SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Rapport de CPA

Etude sur les très grands graphes

Encadrant : Maximilien Danisch
Par : BOUCHENAKI Habib 3522424
ET
MEYNIER Florent 3700051
15 mars 2021

TME 1 : Handling a large graph

Démarche :

Pour ce TME d'introduction, le premier objectif était de pouvoir lire, charger et récupérer des informations sur de très grands graphes (Amazon, Livejournal, Orkut, Friendster). Ainsi, dans un premier temps, nous avons créer une classe JAVA capable de lire et de compter le nombre de noeuds et de liens dans un graphe. Ensuite, nous avons utiliser différents types de structures de données (liste de liens, matrice d'adjacence et enfin la liste d'adjacence) afin de les comparer et de déterminer la plus à même à stocker les graphes de manières efficace. Enfin, nous avons utiliser cette structure afin d'effectuer deux opérations sur les graphes, la première étant le BFS nous retournant, le plus petit diamètre dans un graphe. Et la seconde étant la recherche de triangles, nous retournant le nombre de triangles présent dans un graphe.

Algorithmes importants utilisés :

BFS :

```
public int execute(int s)
{
    boolean visited[] = new boolean[graphe.size()];
    int distances [] = new int[graphe.size()];
    Arrays.fill(distances, 0);

    int d = 0;

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

    visited[s] = true;
    queue.add(s);

    while (queue.size() != 0) {
        s = queue.poll();
        if(d==0 || queue.isEmpty())
            System.out.print(s + " ");

        Iterator<Integer> i = (graphe.getGraphe())[s].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                visited[n] = true;
                distances[n] = distances[s]+1;
                queue.add(n);
            }
            d=s;
        }
    }
    System.out.println();
    System.out.println(distances[d]);
    return d;
}
```

Triangles :

```java
public List<Triangle> findTriangles()
{
    List<Triangle> res = new ArrayList<Triangle>();

    for(Edge e : graph.getEdges())
    {
        int u = e.getFrom();
        int v = e.getTo();
        int i = 0, j = 0;
        List<Integer> vu = graph.getGraphe()[u];
        List<Integer> vv = graph.getGraphe()[v];
        while(i < vu.size() && vu.get(i)  <= v)
        {
            i++;
        }
        while(j < vv.size() && vv.get(j) <= u)
        {
            j++;
        }
        while(i < vu.size() && j < vv.size())
        {
            if(vu.get(i) < vv.get(j))
            {
                i++;
            } else if(vu.get(i) > vv.get(j))
            {
                j++;
            } else
            {
                res.add(new Triangle(u, v, vu.get(i)));
                i++;
                j++;
            }
        }
    }
    return res;
}
```

Résultats :

Pour la première question du TME les résultats trouvés sont :

```
------------------------------
File: com-amazon.ungraph.txt
Nodes count: 334863
Edge count: 925872
------------------------------
------------------------------
File: com-lj.ungraph.txt
Nodes count: 3997962
Edge count: 34681189
------------------------------
------------------------------
File: com-orkut.ungraph.txt
Nodes count: 3072441
Edge count: 117185083
------------------------------
------------------------------
File: com-friendster.ungraph.txt
Nodes count: 65608366
Edge count: 1806067135
------------------------------
```

Pour la deuxième question, seule la adjacency array nous a permis de charger les 4 graphes, edgeList ne stockait pas plus que LiveJournal et adjacency Matrix ne stockait même pas amazon (max noeuds stockés $\approx$ 50000).

