



RAPPORT PROJET SEMESTRE 4

TEST DU PROJET ALGORITHMIQUE ET KERNELIZATION

Réalisé par

Claire ESPANOL

Southiny PHIAHOUAPHANH

Guillaume CHAFIOL

Florent TORNIL

SEMESTRE 4 DUT INFORMATIQUE

ANNEE UNIVERSITAIRE 2016-2017

SOMMAIRE

Introduction	3
Tests par interface.....	4
Tests de validité.....	4
Tests de facilité d'utilisation.....	6
Tests de performance.....	9
Fiabilité.....	13
Sécurité.....	13
Maintenabilité.....	13
Portabilité.....	13
Tests unitaires.....	14
Classe Couple.....	14
Classe Test.....	17
Classe Kernel.....	20
Classe Regle.....	27
Tests de performance.....	30
Conclusion.....	33
Annexes.....	35

PROJET DE TEST

ALGORITHMIQUE ET KERNELIZATION

Au cours de notre semestre 4 de DUT Informatique, nous avons testé un projet réalisé par d'autres étudiants de l'IUT.

Ce projet traite un problème particulier : l'existence ou non d'un vertex cover.

Tout d'abord, il nous a fallu comprendre le but du projet et prendre en main un code que nous n'avons pas réalisé. L'algorithme étant codé en langage JAVA, nous avons pu utiliser les connaissances que nous avons acquises en cours de Validation et Test.

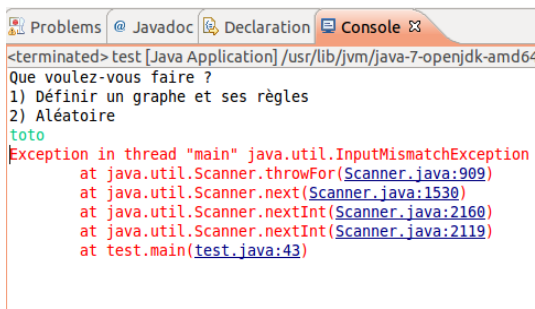
Le programme que nous avons dû tester permet de simplifier un graphe afin de faciliter son traitement, en appliquant plusieurs règles. Nous allons donc vérifier que les fonctions de ce programme permettent de simplifier correctement et efficacement un graphe, tout en respectant les conditions de simplification. Pour cela, nous allons dans un premier temps réaliser des tests via l'interface proposée, c'est à dire le terminal. Puis, nous allons effectuer des tests unitaires pour vérifier plus précisément chaque fonction.

Analyse de Code : Tests par interface

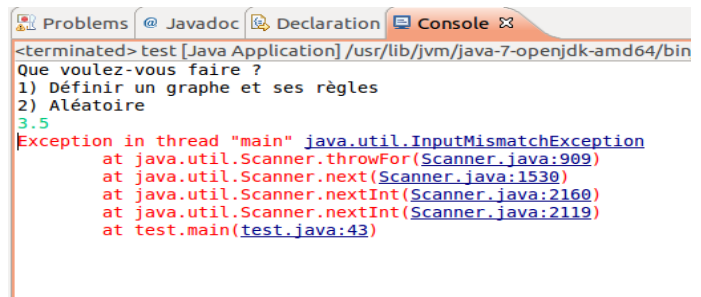
Tests de validité

- Choix du mode de fonctionnement

Nous avons testé le programme en saisissant différents paramètres non conformes aux données attendues afin de voir si le programme gère les cas d'erreurs.



```
<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
toto
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:909)
    at java.util.Scanner.next(Scanner.java:1530)
    at java.util.Scanner.nextInt(Scanner.java:2160)
    at java.util.Scanner.nextInt(Scanner.java:2119)
    at test.main(test.java:43)
```



```
<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
3.5
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:909)
    at java.util.Scanner.next(Scanner.java:1530)
    at java.util.Scanner.nextInt(Scanner.java:2160)
    at java.util.Scanner.nextInt(Scanner.java:2119)
    at test.main(test.java:43)
```

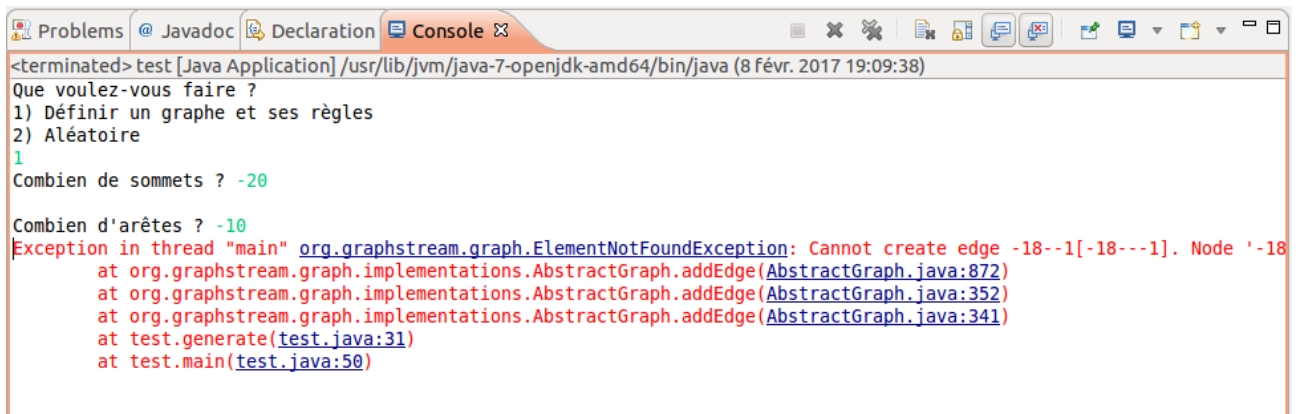
Nous pouvons voir que si nous ne saisissons pas un nombre entier, nous obtenons une exception. Nous pouvons expliquer cette erreur par la lecture des entrées claviers. En effet, le programme récupère les données saisies comme des entiers, comme nous pouvons le voir dans le main. Il n'y a pas de cas d'erreurs traitant les données d'un autre type.

```
public static void main(String args[]) {
    System.out.println("Que voulez-vous faire ?\n1) Définir un graphe et ses règles\n2) Aléatoire");
    Scanner choix = new Scanner(System.in);
    int x = -1, k, sommets, aretes;
    Couple c, c1;
    while(x<1 || x>2) x = choix.nextInt();
```

- Définition du graphe

a) Des problèmes

Il y a le même problème lors du choix du mode « 1 » pour la définition du graphe. Il n'y a pas de contrôle de type. Le problème est même accentué puisque les nombres négatifs entiers ne sont pas gérés. Il aurait fallu contrôler le nombre de sommets et d'arêtes, soit par une exception, soit en demandant à l'utilisateur de saisir une autre valeur tant qu'il n'a pas saisi la bonne.



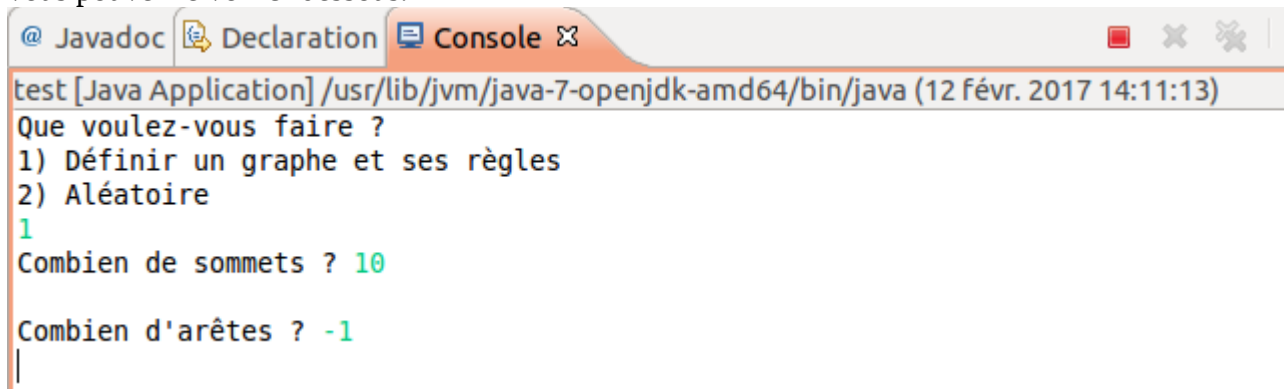
```
<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 19:09:38)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? -20
Combien d'arêtes ? -10
Exception in thread "main" org.graphstream.graph.ElementNotFoundException: Cannot create edge -18--1[-18---1]. Node '-18'
    at org.graphstream.graph.implementations.AbstractGraph.addEdge(AbstractGraph.java:872)
    at org.graphstream.graph.implementations.AbstractGraph.addEdge(AbstractGraph.java:352)
    at org.graphstream.graph.implementations.AbstractGraph.addEdge(AbstractGraph.java:341)
    at test.generate(test.java:31)
    at test.main(test.java:50)
```

Lorsque nous saisissons un nombre d'arêtes incohérent, le système nous envoie un message d'erreur mais le programme continue de s'exécuter. Il manque un arrêt du programme (possible avec `System.exit`)



```
<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 19:33:48)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 5
Combien d'arêtes ? 3000
Erreur nombre de relations trop grand
Définissez un k pour la Kernelization : 2
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les règles 0,1 et 2) 012
|
```

