Théorie des Graphes - Rapport de Projet

Gaëtan Ferry, Tristan Louet, Adrien Smondack

Janvier 2013

Résumé

Ce projet, proposé par Carla Selmi, à été réalisé dans le cadre du module d'Algorithmique 3 - Théorie des Graphes.
Ce rapport à été rédigé en LaTeX.

Sommaire

Introduction

Deux types de coloriage de graphe non orienté ont été très étudié : celui qui permet d'associer une couleur à chaque sommet de telle manière qu'il n'y ait pas deux sommets adjacents de la même couleur et celui qui permet de colorier chaque arêtes de telle manière qu'il n'y ait pas deux arêtes consécutives de la même couleur. Le graphe est supposé non réflexif. Le principal problème qui se pose lors de la coloration d'un graphe est celui de la minimisation des couleurs, problème qui n'est pas simple dans la mesure où la contrainte interdisant deux couleurs consécutives aurait tendance à faire augmenter le nombre de couleurs à utiliser.

L'objectif de ce projet était de réaliser une application permettant de dessiner avec la souris puis de colorier un graphe non orienté. L'application implante des graphes de deux manières différentes : une implantation par liste de successeurs et une implantation par liste d'adjacence. De plus, elle permet de sauvegarder et de charger des graphes (dans la mesure où on à utilisé cette application même pour les créer) entre d'autres fonctionnalités détaillées plus tard.

Chapitre 1

Réflexion sur les coloriages de graphes

1.1 Pourquoi les coloriages de graphes ont été étudier ?

Rappelons nous que la théorie des graphes est née au cours du XVII ème siècle avec le problème des sept ponts de Könisberg. Ce problème consistait à trouver un chemin permettant de visiter chacun des sept ponts sans passer deux fois par le même. Depuis, de nombreux mathématiciens ont développé ce domaine et ses applications, si bien qu'aujourd'hui, de nombreux problèmes peuvent être résolus avec des graphes.

L'étude de la coloration des graphes fut lancée lors de la deuxième moitié du XIX$^{\text{ème}}$ siècle par Francis Guthrie, alors qu'il tentait de colorer une carte des comtés d'Angleterre. Le problème était de trouver une méthode de coloriage permettant d'utiliser un minimum de couleurs et tel qu'aucun comté ne soit de la même couleur que l'un de ses voisins. Ceci donnera naissance , par la suite, au « théorème des 4 couleurs », dont le résultat fut démontré seulement au XX$^{\text{ème}}$ siècle par Kenneth Appel et Wolfgang Haken.

1.2 Dans quels genres de problème sont-ils utilisé ?

Comme nous l'avons dit précédemment, la coloration des graphes est notamment utilisée pour les problème de coloration de cartes et autres objets assimilables à des graphes planaires. Mais elle peut être également utilisée dans des domaines plus professionnels tels que la télécommunication ou le transport :

Exemples :

Allocation de fréquences dans les réseaux GSM

Objectif : Attribuer aux antennes relais des bandes de fréquences pour communiquer avec les usagers, tout en minimisant le nombre de fréquences utilisées et les interférences.

Noeuds : les antennes relais.

Arêtes : entre deux antennes trop proches géographiquement l'une de l'autre (niveau d'interférence trop important).

Couleurs : les canaux de fréquences radio.

Allocation de niveau de vol

Objectif : Attribuer un niveau de vol aux avions pour éviter les conflits aériens tout en minimisant le nombre de niveaux de vol utilisés.

Noeuds : les avions.

Arêtes : deux avions proches l'un de l'autre, risquant de se disputer l'espace aérien s'ils sont sur le même niveau de vol.

Couleurs : les niveaux de vol.

Chapitre 2

Les algorithmes de coloriages utilisés

2.1 Structures de données

Noeuds & Arêtes : Commencons par définir ce que sont un noeud et une arête dans un graphe :

- Les noeuds sont des objets uniques représentant un élément du type étudié (ville, valeurs, etc).
- Les arêtes traduisent une relation entre deux noeuds.

