UCLouvain - EPL

LINFO1104

Reasoning about a highly connected world

Report of the project

Professeur :
Van Roy Peter

Assistants :
Darockzy Balint
Gillard Xavier

Auteurs :
Ben Haddou Mehdi - 19912000
Hennebo Eliot - 43762000

1 Introduction

Dans le monde d'aujourd'hui, **les réseaux sociaux occupent une place de plus en plus importante**. Dans le cadre du cours de Reasoning about a highly connected world : graph theory, game theory, and networks, il nous a été demandé d'analyser le réseau social de Dribbble, qui vise à connecter les designers à travers le monde en les laissant partager leurs créations.

Les analyses qui ont été faites peuvent être séparées **en trois parties** ;

1. L'analyse en surface du graph.
2. La recherche des top designers via l'algorithme PageRank.
3. L'analyse des liens qui se créent entre les designers au travers du temps.

En ce qui concerne la structure plus spécifique de ce rapport, nous avons décidé de diviser chaque tâche en deux sections **pour plus de clarté** :

1. Analyse de la tâche, description et interprétation des résulats
2. Description du code et de l'implémentation

Le code de ce projet sera en Java et utilisera donc une structure de données qui nous est propre et non pas tirée de la librairie Networkx en Python.

2 Tâche 1

2.1 Analyse & résulats

2.1.1 Analyse

Dans cette tâche, nous avons dû analyser différentes informations du graph. Le graph est initialement un graph dirigé. En effet, la structure typique Follower $\longrightarrow$ Followed fait que ce graph est sans nul doute un graph dirigé.

Néanmoins, **nous avons décidé pour cette tâche** de transformer celui-ci en graph non-dirigé. Même si cela peut paraitre contre-intuitif pour ce type de graphe, la raison est que le calcul du nombre de composants dans un graph dirigé peut se faire de deux façons :

1. **Composants faiblement connexes** : Un graph orienté est faiblement connexe s'il y a un chemin entre n'importe quelle paire de sommets dans le graph si l'on ne considère plus l'orientation des edges.
2. **Composants fortement connexes** : Un graph orienté est fortement connexe s'il y a un chemin entre n'importe quelle paire de sommets dans le graph en considérant l'orientation des edges.

Ce que nous devons implémenter dans le cadre de cette tâche représente le comptage des composants faiblement connexes, avec cela, on peut partir de la définition et transformer notre graph en graph non dirigé.

2.1.2 Résultats

Une fois notre graph construit, nous pouvons commencer par compter le nombre de noeuds et d'edges :

➔ Nodes : 149 077
➔ Edges : 129 491

Ensuite, il nous est demandé de compter le nombre de composants, de bridges et de local bridges :

➔ Components : 19 593
➔ Bridges : 129 460
➔ Local bridges : 129 485

2.1.3 Intérprétation

De ces chiffres, nous pouvons facilement en déduire qu'il y a très peu de connexions entre les différents designers. En effet, en calculant le nombre moyen d'edges par noeud avec via la formule $\frac{E}{V}$ avec E le nombre d'edges et V le nombre de noeuds, nous arrivons à un résultat de ≈**1.15 edges par nodes**.

Nous pouvons également voir qu'il n'y a que 6 edges qui ne sont ni des bridges ou des local bridges ce qui est cohéhent avec le taux très faible d'edges moyen par designers.

2.2 Implémentation

Pour notre implémentation du graph, nous avons décidé de le représenter sous forme d'une HashMap<Integer,HashSet<Integer», cette structure de données offre une grande flexibilité dans la création du graph car elle ne nécessite pas une taille initiale comme par exemple un tableau, et elle permet de gérer parfaitement le principe de dictionnaire dont nous avons besoin pour accéder à une liste de voisins d'un noeud en particulier pour le graph.

```
HashMap<Integer, HashSet<Integer>> graph = createGraph();

public static HashMap<Integer, HashSet<Integer>> createGraph(){
    String RESOURCES_PATH = "src/main/resources/";
    String FOLLOWERS_FILE = RESOURCES_PATH + "followers.csv";
    HashMap<Integer, HashSet<Integer>> graph = new HashMap<>();
    try {
        CSVReader reader = new CSVReader(new FileReader(FOLLOWERS_FILE));
        reader.skip(1);

        String[] line = reader.readNext();
        while(line != null){
            int follower = Integer.parseInt(line[0]);
            int followed = Integer.parseInt(line[1]);

            if (!graph.containsKey(follower)) {
                graph.put(follower, new HashSet<>());
            }

            if (!graph.containsKey(followed)) {
                graph.put(followed, new HashSet<>());
            }

            graph.get(follower).add(followed);
            graph.get(followed).add(follower);
            line = reader.readNext();
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return graph;
}
```

