1 Objectif

Fouille du web (web mining), analyse des réseaux sociaux. Détection des communautés avec Python. Utilisation du package igraph.

La détection de communautés dans les réseaux sociaux a pour objectif d'identifier les groupes d'individus entretenant des relations privilégiées. Ce thème connaît une recrudescence d'intérêt ces dernières années avec le développement des médiaux sociaux (Twitter, Facebook, etc.), multipliant les opportunités d'interactions entre les individus. Un réseau social est souvent représenté par un graphe où les sommets (nœuds) représentent les individus, les liens qu'il entretiennent sont matérialisés par les arêtes. Une communauté correspond à un groupe de nœuds présentant une forte densité de connexions.

Ce tutoriel vient en complément de mon support de cours accessible en ligne (COURS) qui nous servira de référence. Nous nous plaçons dans une situation particulière où le graphe est non orienté, les liaisons entre les individus – lorsqu'elles existent – sont symétriques et non pondérées c.-à-d. les connexions ont tous la même intensité.

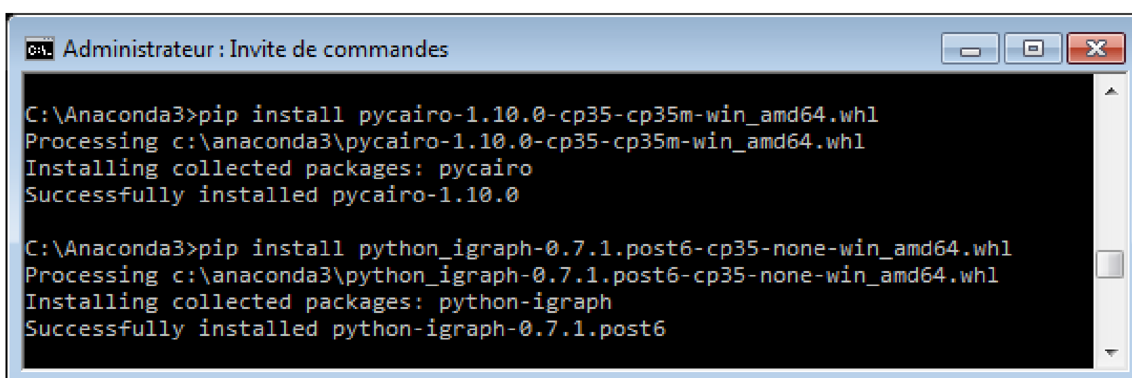
Nous travaillerons sous **Python** et nous utiliserons le package igraph.

2 Installation du package igraph

L'installation du package **igraph** semble a priori facile à l'aide de l'utilitaire **pip** (http://igraph.org/python/). En réalité j'ai rencontré de nombreux problèmes qui m'ont dissuadé de l'utiliser pour mes enseignements à l'Université. En effet, la durée d'une séance de TD (travaux dirigé) est limitée. Il est hors de question de mettre les étudiants aux prises avec une installation récalcitrante empiétant sur le temps que l'on pourrait consacrer à l'apprentissage des techniques d'analyse des réseaux sociaux. Pour mon cours, je me suis donc tourné vers R et le package associé (http://igraph.org/r/), qui est plus facile à utiliser, et surtout qui ne nécessite pas de préparations tarabiscotées pour être opérationnel.

Je n'ai pas cette contrainte pour les tutoriels où je dispose de plus de latitude pour explorer de nouveaux outils. En l'occurrence, s'agissant de **igraph** pour Python, j'ai dû procéder à une série de manipulations avant de pouvoir disposer d'un environnement pleinement exploitable.

- J'utilise à ce jour (10/04/2017) la distribution Anaconda 4.3.1 basée sur **Python 3.6**. Malgré tous mes efforts, je n'ai pas sur installer et faire fonctionner le package **igraph**. Je me suis donc tourné vers Anaconda 4.0.0 parce qu'elle est basée sur **Python 3.5**.
- Il m'a dès lors été possible d'utiliser les packages compilées pour Windows accessibles sur le site de Christoph Gohlke (http://www.lfd.uci.edu/~gohlke/pythonlibs/). Les deux librairies sont à utiliser conjointement, les versions sont spécifiques à Python 3.5[1] :
  - Pycairo nécessaire aux représentations graphiques des graphes (pycairo-1.10.0-cp35-cp35m-win_amd64.whl) ;
  - Igraph (python_igraph-0.7.1.post6-cp35-none-win_amd64.whl).
- Pour installer ces packages, il faut ouvrir le shell DOS en mode administrateur et invoquer l'utilitaire **pip** avec la commande « **pip install** *nom du package* ».



Il ne reste plus qu'à lancer notre outil de travail favori. J'utilise l'EDI (environnement de développement intégré) Spyder en ce qui me concerne. Il est installé automatiquement avec la distribution Anaconda.