Lorsque nous saisissons un nombre d'arêtes négatif, le programme ne se termine jamais comme vous pouvez le voir ci-dessous.



```
@ Javadoc Declaration Console
test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (12 févr. 2017 14:11:13)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 10
Combien d'arêtes ? -1
|
```

b) Quelques incohérences

Nous pouvons tester le programme en ayant un paramètre de kernelization négatif ce qui est incohérent puisque, nous ne pourrions jamais recouvrir toutes les arêtes du graphe avec un paramètre négatif.



```
Problems @ Javadoc Declaration Console
test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 20:22:01)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 5
Combien d'arêtes ? 4
Définissez un k pour la Kernelization : -11
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les règles 0,1 et 2) 012
Fin de l'optimisation
```

De la même manière, nous pouvons saisir un paramètre de kernelization très grand ce qui est inutile puisque nous pouvons recouvrir toutes les arêtes en prenant un paramètre égal à l'ordre du graphe.

```
Problems @ Javadoc Declaration Console X
test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 20:28:47)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 5
Combien d'arêtes ? 4
Définissez un k pour la Kernelization : 10000
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les règles 0,1 et 2) 012
Regle 0
k=10000
```

Il faudrait donc veiller à ce que le paramètre saisi soit cohérent avec l'ordre du graphe. Ensuite, si nous saisissons des règles différentes de celles indiquées, le programme ne se termine jamais. Il n'y a pas de gestion des paramètres qui ne sont pas dans la norme.

```
<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (14 févr. 2017 17:57:14)
```

```
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 2
Combien d'arêtes ? 1
Définissez un k pour la Kernelization : 2
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les règles 0,1 et 2) 456
```

- Tests sur le fonctionnement du programme

Le fonctionnement du mode aléatoire du programme est cohérent. Les relations entre les sommets changent à chaque exécution. Le fonctionnement du mode « définition du graphe » fonctionne bien du moment que l'utilisateur saisit des paramètres cohérents. Le programme fonctionne correctement du moment que l'utilisateur en a un usage normal. Dès qu'il saisit des données incohérentes, le programme ne fonctionne plus.

Tests de facilité d'utilisation

- Première utilisation

Il est à noter que la première utilisation du programme n'est pas facile surtout pour un novice en informatique. En effet, le programme ne fonctionne pas directement. La librairie GraphStream n'est pas inclus dans les sources, il faut donc aller la chercher sur Internet et l'inclure dans le programme (ce qu'un novice ne peut pas faire). Un manuel d'utilisation serait un plus pour prendre en main le programme.

- Saisie du mode de fonctionnement du programme

Lorsque nous saisissons un nombre entier qui n'est pas 1 ou 2 comme demandé sur l'interface, le programme tourne jusqu'à ce que l'on saisisse la bonne valeur.

```
Problems @ Javadoc Declaration
test [Java Application] /usr/lib/jvm/java-7-
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
3
|
```

```
Problems @ Javadoc Declaration
test [Java Application] /usr/lib/jvm/java-7-c
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
-1
|
```

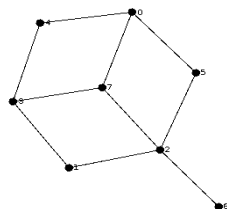
Nous avons trouvé qu'il y avait un manque d'intuitivité. En effet, un utilisateur novice peut se demander pourquoi le programme ne fonctionne pas. Un message « veuillez saisir 1 ou 2 » aurait été apprécié.

- Mode aléatoire

Nous avons testé le mode aléatoire du programme. Lorsque nous arrivons sur l'interface, le choix numéro 2 s'appelle « aléatoire ». Avant de tester le programme, nous nous attendions à un graphe totalement aléatoire. Cependant, le nombre de sommets et d'arêtes reste fixe, seules les relations entre les sommets changent.

```
Problems @ Javadoc Declaration Console
test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/j
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
```

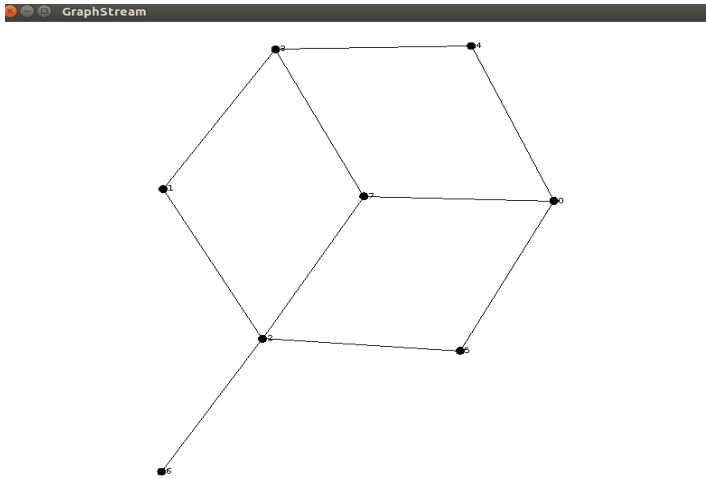
Lorsque nous lançons le mode aléatoire un graphe apparaît puis il faut appuyer sur la touche entrée dans la console pour appliquer la règle de transformation du graphe suivante. Un message nous l'indiquant aurait été le bienvenue.



Premier graphe obtenu

Etat du programme

```
Problems @ Javadoc Declaration Console
test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 18:12:59)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
2
Nous prenons un graphe à 9 sommets et 10 arêtes, et un k=4. Nous appliquons toutes les règles.
```



Deuxième graphe obtenu
Après appui sur Entrée

Etat du programme

```

Problems @ Javadoc Declaration Console
<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 18:30:01)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
2
Nous prenons un graphe à 9 sommets et 10 arêtes, et un k=4. Nous appliquons toutes les règles.
Règle 0
k=4

```

Nous trouvons également dommage de ne pas pouvoir choisir les relations entre les sommets. Il est uniquement possible de choisir le nombre de sommets, d'arêtes et le paramètre de kernelization.

```

<terminated> test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 févr. 2017 19:06:47)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 5
Combien d'arêtes ? 4
Définissez un k pour la Kernelization : 2
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les règles 0,1 et 2) 012

```

- Fin du programme

Nous pouvons constater que le programme ne se termine pas comme vous pouvez le voir sur l'image ci-dessous.


```

@ Javadoc Declaration Console
test [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (9 févr. 2017 17:16:13)
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les 0,1 et 2) 012

Regle 0
k=2

Regle 1
k=1

Regle 0
k=1

Regle 0
k=1

Fin de l'optimisation

```

Programme non terminé

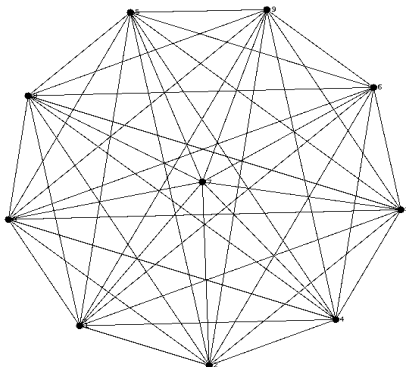
Nous pouvons toutefois lier ce problème à l'utilisation de la librairie GraphStream qui fait que le programme reste actif.

Tests de performance

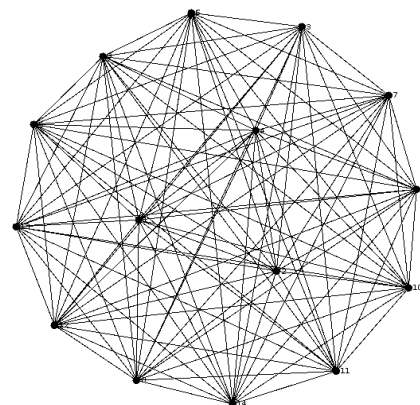
- Tests de la librairie GraphStream

a) lisibilité des graphes

Nous avons testé la lisibilité des graphes obtenus avec GrapStream. Un graphe complet d'ordre inférieur ou égal à 10 reste lisible. Au delà, il est difficile de voir les différentes arêtes. Nous pouvons donc conclure que la librairie est utile pour visualiser des graphes d'ordre plutôt petit.




Graphe complet d'ordre 10



Graphe complet d'ordre 15

Pour une utilisation de graphes plutôt grands (ordre supérieur à 15 pour un graphe complet), il est possible d'utiliser le même programme. En effet, la librairie GraphStream propose d'embellir les graphes en jouant sur les couleurs ou la taille des sommets par exemple.

b) affichage des graphes

Lorsque nous voulons afficher des graphes d'ordre relativement grand (aux alentours de 15 000 sommets), nous avons une image de chargement de la librairie  comme ceci pendant environ une minute avant qu'un graphe ne s'affiche (graphe illisible).

c) Limite des valeurs saisies

Tout d'abord, lorsque nous saisissons une valeur de sommets importante et un petit nombre d'arêtes, nous voyons un message « nombre de relations trop grand ». Cependant ce message s'affiche quand le nombre d'arêtes est supérieur à $(n * (n-1)) / 2$ où n représente le nombre d'arêtes. Nous pouvons voir une première incohérence lorsque les valeurs saisies sont grandes.

```
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 1000000

Combien d'arêtes ? 20
Erreur nombre de relations trop grand

Définissez un k pour la Kernelization : █
```

Ensuite, lorsque nous prenons un nombre de sommets encore plus grand, le programme se bloque indéfiniment.

```
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Aléatoire
1
Combien de sommets ? 1000000000

Combien d'arêtes ? 20
█
```

Puis pour finir, lorsque la valeur saisie est vraiment trop grande, nous avons un message d'erreur.

```
Exception in thread "main" java.util.InputMismatchException: For input string: "
100000000000"
    at java.util.Scanner.nextInt(Scanner.java:2166)
    at java.util.Scanner.nextInt(Scanner.java:2119)
    at test.main(test.java:47)
claire@claire-Inspiron-5558:~/Bureau/AlgoKernel$ █
```

Pour conclure, il ne faut pas permettre à l'utilisateur de saisir n'importe quelle valeur sous peine de voir de nombreuses erreurs apparaître.

