Rapport projet A4

Mesures sur les grands graphes de terrain

Programmation en java

Réalisé par:

TOURE Sidi

4A ILC, ESIREM

Tables des matières

A. INTRODUCTION……3
B. Réalisation……..3
1. Conception……3
2. Réalisation……4
3. Code et détails……4
4. Application sur des graphes……..8
- Exemple1 sur graphe de test……8
- Exemple2 sur grand graphe……9

C. Conclusion……10

- Bibliographie……11

A. Introduction

Associés au Big Data, les grands. Graphes (réseaux sociaux, infrastructure, biologie, etc) font. L'objet d'une attention grandissant dans différents domaines. Si un graphe ne possède pas de circuit(c'est à dire est une forêt ou un arbre), alors de nombreux calculs sur celui-ci deviennent plus faciles et rapides à opérer. De nombreuses façons de mesurer de combien un graphe "s'éloigne" d'un ambrer ont été proposées, parmi elles l'hyperbolicité et la dégénérescence sont les deux mesures les plus courantes et pour lesquelles on dispose d'algorithmes performant pour les calculer.

Ainsi, l'objectif de ce projet est d'implanter les algorithmes de calcul d'hyperbolicité te. De dégénérescence pour les tester sur des graphes réels (potentiellement assez grands) et. Ainsi vérifier en quelles mesures les graphes réels sont proches des arbres. Programmé en java.

B. Réalisation

1. Conception

À partir d'un modèle de graphe, nous allons créer une instance de graphe.

Un graphe est constitué d'un ensemble de Nœuds et d'Arcs. Chaque Arc est constitué d'un Nœud Origine et d'un Nœud Extrémité. Un nom est associé aux Nœuds et aux Arcs.

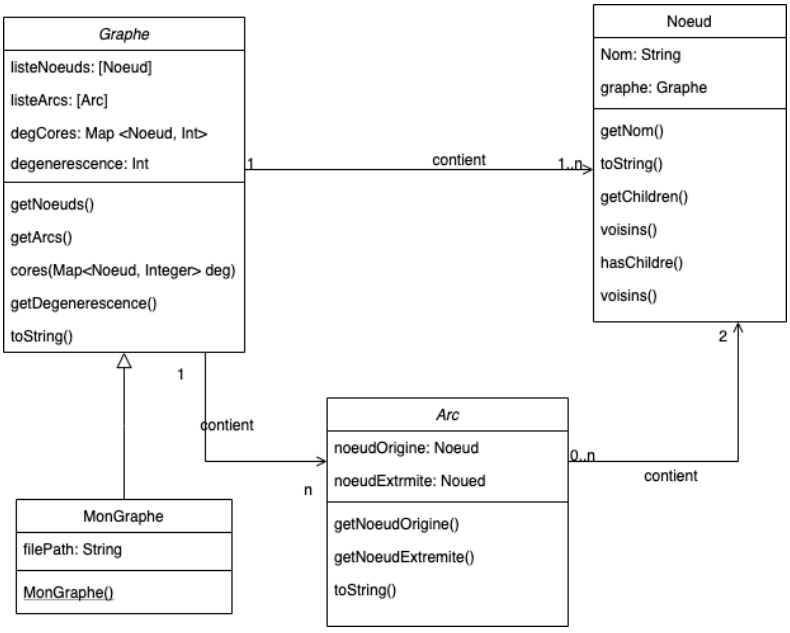


Fig1: diagramme de class du graphe

2. definition

Dans cette partie, on parlera de la réalisation du projet et le résultat obtenu à l'issue de celui-ci. En effet, afin de connaitre de combien un graphe "s'éloigne" d'un arbre, nous implanteront de ce qui suivra un algorithme très efficient de calcul de **dégénérescence**.
En théorie de graphe, la **dégénérescence** est un paramètre associé à un graphe non orienté. Un graphe est ***k*-dégénéré** si tout sous-graphe contient un nœud de degré inférieur ou égal à *k*, et la dégénérescence d'un graphe est le plus petit *k* tel qu'il est *k*-dégénéré. On peut de façon équivalente définir le paramètre en utilisant un ordre sur les sommets, on trouve alors le terme **nombre de marquage**.