3 Données « Club de karaté de Zachary »

Le « Club de Karaté de Zachary » est composé de 34 membres. Il s'agit d'un exemple de réseau social bien connu (https://en.wikipedia.org/wiki/Zachary's_karate_club). Un conflit entre l'administrateur et l'entraîneur (ouh là là, ça rappelle certains clubs de foot ça…) a abouti à une scission de la structure : une partie des membres a suivi l'entraîneur (ah non, on n'a pas ça dans le foot, on devrait…) ; tandis que dans l'autre, certains ont trouvé un nouvel instructeur, d'autres ont abandonné le Karaté (voilà où peut mener l'impéritie des dirigeants…).

[1] Pycairo est disponible pour Python 3.6, mais pas igraph (10/04/2017)…

Le réseau est décrit par une **matrice d'adjacence** booléenne (valeurs 1/0) symétrique indiquant les relations privilégiées (ou non) qu'entretiennent les membres.

Pers	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12	p13	p14	p15	p16	p17	p18	p19	p20	p21	p22	p23	p24	p25	p26	p27	p28	p29	p30	p31	p32	p33	p34
p1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0
p2	1	0	1	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0
p3	1	1	0	1	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0
p4	1	1	1	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p5	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p6	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p7	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p8	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p9	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
p10	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
p11	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p13	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p14	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
p15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
p16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
p17	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p18	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
p20	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
p21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
p22	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
p24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1
p25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0
p26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0
p27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
p28	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
p29	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
p30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1
p31	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
p32	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	1
p33	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	0	1
p34	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1	0	1	1	0	0	1	1	1	1	1	1	1	0

Les noms des individus ont été anonymisés pour corser l'affaire.

Dans un premier temps, notre tâche consistera à identifier les deux leaders en utilisant la notion de centralité. Dans un deuxième temps, nous essayerons de faire ressortir les communautés issues de la partition du club.

4 Importation des Données et représentation du graphe social

4.1 Importation et vérification des données

Nous importons les données dans une structure DataFrame (librairie Pandas) à l'aide de la commande read_table().

```
#changement de répertoire
import os
os.chdir("C:/data")

#chargement des données – libraire pandas
#header = 0 ➔ première ligne (n°0) = étiquette des colonnes
#index_col =  0 ➔ première colonne (n°0) = étiquette des lignes
```

```
import pandas
dfAdj = pandas.read_table("karate.txt",header=0,index_col=0)
```

Nous affichons quelques informations pour vérifier les données. Nous récupérons au passage le nombre d'individus n = 34.

```
#affichage des informations
n = dfAdj.shape[0]
print(n)
print(dfAdj.index)
```

.index contient les étiquettes des individus.

```
>>> print(dfAdj.index)
Index(['p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8', 'p9', 'p10', 'p11',
       'p12', 'p13', 'p14', 'p15', 'p16', 'p17', 'p18', 'p19', 'p20', 'p21',
       'p22', 'p23', 'p24', 'p25', 'p26', 'p27', 'p28', 'p29', 'p30', 'p31',
       'p32', 'p33', 'p34'],
      dtype='object', name='Personnes')
```

Nous transformons la structure DataFrame (librairie pandas) en une matrice de type numpy.

```
#transformer en matrice
MAdj = dfAdj.as_matrix()
print(MAdj)
```

Toutes les valeurs ne sont pas affichées. Nous constatons néanmoins que nous avons bien une matrice.

```
>>> print(MAdj)
[[0 1 1 ..., 1 0 0]
 [1 0 1 ..., 0 0 0]
 [1 1 0 ..., 0 1 0]
 ...,
 [1 0 0 ..., 0 1 1]
 [0 0 1 ..., 1 0 1]
 [0 0 0 ..., 1 1 0]]
```

4.2 Construction du graphe et représentation graphique

Nous importons la librairie igraph, nous vérifions le numéro de version.

```
#igraph
import igraph
print(igraph.__version__)
```

Nous disposons de la **version 0.7.1** dans ce tutoriel. Il faudra vérifier ce numéro si vous tentez de reproduire les calculs et que vous obtenez des résultats incohérents.

Nous créons le graphe à partir de la matrice d'adjacence, il est non-orienté.

```
#création du graphe
g = igraph.Graph.Adjacency(MAdj.tolist(),mode=igraph.ADJ_UNDIRECTED)
print(g)
```

Python affiche les p = 78 connexions existantes entre les n = 34 sommets (n° 0 à 33).

```
>>> print(g)
IGRAPH U--- 34 78 --
+ edges:
 0 --  1  2  3  4  5  6  7  8 10 11 12 13 17 19 21 31      25 -- 23 24 31
 1 --  0  2  3  7 13 17 19 21 30                           26 -- 29 33
 2 --  0  1  3  7  8  9 13 27 28 32                        27 --  2 23 24 33
 3 --  0  1  2  7 12 13                                    28 --  2 31 33
 4 --  0  6 10                                             29 -- 23 26 32 33
 5 --  0  6 10 16                                          30 --  1  8 32 33
 6 --  0  4  5 16                                          31 --  0 24 25 28 32 33
 7 --  0  1  2  3                                           32 --  2  8 14 15 18 20 22 23 29
30 31 33
 8 --  0  2 30 32 33                                       33 --  8  9 13 14
15 18 19 20 22 23 26 27 28 29 30 31 32
 9 --  2 33
10 --  0  4  5
11 --  0
12 --  0  3
13 --  0  1  2  3 33
14 -- 32 33
15 -- 32 33
16 --  5  6
17 --  0  1
18 -- 32 33
19 --  0  1 33
20 -- 32 33
21 --  0  1
22 -- 32 33
23 -- 25 27 29 32 33
24 -- 25 27 31
```