Pour représenter ces éléments, nous avons définit les structures de données suivantes :

```
Type Node =
    Enregistrement
      num : entier ;
      color : couleur ;
    fin ;
Type Vertex =
    Enregistrement
      nodes : tableau[2] de Node ;
      color : couleur ;
    fin ;
```

Sur lesquelles on définis les opérations suivantes :

```
Node :
    {Retourne la couleur du sommet}
  - getColor() : couleur ;
    {Retourne le numéro du sommet}
  - getNumber() : entier ;
  - {Donne la couleur c au sommet}
    setColor(couleur c) ;
  - {Donne le numéro num au sommet}
    setNumber(entier num) ;
```

Vertex :
{Retourne la couleur de l'arête}
- getColor() : couleur ;
{Retourne le tableau des Node en relation par cette arête}
- getNodes() : tableau[2] de Node
{Donne la couleur c à l'arête}
- setColor(couleur c) ;

Graphe : Un graphe est un ensemble de noeuds reliés par des arêtes. Pour représenter un graphe, il existe deux manières principales de procéder : par liste de successeurs ou par matrice d'adjacence.

Nous avons donc implanter les graphes en utilisant ces deux standards :

Type ListGraph =
Enregistrement
lists : tableau[NodesNb] d'ensembles de Node ;
vertexes : ensemble de Vertex ;
fin ;

Type MatrixGraph =
Enregistrement
matrix : tableau[NodesNb][NodesNb] d'entiers ;
vertexes : ensemble de Vertex ;
fin ;

Sur lesquels ont défini les opérations suivantes :
{Retourne l'ensemble des noeuds du graphe}
- getNodes() : ensemble de Node ;
{Retourne l'ensemble des arêtes du graphe}
- getVertexes() : ensemble de Vertex ;
{Retourne l'ensemble des arêtes de n}
- getVertexes(Node n) : ensemble de Vertex ;
{Retourne le nombre de noeuds du graphe}
- getNodesNb() : entier ;
{Retourne le nombre d'arêtes de n}
- getVertexesNb(Node n) : entier ;
{Retourne le nombre d'arêtes du graphe}
- getVertexesNb() : entier ;
{Retourne true si n1 et n2 sont reliés par une arête quelconque}
- areAdjacent(Node n1, n2) : booléen ;
{Retourne true si v1 et v2 ont un noeud en commun}
- areAdjacent(Vertex v1, v2) : booléen ;
{Retourne l'ensemble des noeuds adjacents à n}
- getAdjacent(Node n) : ensemble de Node ;
{Retourne l'ensemble des arêtes adjacentes à v}
- getAdjacent(Vertex v) : ensemble de Vertex ;
{Ajoute un nouveau noeud au graphe}
- addNode() ;
{Supprime n du graphe, ainsi que ses arêtes}
- removeNode(Node n) ;
{Supprime n ainsi que ses arêtes}
- removeNodes(ensemble de Node n) ;

{Ajoute v au graphe}
- addVertex(Vertex v) ;
{Supprime v du graphe}
- removeVertex(Vertex v) ;
{Supprime v du graphe}
- removeVertexes(ensemble de Vertex v) ;
{Colore le noeud n avec la couleur c}
- colorizeNode(Node n, couleur c) ;
{Colore tous les noeuds du graphe}
- colorizeNodes() ;
{Colore toutes les arêtes du graphe}
- colorizeVertexes() ;

2.2 Coloriage des noeuds

Pour le coloriage des noeuds d'un graphe, on utilise l'algorithme DSATUR qui consiste à calculer à chaque nouvelle étape les degrés de saturation afin de déterminer quel noeud doit être colorié.

Cet algorithme est l'un des plus performant qui ait été mis au point à ce jour (avec celui de Welsh et Powel), capable de fournir en un temps polynomial une coloration correcte dans 90% des cas. Cependant cela ne signifie pas que la coloration sera optimale dans tous les cas, l'ordre dans lequel on colore les noeuds influe sur l'efficacité de l'algorithme. Les 10% restants sont des cas irréalisables. DSATUR appartient à la catégorie des algorithmes dits heuristiques, c'est à dire fournissant une solution non optimale mais réalisable.