Résultats BFS :

```
------------------------------
File: com-amazon.ungraph.txt g1: 925872
474072 262774 150736 34 150736 262774 99182 333319
47
333319 287820 303444 418390 419606 262774 150736
47
150736 262774 99182 333319
47
333319 287820 303444 418390 419606 262774 150736
47
150736 262774 99182 333319
47
333319 287820 303444 418390 419606 262774 150736
47
150736 262774 99182 333319
47
333319 287820 303444 418390 419606 262774 150736
```

```
    47
    150736 262774 99182 333319
    47
    time elapsed 1203 ms 408549 ns
    ------------------------------
    ------------------------------
    File: com-lj.ungraph.txt g1: 34681189
    1041487 1961 3911578
    12
    3911578  3911576  3911577  3911579  3911580  3911581  3163482  1072389
3847908
    21
    3847908 3847906 3847907 50 3911578
    21
    3911578  3911576  3911577  3911579  3911580  3911581  3163482  1072389
3847908
    21
    3847908 3847906 3847907 50 3911578
    21
    3911578  3911576  3911577  3911579  3911580  3911581  3163482  1072389
3847908
    21
    3847908 3847906 3847907 50 3911578
    21
    3911578  3911576  3911577  3911579  3911580  3911581  3163482  1072389
3847908
    21
    3847908 3847906 3847907 50 3911578
    21
    3911578  3911576  3911577  3911579  3911580  3911581  3163482  1072389
3847908
    21
    time elapsed 23528 ms 269433 ns
    ------------------------------
    ------------------------------
    File: com-orkut.ungraph.txt g1: 117185083
    1096342 2246285
    7
    2246285 64579 1044 2247779
    9
    2247779 64714 2251796
    9
```

```
2251796 65090 2250603
9
2250603 65080 1129 2363575
10
2363575 73114 2250603
10
2250603 65080 1129 2363575
10
2363575 73114 2250603
10
2250603 65080 1129 2363575
10
2363575 73114 2250603
10
time elapsed 69568 ms 875964 ns
------------------------------
```

Ainsi, on a pour :

- Amazon le diamètre minimum qui vaut : 47
- Live Journal le diamètre minimum qui vaut : 21
- Orkut le diamètre minimum qui vaut 10
- Friendster trop gros graphe notre RAM ne suffit pas à le charger.

Résultats Triangles :

Amazon : Find 667129 triangles in 759ms

Lj : Find 177820130 triangles in 4857855ms

Orkut : none

Friendster : none

Ainsi on a pour :

- Amazon le nombre de triangles qui vaut : 667129
- Live Journal le nombre de triangles qui vaut : 177820130
- Orkut trop gros fichier pour notre RAM.
- Friendster trop gros fichier pour notre RAM.

TME 2 : PageRank

Démarche :

L'objectif de ce TME est d'implémenter un algorithme qui va permettre de noter des pages webs et ainsi permettre à un marcheur aléatoire de choisir son chemin en fonction de cette note. Pour calculer ces notes, nous implémentons l'algorithme Power Iteration en utilisant un certain alpha. Cet algorithme se termine lorsque les différentes notes des pages convergent.

Algorithme importants :

PowerIteration :

```java
public Double[] powerIteration(double alpha, int t)
{
    List<Value> []matrix = transition();
    Double P[] = new Double[graph.size()];

    double unN = 1.0 / nbNodes;
    for(int i = 0; i < graph.size(); i++)
    {
        P[i] = unN;
    }
    display(P);
    for(int i = 0; i < t; i++)
    {
        P = matVectProd(matrix, P);
        display(P);
        double norme = 0.0;
        for(int p = 0; p < P.length; p++)
        {
            P[p] = (1 - alpha) * P[p] + alpha * unN;
            norme += (P[p] * P[p]);
        }
        norme = Math.sqrt(norme);
        for(int p = 0; p < P.length; p++)
        {
            P[p] += ((1 - norme) / nbNodes);
        }
    }
    return P;
}
```

```java
private Double[] matVectProd(List<Value>[] matrix, Double[] A)
{
    Double []B = new Double[A.length];
    for(int i = 0; i < B.length; i++)
    {
        B[i] = 0.0;
    }
    for(Edge e : graph.getEdges())
    {
        Value v = getValueFromMatrix(matrix, e.getFrom(), e.getTo());
        if(v == null)
        {
            continue;
        }
        B[e.getFrom()] += v.getDegree() * A[e.getTo()];
    }
    return B;
}

@SuppressWarnings("unchecked")
private List<Value>[] transition()
{
    List<Value> []matrix = new ArrayList[graph.size() + 1];
    for(int i = 0; i < graph.size() + 1; i++)
    {
        matrix[i] = new ArrayList<Value>();
    }
    for(Edge e : graph.getEdges())
    {
        matrix[e.getFrom()].add(new Value(e.getTo(), graph.getGraphe()[e.getTo()].size()));
    }
    return matrix;
}

private Value getValueFromMatrix(List<Value>[] matrix, int i, int j)
{
    for(int to = 0; to < matrix[i].size(); to++)
    {
        if(matrix[i].get(to).getNode() == j)
        {
            return matrix[i].get(to);
        }
    }
    return null;
}
```

Résultats :

Nous n'avons pas réussi à implémenter correctement l'algorithme du cours. Les résultats obtenus ne sont pas cohérents avec ce qui est attendu. Lors de nos recherches pour tenter de faire fonctionner l'algorithme, nous sommes tomber sur la librairie Python networkx qui implémentait l'algorithme du PageRank. Nous l'avons exécuté et avons pu observer le résultat sur les fichiers fournis. Nous avons tout de même essayé après cela de corriger les bugs, mais sans succès.