- Temps d'exécution

Avec la commande « time » du terminal, nous avons testé tout d'abord le temps de fonctionnement (en seconde) du mode aléatoire. Ce temps est plutôt très efficace. Nous avons pu constater que ce temps varie de façon croissante en fonction du nombre de règles appliquées. Le temps est d'environ 21.15 s.

Temps réel : 21,15 s
Temps utilisateur : 0
Temps système : 0

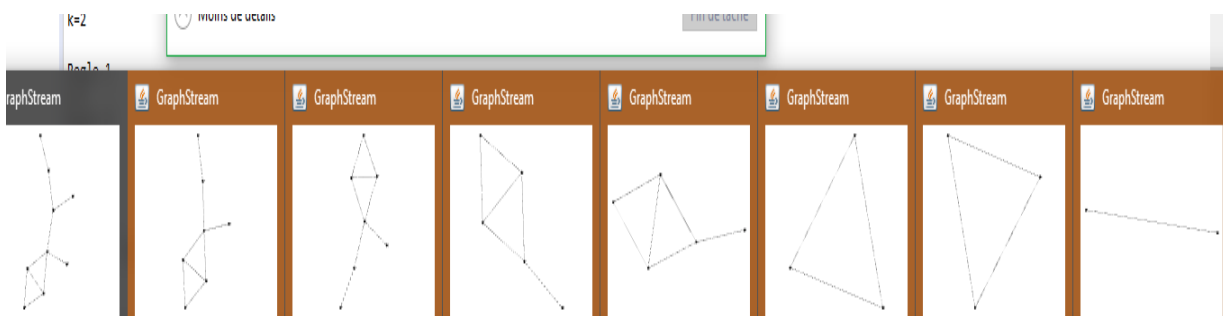
Nous avons ensuite testé le temps de fonctionnement du mode « définition du graphe ». Ce temps est correct du moment que nous choisissons des graphes d'ordre petit. (le temps d'exécution réel d'un graphe complet d'ordre 12 est d'environ 20s). Pour un graphe d'ordre 100 ayant 50 arêtes, le temps d'exécution réel est d'environ 50 secondes. En augmentant la taille du graphe, le temps d'exécution augmente mais de façon modérée. Le temps d'exécution est donc satisfaisant pour ce programme

- Ressources

Lorsque le programme est lancé avec un graphe de petite taille, celui-ci s'exécute rapidement. Nous avons cependant remarqué qu'à chaque fois qu'une règle était appliquée, une fenêtre montrant le nouveau graphe obtenu était ouverte, mais cette fenêtre ne se ferme qu'à la fin du programme. Dans un graphe de taille raisonnable (une dizaine de sommets), nous obtenons un résultat très peu gourmand en ressources :

Nom	Processeur	Mémoire	Disque	Réseau
Applications (6)				
> eclipse.exe	0%	429,6 Mo	0 Mo/s	0 Mbits/s
> Explorateur Windows	0%	42,1 Mo	0 Mo/s	0 Mbits/s
> Firefox (32 bits)	0,3%	466,7 Mo	0 Mo/s	0 Mbits/s
> Java(TM) Platform SE...	1,8%	58,1 Mo	0,1 Mo/s	0 Mbits/s
> OpenOffice 4.1.2 (32 ...	0%	44,1 Mo	0 Mo/s	0 Mbits/s

Cependant, il y a déjà un nombre non négligeable de fenêtres graphiques montrant chaque graphe ayant été généré lors de la kernelization :



Nous avons donc décidé d'essayer ce programme avec un graphe de taille beaucoup plus importante (10 000 sommets, et 10 000 arrêtes). L'exécution du programme en lui-même n'a pas posé de problèmes majeurs. Cependant, l'ouverture multiple des fenêtres graphiques provoque de très importants problèmes de performance comme le montre les images ci-dessous :

Au début, il n'y a aucun problème majeur de performances

> Java(TM) Platform SE binary

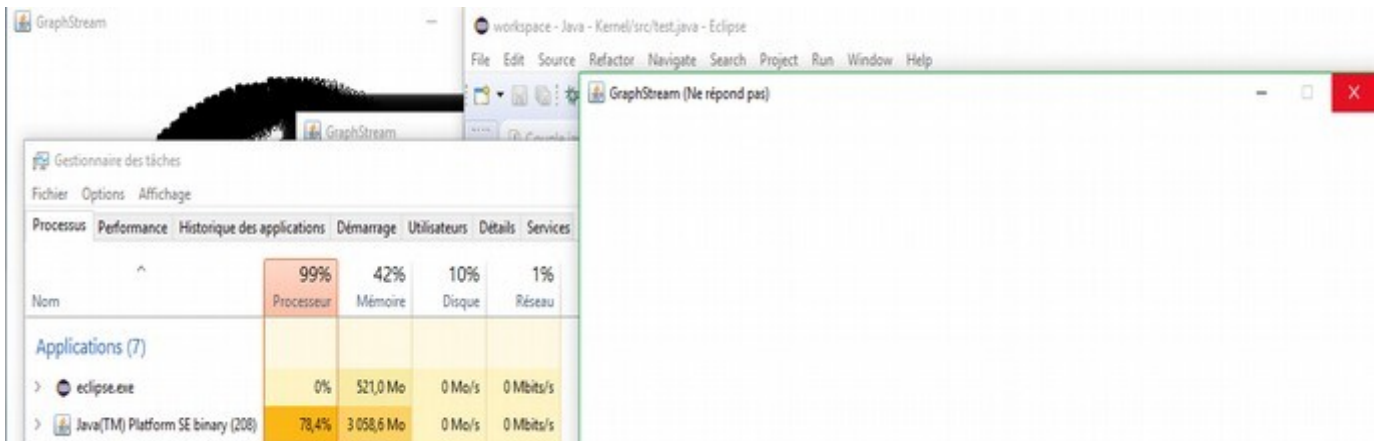
36,1%

217,8 Mo

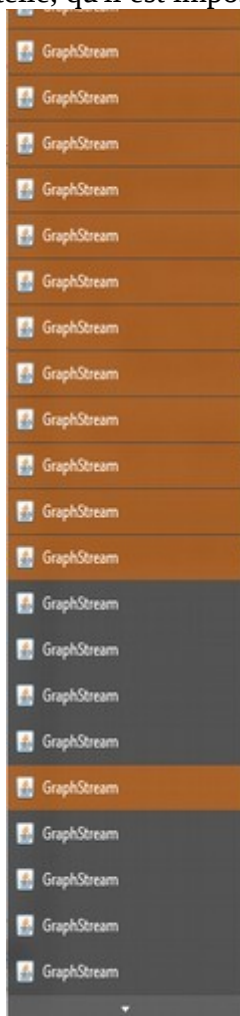
0 Mo/s

0 Mbits/s

Mais après quelques simulations de plus, il est impossible de continuer la simulation, car les fenêtres graphiques ne peuvent plus s'ouvrir :



Les performances de la machine sont mises à rude épreuve. De plus, la quantité de graphes affichée est telle, qu'il est impossible de s'y retrouver :



Au final, si nous continuons vraiment l'exécution du programme, une exception est levée, par la bibliothèque graphstream, car la Carte Graphique n'a pas suffisamment d'espace pour afficher le nouveau graphe :

```
Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.util.HashMap.newNode(Unknown Source)
    at java.util.HashMap.putVal(Unknown Source)
    at java.util.HashMap.put(Unknown Source)
    at org.graphstream.ui.graphicGraph.StyleGroup.addElement(StyleGroup.java:428)
    at org.graphstream.ui.graphicGraph.StyleGroupSet.addElement(StyleGroupSet.java:648)
    at org.graphstream.ui.graphicGraph.StyleGroupSet.addElement(StyleGroupSet.java:632)
    at org.graphstream.ui.graphicGraph.GraphicGraph.addEdge(GraphicGraph.java:701)
    at org.graphstream.util.GraphListeners.edgeAdded(GraphListeners.java:353)
    at org.graphstream.ui.graphicGraph.GraphicGraph.edgeAdded(GraphicGraph.java:1098)
    at org.graphstream.stream.SourceBase.sendEdgeAdded(SourceBase.java:410)
    at org.graphstream.stream.thread.ThreadProxyPipe.processMessage(ThreadProxyPipe.java:510)
    at org.graphstream.stream.thread.ThreadProxyPipe.pump(ThreadProxyPipe.java:270)
    at org.graphstream.ui.view.Viewer.actionPerformed(Viewer.java:534)
    at javax.swing.Timer.fireActionPerformed(Unknown Source)
    at javax.swing.Timer$DoPostEvent.run(Unknown Source)
    at java.awt.event.InvocationEvent.dispatch(Unknown Source)
    at java.awt.EventQueue.dispatchEventImpl(Unknown Source)
    at java.awt.EventQueue.access$500(Unknown Source)
    at java.awt.EventQueue$3.run(Unknown Source)
    at java.awt.EventQueue$3.run(Unknown Source)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.security.ProtectionDomain$JavaSecurityAccessImpl.doIntersectionPrivilege(Unknown Source)
    at java.awt.EventQueue.dispatchEvent(Unknown Source)
    at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
    at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
    at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
    at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
    at java.awt.EventDispatchThread.run(Unknown Source)
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.util.HashMap.newNode(Unknown Source)
    at java.util.HashMap.putVal(Unknown Source)
    at java.util.HashMap.put(Unknown Source)
    at org.graphstream.graph.implementations.SingleNode.addEdgeCallback(SingleNode.java:105)
    at org.graphstream.graph.implementations.AbstractGraph.addEdge(AbstractGraph.java:894)
    at org.graphstream.graph.implementations.AbstractGraph.addEdge(AbstractGraph.java:352)
    at org.graphstream.graph.implementations.Graphs.clone(Graphs.java:173)
    at Couple.<init>(Couple.java:12)
    at Regle.appliquerRegle(Regle.java:64)
    at Kernel.appliquerRegle(Kernel.java:39)
    at Kernel.appliquerString(Kernel.java:95)
```

Pour conclure, nous pouvons dire que le programme est assez bien optimisé. Pour les graphes de grande taille, à condition de ne pas afficher tous les graphes à chaque itération, car cela demande énormément de ressources.