Par exemple, l'individu n°10 (correspondant à l'étiquette p11) est relié aux individus n° 0, 4 et 5 (en rouge dans la sortie Python ci-dessus). Pour faciliter les manipulations, nous pouvons attribuer des noms aux sommets du graphe.

```
#attribuer des noms aux sommets
g.vs["name"] = dfAdj.index.tolist()
print(g.vs[10]["name"])
```

Nous constatons ainsi que le nom de l'individu n°10 est bien p11.

```
>>> print(g.vs[10]["name"])
p11
```

Les accès sont plus faciles et lisibles. Pour disposer du nombre de voisins de p11 par exemple, nous ferons :

```
#nombre de voisins de p11
print(g.neighborhood_size("p11"))
```

Il y en a **4** apparemment…

```
>>> print(g.neighborhood_size("p11"))
4
```

… parce qu'il se compte lui-même. En effet, si l'on affiche la liste :

```
#voisinage de p11
print(g.neighborhood("p11"))
```

Nous retrouvons l'individu n°10 dans le lot.

```
>>> print(g.neighborhood("p11"))
[10, 0, 4, 5]
```

La lecture est plus facile si nous affichons plutôt leurs noms :

```
#si on souhaite avoir leurs noms
print(g.vs[g.neighborhood("p11")]['name'])
```

On y observe p11.

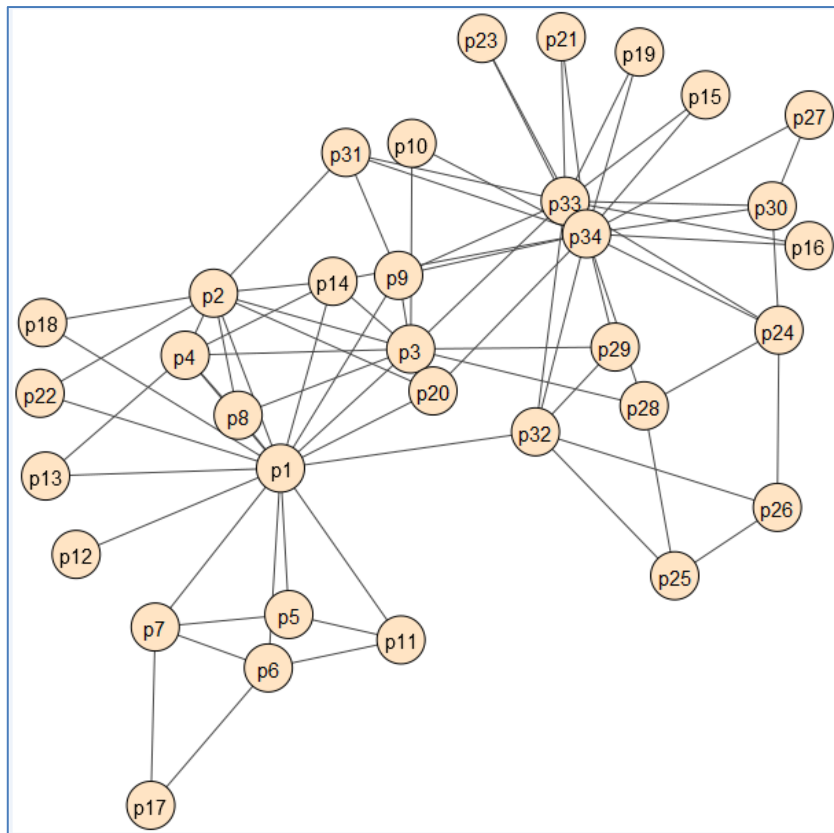
```
>>> print(g.vs[g.neighborhood("p11")]['name'])
['p11', 'p1', 'p5', 'p6']
```