Considerons le graphe $G = (S, A)$. Pour chaque noeud $s \in S$, on calcule le degré de saturation $DSAT(s)$ de la manière suivante :

Soit $D(s)$ le degré de s, et $C(s)$ le nombre de couleurs utilisées dans le voisinage de s (excepté le noir, étant la couleur par défaut).

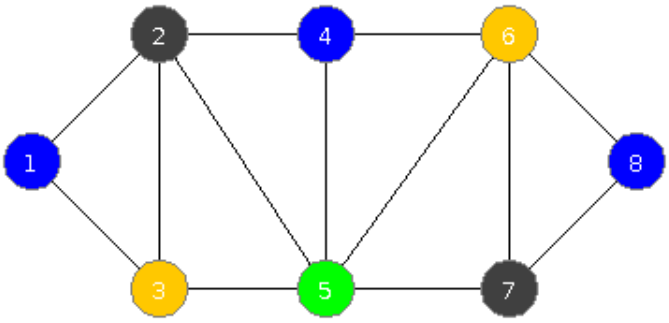
$$DSAT(s) = \begin{cases} D(s) & \text{Si aucun voisin de s n'est coloré} \\ C(s) & \text{Sinon} \end{cases}$$

DSATUR est un algorithme de coloration séquentiel, c'est à dire qu'il ne colorie qu'un seul noeud à la fois, les propriétés suivantes devant être vérifiées en permanence :

- Initialement, le graphe ne doit pas être colorié.
- On colorie un sommet non coloré.
- L'algorithme prend fin quand tous les noeuds sont colorés.

Algorithme DSATUR :

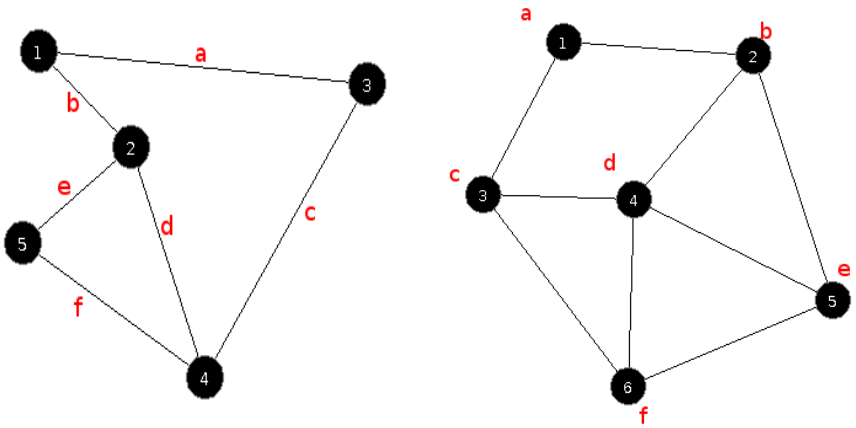
1. Ordonner les sommets par ordre décroissant de degré.
2. Colorer un sommet de degré maximum avec la première couleur disponible.
3. Choisir un sommet non coloré avec un $DSAT$ maximum, en cas de conflit on choisit celui de degré maximum.
4. Colorier ce dommet par la première couleur disponible.
5. S'il reste des noeuds à colorier, alors on revient à l'étape 3.



2.3 Coloriage des arcs

Le coloriage des arcs est un problème analogue à celui du coloriage des noeuds dans le sens où il est possible de construire un graphe dans lequel chaque noeud représente un arc du graphe d'origine et où deux noeuds sont en relation (c'est à dire reliés par un arc) lorsque les arcs associés à ces noeuds sont adjacents dans le graphe d'origine.

Ainsi la convertion du graphe de gauche ci-dessous, donnera le graphe de droite :



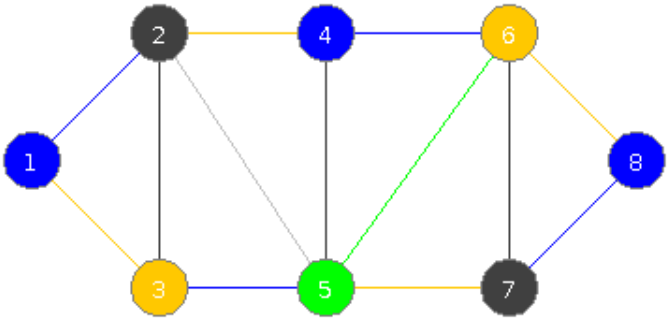
Dans ce but, nous avons mis au point un algorithme de convertion de graphe que nous décrirons comme suit :