Une fois le graph créé, on peut calculer les différentes informations dont nous avons besoin. Nous avons donc créé des méthodes adaptées et qui renvoient les différents résultats.

```
int nodes = graph.size();
int edges = countEdges(graph);
int components = countComponents(graph);
```

```
static int bridges = 0; bridges(graph);
int localBridges = localBridges(graph);
```

Pour compter les différents edges, on itère simplement sur toutes les clés du graph et on ajoute la taille de chaque set à un compteur.

```
public static int countEdges(HashMap<Integer, HashSet<Integer>> graph){
    int count = 0;
    for (int v : graph.keySet()) {
        count += graph.get(v).size();
    }
    return count;
}
```

Pour compter les différents composants, on crée une méthode qui itère sur chaque clé du graph et qui pour chaque clé du graph si elle n'a pas encore été visitée lance un DFS (Depth First Search), ce DFS va parcourir le graph et marquer tous les noeuds qu'il rencontre. Cela aura pour conséquence de marquer tous les noeuds d'un même composant. Cela veut dire que le nombre de fois que l'on lance le DFS est égal au nombre de composants du graph.

```
public static int countComponents(HashMap<Integer, HashSet<Integer>> graph) {
    int count = 0;
    marked = new HashSet<>();
    for (int v : graph.keySet()) {
        if (!marked.contains(v) && !graph.get(v).isEmpty()){
            dfs(graph, v);
            count++;
        }
    }
    return count;
}
public static void dfs(HashMap<Integer, HashSet<Integer>> graph, int src){
    marked.add(src);
    if (graph.get(src) != null){
        for (int v : graph.get(src)) {
            if (!marked.contains(v)) {
                dfs(graph, v);
            }
        }
    }
}
```

Nous effectuons une traversée DFS du graphe donné. Dans un arbre DFS, une arête (u, v) (u est le parent de v dans l'arbre DFS) est un pont s'il n'existe pas d'autre alternative pour atteindre u ou un ancêtre de u à partir du sous-arbre enraciné avec v. La valeur low[v] indique le sommet visité le plus tôt atteignable à partir du sous-arbre enraciné avec v. La condition pour qu'une arête (u, v) soit un pont est "low[v] > disc[u]".

```
public static void bridges(HashMap<Integer, HashSet<Integer>> graph)
{
    HashSet<Integer> visited = new HashSet<>();
    HashMap<Integer, Integer> disc = new HashMap<>();
    HashMap<Integer, Integer> low = new HashMap<>();
    HashMap<Integer, Integer> parent = new HashMap<>();

    for (int v : graph.keySet()) {
        parent.put(v, NIL);
    }
```

```
    for (int v : graph.keySet()) {
        if (!visited.contains(v)){
            bridgesUtil(graph, v, visited, disc, low, parent);
        }
    }
}
public static void bridgesUtil(HashMap<Integer, HashSet<Integer>> graph, int u, HashSet<Integer> visited,
                               HashMap<Integer, Integer> disc, HashMap<Integer, Integer> low,
                               HashMap<Integer, Integer> parent)
{
    time++;
    visited.add(u);
    disc.put(u, time);
    low.put(u, time);

    for (int v : graph.get(u)) {
        if (!visited.contains(v)) {
            parent.put(v, u);
            bridgesUtil(graph, v, visited, disc, low, parent);

            low.put(u, Math.min(low.get(u), low.get(v)));

            if (low.get(v) > disc.get(u)) {
                count++;
            }
        }
        else if (v != parent.get(u)) {
            low.put(u, Math.min(low.get(u), disc.get(v)));
        }
    }
}
```

Pour compter le nombre de local bridges, on va simplement compter toutes les paires de voisins qui n'ont pas d'amis en commun. Encore une fois, grâce à l'implémentation de HashSet, cette méthode s'effectue très simplement et avec une excellente complexité temporelle.

```
public static int localBridges(HashMap<Integer, HashSet<Integer>> graph){
    int count = 0;
    for (int v : graph.keySet()) {
        for (int w : graph.get(v)) {
            if (compareNeighbours(graph.get(v), graph.get(w))){
                count++;
            }
        }
    }
    return count/2;
}

public static boolean compareNeighbours(HashSet<Integer> a, HashSet<Integer> b){
    for (int x : a) {
        if (b.contains(x)){
            return false;
        }
    }
    return true;
}
```