4.3 Représentation graphique

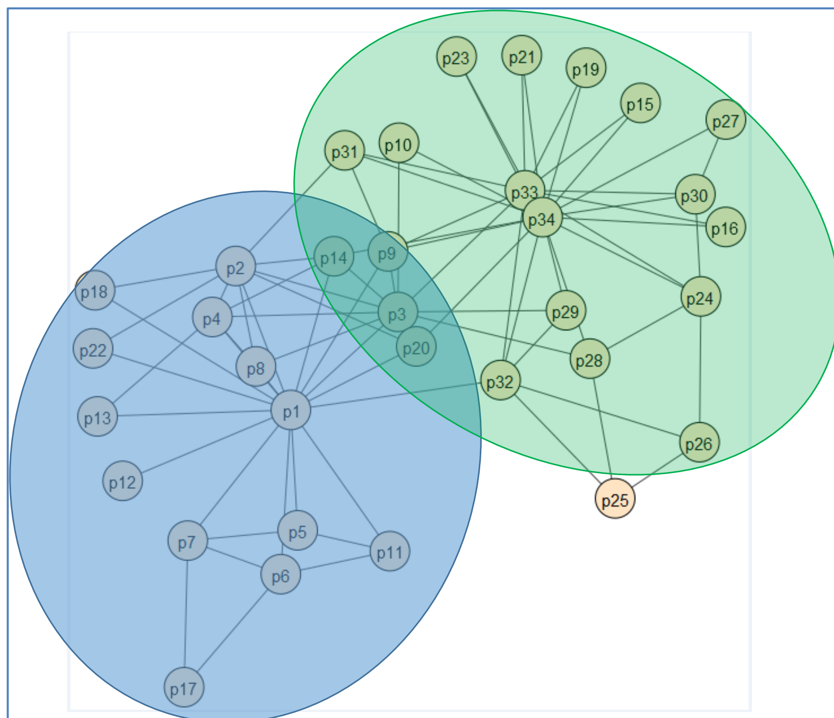
Pour rendre la représentation graphique plus lisible, nous devons tout d'abord étiqueter les sommets (dans le paragraphe précédent, nous les avions nommés) avant de lancer la commande plot(). Les options invoquées lors de l'appel de plot() sont essentiellement d'ordre cosmétique.

```
#attribuer des étiquettes aux sommets
g.vs['label'] = dfAdj.index.tolist()
print(g.vs[0]['label'])

#affichage du graphe
obj = igraph.plot(g,vertex_label_size=15,vertex_size=35,vertex_color='#ffe4c4')
obj.show()
```



Visuellement, on croit deviner les deux blocs opposés, ainsi que les individus centraux au sein d'entre eux. Nous verrons si cette intuition est confirmée par le calcul dans la section suivante.



5 Notion de centralité

5.1 Identification des nœuds centraux

La centralité indique l'importance d'un sommet dans un graphe. Elle peut être quantifié simplement par son voisinage, un nœud est central s'il a beaucoup de voisins (degré de centralité) ; elle peut l'être en termes de distance, un nœud central est peu éloigné des autres (centralité closeness) ; ou plus subtilement, il constitue un nœud de passage du plus court chemin pour transiter d'un sommet à l'autre (centralité betweenness) (COURS, page 9). Dans ce qui suit, nous mettrons en évidence les 5 individus les plus « centraux » au regard de ces différentes mesures.

5.1.1 Degré de centralité

Les informations relatives aux sommets sont accessibles à travers la propriété .vs de l'objet graphe. Ainsi, pour obtenir les degrés de centralité, nous ferons :

```
#degré de centralité
print(g.vs.degree())
```

Nous obtenons :

```
>>> print(g.vs.degree())
[16, 9, 10, 6, 3, 4, 4, 4, 5, 2, 3, 1, 2, 5, 2, 2, 2, 2, 2, 3, 2, 2, 2, 5, 3, 3, 2,
4, 3, 4, 4, 6, 12, 17]
```

La lecture est plus facile si nous affichons les valeurs les plus importantes par ordre décroissant en les associant aux noms des nœuds.

```
#affichage des 5 premiers sommets, tri de manière décroissante
valeurs = {'node_name':g.vs['name'],'degree':g.vs.degree()}
print(pandas.DataFrame.from_dict(valeurs).sort_values(by='degree',ascending=False).iloc[:5,:])
```

Les sommets 'p34', 'p1', 'p33', 'p3' et 'p2' sont mis en exergue.

```
    degree node_name
33      17       p34
0       16        p1
32      12       p33
2       10        p3
1        9        p2
```

5.1.2 Centralité closeness

Nous réalisons les mêmes opérations pour ce second indicateur.

```
#closeness centralité
```

```
valeurs = {'node_name':g.vs['name'],'closeness':g.vs.closeness()}
print(pandas.DataFrame.from_dict(valeurs).sort_values(by='closeness',ascending=False).iloc[:5,:])
```

Nous obtenons :

```
    closeness node_name
0    0.568966        p1
2    0.559322        p3
33   0.550000       p34
31   0.540984       p32
13   0.515625       p14
```

5.1.3 Centralité betweenness

Enfin, pour le 3ème indicateur :

```
#betweenness centralité
valeurs = {'node_name':g.vs['name'],'betweenness':g.vs.betweenness()}
print(pandas.DataFrame.from_dict(valeurs).sort_values(by='betweenness',ascending=False).iloc[:5,:])
```