TME 3 : Practical Work

Démarche :

L'objectif de ce TME est de détecter des communautés dans un graphe. Pour cela, nous avons crée notre propre graphe avec des clusters déjà présents. Chaque noeud avec une probabilité p d'avoir un lien avec un autre noeud du cluster et une probabilité q d'avoir un lien avec un noeud d'un autre cluster. Une fois ce graphe généré, nous l'avons utilisé afin d'appliquer nos algorithmes de détection de communauté. Nous avons implémenté un premier algorithme qui utilise le principe de propagation. Nous avions ensuite dû réfléchir à implémenter notre propre algorithme de détection de communauté. Pour cela nous nous sommes un peu inspiré du fonctionnement de l'algorithme de propagation que nous avons appliqué sur les triangles du graphe. Pour finir, nous avons implémenté l'algorithme de Louvain et avons comparé les résultats de ces 3 algorithmes en affichant les graphes.

Algorithmes importants :

LabelPropagation :

```java
public void propagate()
{
    liste = new ArrayList<>();
    labels = new ArrayList<>();
    boolean swap = false;

    for(int i = 0; i < graph.size(); i++)
    {
        liste.add(i);
        labels.add(i);
    }
    do
    {
        swap = false;
        Collections.shuffle(liste);

        for(Integer u : liste)
        {
            if(u == 0)
            {
                break;
            }
            int max = 0;
            int idMax = 0;
            Map<Integer,Integer> m = new HashMap<>();

            for(Integer v : graph.getGraphe()[u])
            {
                int lv = labels.get(v);
                if(m.containsKey(lv))
                {
                    int nbV = m.get(lv)+1;
                    m.put(lv, nbV);
                    if(nbV>max)
                    {
                        max = nbV;
                        idMax = lv;
                    }
                }else
                {
                    m.put(lv, 1);
                    if(idMax == 0)
                    {
                        idMax = lv;
                        max = 1;
                    }
                }
            }
            if(idMax == 0)
            {
                continue;
            }
            if(labels.get(u) != idMax)
            {
                labels.set(u, idMax);
                swap = true;
            }
        }
    }while(swap);
}
```

