
			while(itSom.hasNext()) {
				array[2] = array[2]+ ""+ itSom.next().num();

				if(itSom.hasNext()) {
					array[2] = array[2]+", ";
				}

			}
		}
```

```
        }

        public int getColumnCount() {
            return columnNames.length;
        }

        public int getRowCount() {
            return resultat.length;
        }

        public String getColumnName(int col) {
            return columnNames[col];
        }

        public Object getValueAt(int row, int col) {
            return resultat[row][col];
        }

        public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
        }

        public boolean isCellEditable(int row, int col) {
            if (col < 1) {
                return false;
            } else {
                return true;
            }
        }

        public void setValueAt(Object value, int row, int col) {

            resultat[row][col] = value;
            fireTableCellUpdated(row, col);

        }

    }

       private Color couleur(HashSet<Sommet> set) {
              Iterator<Sommet> it = set.iterator();

              return it.next().getCouleur();
        }

}
```

Classe colorierAction

```
import java.awt.Color;
import java.awt.Component;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.util.HashSet;
```

```java
import java.util.Iterator;

import javax.swing.AbstractAction;
import javax.swing.JFrame;
import javax.swing.table.TableModel;


public class colorierAction extends AbstractAction {
	//attributs
	TableModel table;
	Graphe graphe;
	HashSet[] tableau;
	JFrame fenetre;
	JFrame fenetreDeBase;

	//constructeurs
	public colorierAction(JFrame f, JFrame fbase, String nom, TableModel t, HashSet[]
tab, Graphe g) {
		super(nom);
		table = t;
		tableau = tab;
		fenetre = f;
		fenetreDeBase = fbase;
		graphe = g;
	}

	//méthodes
	public void actionPerformed(ActionEvent arg0) {

		int i=0;

		while (i !=tableau.length) {
			Color couleur = (Color) table.getValueAt(i, 1);
			Iterator<Sommet> itSom = tableau[i].iterator();
			while(itSom.hasNext()){
				Sommet reference = itSom.next();
				Iterator<Sommet> itGrph = graphe.getSommets().iterator();
				while(itGrph.hasNext()) {
					Sommet som = itGrph.next();
					if (som.equals(reference)) {
						som.setColor(couleur);
					}
				}
			}

			i++;
		}

		fenetre.setVisible(false);
		fenetreDeBase.repaint();
	}


}
```

Classe ColorEditor

```
import javax.swing.*;
import javax.swing.table.TableCellEditor;

import java.awt.Color;
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ColorEditor extends AbstractCellEditor
                         implements TableCellEditor,
                                    ActionListener {
    Color currentColor;
    JButton button;
    JColorChooser colorChooser;
    JDialog dialog;
    protected static final String EDIT = "edit";

    public ColorEditor() {
        button = new JButton();
        button.setActionCommand(EDIT);
        button.addActionListener(this);
        button.setBorderPainted(false);

        //Set up the dialog that the button brings up.
        colorChooser = new JColorChooser();
        dialog = JColorChooser.createDialog(button,
                                        "Choisissez une couleur",
                                        true,  //modal
                                        colorChooser,
                                        this,  //OK button handler
                                        null); //no CANCEL button handler
    }

    public void actionPerformed(ActionEvent e) {
        if (EDIT.equals(e.getActionCommand())) {
            //The user has clicked the cell, so
            //bring up the dialog.
            button.setBackground(currentColor);
            colorChooser.setColor(currentColor);
            dialog.setVisible(true);

            fireEditingStopped(); //Make the renderer reappear.

        } else { //User pressed dialog's "OK" button.
            currentColor = colorChooser.getColor();
        }
    }

    //Implement the one CellEditor method that AbstractCellEditor doesn't.
    public Object getCellEditorValue() {
        return currentColor;
    }

    //Implement the one method defined by TableCellEditor.
    public Component getTableCellEditorComponent(JTable table,
```

```
                                                    Object value,
                                                    boolean isSelected,
                                                    int row,
                                                    int column) {
        currentColor = (Color)value;
        return button;
    }
}
```

Classe ColorRenderer

```
/*
 * ColorRenderer.java (compiles with releases 1.2, 1.3, and 1.4) is used by
 * TableDialogEditDemo.java.
 */

import javax.swing.BorderFactory;
import javax.swing.JLabel;
import javax.swing.JTable;
import javax.swing.border.Border;
import javax.swing.table.TableCellRenderer;
import java.awt.Color;
import java.awt.Component;

public class ColorRenderer extends JLabel
                           implements TableCellRenderer {
    Border unselectedBorder = null;
    Border selectedBorder = null;
    boolean isBordered = true;

    public ColorRenderer(boolean isBordered) {
        this.isBordered = isBordered;
        setOpaque(true); //MUST do this for background to show up.
    }

    public Component getTableCellRendererComponent(
                            JTable table, Object color,
                            boolean isSelected, boolean hasFocus,
                            int row, int column) {
        Color newColor = (Color)color;
        setBackground(newColor);
        if (isBordered) {
            if (isSelected) {
                if (selectedBorder == null) {
                    selectedBorder = BorderFactory.createMatteBorder(2,5,2,5,
                                              table.getSelectionBackground());
                }
                setBorder(selectedBorder);
            } else {
                if (unselectedBorder == null) {
                    unselectedBorder = BorderFactory.createMatteBorder(2,5,2,5,
                                              table.getBackground());
                }
                setBorder(unselectedBorder);
            }
```

```
        }

        setToolTipText("RGB value: " + newColor.getRed() + ", "
                                     + newColor.getGreen() + ", "
                                     + newColor.getBlue());
        return this;
    }
}
```