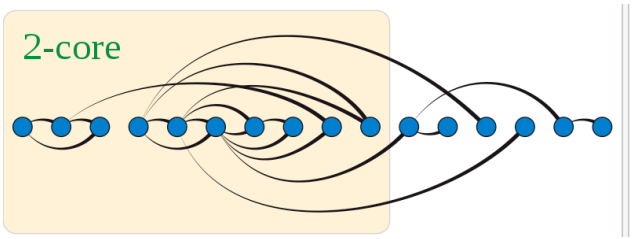


Figure2: un graphe non dégénéré dans l'ordre

à partir de la définition de la dégénérescence, on pourra définir la dégénérescence comme étant le plus grand K, telle qu'il existe K-core sous graphe de notre graphe.

3.Code

Dans notre l'algorithme, nous allons décrire une implémentation de l'algorithme dans un langage de type JAVA pour le cas d'un graphe simple non orienté G = (V, E), E est l'ensemble des arêtes.
Le graphe de structure est utilisé pour représenter un graphe donné G = (V, L). Nous ne décrirons pas la structure en détail, car il existe plusieurs possibilités, comment la mettre en œuvre. Nous supposons que les sommets de G sont numérotés de 1 à n. L'utilisateur doit également fournir la taille de la fonction, qui renvoie le nombre de sommets dans le graphique donné, et la fonction dans Voisins, qui renvoie le prochain voisin non encore visité d'un sommet donné dans le graphique donné. En utilisant une représentation adéquate du graphe G (listes de voisins), nous pouvons implémenter les deux fonctions pour qu'elles s'exécutent en un temps constant.
Deux types différents de tableaux entiers (sommets et degCores) sont également introduits. Les deux sont de longueur n. La seule différence est la façon dont nous indexons leurs éléments. Nous commençons avec l'index 1 dans Sommets et avec l'index 0 dans degreCores.

L'algorithme est implémenté par les cœurs de procédure. Son entrée est un graphe G, représenté par la variable g de type graphe; la sortie est un tableau de type **<Noeud,Int>** contenant le nombre de base pour chaque sommet du graphique G.
Nous avons également besoin (22-27 du code ci-dessous) de variables entières, 3variables Noeud, 2 dictionnaire et une liste supplémentaire (voir figure 3). Le tableau vert contient l'ensemble des sommets, triés par leurs degrés. Les positions des sommets dans le tableau vert sont stockées dans le tableau positions. Le bac de tableau contient pour chaque degré possible la position du premier sommet de ce degré dans le tableau **Sommet**.
Dans une implémentation réelle de l'algorithme proposé, des tableaux alloués dynamiquement devraient être utilisés. Pour simplifier notre description de l'algorithme, nous les avons remplacés par statiques. Au début, nous devons initialiser certaines variables et tableaux locaux (29-39). On détermine d'abord n, le nombre de sommets du graphe g. Ensuite, nous calculons son degré pour chaque sommet v dans le graphique g et le stockons dans le tableau **deg**. Simultanément, nous avons également calculer le degré maximum **maxdegre**.
Puisque les valeurs des degrés sont des entiers de l'intervalle 0 .. n - 1, nous pouvons trier les sommets dans l'ordre croissant de leurs degrés en temps linéaire à l'aide d'une variante de bin-sort (16-28).
D'abord, nous comptons (16-17) combien de sommets seront dans chaque bac (le bac se compose de sommets du même degré). Les bacs(cad les ensembles de sommets ayant les même degré) sont numérotés de 0 à **maxdegre**. A partir des tailles de bacs, nous pouvons déterminer (56-63) les positions de départ des bacs dans le tableau de sommet. Le bac 0 commence à la position 1, tandis que les autres bacs commencent à la position, égale à la somme de la position de départ et de la taille du bac précédent.