- Fiabilité

Nous pouvons dire que le programme est correct puisqu'il optimise parfaitement les graphes, et permet de trouver s'il existe, un vertex cover sur ce graphe. Cependant, il n'est pas assez robuste puisqu'il ne résiste pas aux événements exceptionnels (pas de gestion d'exceptions) tels que la saisie de données erronées ou incohérentes.

- Sécurité

Les variables de classe sont privées pour plus de sécurité. Certaines variables sont immuables (mot clé final), ainsi leurs valeurs, ne seront pas changées au cours du programme.

```
private int numRegle;  
private final static int nombreRegles = 3;
```

```
... - - - ...
```

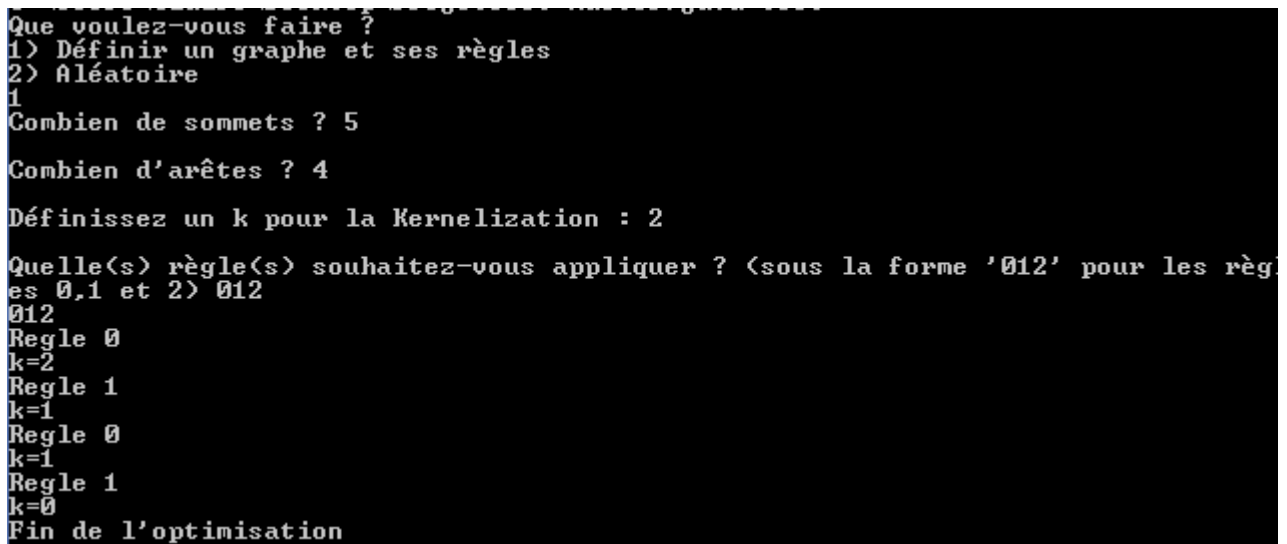
La variable nombreRegles vaudra toujours 3.

- Maintenabilité

Le programme étant décomposé en classe, il sera plutôt facile d'ajouter de nouvelles fonctionnalités ou d'apporter des modifications à une fonctionnalité existante. La maintenabilité dépend aussi de la librairie GraphStream, c'est à dire que si elle évolue, le programme devra par conséquent évoluer.

- Portabilité

Le programme fonctionne normalement (lors d'une utilisation normale) sous Linux. Il en va de même sur Windows. Le programme fonctionne normalement lorsque nous utilisons le terminal Windows.



```
Que voulez-vous faire ?  
1) Définir un graphe et ses règles  
2) Aléatoire  
1  
Combien de sommets ? 5  
Combien d'arêtes ? 4  
Définissez un k pour la Kernelization : 2  
Quelle(s) règle(s) souhaitez-vous appliquer ? (sous la forme '012' pour les règles 0,1 et 2) 012  
012  
Règle 0  
k=2  
Règle 1  
k=1  
Règle 0  
k=1  
Règle 1  
k=0  
Fin de l'optimisation
```

Cependant sur Eclipse, l'encodage des caractères ne fonctionne pas bien. Nous pouvons voir des conflits d'encodage de caractères lorsque nous passons de Linux à Windows ci dessous. Ce problème est lié à Eclipse et peut se changer dans les paramètres du logiciel.

```
Problems @ Javadoc Declaration Console
<terminated> test [Java Application] C:\Program Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (9 févr. 2017 à 21:58:32)
Que voulez-vous faire ?
1) Définir un graphe et ses règles
2) Algorithme
2
Nous prenons un graphe à 9 sommets et 10 arêtes, et un k=4. Nous appliquons toutes les règles.
```

Analyse de code : Tests unitaires

Nous allons présenter les tests unitaires que nous avons réalisés pour ce projet. Pour cela, nous allons expliquer quelques tests pour chaque classe que nous avons testée. Pour les tests soulevant une erreur, la ligne de test générant l'erreur sera surlignée en orange et nous tenterons de proposer une solution aux problèmes trouvés.

L'ensemble des tests est présent sur les fichiers de code.

I) Classe Couple

1) Tests corrects

- fonction getK()

Nous testons la fonction avec un couple c4 ayant un graphe g3 à 6 sommets et un paramètre k1=4. Le test s'effectue normalement puisque la fonction getK retourne bien k1.

```
@Test
public void getKNormal() {
    assertEquals(c4.getK(), k1);
}
```

- Constructeur Couple

Nous créons un couple à partir d'un graphe g3 avec 6 sommets et d'un k=4. Le k est compris entre 1 et l'ordre du graphe g3. Le test s'effectue normalement. Le couple créé n'est pas nul et le k du couple vaut bien 4.

```
@Test
public void creerCoupleOkK() {
    Graph g = new SingleGraph("graphe de test");
    Couple c = new Couple(g3, 4);
    assertTrue(c.getK() == 4);
    assertNotNull(c);
}
```

- fonction afficherGraph()

Nous testons uniquement d'appeler la méthode et nous confirmons le test si la fonction ne renvoie pas d'erreur car cette fonction ne renvoie rien et ne modifie aucun attribut, aussi bien sur l'objet Couple que sur l'objet Graph du couple.

```
@Test
public void afficherGraph1Normal() {
    c4.afficherGraph();

    assertFalse(1 == 2);
}
```

- Fonction getG ()

Nous testons avec un .toString() afin de vérifier que les deux graphes sont bien identiques bien que le graphe du couple ait été cloné. Le test s'effectue normalement, les deux graphes sont identiques.

```
@Test
public void getGNormal() {
    assertEquals(c4.getG().toString(), g3.toString());
}
```

2) Tests comportant des erreurs

- Constructeur Couple

Ce test a pour but de vérifier le fonctionnement du constructeur avec un paramètre de kernelization k négatif. En effet, il n'est pas cohérent de pouvoir créer un couple avec un k négatif. Nous ne pourrions pas avoir de vertex cover de taille k=-1 puisqu'il est impossible d'avoir une couverture maximale des arêtes en utilisant -1 sommets.

```
@Test
public void creerCoupleKNegatif() {
    Graph g = new SingleGraph("A modifier");
    Couple c = new Couple(g, -4);
    assertEquals(-4, c.getK());
}
```

Nous proposons comme solution de normaliser les valeurs du paramètre k. En effet, lorsque l'utilisateur saisit une valeur hors normes c'est à dire plus petite que 0, il faudrait rajouter une condition pour passer le k à 1. Ainsi lorsqu'une valeur négative est passée le couple créé aura pour paramètre le graphe g et un k valant 1.

Nous avons testé le constructeur en passant un graphe null. Le test renvoie un NullPointerException. Ce problème est dû à la fonction Graphs.clone() qui renvoie cette erreur si le graphe passé en argument est null.

```

@Test
public void creerCoupleGNull() {
    Couple c = new Couple(null, 4);
    assertNotEquals(c.getG(), null);
}

```

Nous proposons deux solutions. Soit utiliser une condition de telle façon que si g est null alors on initialise à nouveau le graphe g de la façon suivante `g=new SingleGraph (« mon graph »)` ; La deuxième solution est de renvoyer une exception plutôt que de renvoyer un null pointer Exception. Nous pouvons mettre en œuvre cette solution en utilisant un bloc `try {...}catch {...}` c'est à dire

```

try {
    Initialisation du couple
}catch (nomException) {
    Ecriture du message d'erreur
}

```

Nous avons ensuite testé le constructeur avec un k largement supérieur à l'ordre du graphe en paramètre. Le graphe g3 est d'ordre 6. Nous nous attendons à une normalisation de k c'est à dire que le k soit au plus égal à l'ordre du graphe. Le test renvoie une erreur puisque le paramètre de kernelization n'est pas modifié. Il paraît plus cohérent de chercher un vertex cover avec un paramètre de kernelization égal à l'ordre du graphe puisqu'une couverture maximale peut être obtenu en utilisant tous les sommets. Il est donc inutile de chercher un vertex cover pour un k supérieur à l'ordre du graphe.

```

@Test
public void creerCoupleKGrand() {

    Couple c = new Couple(g3, 1000);
    assertEquals(c.getK(),c.getG().getNodeCount() );
}

```

Nous pouvons proposer comme solution d'ajouter une condition dans le constructeur telle que si le k passé en paramètre est supérieur au nombre de sommets du graphe passé en paramètre alors le k prend la valeur de l'ordre du graphe.