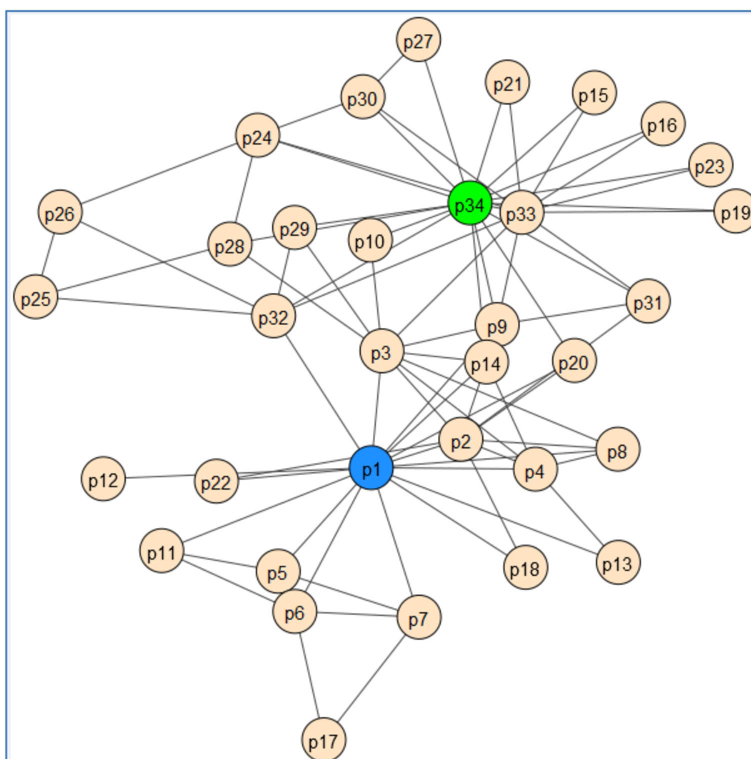
Nous observons une certaine cohérence avec les résultats précédents.

```
    betweenness node_name
0    231.071429        p1
33   160.551587       p34
32    76.690476       p33
2     75.850794        p3
31    73.009524       p32
```

Manifestement, les individus p1 et p34 sont au centre (c'est le cas de le dire) des relations entre les membres du club. Il est fort à parier qu'ils correspondent (dans le désordre) à l'administrateur et de l'entraîneur du club (c'est le cas). Nous les faisons ressortir dans le graphe social en utilisant des jeux de couleurs adéquats.

```
#graphe mettant en évidence p1 (n°0) et p34 (n°33)
g.vs['color'] = '#ffe4c4'
g.vs[0]['color'] = '#1e90ff'
g.vs[33]['color'] = 'green'
obj = igraph.plot(g,vertex_label_color='black',vertex_label_size=15,vertex_size=35)
obj.show()
```

p1 et p34 sont certes centraux, mais ils disposent de nœuds périphériques différents. On devine déjà à peu près les communautés qui émergeront lors de la partition du graphe.



5.2 Individus relais

Les problèmes surgissent souvent lorsque les personnes ne se parlent plus (dit le sage). Je me suis demandé si certains individus pouvaient jouer le rôle de relais entre ces deux leaders. Pour ce faire, affichons les plus courts chemins menant de p1 à p34 (et inversement puisque le graphe est non orienté).

```
#plus court chemin entre p1 et p34
print(g.get_all_shortest_paths('p1','p34'))
```

Plusieurs individus peuvent jouer le rôle d'intermédiaires directs entre 'p1' et 'p34'…

```
>>> print(g.get_all_shortest_paths('p1','p34'))
[[0, 31, 33], [0, 19, 33], [0, 13, 33], [0, 8, 33]]
```

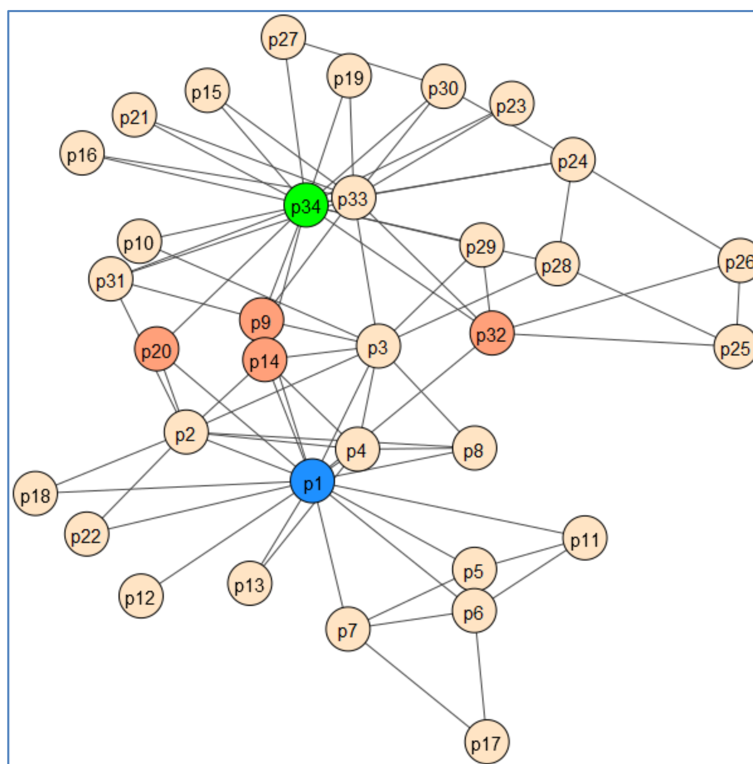
…, à savoir : les n°31, 19, 13 et 8.