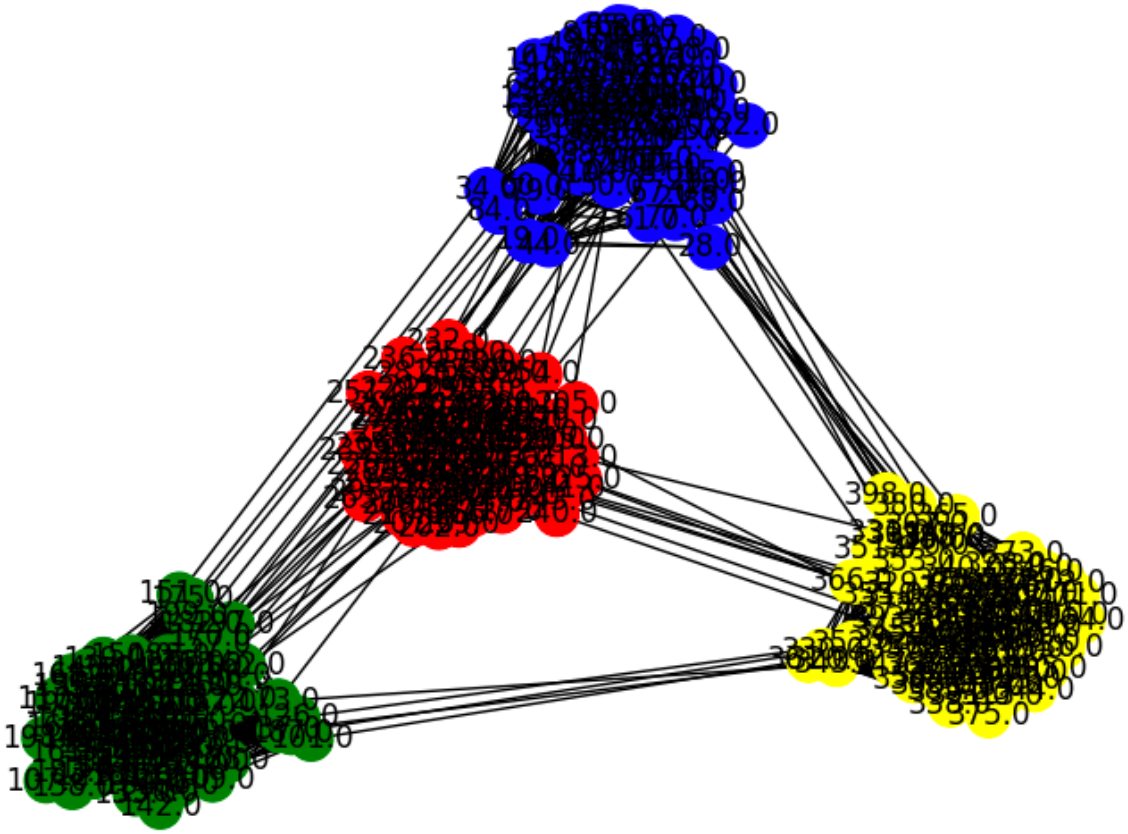
Algorithme personnel :

```java
public void propagate()
{
    List<Triangle> list = new FindTriangles(graph).findTriangles();
    labels = new ArrayList<Integer>();
    boolean swap;

    for(int i = 0; i < graph.size(); i++)
    {
        labels.add(i);
    }
    List<Triangle> traite = new ArrayList<Triangle>();
    do
    {
        swap = false;

        for(Triangle t : list)
        {
            if(labels.get(t.getN1()) == labels.get(t.getN2()) && labels.get(t.getN1()) != labels.get(t.getN3()))
            {
                if(traite.contains(t))
                {
                    continue;
                }
                traite.add(t);
                swap = true;
                labels.set(t.getN3(), labels.get(t.getN1()));
            } else if (labels.get(t.getN2()) == labels.get(t.getN3()) && labels.get(t.getN2()) != labels.get(t.getN1()))
            {
                if(traite.contains(t))
                {
                    continue;
                }
                traite.add(t);
                swap = true;
                labels.set(t.getN1(), labels.get(t.getN2()));

            } else if(labels.get(t.getN1()) == labels.get(t.getN3()) && labels.get(t.getN1()) != labels.get(t.getN2()))
            {
                if(traite.contains(t))
                {
                    continue;
                }
                traite.add(t);
                swap = true;
                labels.set(t.getN2(), labels.get(t.getN1()));
            } else {
                int rand = new Random().nextInt(3);
                switch(rand)
                {
                case 0:
                    labels.set(t.getN1(), labels.get(t.getN2()));
                    break;
                case 1:
                    labels.set(t.getN2(), labels.get(t.getN3()));
                    break;
                case 2:
                    labels.set(t.getN3(), labels.get(t.getN1()));
                    break;
                default:
                    break;
                }
            }
        }
    } while(swap);
}
```