- Fonction `getK()`

Ensuite, nous avons testé cette fonction avec le couple suivant c3 dont le paramètre de kernelization est 1000 et un graphe g3 ayant 6 sommets. Le but de ce test est de vérifier que le k retourné est cohérent avec l'ordre du couple. Nous attendons donc une normalisation de k c'est à dire que le k soit égal à l'ordre du graphe. Le test réalisé renvoie une erreur, le k n'est pas normalisé, le k retourné n'est donc pas modifié.


```
@Test
public void testGetKGrand() {
    assertEquals("doivent être égaux", c3.getK(),g3.getNodeCount());
}
```

Nous avons de plus, effectué le même test mais cette fois avec un k négatif. Le but de ce test est de vérifier que le k retourné est cohérent avec l'ordre du couple. Nous attendons donc une normalisation de k c'est à dire que le k soit égal à 1. Le test réalisé retourne une erreur, le k n'est pas normalisé, le k retourné n'est donc pas modifié.

```
@Test
public void testGetKNégatif() {
    Couple c= new Couple (g3, -4);
    assertEquals("doivent être égaux",c.getK(),1 );
}
```

En résolvant les problèmes de la fonction constructeur, nous solutionnons les problèmes de la fonction getK().

II) Classe test (fonction generate)

1) Tests corrects

Nous testons tout d'abord que le graphe créé ne soit pas null. Le test s'effectue normalement.

```
@Test
public void testGraphNull() {
    Graph g10=test.generate(10, 1);
    assertNotNull(g10);
}
```

Nous testons ensuite la fonction avec un nombre de sommets et d'arêtes dans les normes. Le test s'effectue normalement et retourne donc bien 10 sommets.

```
@Test
public void testSommetOkAretesOk() {
    Graph g4=test.generate(10, 1);

    assertEquals("sont égaux", g4.getNodeCount(), 10);
}
```

Nous testons le nombre d'arêtes du graphe créé. Le test s'effectue normalement et retourne bien 5 arêtes.

```
@Test
public void testAretesNormes() {
    Graph g5=test.generate(10,5);
    assertEquals("sont égaux", g5.getEdgeCount(),5);
}
```

Ensuite, nous allons tester le degré d'un sommet du graphe retourné. Nous générons un graphe d'ordre 5 ayant 5 arêtes. Le degré d'un sommet du graphe doit donc être compris entre 0 et 4. Nous testons le degré sur le sommet « 0 » que nous récupérons grâce à la fonction getNode(). Le test s'effectue normalement, le degré du sommet 0 est donc correct.

```
@Test
public void DegreSommet() {
    Graph g18=test.generate(5, 5);
    Node n1=g18.getNode("0");
    assertTrue("relations vraies", n1.getDegree()>=0 );
    assertTrue("relations vraies", n1.getDegree()<=4);
}
```

Nous testons ensuite l'unicité d'une arête. Nous générons un graphe complet d'ordre 3. Ce test va compter le nombre d'arêtes et vérifier que ce nombre est égal au nombre d'arêtes total. Ensuite comme chaque arête doit être unique, nous savons que pour une arête d'extrémité x y nous avons deux possibilités l'arête x-y ou l'arête y-x. Il faut donc vérifier que ces arêtes ne soient pas égales pour être sûr qu'une seule arête soit créée. Si les deux arêtes sont nulles, l'arête n'existe pas, si seulement une des deux est null alors elle existe, si aucune possibilité est nulle, l'arête n'est pas unique, il y a un doublon. Nous allons tester l'unicité des arêtes pour chaque arête possible. Nous sommes sûrs que deux possibilités d'arêtes ne peuvent pas être égales puisque nous avons généré un graphe complet.

```
@Test
public void testUniciteArete() {
    Graph g19=test.generate(3, 3);
    assertEquals(g19.getEdgeCount(),3);
    Edge a1=g19.getEdge("0-1");
    Edge a2=g19.getEdge("1-0");
    assertTrue (a1!=a2 && ( a1==null || a2==null) );
    Edge a3=g19.getEdge("1-2");
    Edge a4=g19.getEdge("2-1");
    assertTrue (a3!=a4 && ( a3==null || a4==null) );
    Edge a5=g19.getEdge("2-0");
    Edge a6=g19.getEdge("0-2");
    assertTrue (a5!=a6 && ( a5==null || a6==null) );
}
```

Ensuite, nous testons la fonction avec un nombre d'arêtes trop grand. Le test s'effectue normalement et retourne un graphe vide juste initialisé avec son nom. Il est à noter que générer un graphe vide n'est pas la seule solution possible. Nous pouvons choisir de générer le graphe après la vérification de la condition sur le nombre d'arêtes (actuellement le graphe est généré avant la vérification de la condition). Nous pouvons expliquer la création du graphe vide avec le code de la fonction générant le graphe. En effet, le nombre d'arêtes étant trop grand, nous ne rentrons pas dans la boucle créant les sommets et les arêtes donc nous ne créons pas d'arêtes.

```

@Test
public void testGraphAretesNon() {
    Graph g11=test.generate(10, 200);
    assertNotNull(g11);
    assertEquals(g11.toString(), "EnMarche");
}

```

Puis nous effectuons un test avec un nombre de sommets négatif. Le test s'effectue normalement et le nombre de sommets est égal à 0. Nous pouvons l'expliquer avec le code de la fonction générant le graphe. En effet, avec un nombre de sommets négatif, nous ne rentrons pas dans la boucle for permettant de créer les sommets donc nous obtenons au final 0 sommets.

```

public void testSommetsHorsNormes() {
    /* initialisation d'un graphe avec un nombre de sommets anomal */
    Graph g3=test.generate(-10, 0);
    assertNotEquals("relations vraie", g3.getNodeCount(), -10);
    assertEquals("sont égaux", g3.getNodeCount(), 0);
}

```

2) Tests comportant des erreurs

Nous testons de générer un graphe avec 10 sommets et -5 arêtes. Pour réaliser ce test, nous avons mis un timeout de 10 000 ms soit 10 s. Le timeout permet de stopper le test quand il dépasse 10 secondes et de le passer en erreur. Comme vous pouvez le voir sur l'image le test est passé en erreur car la fonction generate a mis plus de 10 secondes.

```

@Test (timeout=10000)
public void testSommets0kAretesNon() {
    Graph g4=test.generate(10, -5);
    assertEquals("sont égaux", g4.getEdgeCount(), -5);
}

```

Si nous enlevons le timeout, ce test fait une boucle infinie. Nous pouvons expliquer cette boucle infinie lors de la création des arêtes. En effet, nous allons rentrer dans la boucle permettant de créer des arêtes. Cependant le compteur étant initialisé à 0, la boucle while (cpt !=edge) ne s'arrête jamais lorsque le nombre d'arêtes est négatif. En effet le compteur augmentera sans cesse au cours du programme.

Nous proposons comme solution d'ajouter une condition sur le nombre d'arêtes au début du programme telle que si le nombre d'arêtes est négatif, alors nous normalisons ce nombre.

III) Classe Kernel

Pour tester, les fonctions de cette classe, il est souvent nécessaire d'appuyer sur la touche entrée dans la console.

1) Tests corrects

- Constructeur Kernel

Nous créons un nouveau kernel et nous vérifions que le kernel ainsi que la liste des règles ne soient pas null. Nous vérifions également que la taille de la liste est égale à 0. Le test se déroule normalement.

```
@Test
public void testConstructeur() {
    Kernel k = new Kernel();

    assertNotNull(k);
    assertNotNull(k.liste);

    assertTrue(k.liste.size() == 0);
}
```

- AjoutRegle (int r)

Nous testons la fonction à partir de deux kernel vide (ke1 et ke2). Nous ajoutons une règle à chaque kernel. Nous vérifions que la taille de la liste de règles des deux kernel est égale et que ces listes ne soient pas nulles. Le test s'effectue normalement.

```
@Test
public void testAjoutRegleNormal() {

    assertEquals(ke1.liste, ke2.liste);

    ke1.ajoutRegle(0);
    ke2.ajoutRegle(1);

    assertNotEquals(ke1.liste, ke2.liste);
    assertEquals(ke1.liste.size(), ke2.liste.size());
    assertEquals(ke1.liste.size(), 1);
}
```

- appliquerRegle(Couple c)

Nous testons la fonction avec un couple c1 qui contient un k positif et un graphe g1 (voir annexe illustration 1) Le graphe a 10 sommets et le sommet 0 est de degré 0. keg0 est un kernel contenant la règle 0.