Mettons-les en évidence dans le graphe :

```
#mettre en évidence les individus relais
for v in g.get_all_shortest_paths('p1','p34'):
    g.vs[v[1]]['color'] = '#ffa07a'

obj = igraph.plot(g,vertex_label_color='black',vertex_label_size=15,vertex_size=35)
obj.show()
```

Si un rapprochement est possible, ces individus peuvent avoir un rôle à jouer.



6 Découverte de communautés

Il existe plusieurs techniques de subdivision des graphes permettant de délimiter les communautés (COURS, pages 10 à 28). Dans ce tutoriel, nous nous intéressons à une approche divisive basée sur la notion de « edge betweenness » (COURS, pages 19 à 21). Nous obtenons une partition nette (*crisp*) c.-à-d. un individu appartient à un et un seul groupe. La solution a le mérite de la simplicité. Mais, tout comme en classification automatique (*clustering*), nous savons qu'elle ne correspond pas toujours à la réalité. Nous pouvons avoir des communautés chevauchantes c.-à-d. un individu peut appartenir à différentes classes à des degrés divers. Dans notre exemple en l'occurrence, la question ne se pose pas. Après l'éclatement du club, les personnes ont choisi leur camp.

6.1 Edge betweenness

L'*edge betweenness* représente la fréquence avec laquelle une connexion est empruntée lorsque l'on considère les plus courts chemins entre chaque paire de nœuds. Plus la valeur est élevée, plus la connexion est importante car on peut considérer qu'elle établit un « pont » entre des sommets voire des groupes de sommets.

Nous calculons les valeurs *edge betweenness* de chaque connexion.

```
#récupération des edge betweenness
```

```
eb = g.edge_betweenness()
print(eb)
```

On a un peu du mal à s'y retrouver…

```
[14.166666666666664, 43.638888888888886, 11.5, 29.333333333333332,
43.83333333333333, 43.833333333333336, 12.80238095238095, 41.64841269841271,
29.333333333333332, 33.0, 26.099999999999994, 23.77063492063493,
22.509523809523817, 25.770634920634926, 22.509523809523813, 71.39285714285714,
13.033333333333335, 4.333333333333333, 4.164285714285714, 6.959523809523811,
10.490476190476187, 8.20952380952381, 10.490476190476189, 18.10952380952381,
12.583333333333332, 14.145238095238092, 5.147619047619047, 17.28095238095238,
4.28095238095238, 23.10873015873016, 12.780952380952376, 38.70158730158729,
1.8880952380952383, 6.899999999999997, 8.37142857142857, 2.6666666666666665,
1.6666666666666665, 1.6666666666666667, 2.6666666666666665, 16.5, 16.5, 5.5,
17.077777777777776, 22.684920634920633, 16.614285714285714, 38.04920634920634,
13.511111111111113, 19.488888888888887, 13.511111111111113, 19.488888888888887,
13.511111111111113, 19.488888888888887, 33.31349206349207, 13.511111111111113,
19.488888888888887, 13.511111111111111, 19.488888888888887, 11.094444444444443,
5.9111111111111105, 3.7333333333333334, 12.533333333333331, 18.327777777777783,
2.3666666666666667, 10.466666666666665, 22.5, 23.59444444444445,
2.5428571428571427, 30.457142857142856, 17.097619047619048, 8.333333333333332,
13.78095238095238, 13.087301587301585, 16.72222222222222, 9.566666666666666,
15.042857142857144, 23.244444444444447, 29.95396825396826, 4.614285714285714]
```

Identifions la connexion la plus forte au sens de notre critère…

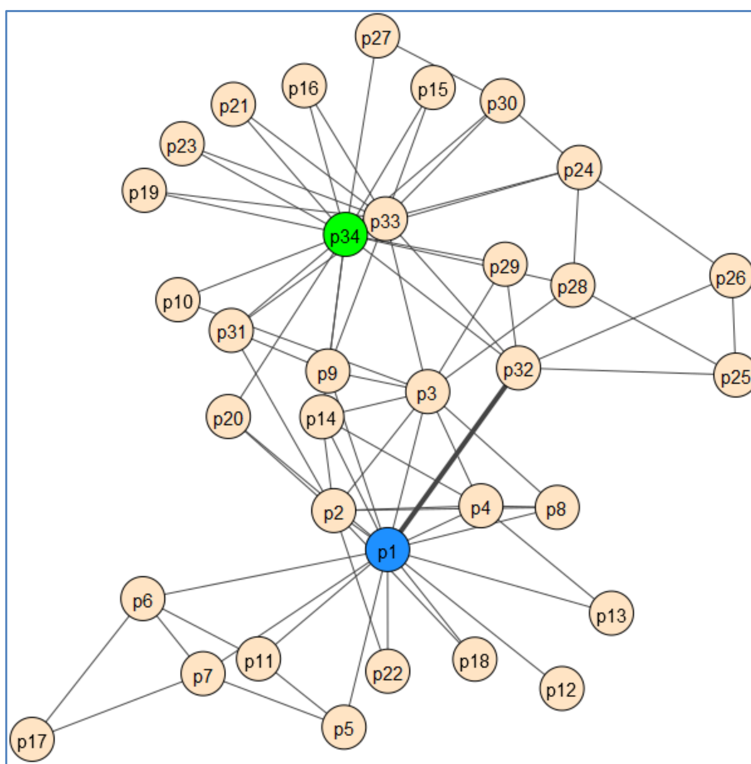
```
#transformer en vecteur numpy
import numpy
ebn = numpy.array(eb)
print(ebn)

#récupérer l'indice de la valeur maximale
iebn = numpy.argmax(ebn)
print(iebn)
```