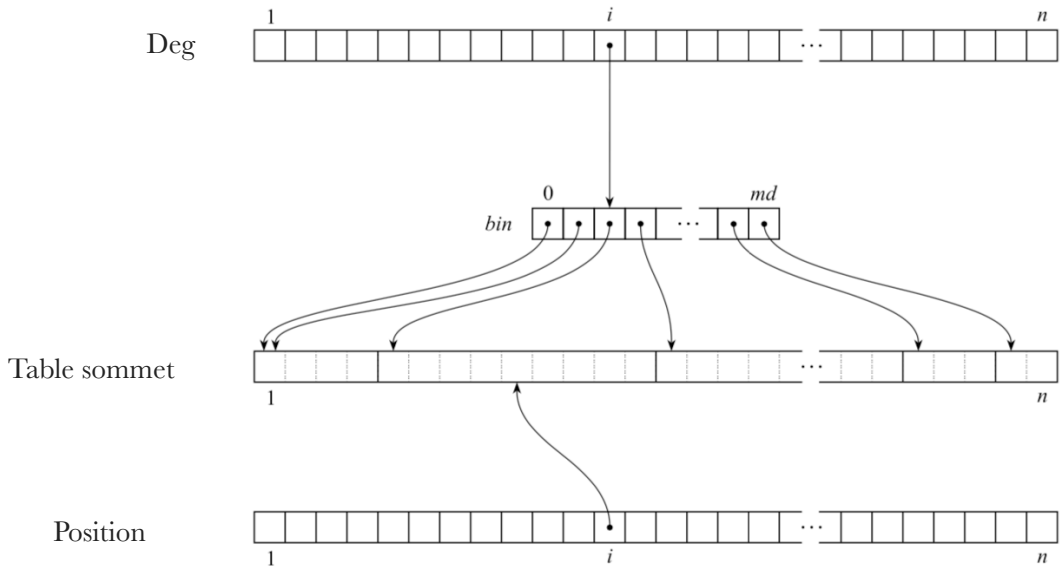
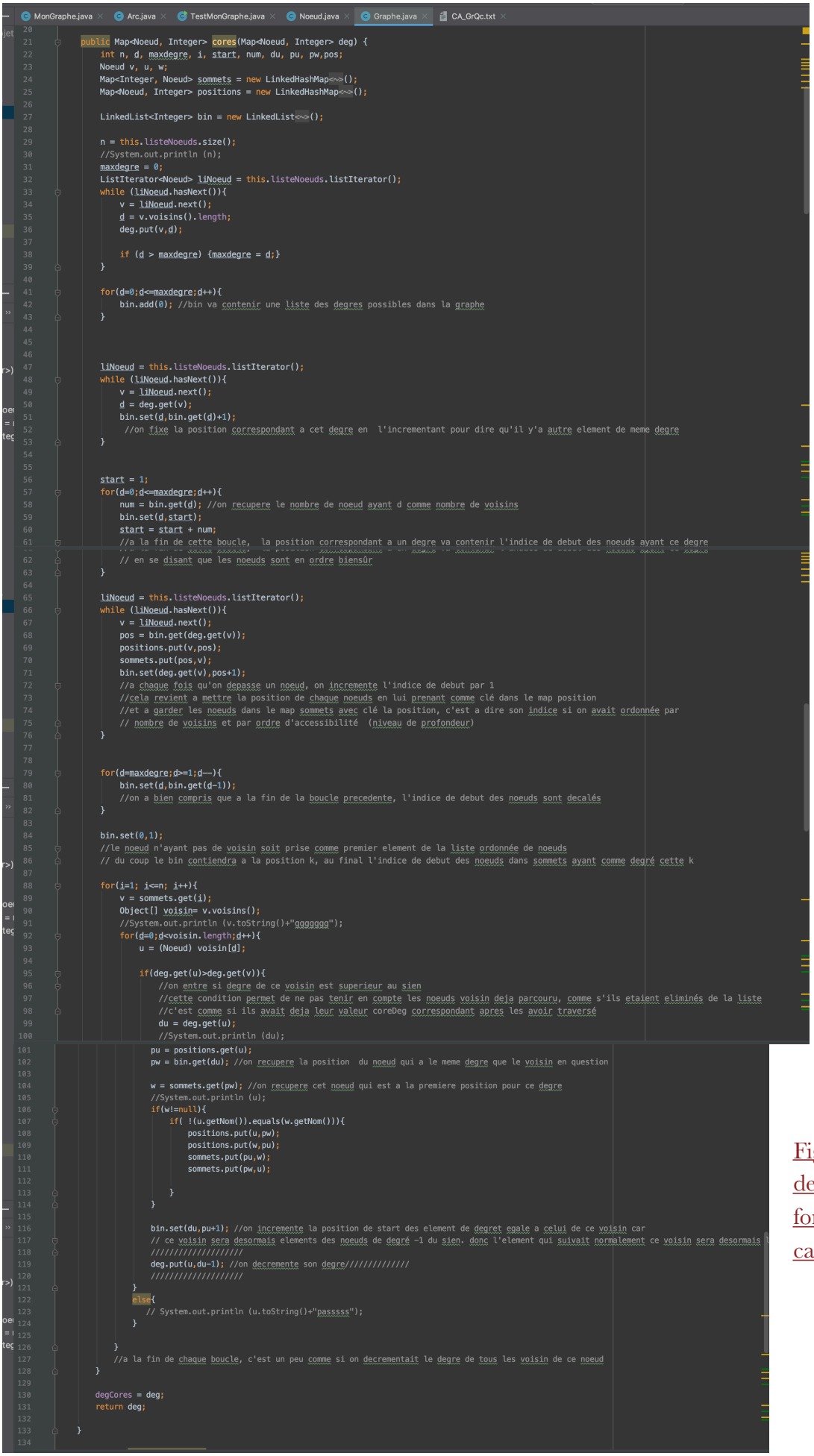