Nous devons donc vérifier que la suppression du sommet « 0 » s'effectue normalement. Pour cela,

le nombre de sommets du graphe du couple final doit être égal au nombre de sommets du graphe initial - 1. Ensuite le sommet « 0 » doit être null et le k doit être inchangé. Le test se déroule parfaitement.

```
@Test
public void testAppliquerRegle0() {

    Couple cRes = keg0.appliquerRegle(c1);

    assertTrue(cRes.getG().getNodeCount() == c1.getG().getNodeCount() - 1);

    assertNull(cRes.getG().getNode("0"));
    assertNotNull(cRes.getG().getNode("6"));

    assertEquals(cRes.getK(), c1.getK());
}
```

- Fonction appliquerRegleB(Couple c)

Nous testons la fonction avec un couple c1 qui contient un k positif et un graphe g1 (voir annexe illustration 1). Le graphe a 10 sommets et le sommet 6 est le seul voisin du sommet 8 de degré 1. keg1 est un kernel contenant la règle 1. Nous devons donc vérifier que le sommet 6 soit bien supprimé. Pour cela, le nombre de sommets du graphe du couple final doit être égal au nombre de sommets du graphe initial - 1. Le sommet 6 doit être null et d'après les règles énoncées dans le rapport le k du couple final doit diminuer d'un. Les autres sommets comme le sommet 0 ne doivent pas être supprimés donc ils ne doivent pas être nuls. Le test s'effectue normalement.

```
@Test
public void testAppliquerRegleB1() {
    Couple cRes = keg1.appliquerRegleB(c1);

    assertTrue(cRes.getG().getNodeCount() == c1.getG().getNodeCount() - 1);

    assertNull(cRes.getG().getNode("6"));
    assertNotNull(cRes.getG().getNode("0"));

    assertEquals(cRes.getK(), c1.getK() - 1);
}
/*
```

Nous vérifions si le graphe est modifié par une règle qui ne s'applique pas à lui-même, c'est à dire, si nous appliquons la règle 1 sur g2 (voir annexe illustration 2). Le résultat est bien celui attendu, c'est à dire que ni le graphe ni le k n'est modifié. Cependant, nous rentrons dans la boucle, car il faut appuyer sur entrée pour continuer, ce qui est une perte de performance, car il n'y avait pas besoin de rentrer dans la boucle.

```
@Test
public void testAppliquerRegleB1Inutile() {
    Couple cRes = keg1.appliquerRegleB(c2);

    assertEquals(cRes.getG().toString(), c2.getG().toString());

    assertFalse(cRes.getK() != c2.getK());
}
```

- Fonction appliquerString (Couple c, String s)

Nous testons la fonction avec le couple c2 qui contient un k positif ainsi qu'un graphe g2 (voir annexe illustration 2). Ce graphe contient le sommet 0 du graphe g2 qui doit être supprimé par la fonction car il est de degré 4, ce qui est supérieur à $k=3$. Nous vérifions alors que le nombre de sommets du graphe du couple final soit égal au nombre de sommets du graphe initial - 1. Le sommet 0 doit être null puisqu'il doit être supprimé et les autres sommets comme le sommet 1 ne doivent pas être null car ils ne sont pas supprimés. Le k du couple final doit être décrémenté de 1. Le test s'effectue normalement.

```
@Test
public void testAppliquerString2() {
    Couple cRes = Kernel.appliquerString(c2, "2");

    assertTrue(cRes.getG().getNodeCount() == c2.getG().getNodeCount() - 1);

    assertNull(cRes.getG().getNode("0"));
    assertNotNull(cRes.getG().getNode("1"));

    assertEquals(cRes.getK(), c2.getK() - 1);
}
```

Nous testons la fonction avec une chaîne de caractères comme « toto » qui ne correspond à aucune règle. La fonction AppliquerString ne doit rien faire. Le nombre de sommets du graphe du couple final doit être identique au graphe de départ. De même le k ne doit pas changer. Le test s'effectue normalement. Nous pouvons expliquer pourquoi la fonction ne génère pas d'erreur grâce à la méthode getNumericValue utilisée comme ceci :

```
if(b >= 0 && b < Regle.getNombre()) ker.ajoutRegle(b);
```

La méthode getNumericValue convertit une chaîne de caractères en valeur numérique lorsque la chaîne possède une valeur numérique. Si la chaîne ne possède pas de valeur numérique (comme « toto »), alors getNumericValue renvoie -1 ce qui fait qu'il n'y a pas de règle négative créée et pas d'exceptions.

```
@Test
public void testAppliquerStringLettres() {

    Couple cRes = Kernel.appliquerString(c1, "toto");

    assertEquals(cRes.getG().toString(), c1.getG().toString());

    assertFalse(cRes.getK() != c1.getK());
}
```

Nous vérifions si le graphe est modifié par une règle qui ne s'applique pas à lui-même, c'est à dire, si nous appliquons la règle 0 sur g2 (voir annexe illustration 2)

Le résultat est bien celui attendu, c'est à dire que ni le graphe ni le k n'est modifié. Cependant, nous rentrons dans la boucle, car il faut appuyer sur entrée pour continuer, ce qui est une perte de performances, car il n'y avait pas besoin de rentrer dans la boucle.

```

@Test
public void testAppliquerString0Inutile() {
    Couple cRes = Kernel.appliquerString(c2, "0");

    assertEquals(cRes.getG().toString(), c2.getG().toString());

    assertFalse(cRes.getK() != c2.getK());
}

```

2) Tests comportant des erreurs

- fonction ajoutRegle (Regle r)

Nous essayons de mettre deux fois la même règle au sein d'un Kernel. Le résultat attendu de la part du programme est que le Kernel contienne la règle demandée une seule fois, et ignore la seconde demande. Le résultat obtenu est différent de celui attendu. En effet, si nous utilisons le même objet Regle lors de la création, celle-ci est bien unique. Cependant, en créant une deuxième Regle avec le même numéro, nous obtenons une Regle du même numéro, mais correspondant à un objet différent de la précédente, et qui peut donc être ajoutée à la liste. Nous pouvons expliquer cette erreur avec le code de la fonction ajoutRegle et plus particulièrement cette ligne :

```

this.liste.contains(r1)

```

Avec .contains(), seules les références des objets sont comparées et comme elles ne sont jamais égales, la règle est ajoutée.

```

@Test
public void testAjoutRegleDouble2() {
    Regle r1 = new Regle(0);

    assertTrue(kel.liste.isEmpty());

    kel.ajoutRegle(r1);
    assertFalse(kel.liste.isEmpty());

    assertEquals(kel.liste.size(), 1);

    kel.ajoutRegle(r1);
    assertEquals(kel.liste.size(), 2);

    Regle r2 = new Regle(0);

    kel.ajoutRegle(r2);
    assertEquals(kel.liste.size(), 2);
}

```

Nous proposons comme solution de redéfinir la méthode .equals() de Regle car la fonction .contains() fait appel à .equals(). .equals() renverrait alors true si les règles ont le même numéro et false sinon. Ainsi lorsque nous ferons this.liste.contains(r1), .contains() appellera .equals() qui renverra true si la liste de l'objet this contient déjà une Regle avec le même numéro que la Regle r1.

- AppliquerRegle(int r1)

Nous avons tenté d'ajouter deux fois une Regle avec le même numéro. Il est possible d'ajouter deux

fois la même règle. Les raisons sont les mêmes que pour la fonction `appliquerRegle(Regle r1)` et la solution proposée est la même.

```
@Test
public void testAjoutRegleDouble() {

    assertTrue(ke1.liste.isEmpty());

    ke1.ajoutRegle(0);
    assertFalse(ke1.liste.isEmpty());

    assertEquals(ke1.liste.size(), 1);

    ke1.ajoutRegle(0);
    assertEquals(ke1.liste.size(), 2);

}
```

Ensuite, nous avons testé la fonction en passant une Règle nulle. Le test ajoute une Règle Nulle à la liste.

```
@Test
public void testAjoutRegleNull() {

    assertTrue(ke1.liste.isEmpty());

    ke1.ajoutRegle(null);
    assertTrue(ke1.liste.isEmpty());

    Regle r2 = new Regle(0);

    ke1.ajoutRegle(r2);
    assertEquals(ke1.liste.size(), 1);

}
```

Nous proposons comme solution de ne pas ajouter la Règle si elle est nulle ou de lever une exception avec un bloc `try {...} catch {...}`

- Fonction `appliquerRegleB(Couple c)`

Nous testons la fonction avec un kernel `keg0` qui contient une Règle 0. Nous testons cette règle sur un couple `test1` ayant un paramètre `k` négatif et un graphe `g1` (voir annexe illustration 1). Le graphe a 10 sommets et le sommet 0 est de degré 0. Le test ne fonctionne pas puisque nous nous attendions à un nombre de sommets non modifié.


```

@Test
public void testAppliquerRegleB0KNegSommet() {
    Couple cRes = keg0.appliquerRegleB(test1);
    assertEquals(cRes.getG().getNodeCount(), test1.getG().getNodeCount());
}

```

Ensuite, nous pouvons voir que le sommet 0 est null donc le sommet « 0 » a été supprimé ce qui est différents des attentes puisque nous ne nous attendions à aucune suppression de sommets.

```

@Test
public void testAppliquerRegleB0KNegNull() {
    Couple cRes = keg0.appliquerRegleB(test1);
    assertNotNull(cRes.getG().getNode("0"));
}

```

Nous proposons comme solution d'ajouter une condition dans la fonction Regle.appliquerRegle(Couple c) afin de ne pas appliquer la règle 0 si le k est négatif. Cette condition est présente pour les Regle 1 et 2 mais pas pour la règle 0 ce qui n'est pas cohérent avec les indications données sur le rapport (les Regles ne s'appliquent pas lorsque le k est négatif).