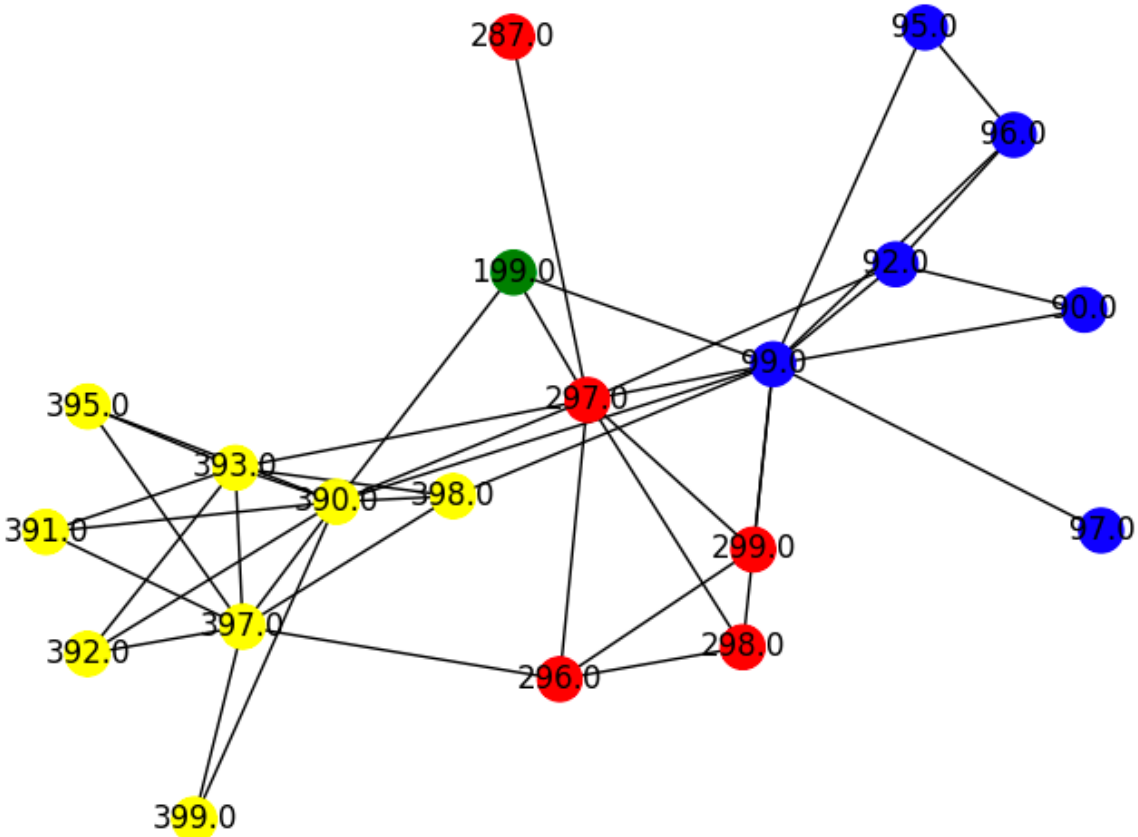
Algorithme Louvain :

```
public void propagate()
{
    comm = new ArrayList<Integer>();
    modC = new int[graph.size()];

    boolean swap = false;
    for(int i = 0; i < graph.size(); i++)
    {
        comm.add(i);
        modC[i] = graph.getGraphe()[i].size();
    }
    double q = Double.MIN_VALUE;
    do {
        swap = false;
        System.out.println("");
        afficheLabels();
        for(int i = 1; i < graph.size(); i++)
        {
            int commI = comm.get(i);
            modC[comm.get(i)] -=  graph.getGraphe()[i].size();

            double maxQ = Double.MIN_VALUE;
            int bestComm = commI;
            for(Integer j : graph.getGraphe()[i])
            {
                q = modularity(i, j);
                if(q >= maxQ)
                {
                    maxQ = q;
                    bestComm = comm.get(j);
                    //comm.set(i, comm.get(j));
                }
            }
            modC[bestComm] += graph.getGraphe()[i].size();
            if(commI != bestComm)
            {
                swap = true;
                comm.set(i, bestComm);
            }
        }
    }while(swap);
    afficheLabels();
}
```

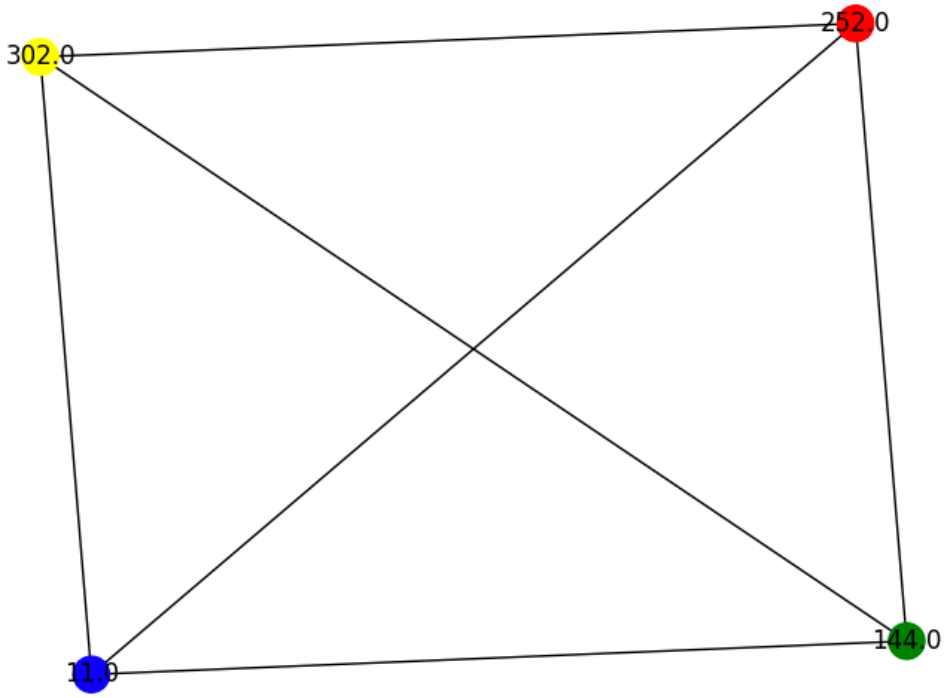
Graphe généré :