Fig3: les Tableaux




```java
public Map<Noeud, Integer> cores(Map<Noeud, Integer> deg) {
    int n, d, maxdegre, i, start, num, du, pu, pw,pos;
    Noeud v, u, w;
    Map<Integer, Noeud> sommets = new LinkedHashMap<>();
    Map<Noeud, Integer> positions = new LinkedHashMap<>();

    LinkedList<Integer> bin = new LinkedList<>();

    n = this.listeNoeuds.size();
    //System.out.println (n);
    maxdegre = 0;
    ListIterator<Noeud> liNoeud = this.listeNoeuds.listIterator();
    while (liNoeud.hasNext()){
        v = liNoeud.next();
        d = v.voisins().length;
        deg.put(v,d);

        if (d > maxdegre) {maxdegre = d;}
    }

    for(d=0;d<=maxdegre;d++){
        bin.add(0); //bin va contenir une liste des degres possibles dans la graphe
    }



    liNoeud = this.listeNoeuds.listIterator();
    while (liNoeud.hasNext()){
        v = liNoeud.next();
        d = deg.get(v);
        bin.set(d,bin.get(d)+1);
         //on fixe la position correspondant a cet degre en  l'incrementant pour dire qu'il y'a autre element de meme degre
    }


    start = 1;
    for(d=0;d<=maxdegre;d++){
        num = bin.get(d); //on recupere le nombre de noeud ayant d comme nombre de voisins
        bin.set(d,start);
        start = start + num;
        //a la fin de cette boucle,  la position correspondant a un degre va contenir l'indice de debut des noeuds ayant ce degre
        // en se disant que les noeuds sont en ordre biensûr
    }

    liNoeud = this.listeNoeuds.listIterator();
    while (liNoeud.hasNext()){
        v = liNoeud.next();
        pos = bin.get(deg.get(v));
        positions.put(v,pos);
        sommets.put(pos,v);
        bin.set(deg.get(v),pos+1);
        //a chaque fois qu'on depasse un noeud, on incremente l'indice de debut par 1
        //cela revient a mettre la position de chaque noeuds en lui prenant comme clé dans le map position
        //et a garder les noeuds dans le map sommets avec clé la position, c'est a dire son indice si on avait ordonnée par
        // nombre de voisins et par ordre d'accessibilité  (niveau de profondeur)
    }


    for(d=maxdegre;d>=1;d--){
        bin.set(d,bin.get(d-1));
        //on a bien compris que a la fin de la boucle precedente, l'indice de debut des noeuds sont decalés
    }

    bin.set(0,1);
    //le noeud n'ayant pas de voisin soit prise comme premier element de la liste ordonnée de noeuds
    // du coup le bin contiendra a la position k, au final l'indice de debut des noeuds dans sommets ayant comme degré cette k

    for(i=1; i<=n; i++){
        v = sommets.get(i);
        Object[] voisin= v.voisins();
        //System.out.println (v.toString()+"ggggggg");
        for(d=0;d<voisin.length;d++){
            u = (Noeud) voisin[d];

            if(deg.get(u)>deg.get(v)){
                //on entre si degre de ce voisin est superieur au sien
                //cette condition permet de ne pas tenir en compte les noeuds voisin deja parcouru, comme s'ils etaient eliminés de la liste
                //c'est comme si ils avait deja leur valeur coreDeg correspondant apres les avoir traversé
                du = deg.get(u);
                //System.out.println (du);
                pu = positions.get(u);
                pw = bin.get(du); //on recupere la position  du noeud qui a le meme degre que le voisin en question

                w = sommets.get(pw); //on recupere cet noeud qui est a la premiere position pour ce degre
                //System.out.println (u);
                if(w!=null){
                    if( !(u.getNom()).equals(w.getNom())){
                        positions.put(u,pw);
                        positions.put(w,pu);
                        sommets.put(pu,w);
                        sommets.put(pw,u);

                    }
                }

                bin.set(du,pu+1); //on incremente la position de start des element de degret egale a celui de ce voisin car
                // ce voisin sera desormais elements des noeuds de degré -1 du sien. donc l'element qui suivait normalement ce voisin sera desormais
                ////////////////////
                deg.put(u,du-1); //on decremente son degre//////////////
                ////////////////////
            }
            else{
               // System.out.println (u.toString()+"passsss");
            }

        }
        //a la fin de chaque boucle, c'est un peu comme si on decrementait le degre de tous les voisin de ce noeud
    }

    degCores = deg;
    return deg;

}
```