3 Tâche 2

3.1 Analyse & résulats

3.1.1 Analyse

Dans la deuxième tâche, il nous est demandé de déterminer **les Designers les plus influents** du plus grand composant du réseau social. Pour ce faire, nous avons extrait le plus grand composant du graphe et nous avons implémenté l'algorithme de PageRank que nous avons appliqué sur ce sous-graph. L'algorithme de PageRank peut être implémenté de façon à éviter les noeuds entonnoirs qui absorberaient l'entièreté des scores du graph, cette façon s'illustre par un système d'évaporation.

Nous avons calculé que ce sous-graphe est composé de 23 381 noeuds et 23 388 edges.

3.1.2 Résultats

Nous avons décidé d'implémenter les deux façons et d'imprimer les résultats, la première sans évaporation et la seconde avec évaporation :

Les 20 designers les plus influents selon PageRank sans évaporation						
Top	Designer	Score PageRank	Shots	Likes	Moyenne	Localisation
1	57915	0,00025856766	0	0	0	Munich
2	181406	0,00023550852	0	0	0	Miami
3	320575	0,00022477449	0	0	0	Toronto
4	70408	0,00021775829	0	0	0	Shanghai
5	99199	0,00021427352	0	0	0	Poland
6	446729	0,00021398603	0	0	0	Moscow
7	195269	0,00019603880	0	0	0	Prague
8	626813	0,00019258098	0	0	0	Wisconsin
9	159858	0,00019251167	0	0	0	null
10	577047	0,00018536748	0	0	0	sweden
11	53229	0,00017598456	0	0	0	Bulgaria
12	71026	0,00017333514	0	0	0	Los Angeles
13	65955	0,00017261092	0	0	0	San Francisco
14	44717	0,00017232966	0	0	0	null
15	312654	0,00017194980	0	0	0	San Francisco
16	179980	0,00017177277	0	0	0	China
17	284417	0,00017174783	0	0	0	Southsea / London UK
18	170142	0,00017141138	0	0	0	Chengdu
19	67857	0,00017131311	0	0	0	hangzhou
20	172022	0,00017128835	0	0	0	San Francisco

Les 20 designers les plus influents selon PageRank avec évaporation						
Top	Designer	Score PageRank	Shots	Likes	Moyenne	Localisation
1	57915	0,00020143710	0	0	0	Munich
2	181406	0,00018792004	0	0	0	Miami
3	320575	0,00018158415	0	0	0	Toronto
4	70408	0,00017745353	0	0	0	Shanghai
5	99199	0,00017539720	0	0	0	Poland
6	446729	0,00017522655	0	0	0	Moscow
7	195269	0,00016322618	0	0	0	Prague
8	626813	0,00016118390	0	0	0	Wisconsin
9	159858	0,00016114334	0	0	0	null
10	577047	0,00015645732	0	0	0	sweden
11	53229	0,00015029369	0	0	0	Bulgaria
12	71026	0,00014856506	0	0	0	Los Angeles
13	44717	0,00014788848	0	0	0	null
14	312654	0,00014765550	0	0	0	San Francisco
15	179980	0,00014753941	0	0	0	China
16	284417	0,00014752329	0	0	0	Southsea / London UK
17	170142	0,00014730322	0	0	0	Chengdu
18	67857	0,00014723639	0	0	0	hangzhou
19	172022	0,00014722101	0	0	0	San Francisco
20	108590	0,00014716543	0	0	0	Bordeaux

3.1.3 Interprétation

On peut constater que **les scores PageRank sont très faibles**, et cela peut s'expliquer simplement par le fait que les Designers se suivent très peu entre eux, on le constate par les informations générales du composant, en effet, le nombre d'edges par noeud est de ≈ **1.00**. Ce qui veut dire qu'un Designer en suivra en moyenne un autre, en partant de ce principe, des designers qui parviennent à avoir plus de 1 follower se retrouveraient dans la moyenne haute du classement, alors que ce nombre de follower est pourtant très faible.