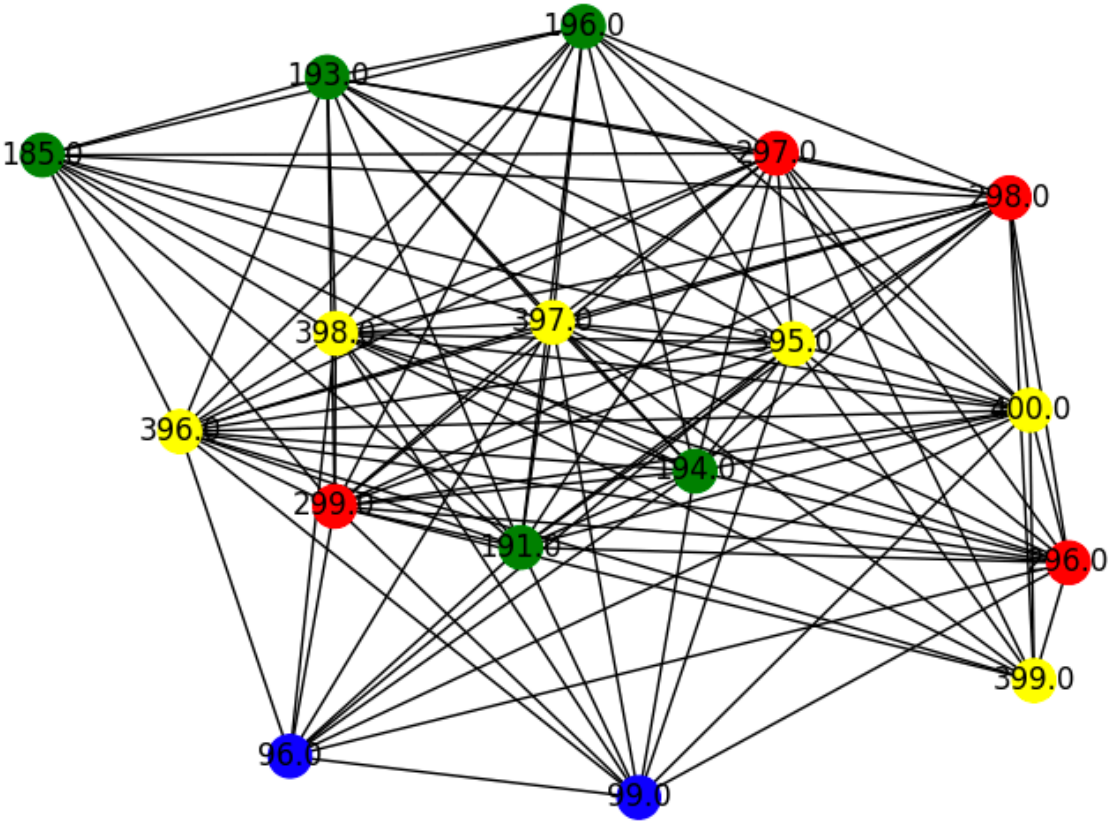
Résultat LabelPropagation :



Résultat Algorithme personnel :



Résultat Louvain :



Discussion sur les Algorithmes :

En augmentant le ratio p/q, les clusters seront de plus en plus distinct. En effet, les noeuds d'un même cluster auront une probabilité plus élevé d'avoir un lien avec un autre noeud de ce cluster, et aura donc une probabilité plus faible d'avoir un lien avec un noeud d'un autre cluster. A l'inverse, en diminuant le ratio p/q, les clusters seront moins distincts car les noeuds auront de plus en plus de liens avec les noeuds d'un autre cluster.