Soit $G = (S, A)$ et $G' = (S', A')$ deux graphes, où S et S' sont les ensembles des sommets respectifs de G et G', A et A' sont les ensembles des arêtes respectifs de G et G', et $t : A \to S'$ une application telle que $\forall\ a \in A$, $\exists$ un unique $s' \in S'$ tel que $a \xrightarrow{t} s'$.

Algorithme CONVERT_GRAPH :

1. $\forall\ a \in A$, créer un sommet $s' \in S'$.
2. $\forall\ a1, a2 \in A$ tel que $a1$ et $a2$ sont adjacentes, créer une arête entre $t(a1)$ et $t(a2)$.
3. Colorier G' (via DSATUR).

A ce stade, nous avons un graphe G' dont les couleurs des sommets correspondent à celles des arêtes de G qui leurs sont associés par la relation t, et donc $\forall\ s' \in S'$, il suffit de colorer $t^{-1}(s')$ de la couleur de s'.



Chapitre 3

Méthode de codage pour la sauvegarde des fichiers

Pour la sauvegarde de graphe, nous avons décidé de faire hériter notre interface *Graph* de l'interface *Serializable*. Cette interface ne donne accès à aucune méthode supplémentaire mais permet simplement de désigner une classe comme sérialisable.

La sérialisation fait appel aux objets *ObjectOutputStream* et *ObjectInputStream*. Ces classes permettent respectivement de sérialiser et désérialiser un objet. Si l'on tente de sérialiser un objet non sérialisable, une exception de type NotSerializableException est levée.

La sérialisation consiste à sauvegarder un objet complet sur un fichier, d'où il pourra être restauré à tout moment (désérialisation), en sauvegardant tous les attributs. Par conséquent, les classes *PositionedNode* et *Vertex* sont aussi sérialisables puisque les graphes sont des ensembles de noeuds et d'arêtes.

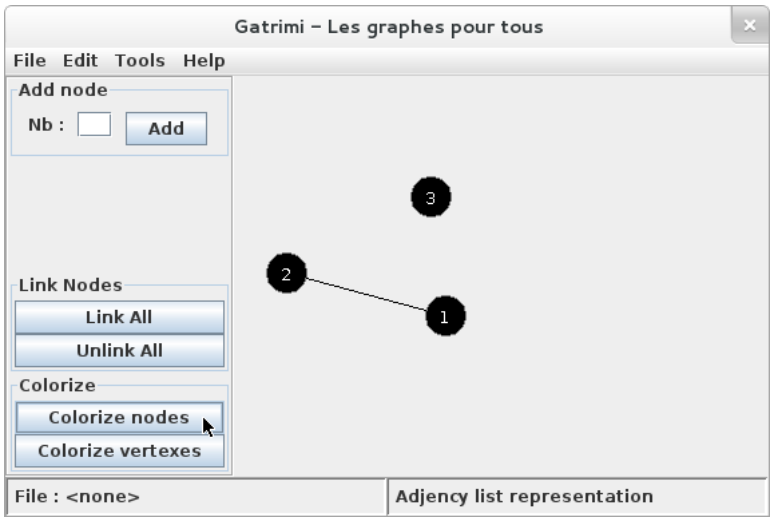
Lors de la sauvegarde, les graphes sont toujours sauvegardés en mode liste de successeurs. Ce format standard permet de ne pas se poser de question lors du chargement, le graphe est convertit dans le format utilisé à ce moment là. Cela permet également de garder une représentation creuse du graphe en question, ce qui représente un gain de mémoire par rapport à la représentation matricielle.