Nous testons la fonction avec un couple null et une exception nullPointer Exception est levée.

```

@Test
public void testAppliquerRegleBNull() {
    Couple cRes = keg1.appliquerRegleB(null);

    assertNotNull(cRes);
}

```

En solution, nous pouvons mettre une condition de telle sorte que si la chaîne est nulle, aucune règle n'est créée. Sinon, nous pouvons si la chaîne est nulle, choisir d'appliquer une règle en particulier comme la règle 0 par exemple. Une autre façon possible est de lever une exception avec un bloc try{...} catch {...}.

- Fonction appliquerString (Couple c, String s)

Nous testons la fonction avec une chaîne null et une exception nullPointer Exception est levée.

```

@Test
public void testAppliquerStringNull() {
    Couple cRes = Kernel.appliquerString(c1, null);

    assertNotNull(cRes);
}

```

La solution proposée pour la gestion du null est la même que pour la fonction appliquerRegleB(couple c).

Nous testons la fonction avec un couple null.

```
@Test
public void testAppliquerCoupleNull() {
    Couple cRes = Kernel.appliquerString(null, "0");

    assertNotNull(cRes);
}
```

En solution, nous pouvons modifier le constructeur Couple comme indiqué dans les tests de la classe Couple.

- Fonction kernelization (Couple c)

Nous testons la fonction avec un couple null et une exception nullPointer Exception est levée.

```
@Test
public void testKernelisationNull() {

    assertFalse(Kernel.kernelization(null));

}
```

En solution, nous pouvons modifier le constructeur Couple comme indiqué dans les tests de la classe Couple.

Ce test permet de vérifier le fonctionnement de kernelisation de Kernel. Nous testons avec un couple c2 ayant un graphe g2 et un k=3.

Le graphe g2 (voir annexe illustration 2) est fait de telle sorte qu'il soit possible de trouver un vertex cover de taille 3, cette fonction est donc supposée renvoyer vrai.

La fonction retourne cependant faux. Cela est dû au fait que, après l'application des règles, le graphe n'est pas vide, c'est à dire, qu'il reste encore un sommet, de degré 0, qui devrait être enlevé par la règle 0 afin de rendre la condition vraie (0 sommets et k=0).

```
@Test
public void testKernelisationOui() {

    assertTrue(Kernel.kernelization(c2));

}
```

Nous pouvons proposer comme solution d'appliquer à nouveau la Règle 0 a la fin de la simulation pour remplir la condition.

- Fonction appliquerRegle (Couple c)

Nous testons la fonction avec un couple null et une exception NullPointerException est levée.

```
@Test
public void testAppliquerRegleNull() {
    Couple cRes = keg1.appliquerRegle(null);

    assertNotNull(cRes);
}
```

En solution, nous pouvons mettre une condition de telle sorte que si la chaîne est nulle, aucune règle n'est créée. Sinon, nous pouvons si la chaîne est nulle, choisir d'appliquer une règle en particulier comme la règle 0 par exemple. Une autre façon possible est de lever une exception avec un bloc try {...} catch {...}.

IV Classe Regle

1) Tests corrects

- Constructeur Regle

Nous testons le constructeur de Regle. Les Regles créées ne doivent pas être nulles et le numéro de la règle doit correspondre à celui passé en paramètre. Le test s'effectue normalement.

```
@Test
public void testBonneNumRegle() {
    Regle r0 = new Regle(0);
    Regle r1 = new Regle(1);
    Regle r2 = new Regle(2);

    assertNotNull("Ne doit pas être null", r0);
    assertNotNull("Ne doit pas être null", r1);
    assertNotNull("Ne doit pas être null", r2);

    assertEquals("doit être égaux", r0.getNum(), 0);
    assertEquals("doit être égaux", r1.getNum(), 1);
    assertEquals("doit être égaux", r2.getNum(), 2);
}
```

- Fonction appliquerRegle

Pour cette fonction, nous avons vérifié que les règles 0, 1 ou 2 soient bien appliquées de la même manière que pour les tests effectués pour Kernel.appliquerRegle puisque ces deux fonctions se ressemblent. Les tests s'effectuent normalement et les sommets voulus sont bien supprimés.

Nous avons testé la fonction `appliquerRegle` avec un couple `c1Negatif`. Ce couple contient le graphe `g1` (voir annexe illustration 1) et un `k` valant `-1`. La fonction ne doit rien faire c'est à dire que aucun sommet ne doit être supprimé et le `k` doit rester inchangé. Le test s'effectue normalement.

```
@Test
public void testAppliquerRegle1Negatif() {

    int nodesInit = c1Negatif.getG().getNodeCount();
    Couple cRes = r1.appliquerRegle(c1Negatif);

    assertTrue(cRes.getG().getNodeCount() == nodesInit);

    assertNotNull(cRes.getG().getNode("6"));
    assertNotNull(cRes.getG().getNode("0"));

    assertEquals(cRes.getK(), c1Negatif.getK());
}
```

2) Tests comportant des erreurs

- Constructeur Regle

Nous avons testé le constructeur avec des numéros de règles qui ne correspondent à aucune règle appliquée par l'algorithme. Nous avons deux possibilités que nous pouvons attendre du programme. Soit le numéro de la règle est normalisé à 0 (s'il est différent de 0, 1 ou 2), soit aucun objet `Regle` n'est créé donc la règle doit valoir `null`.

```
@Test
public void testMauvaiseNumRegle() {
    Regle r0 = new Regle(3);
    Regle r1 = new Regle(-1);
    Regle r2 = new Regle(100);

    assertTrue((r0.getNum()==0 && r0!=null) || (r0==null));
    assertTrue((r1.getNum()==0 && r1!=null) || (r1==null));
    assertTrue((r2.getNum()==0 && r2!=null) || (r2==null));
}
```

Nous proposons comme solution d'ajouter une condition dans le constructeur de `Regle` telle que si le numéro de la règle ne vaut pas 0,1 ou 2, alors il vaut 0.

- `appliquerRegle(couple c)`

Nous testons la fonction avec le couple `c1` négatif comme expliqué précédemment pour le test de la fonction `appliquerRegle(couple c)` avec la `Regle 1` (test qui fonctionne). Nous nous attendions à avoir le même graphe qu'au départ puisque le `k` est négatif. Le test ne fonctionne pas puisque un sommet a été supprimé, le sommet 0. Le nombre de sommet diminue.

```

@Test
public void testAppliquerRegle0NegatifSommet() {

    int nodesInit = c1Negatif.getG().getNodeCount();
    Couple cRes = r0.appliquerRegle(c1Negatif);
    assertEquals(cRes.getK(), c1Negatif.getK());
    assertTrue(cRes.getG().getNodeCount() == nodesInit);

}

```

```

@Test
public void testAppliquerRegle0NegatifNull() {
    Couple cRes = r0.appliquerRegle(c1Negatif);
    assertNotNull(cRes.getG().getNode("6"));
    assertNotNull(cRes.getG().getNode("0"));

}

```

Nous proposons comme solution d'ajouter une condition qui vérifie si k est positif avant d'appliquer la règle 0.

Ensuite, nous testons la fonction en passant un couple null. Une exception NullPointerException est levée .

```

@Test
public void testAppliquerRegleNull() {
    Couple cRes = r1.appliquerRegle(null);

    assertNotNull(cRes);
}

```

Nous proposons comme solution de faire une condition qui ne crée pas de couple si le couple passé en paramètre du constructeur couple est null ou de lever une exception avec un bloc try{...}catch{...]

- fonction getNombre()

Nous testons la fonction avec la règle r3 qui porte le numéro 3. Nous nous attendions à ce que r3 soit null ou normalisée c'est à dire que pour toute valeur différente de 0,1 ou 2, la fonction renvoie 0.

```

@Test
public void testGetNombreMauvaiseRègle() {
    assertTrue(r3==null || r3.getNum()==0);
}

```

Il suffit d'ajouter une condition dans le constructeur pour normaliser les règles créées comme proposé précédemment.

V) Tests de performance

Les tests de performance nous donne une indication mais les résultats peuvent varier car ils dépendent de la machine virtuelle de Java.

- Classe Test

Ce test permet de vérifier la performance de la fonction generate. Nous testons avec un graphe à 7 000 000 et 1 arête. Le test s'effectue dans le temps demandé.

Mais il ne crée aucun sommet du graphe ce qui n'est pas conforme à ce que nous pouvions attendre c'est à dire un graphe à 7 000 000 de sommets. Il faut donc contrôler les valeurs, d'autant plus que sur l'interface, l'utilisateur peut saisir sa valeur qui sera utilisée dans cette fonction.

```
@Test(timeout=10000)
public void testGenererGraphe2() {
    Graph GTEST2=test.generate(7000000,1);
    assertEquals(GTEST2.getNodeCount(),0);
    assertEquals(GTEST2.getNodeCount(),7000000);
}
```

- Classe Kernel

Ce test permet de vérifier la performance de la fonction ajout règle. Nous allons tester d'ajouter 400 000 règles 0 en moins de 10 seconde. Ce test passe en erreur après 10 secondes.

```
@Test(timeout=10000)
public void testPerfAjoutRegle() {
    Kernel keg0test = new Kernel();
    for(int i=0; i<400000 ; i++) {
        keg0test.ajoutRegle(0);
    }
}
```

Si nous relançons le test en essayant de créer 200 000 règles, le test ne passe pas en erreur en moins de 10 secondes. Il est donc possible de créer 200 000 règles rapidement. Ensuite, nous avons relancé le test de la fonction AjoutRegle en mettant en commentaire la ligne contenant la méthode .contains(), c'est à dire que nous avons testé la fonction ajoutRegle comme ceci

```
public void ajoutRegle(int r){
    Regle r1 = new Regle(r);
    // l'opération contains n'est pas utilisé pour le deuxième test
    //if(!this.liste.contains(r1))
    this.liste.add(r1);
}
```

Sans le contains, nous pouvons créer 25 000 000 règles sans problème de temps

La vérification faite par contains prend énormément de temps car il faut parcourir toute l'arraylist à chaque itération. De plus comme le contains est mal défini, il est inutile et provoque une perte de temps

- Comparaison entre AppliquerRegleB et AppliquerString de Kernel

Ce test permet de vérifier la performance de la fonction appliquerRègle avec la règle 0. Nous voulons appliquer 1 fois la règle 0 sur un graphe à 30 sommets.