- ➔ C'est ainsi que l'on remarque que les 20 premiers designers du classement **n'ont posté aucune publication** et se retrouvent pourtant en tête du classement, ces personnes pourraient être des Designers déjà réputés hors de Dribbble et ils n'ont donc pas besoin d'être actifs pour attirer des followers.
- ➔ Une autre possibilité est que **l'on observe un effet de cascade**, c'est à dire qu'une personne qui parviendrait à obtenir quelques followers et dépasser la moyenne par chance, parviendrait à être plus visible sur la plateforme et attirer plus de monde sur son profil. On observe ce phénomène très fréquemment sur la plateforme YouTube, qui fonctionne par référencement des vidéos et chaines qui sont "tendances". Un autre exemple est celui du restaurant, si un client doit choisir entre 2 restaurants qui lui semblent au même niveau, il se dirigera vers celui qui attire le plus de monde, on peut faire le parallèle avec Dribbble, un utilisateur qui voit passer 2 profils vides dont un avec un nombre de followers supérieur à la moyenne et l'autre sans follower, préfèrera choisir celui avec un grand nombre de followers car il pensera intrinsèquement que la personne est suivie pour une raison.

3.2 Implémentation

Pour cette partie, la première grande étape consiste à trouver le plus grand composant du graph. Pour ce faire, on modifie légèrement les méthodes que l'on utilisait pour compter le nombre de composants afin de les réutiliser pour trouver le plus grand composant, pour ça on ajoute les noeuds des composants successivement dans le HashSet "largestCCMarked", que l'on modifie à chaque fois que l'on trouve un composant plus grand, à la fin du parcours sur tout le graph, le plus grand composant se retrouve dans ce HashSet, et donc on reconstruit le graph à partir du fichier "followers.csv" en ne prenant que les noeuds qui sont contenus dans ce HashSet.

```
static HashSet<Integer> marked = new HashSet();
static HashSet<Integer> lastMarked = new HashSet();

public static HashMap<Integer, HashSet<Integer>> largestComponent(
                HashMap<Integer, HashSet<Integer>> graph){

    marked = new HashSet<>();
    HashSet<Integer> largestCCMarked = new HashSet<>();

    int max = Integer.MIN_VALUE;
    for (int v : graph.keySet()) {
        if (!marked.contains(v) && !graph.get(v).isEmpty()) {
            lastMarked.clear();
            dfsLargest(graph, v);

            if(lastMarked.size() > max){
                max = lastMarked.size();
                largestCCMarked = (HashSet<Integer>) lastMarked.clone();
            }
        }
    }
    return createGraphWithMarked(largestCCMarked);
}
public static void dfsLargest(HashMap<Integer, HashSet<Integer>> graph, int src){
    lastMarked.add(src);
    marked.add(src);
    if (graph.get(src) != null){
        for (int v : graph.get(src)) {
            if (!marked.contains(v)) {
                dfsLargest(graph, v);
            }
        }
    }
}
```

Cette méthode est simplement une méthode utilitaire qui va nous servir dans l'algorithme de PageRank lorsque l'on aura besoin des "in nodes" d'un noeud, en renversant le graph, les "in nodes" deviennent des "out nodes", ce qui fait qu'on peut les récupérer très facilement dans le graph en cherchant le noeud dans la HashMap, ce qui nous renvoie le HashSet de "in nodes".

```
public static HashMap<Integer,HashSet<Integer>> reverseGraph(
                HashMap<Integer,HashSet<Integer>> graph){

    HashMap<Integer,HashSet<Integer>> reverse = new HashMap<>();
    for (int v : graph.keySet()) {
        reverse.put(v, new HashSet<>());
    }
```

```
    for(int v: graph.keySet()){
        for(int w: graph.get(v)){
            if (reverse.containsKey(w)){
                reverse.get(w).add(v);
            }
        }
    }
    return reverse;
}
```