Chapitre 4

La structure du programme

4.1 La fenêtre d'application

La fenêtre de l'application de décompose en quatre éléments :
- la barre de menu
- la barre d'outils latérale
- la feuille de dessin
- la barre d'état



Arborescence graphique de répartition des composants :

La barre de menu : Il s'agit d'une barre de menu standard décomposée en quatre parties.
- Le menu *File*, contenant tous les éléments nécéssaire au fonctionnement de base de l'application
  1. New : Supprime le graphe en cours d'édition dans la fenêtre de dessin et le remplace par un nouveau graphe vierge. Si le graphe en cours d'édition était associé à un fichier, et n'était pas sauvegardé, une fenêtre de confirmation apparaîtra vous permettant de changer d'avis

2. Open... : Supprime le graphe en cours d'édition dans la fenêtre de dessin et le remplace par un graphe précédemment sauvegardé, que vous pourrez choisir grace à une fenêtre de sélection de fichier. Si le graphe en cours d'édition était associé à un fichier, et n'était pas sauvegardé, une fenêtre de confirmation apparaîtra vous permettant de changer d'avis
3. Save : Sauvegarde le graphe en cours d'édition dans le fichier qui lui est associé. Cette élément n'est utilisable que lorsque le graphe en cours d'édition est associé à un fichier. Dans le cas contraire, l'élément sera grisé.
4. Save as... : Fait apparaître un sélectionneur de fichier, vous permettant de sauvegarder le graphe en cours d'édition dans un nouveau fichier
5. Reload : Recharge le graphe en cours d'édition dans l'état qui était le sien lors de la dernière sauvegarde ou ouverture de fichier. Si le graphe en cours d'édition était associé à un fichier, et n'était pas sauvegardé, une fenêtre de confirmation apparaîtra vous permettant de changer d'avis. Cette option n'est disponible que lorsque le graphe en cours d'édition est associé à un fichier
6. Close : Supprime le graphe en cours d'édition et ferme le fichier qui lui est associé. Si le graphe en cours d'édition était associé à un fichier, et n'était pas sauvegardé, une fenêtre de confirmation apparaîtra vous permettant de changer d'avis. Cette option n'est disponible que lorsque le graphe en cours d'édition est associé à un fichier
7. Quit : Ferme l'application. Si le graphe en cours d'édition était associé à un fichier, et n'était pas sauvegardé, une fenêtre de confirmation apparaîtra vous permettant de changer d'avis

– Le menu *Edit*, contenant quelques options d'éditions des graphes
1. Undo : Non encore implémenté. Annule la dernière action exécutée par l'utilisateur
2. Redo : Non encore implémenté. Refait la dernière action annulée par l'utilisateur
3. Clear : Supprime tous les noeuds et arcs du graphe en cours d'édition
4. Preferences : Permet de choisir le mode de représentation interne des graphes. Il y a deux modes de représentation disponibles, le mode de représentation par matrice d'adjacence, et celui par liste de successeurs

– Le menu *Tools*, contenant quelques outils utiles
1. Export PNG : Permet d'enregistrer une image, au format png, du graphe actuellement en cours d'édition. Lors de l'utilisation de cet outil, une fenêtre apparaîtra, vous permettant de choisir le nom de votre future image
2. Random Graph : Permet la génération aléatoire d'un graphe. Le graphe en cours d'édition sera remplacé par un graphe généré aléatoirement dont le nombre de noeuds sera choisit par l'utilisateur à l'aide d'une fenêtre de sélection.

3. Colorize... : Permet de coloriser, au choix, les noeuds ou les arcs du graphe

– Le menu *Help*, fournissant des informations générales sur l'application
  1. About : Fait apparaître une fenêtre contenant des informations générales sur l'application
  2. Contents : Ouvre une boite de dialogue vous invitant à consulter le manuel fournit avec ce rapport.