Fichiers tests utilisés :
generation 400 noeuds (p=0.2 q=0.001)
generation 10000 noeuds (p=0.02 q=0.00001)

Temps de calcul des algorithmes en fonction de la taille des graphes :
400 noeuds :

- LabelPropagation : 58ms
- Triangle : 93ms
- Louvain : 31ms

10000 noeuds (millions de liens):

- LabelPropagation : 1500ms
- Triangle : 16655086ms
- Louvain : 289 ms

Taux de précision des algorithmes en fonction de la taille des graphes : (Accuracy.py)
400 noeuds :

- LabelPropagation : 92.44 %
- Triangle : 96.22 %
- Louvain : 92.44 %

10000 noeuds (millions de liens) :

- LabelPropagation : 91.05 %
- Triangle : 99.57 %
- Louvain : 90.99 %

En Conclusion :

A travers ces résultats, on peut voir que notre algorithme personnel a un taux de précision qui surpasse les deux autres algorithmes aux alentours de 96 % pour des petits graphes et 99 % pour des grands graphes, cependant son temps d'exécution est également drastiquement supérieur. L'algorithme de LabelPropagation quant à lui offre des précisions tout à fait satisfaisantes ($> 90\,\%$) avec un temps d'exécution pouvant aller jusqu'à 1,5 s sur de grands graphes. Ainsi, pour conclure notre étude, c'est l'algorithme de Louvain qui offre les meilleurs performances, avec un taux de précision similaire à l'algorithme de LabelPropagation mais avec un temps d'exécution 5 fois plus faible sur de grands graphes.

TME 4 : Densest subgraph

Démarche :

L'objectif de ce TME est de pouvoir déterminer des sous-graphe en fonction du core déterminé. Pour cela nous avons dû implémenter un algorithme qui va décomposer le graphe en sous-graphe, et en fonction du degrés de chaque noeud ainsi que du degrés des voisins, nous avons calculé le core du graphe. Puis nous avons effectué des calculs sur les résultats de l'algorithme afin de calculer le edge density, average degree density et d'autres données dans le but de pouvoir détecter les noeuds anormaux de notre graphe.

Algorithmes importants :

K-Core Décomposition :

```
public ResultKCore decompose()
{
    List<Integer> l = new ArrayList<Integer>();
    List<Integer> d = new ArrayList<Integer>();
    int dMax = 0;
    for(int i = 0; i < g.size(); i++)
    {
        if(g.getGraphe()[i].size() > dMax)
        {
            dMax = g.getGraphe()[i].size();
        }
    }
    List<List<Integer>> D = new ArrayList<>();
    for(int i = 0; i < dMax + 1; i++)
    {
        D.add(new ArrayList<Integer>());
    }
    for(int i = 0; i < g.getMax(); i++)
    {
        d.add(g.getGraphe()[i].size());
        D.get(d.get(i)).add(i);
        l.add(0);
    }
    int i = 0;
    for(int k = 0; k <= dMax; k++)
    {
        while(!D.get(k).isEmpty())
        {
            i = D.get(k).remove(0);
            l.set(i, k);
            for(int v = 0; v < g.getGraphe()[i].size(); v++)
            {
                int j = g.getGraphe()[i].get(v);
                if(g.getGraphe()[j].size() > k)
                {
                    int dJ = g.getGraphe()[j].size();
                    for(int u = 0; u < D.get(dJ).size(); u++)
                    {
                        if(D.get(dJ).get(u) == j)
                        {
                            D.get(dJ).remove(u);
                                D.get(dJ - 1).add(j);
                                d.set(j, d.get(j) - 1);
                                break;
                            }
                        }
                    }
                }
            }
        }
    }

    return new ResultKCore(dMax, l, (g.getNbEdges() / (g.getNbNodes() * 1.0)), g.getNbEdges() / (g.getNbNodes() * (g.getNbNodes() - 1) / 2.0));
}
```

Résultats :

Amazon :

```
Fichier amazon :
c : 6
Average degree density : 2.7649277465709856
Edge density : 0.003993028829405762
```