On a maintenant tout pour calculer les scores PageRank de tous les noeuds du plus grand composant du graph. Pour ça on implémente simplement l'algorithme tel qu'il est connu, tous les noeuds commencent avec une fraction du fluide réparti sur tout le graph entre les noeuds. On fait circuler ce fluide aux enfants de chaque noeuds en divisant équitablement entre les enfants, et lorsqu'un noeud n'a pas d'enfant il garde son fluide. On a également ajouté l'évaporation en option, c'est à dire qu'à chaque itération, une partie du fluide est redistribuée à tous les noeuds du graph. Cela permet d'éviter les noeuds entonnoirs ou les cycles qui concentrent tout le fluide de par leur position avantageuse.

```
HashMap<Integer, Float> pageRank = computePageRank(largestComponent(graph), 100, 0.85f);
List<Integer> top = getHighestValues(20, (HashMap<Integer, Float>) pageRank.clone());

public static HashMap<Integer, Float> computePageRank(HashMap<Integer,HashSet<Integer>> graph,
                                                      int k, float alpha){
    float[] pageRank = new float[7000000];
    Float[] floatsPageRank = new Float[7000000];

    HashMap<Integer,Float> pageRankValues = new HashMap<>();
    HashMap<Edge,Float> edgePageRankValues = new HashMap<>();
    HashMap<Integer,HashSet<Integer>> reverseGraph = reverseGraph(graph);

    for (int v : graph.keySet()) {
        pageRankValues.put(v, 1.f/graph.size());
        pageRank[v] = 1.f/graph.size();

        for(int w:graph.get(v)){
            edgePageRankValues.put(new Edge(v,w), 1.f/graph.size());
        }
    }

    int out;
    float fluidOut,fluidIn;
    Edge edge;

    for (int i = 0; i < k; i++) {
        for(int v:graph.keySet()){
            out = graph.get(v).size();

            fluidOut = pageRankValues.get(v) / out;
            for(int outgoing :graph.get(v)){
                edge = new Edge(v,outgoing);
                edgePageRankValues.put(edge,fluidOut);
            }
        }

        for(int v:graph.keySet()){
```

```
            fluidIn = 0;

            if(graph.get(v).size() == 0){
                fluidIn = pageRank[v];
            }
            else {
                for (int ingoing : reverseGraph.get(v)) {
                    edge = new Edge(ingoing, v);
                    fluidIn += edgePageRankValues.get(edge);
                }
            }

            fluidIn  = alpha * fluidIn + ((1.f - alpha) / graph.size() );
            pageRank[v] = fluidIn;
            pageRankValues.put(v,fluidIn);

        }
    }

    int count = 0;
    for (float pr:pageRank) {
        floatsPageRank[count++] = new Float(pr);
    }

    return pageRankValues;
}
```

Cette méthode permet simplement de récupérer les meilleurs Designers du graph après que l'on ait calculé le score PageRank de tous les noeuds.

```
public static List<Integer> getHighestValues(int top,HashMap<Integer,Float> values){
    List<Integer> maxs = new ArrayList<>();
    float max;
    int indexMax;
    for (int i = 0; i < top; i++) {
        max = 0f;
        indexMax = 0;
        for (int v:values.keySet()) {
            if(values.get(v) > max){
                max = values.get(v);
                indexMax = v;
            }
        }
        maxs.add(new Integer(indexMax));
        values.remove(indexMax);
    }
    return maxs;
}
```

4 Tâche 3

4.1 Analyse & résulats

4.1.1 Analyse

Dans cette partie, il nous est demandé de trouver les Triadic closure qui se produisent au fur et à mesure de la création du graph dans le temps. Pour cela, nous devons nous servir des TimeStamp fournis avec les edges qui permettent de signaler la date à laquelle a eu lieu l'action du Follow. Ce graph est un graph dirigé.

1. Pour la première partie, nous avons créé un algorithme qui permet de compter le nombre de Triadic Closure, cet algorithme sera détaillé précisément dans la partie implémentation.
2. Pour cette deuxième partie, nous avons simplement appliqué un filtre sur les résultats du premier algorithme, tel que demandé dans l'énoncé, nous prenons les Triadic Closure renvoyés par la première méthode, et nous vérifions si les Designers appartenant au Triadic Closure habitent dans la même ville.
3. Pour cette partie, il faut identifier les Triadic Closures qui se créent au fur et à mesure des mois, pour ce faire on a simplement à trier nos Triadic Closures par mois d'apparition. Ensuite on trie tous les Triadic Closures par ville.
4. Pour cette dernière partie, on doit calculer la moyenne de temps d'apparition des Triadic Closures en fonction des villes.

4.1.2 Résultats

Les résultats indiquent qu'il n'y a pas de Triadic Closure qui se forment durant toute la création du graph dirigé, et ce même en ne filtrant pas sur base des villes. Nous avons pu tester notre algorithme sur un graph de test et celui-ci semblait renvoyer les bons résultats.

4.1.3 Interprétation

Le fait qu'il n'y ait pas de Triadic Closures dans ce graph est cohérent, nous avons vu plus tôt dans l'analyse du sous-graph pour l'implémentation de PageRank, que le nombre d'edges par noeuds étaient de ≈ 1.00. Cela étant, il est très difficile de pouvoir bâtir des Triadic Closures avec si peu de connexions, ceux-ci nécessitent généralement que par exemple dans une amitié, un des deux amis soit lié à un autre ami en plus de celui concerné, afin de pouvoir espérer former un Triadic Closure.