Fig4: le code de la fonction de calcul

Pour éviter un tableau supplémentaire, nous avons utilisé le même tableau (**bin**) pour stocker les positions de départ des bacs. Maintenant, nous pouvons mettre (65-73) sommets du graphe G dans le tableau **Sommet**. Pour chaque sommet, nous savons à quel bac il appartient et quelle est la position de départ de ce bac. Ainsi, nous pouvons mettre le sommet actuel à la bonne place, nous souvenir de sa position dans la table **position** et augmenter la position de départ du bac que nous avons utilisé. Les sommets sont maintenant triés par leurs degrés. Dans la dernière étape de la phase d'initialisation, nous devons récupérer les positions de départ des bacs noté **bin** (79-82). Nous les avons augmentés plusieurs fois à l'étape précédente, lorsque nous avons placé les sommets dans les cases correspondantes. Il est évident que la position de départ modifiée est la position de départ d'origine du bac suivant. Pour restaurer les bonnes positions de départ, nous devons déplacer les valeurs dans le bac du tableau pour une position vers la droite. Nous devons également réinitialiser la position de départ du bac 0 à la valeur 1.
La décomposition des cœurs, implémentant la pour chaque boucle de l'algorithme décrit dans la section 3, se fait dans la boucle principale (32-50) qui parcourt tous les sommets v du graphe g dans l'ordre, déterminé par la table vert. Le nombre de base du sommet actuel v est le degré actuel de ce sommet. Ce nombre est déjà stocké dans le tableau deg. Pour chaque voisin u du sommet v de degré supérieur, nous devons diminuer son degré de 1 et le déplacer d'un bac vers la gauche. Déplacer le sommet u pour un bac vers la gauche est une opération qui peut être effectuée en un temps constant. Nous devons d'abord échanger le sommet u et le premier sommet dans le même bac. Nous devons également échanger leurs positions dans le tableau **positions**. Enfin, nous augmentons la position de départ du bac (nous augmentons le précédent et réduisons le bac actuel pour un élément).
A la fin de l'application de cette méthode sur le graphe, on aura à la sortie que le tableau **degCores** <Noeud, Int> contient tous les noeuds avec leurs correspondant k en termes du dernier K-cores auquel ils appartiendront lorsqu'on procède à une formation progressive des k-core en allant de 0-core au dernier k-core. D'ou il ne restera qu'à appliquer un calcul de la plus grande valeur de cette liste pour afin conclure pour la dégénérescence.

```
public int getDegenerescence(){
    this.cores(this.degCores);
    ListIterator<Noeud> liNoeud = this.listeNoeuds.listIterator();
    while (liNoeud.hasNext()){
        Noeud noeudCourant = liNoeud.next();
        if(degenerescence<degCores.get(noeudCourant)){
            degenerescence=degCores.get(noeudCourant);
        }
    }
    return degenerescence;
}
```