La barre d'outils latérale : La barre d'outils contient quelques outils permettant d'agir directement dur le graphe en cours d'édition.
– Add Node : Cette partie de la barre d'outils, vous permet d'ajouter directement un grand nombre de noeuds au graphe en cours d'édition. Un clic sur le bouton *Add* ajoutera au graphe autant de noeuds que spécifié dans le champ de texte adjacent. Tous les noeuds apparaîtront dans le coin supérieur gauche de la fenêtre de dessin, légèrement espacés les uns des autres
– Link all : Relie tous les noeuds du graphe en cours d'édition les uns avec les autres. Cette option n'est disponible que si le graphe contient au moins un noeud
– Unlink all : Supprime tous les arcs du graphe en cours d'édition. Cette option n'est disponible que si le graphe contient au moins un arc
– Colorize : Permet de coloriser au choix les noeuds ou les arcs du graphe. Ces options ne sont disponibles que si le graphe contient au moins un noeud ou un arc, selon le mode de colorisation choisi

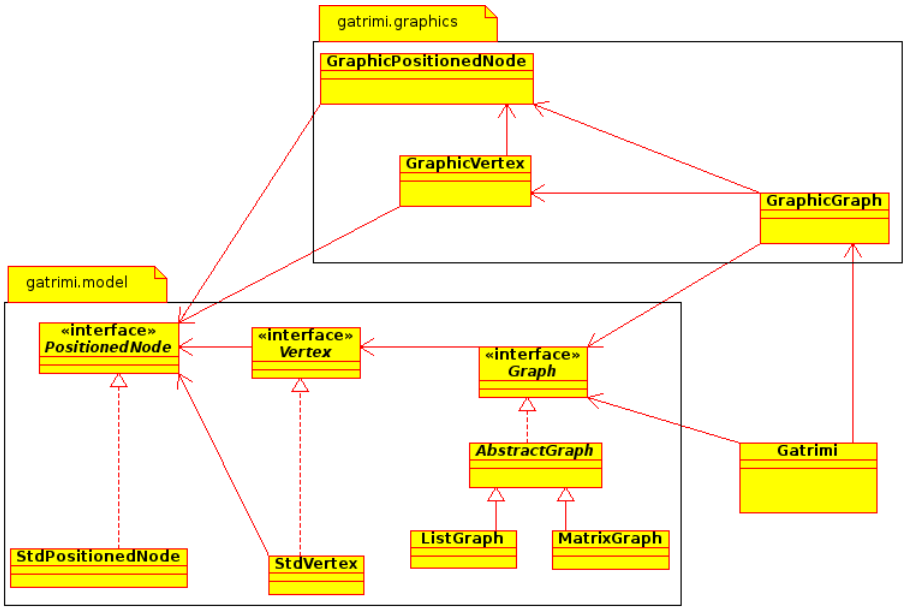
La feuille de dessin : Cette fenetre permet de visualiser le graphe au fur et à mesure de son édition. Elle contient elle aussi des outils de base par l'intermédiaire de menus contextuels.
– Ajout d'un noeud : Pour ajouter un noeud a votre graphe, il vous suffit de cliquer à l'aide du bouton droit de la souris puis de cliquer sur le bouton correspondant du menu contextuel
– Suppression d'un noeud : Pour supprimer un noeud, il suffit de cliquer sur celui-ci à l'aide du bouton droit de la souris puis de cliquer sur le bouton *Delete node* du menu contextuel
– Déplacement d'un noeud : Pour déplacer un noeud, restez cliqué avec le bouton gauche de votre souris sur le noeud en question et déplacez votre souris, avant de relacher le bouton à l'emplacement souhaité
– Ajout d'un arc : Pour ajouter un arc entre deux noeuds n1 et n2, il faut d'abord sélectionner un des deux noeuds en cliquant gauche sur celui-ci (un cadre rouge entourant le noeud apparaîtra), cliquer droit sur le second noeud puis cliquer sur le bouton *Add vertex* du menu contextuel.
– Suppression d'un arc : Pour supprimer un arc entre deux noeuds, il faut utiliser la même procédure que pour l'ajout d'un arc mais en finissant par cliquer sur le bouton *Remove vertex* du menu contextuel
– Suppression de plusieurs arc : Il est possible de supprimer tous les arcs reliés à un noeud. Pour se faire, il suffit de cliquer droit sur le noeud puis de cliquer sur le bouton *Remove all vertexes* du menu contextuel.

La barre d'état : Cette barre fournit quelques informations concernant l'état actuel de l'application. Elle se décompose en deux parties distinctes.