Il s'agit de la connexion n°15 que nous pouvons souligner dans le graphe.

```
#mettre en évidence la connexion n°iebn (n°15)
g.vs['color'] = '#ffe4c4'
g.vs[0]['color'] = '#1e90ff'
g.vs[33]['color'] = 'green'
g.es['width'] = 1
g.es[iebn]['width'] = 4
obj = igraph.plot(g,vertex_label_color='black',vertex_label_size=15,vertex_size=35)
obj.show()
```

Elle relie les sommets p1 et p32.



6.2 Algorithme de construction des communautés

L'approche divisive consiste à retirer itérativement les connexions présentant les valeurs les plus élevées d'*edge betweenness*. Voici le pseudocode de l'algorithme.

```
Fonction Subdiviser(graphe)
Calculer les eb() (edge betweenness)
TANT QUE graphe ≠ singleton
  Retirer le lien avec l'eb() le plus élevé
  Si partition en g1 et g2 Alors
     Subdiviser(g1), Subdiviser(g2)
  Sinon
     Recalculer les eb()
FIN TANT QUE
```

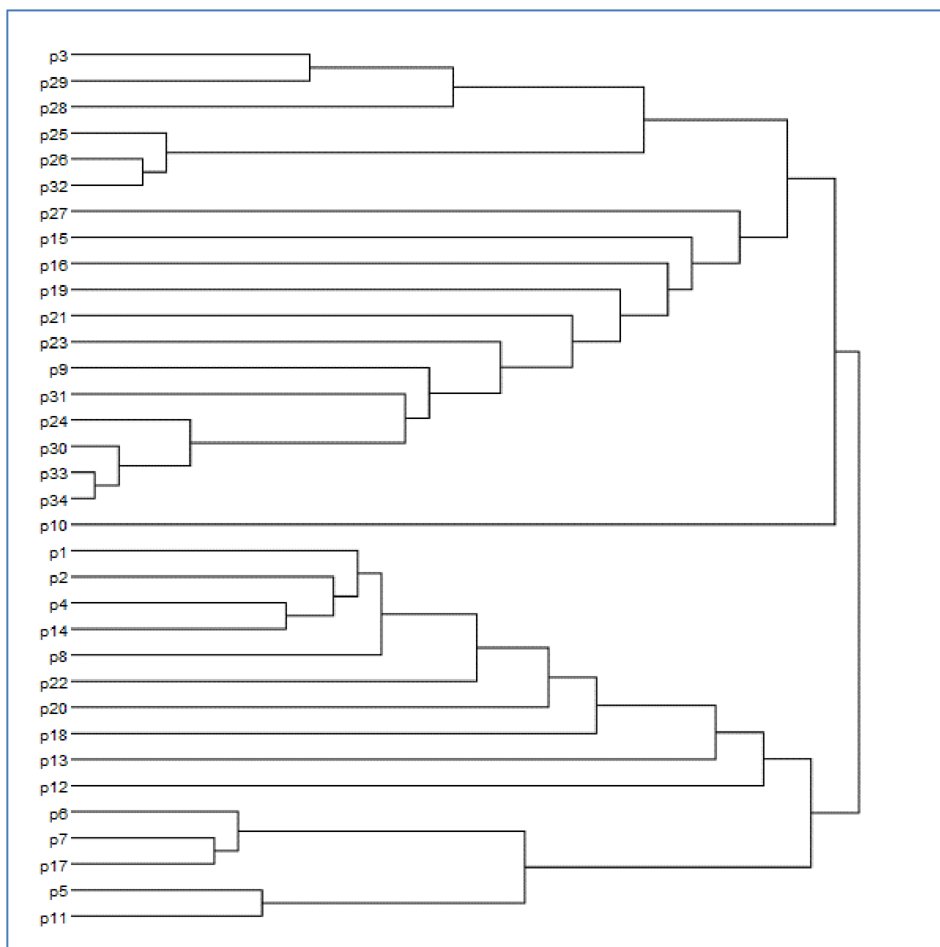
Lors du retrait d'une connexion, nous faisons face à une alternative : soit il n'a pas engendré de partition du graphe, auquel cas nous recalculons les *edge betweenness*, en effet les plus courts chemins entre les paires de sommets peuvent être modifiés, et nous recommençons ; soit il a engendré une partition en deux sous-graphes, nous travaillons alors récursivement sur les sous-groupes. Les subdivisions successives peuvent être illustrées par un dendrogramme qui part de la non-partition (un seul groupe) jusqu'à la partition triviale (un sommet = un groupe).