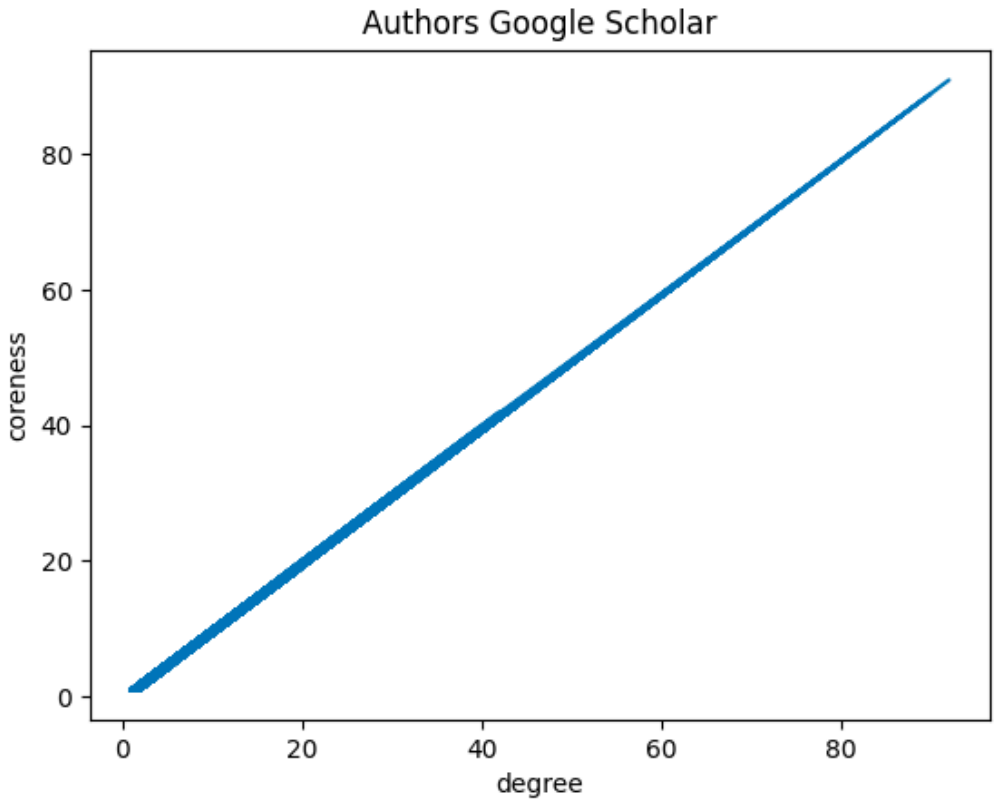
LiveJournal :

```
Fichier lj :
c : 17
Average degree density : 8.674717018320834
Edge density : 0.032674222797734034
```

Orkut : `Trop Long`

Friendster : `Trop gros pour notre RAM.`

Résultats Google Scholar :



Problèmes rencontrés :

Nous avions commencé les 2 premiers TME en C et en Java, mais nous sommes rapidement passé uniquement sur du Java car nous sommes plus à l'aise dans ce langage. Le premier TME s'est fait rapidement et sans réel accroc. Le second TME a été plus complexe, nous avons eu du mal à implémenter les algorithmes, et d'ailleurs nos implémentations ne sont pas correctes. Nous avons mieux réussi le TME3 que le 2, même si nous avons passé beaucoup de temps sur les algos. Nous avons fait pas mal de recherches pour mieux les comprendre et chercher d'autres implémentations plus simples à mettre en oeuvre, tout en respectant les consignes de l'énoncé. Le dernier TME, bien que plus court que les autres n'a pas été plus facile. Nous n'avions pas totalement compris ce qu'il fallait implémenter, ce que nous devions faire des résultats ni comment les représenter. Le manque d'informations sur les différents algos des TME nous a beaucoup ralenti. Plus généralement les algos ont été compliqués à tester car nos ordinateurs ne possédaient pas suffisamment de RAM. Nous avions créé pour chaque TME des petits fichiers sur lesquels itérer, néanmoins, lorsqu'il a fallu tester sur les fichiers que vous nous avez fourni, il fallait pour certains algorithmes plusieurs heures afin d'exécuter l'algorithme dessus. Ce temps nous ne le possédions pas toujours car devions, pendant que les algorithmes tournent, travailler sur d'autres UES. Pour finir, nous avons sûrement consacré plus de 2 fois le temps des TME à cette UE sans parvenir à un résultat convenable (nous ne sommes pas satisfait du rendement). Il a fallu faire des compromis dans nos choix de travail, et avons tenté d'équilibrer nos temps de travail sur chacune des UES, pour ne pas se mettre en retard. Ce paragraphe n'excuse en rien le manque de résultats sur certaines parties des TME mais tente d'en expliquer plusieurs raisons.