La première est la barre de fichier. Cette partie de la barre sert à indiquer à l'utilisateur quel fichier est actuellement associé au graphe en cours d'édition. Si aucun fichier n'est associé au graphe, cette partie de la barre contiendra le texte *<none>*. Lorsqu'un fichier est associé au graphe et que le graphe en cours d'édition ne correspond pas à celui contenu dans le fichier (il n'est pas enregistré), la barre contiendra le symbole *.

La seconde barre contient uniquement le mode de représentation interne des graphes. Si ceux-ci sont représentés par des listes de successeurs, la barre contiendra le texte *Adjency list representation*, si au contraire le mode de représentation choisit est la représentation par matrice d'adjacence, elle contiendra le texte *Matrix representation*.

4.2 Diagramme de l'application



4.3 Problèmes rencontrés

Un problème de hash code... Afin d'implanter des graphes, l'application utilise beaucoup d'ensemble (interface *Set<E>* en Java), que ce soit des ensembles de noeuds ou des ensemble d'arcs. Afin de s'assurer qu'un ensemble ne contient chacun de ses éléments qu'une seule et unique fois, la classe *HashSet<E>* (utilisée afin d'avoir des ensembles ayant une complexité efficace dans ses opérations grâce à l'utilisation en interne de tables de hachage) utilise la méthode *equals*.
Nous avions donc initialement redéfini les méthodes *equals* et *hashCode* afin que deux noeuds ou arcs considérés égaux (pour les noeuds, cela signifie ayant le même numéro et pour deux arcs, cela signifie reliant les mêmes noeuds) aient

aussi la même valeur de hachage.
Cependant, après avoir développé bien plus l'application, nous avons rencontrer des problèmes avec la méthode *removeNode*, enlevant un noeud au graphe et renumérotant tout les autres noeuds afin de garder une cohérence.
Il se trouve qu'en renumérotant les noeuds, nous modifions le comportement de leur méthode *hashCode* puisque celle-ci se basait sur le numéro du noeud. Hors, dans la structure de donnée interne des ensemble de noeuds, le hashCode d'origine (avec le numéro originel du noeud) était utilisé pour repérer celui-ci. Nous nous somme donc finalement rendu compte que les noeuds ne pouvaient plus être enlevés ou récupérés dans l'ensemble des noeuds puisque l'ont ne les recherchait pas avec leur valeur de hachage d'origine mais une nouvelle valeur. C'est pourquoi, compte tenu de l'avancement du développement qui, trop avancé, ne permettait pas de repenser les algorithmes et les structures de données afin de corriger le code, nous avons dû "tricher" en basant finalement la valeur de hachage sur l'adresse mémoire de l'objet Java représentant le noeud, cette adresse ne changeant jamais pour un objet donné. Malheureusement, cela implique que pour deux noeuds égaux (retournant la valeur *true* pour la méthode *equals*) ne retournent pas la même valeur de hachage, ce qui n'est pas appréciable ; nous avons donc dû construire le logiciel compte tenu de ce détail.
Si le temps le permettait, cela serait certainement la première, mais aussi plus difficile, des choses à remanier.

Ouvertures possibles

Plusieurs améliorations pourraient être apportées à l'application. D'un point de vue technique, l'application est sujette à des ralentissements, voire à des plantages complets, lorsque des graphes trop grands sont manipulés. Ainsi, selon la puissance de l'ordinateur supportant l'application, il est possible qu'à partir d'une centaine de noeuds les actions deviennent moins fluides. Au delà, l'application peut parfois ne plus répondre et doit être terminée manuellement. Une amélioration technique de l'application pourrait, à défaut de supprimer ces bugs inhérents à l'implantation de graphes en Java avec une mémoire limitée, permettre au moins une prévention de ces problèmes. Ainsi l'application détectant que certaines limites sont atteintes pourrait avertir l'utilisateur qu'il n'est pas possible de faire un graphe plus gros et désactiver ses fonctions tant que le graphe est ainsi.
D'un point de vue conceptuel, des idées telles que la décoloration de tous les noeuds et arcs du graphe, la coloration automatique (qui se réadapte à chaque ajout ou retrait du graphe), les options Undo/Redo ou encore la sauvegarde conservant la représentation du graphe ont été émises au sein du projet mais sont restées à l'état de concept.

Table des matières