Fig4: fonction de calcul de la dégénérescence

4. APPLICATION SUR DES GRAPHES:

Exemple1: il s'agit d'appliquer les calcul précédent sur le graphe ci dessous.

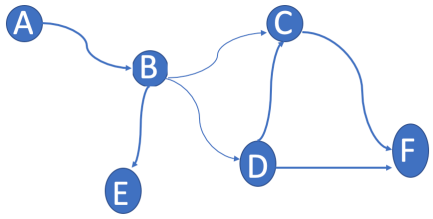


Fig5: graphe simple

L'exécution du code nous donne le résultat suivant

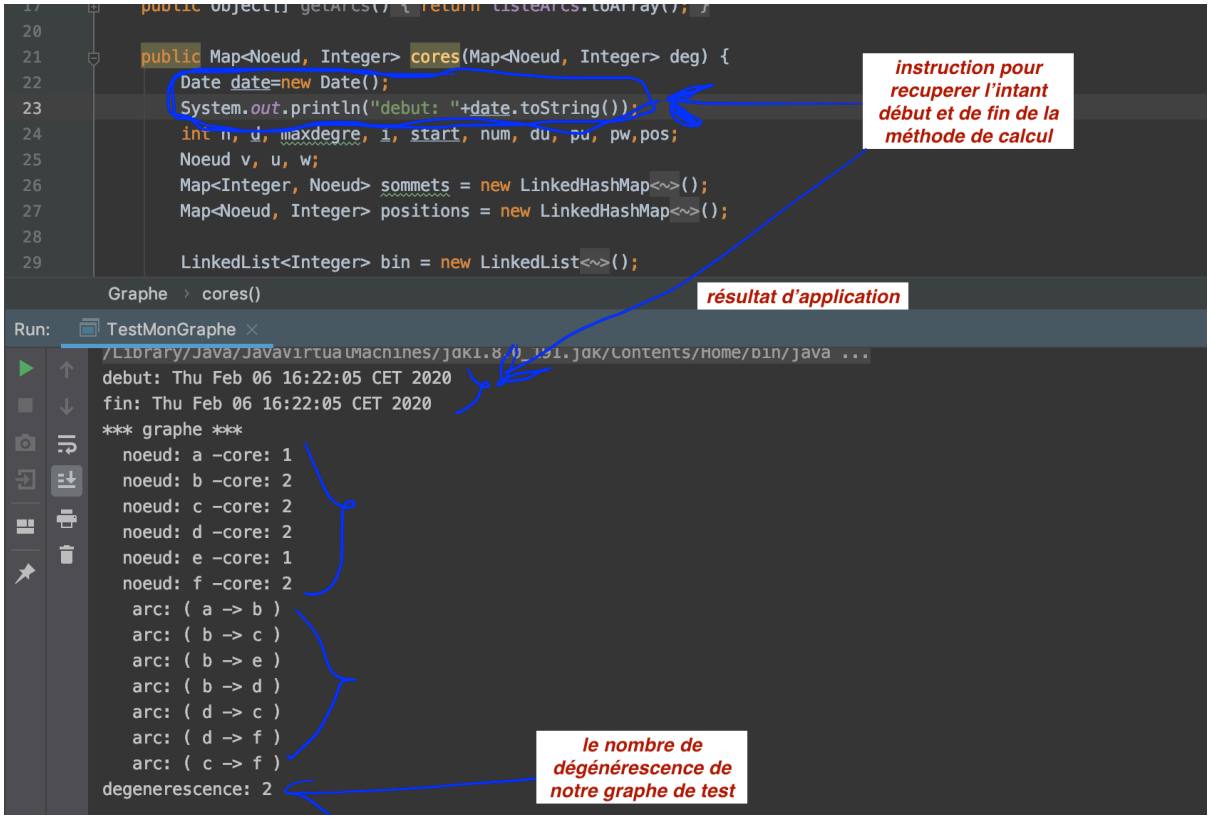


Fig6: résultat d'application du code sur un graphe simple

À l'issue de cela, on remarque que l'algorithme fait le calcul en moins d'une seconde dans ce cas. Ceci est dû au fait que on a pas autant de nombre de Sommets pour que le calcul soit si long que ça. Ainsi, la question qu'on se pose est qu'obtient-on dans le cas d'un graphe ayant un nombre important de Sommets ?
D'où l'utilisation des grand graphe de http://snap.stanford.edu/data

Exemple2: Application sur un grand graphe.
Il s'agit ici de la graphe récupéré de la dataSet de Snap Standford décrit par [Collaboration network of Arxiv General Relativity category](http://snap.stanford.edu/data/ca-GrQc.html).
Ceci peut être considéré comme un grand un graphe grâce à sont grand nombre de Sommets (soit 5242 noeud) et d'arc (soit 14496 edges).

On commence par initié le graphe en respectant la conception du début, le code permettant la lecture du fichier et la construction du graphe est le suivant:

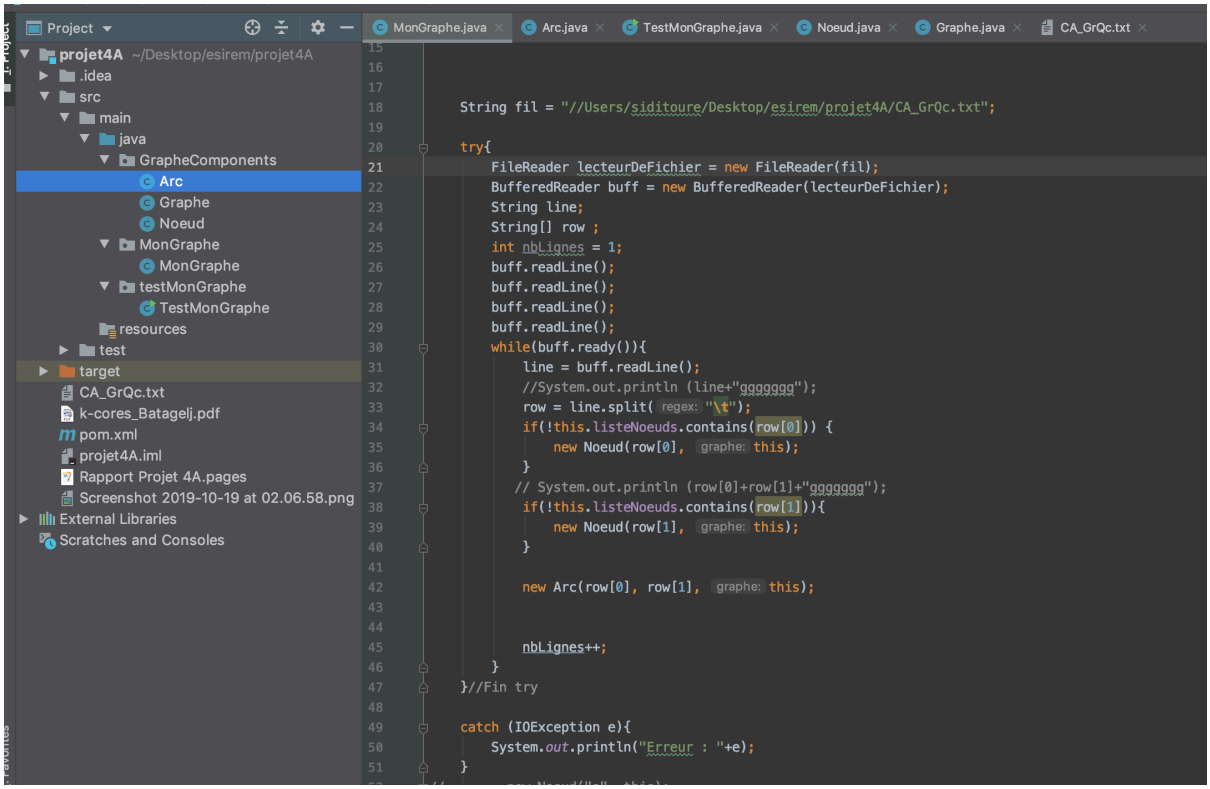


Fig7: code de construction du graphe à partir du fichier CA_GrQc.txt récupéré dans le stock de http://snap.stanford.edu/data.
L'application de la méthode sur celui-ci nous retourne un temps de calcul de la dégénérescence égal à 162 avec un temps de calcul de 56secondes.on **dira donc que notre graphe est 162-dégénéré**

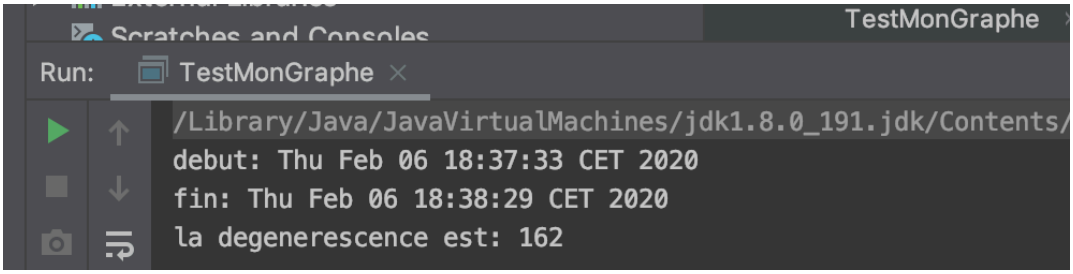


Fig8: temps de calcul sur le grand graphe et dégénérescence

C. Conclusion

En conclusion, d'une part on aura compris que la relation entre la relation entre la determination des k-cores d'un graphe et la dégénérescence de celui-ci nous permet faire le calcul de la dégénérescence afin d'avoir une idée du mesure de combien "s'éloigne" t-il d'un arbre. Part ailleurs, on a remarque que cet algorithme est le mieux adapté pour les petits graphes par défaut du temps de calcul que celui-ci pourra prendre lorsqu'il s'agit d'un grand graphe. Pour finir, on retiendra que cette durée peut néanmoins varier d'un langage de programmation à un autre.

Le calcul **d'hyperbolicité** peut être ajouté dans le code du projet. Pour cela, il suffit d'ajouter une méthode implémentant un algorithme qui calcule celui-ci.

•bibliographie

[1]: definition de la dégénérescence, https://fr.wikipedia.org/wiki/D%C3%A9g%C3%A9n%C3%A9rescence_(th%C3%A9orie_des_graphes)

[2]: Algorithme de calcul d'hyperbolicité, alhttps://hal.archives-ouvertes.fr/hal-00818441/document

[3]: Bases de données de grands graphes SNAP, https://snap.stanford.edu

[4]: Bases de données de grands graphes KONECT, http://konect.uni-koblenz.de/

[5]: Algorithme de calcul de k-core, http://www.cse.cuhk.edu.hk/~cslui/CMSC5734/k-cores_Batagelj.pdf

[6]: Modèle de graphe en java, https://sbachmann.developpez.com/eclipse/applicationgraphe/