4.2 Implémentation

Afin de calculer les triadic closures qui se sont réalisées à travers les années, nous commencons par calculer la plus petite année où un follow a eu lieu afin de pouvoir itérer depuis cette année à 2021. Ensuite, pour chaque année nous parcourons le graph à la recherche de closure probable dans le futur qui se sont créées à l'année courante.

Nous récupérons toutes ces probables closures dans une hashmap indêxée par l'année où la closure peut être réalisée. Exemple : A et B sont connectés et C se connecte à B en 2017, notre hashmap contiendra une ProbableTriadicClosure A,B,C au timestamp du edge B,C dans sa liste de probable closures à l'index 2017.

Ensuite nous itérons pour chaque année sur toutes les ProbableTriadicClosure afin de savoir si certaines ont été réalisées à une date ultérieure. Nous en profitons également pour mettre à jour le timestamp où la closure s'est réalisée ainsi que pour filtrer les closures qui comprennent trois designers de la même ville comme demandé dans la tâche 4.2.

En premier lieu, voici la méthode qui permet d'obtenir les ProbableTriadicClosure à une année donnée entre un follower et son followed.

```
public static HashSet<ProbableTriadicClosure> getProbableClosure(HashMap<Integer,
                HashSet<Follow>> graph,
                int follower,
                int followed, int year){

    HashSet<ProbableTriadicClosure> closures = new HashSet<>();
    HashSet<Follow> followsFromFollower = graph.get(follower);
    HashSet<Follow> followsFromFollowed = graph.get(followed);

    ProbableTriadicClosure closure;
    for(Follow follow:followsFromFollower){
        for(Follow f:followsFromFollowed){
            if(graph.get(f.getFollowed()) != null &&
                    !graph.get(f.getFollowed()).contains(follow.getFollower())){

                if(f.getTimeStamp() >= year && f.getTimeStamp() < year + YEAR_IN_SECONDS) {
                    closure = new ProbableTriadicClosure(
                    follow.getFollower(), follow.getFollowed(),
                    f.getFollower(),
                    f.getTimeStamp());
                    closures.add(closure);
                }

            }
        }
    }

    return closures;
}
```

Vous trouverez donc ci-dessous l'algorithme tel qu'expliqué ci-dessus.

```
public static List<ProbableTriadicClosure> computeTriadicClosure(HashMap<Integer,HashSet<Integer>> G){
    HashMap<Integer, HashSet<Follow>> graph = createFollowGraph(getLargestNodes(G));


    HashMap<Integer,HashSet<ProbableTriadicClosure>> probableOverYears = new HashMap<>();
    HashSet<ProbableTriadicClosure> probables = new HashSet<>();

    int minYear = (getMinYear(graph) / YEAR_IN_SECONDS) * YEAR_IN_SECONDS;
    int year = 1975 + (minYear/YEAR_IN_SECONDS);
    for(int curYear = year; curYear <= 2021 ; curYear++){

        for(int v: graph.keySet()){
            if(graph.get(v).size() >= 2){

                for(Follow follow: graph.get(v)){
                    if(graph.get(follow.getFollowed()) != null
                    && graph.get(follow.getFollowed()).size() >= 1) {
                        probables.addAll(
                                getProbableClosure(graph, v, follow.getFollowed(), curYear)
                        );
                    }
                }
```

```
            }
        }
        probableOverYears.put(curYear,probables);

    }

    List<ProbableTriadicClosure> closuresRealised = new ArrayList<>();

    for(int y :probableOverYears.keySet()){

        for(ProbableTriadicClosure closure: probableOverYears.get(y)) {
            for (Follow follow : graph.get(closure.getDesigner())){

                if (follow.getFollowed() == closure.getProbableFollowed()) {
                    if(areInSameTown(closure)){
                        closure.setTimeStampRealised(follow.getTimeStamp());
                        closuresRealised.add(closure);
                    }
                }

            }
        }

    }

    return closuresRealised;
}
```

5 Conclusion

Ce projet très intéressant nous a permis d'utiliser les algorithmes que nous avons vu en cours sur des données réelles tout en essayant de tirer des conclusions des différents résultats que nous obtenions. Malgré le fait que le Dataset ne permettait pas de visualiser toutes les situations attendues, nous avons pu avoir une vision macroscopique des réseaux sociaux et de l'influence que peuvent avoir des personnes au sein d'un réseau social.