Voyons ce qu'il en est sur les données « Karaté ».

```
#construction de communautés avec edge betweenness
res = g.community_edge_betweenness(directed=False)

#affichage du dendrogramme
igraph.plot(res)
```

Que l'on ne s'y trompe pas, nous sommes sur une démarche divisive.



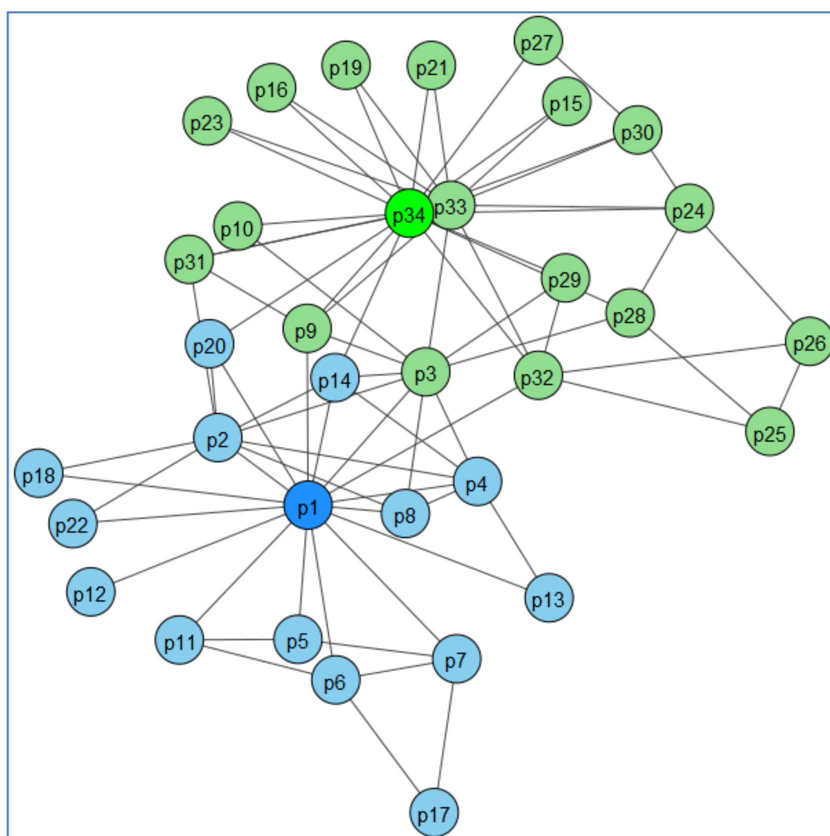
Dans une éventuelle partition en deux groupes, nous remarquons que les fauteurs de troubles que sont les individus p1 et p34 seront effectivement séparés. Nous demandons cette solution explicitement :

```
#couper en 2 groupes
sol = res.as_clustering(n=2)
print(sol)
```

Nous disposons de la liste des individus dans chaque groupe (n°0 et 1) :

```
Clustering with 34 elements and 2 clusters
[0] p1, p2, p4, p5, p6, p7, p8, p11, p12, p13, p14, p17, p18, p20, p22
[1] p3, p9, p10, p15, p16, p19, p21, p23, p24, p25, p26, p27, p28, p29, p30,
    p31, p32, p33, p34
```

Nous pouvons les faire apparaître explicitement dans le graphe :



Parmi les 4 individus qui pouvaient jouer le rôle de relais entre les deux leaders (section 5.2), deux ont pris parti pour p1 (p20 et p14), les deux autres ont choisi p34 (p9 et p32).

7 Conclusion

Dans ce tutoriel, nous n'avons montré qu'une infime partie de la détection de communautés dans les réseaux sociaux. Nous nous sommes placés dans un cadre très simple : nous travaillons à partir d'un graphe non-orienté et non-pondéré. Les extensions se devinent aisément. Dans cette optique, le package **igraph** pour Python semble disposer de ressources insoupçonnées. Pour ma part, il m'a fallu un peu (beaucoup) de temps pour savoir le manipuler correctement, et je pense être très loin d'en avoir fait le tour…

8 Références

- (**COURS**) Rakotomalala R., « Détection de communautés – Diapos », Mars 2017.
- Newman M.E.J., « Finding community structure in networks using eigenvectors of matrices », Physical Review, E 74 036104, 2006.

- Selmane S.A., « Détection et analyse de communautés dans les réseaux sociaux : approche basée sur l'analyse formelle de concepts », Thèse de Doctorat Lyon 2, 2015.
- Tang L., Liu H., « Community detection and mining in social media », Morgan and Claypool Publishers, 2010 (http://dmml.asu.edu/cdm/).
- Wikipédia, « Community structure ».