Nous indiquons une durée de 10 secondes et le test passe en erreur car il prend plus de 10 secondes. Cette fonction n'est pas très performante.

Nous pouvons relancer le test avec un graphe à 15 sommets, il y a donc 6 sommets de degré 0, le test passe en dessous de 10 secondes.

La fonction s'exécute en moins de 10 secondes seulement sur de petits graphes.

```
@Test(timeout=10000)
public void testAppliquerRegle() {
    Graph gtest = new SingleGraph("graphe g1");

    for (int i = 0; i < 30; i++) {
        gtest.addNode(i + "");
    }

    // et on rajoute des aretes

    gtest.addEdge("1-7", "1", "7");
    gtest.addEdge("1-9", "1", "9");
    gtest.addEdge("2-4", "2", "4");
    gtest.addEdge("2-5", "2", "5");
    gtest.addEdge("3-5", "3", "5");
    gtest.addEdge("3-7", "3", "7");
    gtest.addEdge("4-5", "4", "5");
    gtest.addEdge("4-6", "4", "6");
    gtest.addEdge("5-7", "5", "7");
    gtest.addEdge("5-6", "5", "6");
    gtest.addEdge("6-7", "6", "7");
    gtest.addEdge("6-8", "6", "8");
    gtest.addEdge("6-9", "6", "9");
    gtest.addEdge("7-9", "7", "9");

    Couple c0test=new Couple(gtest,2);
    Kernel keg0test = new Kernel();
    keg0test.ajoutRegle(0);

    Couple cRes = keg0test.appliquerRegle(c0test);
}
```

Ce test permet de vérifier la performance de la fonction appliquerRègleB avec la règle 0. Nous voulons appliquer 100 000 fois la règle 0 sur un graphe à 100 000 sommets. Nous indiquons une durée de 10 secondes et le test passe en erreur car il prend plus de 10 secondes. Nous pouvons relancer le test avec un graphe à 10 000 sommets et en souhaitant appliquer 10 000 fois la règle 0. La fonction s'exécute en moins de 10 secondes. AppliquerRegleB est donc beaucoup plus performante niveau temps que la fonction appliquerRegle.

AppliquerRegleB n'affiche pas le graphe après chaque règle ce qui fait gagner du temps et la rend plus performante que AppliquerRegle qui affiche le graphe à chaque itération.

```
@Test(timeout=10000)
public void testAppliquerRegleB() {
    Graph gtest = new SingleGraph("graphe g1");
    for (int i = 0; i < 100000 ; i++) {
        gtest.addNode(i + "");
    }

    // et on rajoute des aretes

    gtest.addEdge("1-7", "1", "7");
    gtest.addEdge("1-9", "1", "9");
    gtest.addEdge("2-4", "2", "4");
    gtest.addEdge("2-5", "2", "5");
    gtest.addEdge("3-5", "3", "5");
    gtest.addEdge("3-7", "3", "7");
    gtest.addEdge("4-5", "4", "5");
    gtest.addEdge("4-6", "4", "6");
    gtest.addEdge("5-7", "5", "7");
    gtest.addEdge("5-6", "5", "6");
    gtest.addEdge("6-7", "6", "7");
    gtest.addEdge("6-8", "6", "8");
    gtest.addEdge("6-9", "6", "9");
    gtest.addEdge("7-9", "7", "9");

    Couple c0test=new Couple(gtest,2);
    Kernel keg0test = new Kernel();
    for(int i=0; i<100000 ; i++) {
        keg0test.ajoutRegle(0);
    }
    Couple cRes = keg0test.appliquerRegleB(c0test);
}
```


CONCLUSION

Nous avons fait deux types de tests différents : des tests par l'interface et des tests unitaires. Concernant les tests par interface, nous pouvons retenir qu'il peut être difficile de lancer le programme puisqu'il n'y a pas de manuel d'utilisation nous indiquant la librairie GraphStream à importer. Ensuite, les tests par interface donnent à l'utilisateur une certaine liberté de saisie. Dans ce programme, les données saisies par l'utilisateur ne sont pas traitées et génèrent souvent des cas d'erreurs. Les tests unitaires révèlent des problèmes concernant des valeurs hors normes (négatives, trop grandes) ou nulles (NullPointerException).

Le programme a un fonctionnement correct lors d'un usage normal et pour finir ses performances sont largement satisfaisantes.

Il existe donc des possibilités d'améliorations comme par exemple la gestion des exceptions ou des valeurs hors normes. Il est possible de rendre plus lisible l'affichage des graphes ayant un ordre relativement grand.

ANNEXES

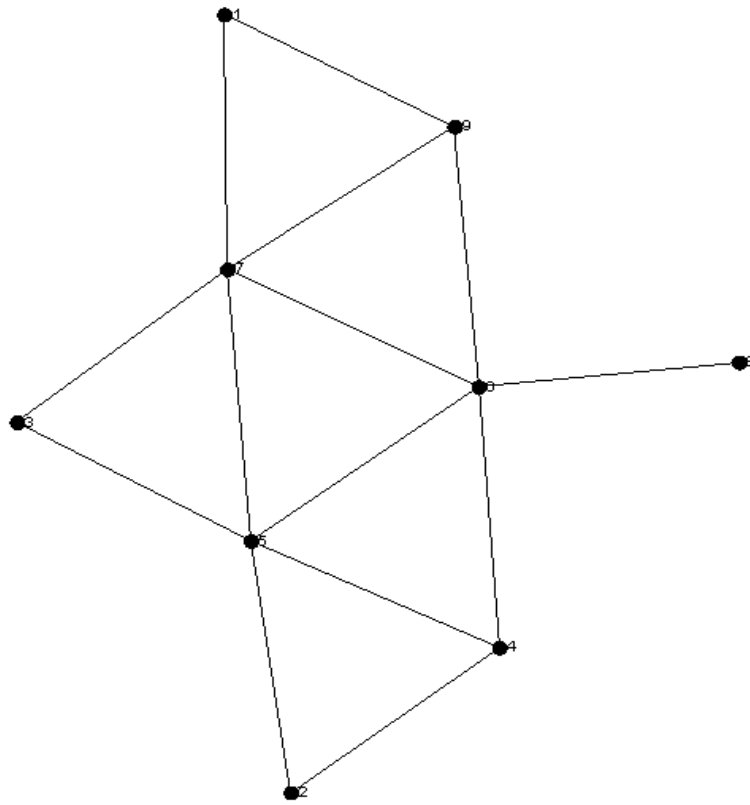


Illustration 1: Graphe g_1

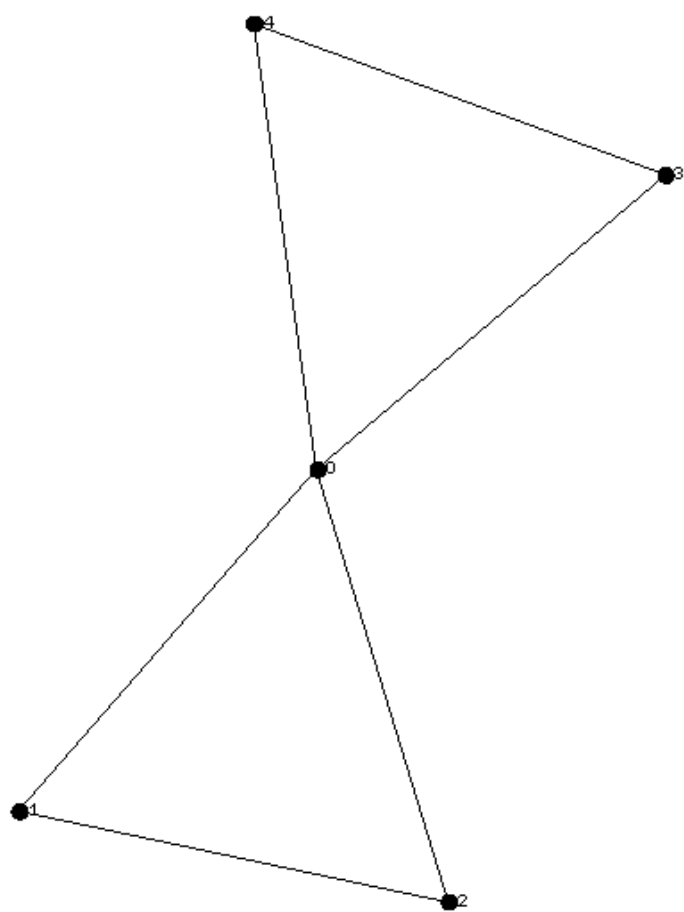


Illustration 2: graphe g2