RAPPORT PROJET FINAL : GRAPHES (MODELÈS ET ALGORITHMES)

EA MAP572 2019-2020

29 novembre 2019

Achraf AZIZE

TABLE DES MATIÈRES

1 ABSTRACT

Ce document constitue un rapport sur les travaux réalisés pendant les trois dernières séances du EA MAP572, dans le cadre du projet final, concernant différents algorithmes et modèles sur les graphes.

Toutes les quatre sections du TP seront abordés, à savoir :

— Les graphes à attachement préférentiel : modèle, simulation et

— Coder un programme qui va calculer les prix pour les contrats actuels et renvoyer un fichier CSV identique à celui fournit par iVector, en moins de temps

Ce document est totalement indépendent, avec tous les codes et scripts utiles. Cependant, un notebook jupyter est joint à ce rapport, pour pouvoir executer directement les tests, et tester la performence.

2 GRAPHES PAR ATTACHEMENT PRÉFÉRENTIEL

Nous allons commencer par le graphe à attachement préférentiel, introduit par A.-L. Barabási et R.Albert pour modéliser de façon dynamique la création de grands réseaux d'interactions.

2.1 Le Modèle

Soit $n \geq 1$ un entier, nous allons construire de façon dynamique un graphe aléatoire G_n à n sommets $\{1, ..., n\}$ et n arêtes :

1. On part du graphe G_1 qui contient un unique sommet $\{1\}$, et dont l'unique arête est une boucle $1 \to 1$;
2. Connaissant G_k, on rajoute un sommet $k+1$ et une nouvelle arête $k+1 \to v_{k+1}$, où $v_{k+1} \in \{1, 2, ..., k\}$ est aléatoire, indépendant de G_k, et tiré de la façon suivante :

$$\mathbb{P}(v_{k+1} = i) = \frac{degre(i)}{\sum_j degre(j)} = \frac{degre(i)}{2k-1}$$

où **degre(i)** est le nombre d'arêtes **distinctes** qui touchent le sommet i.

2.2 Simulation

Voici le script Python pour simuler la matrice d'adjacence d'un graphe à attachement préférentiel, avec **Adjacence(i,j)**=1 si i et j sont voisins dans le graph :

```
import numpy as np

def Gr(n):
    if n==1:       # cas n=1
        return [[1]]
    deg=[0,1]      # initialistion de la liste des degres
    A=np.zeros((n,n))
    A[0][0]=1      # connecter 1 avec 1
    for i in range(1,n):    # pour chaque element
        # on calcule la liste des probabilites cumules
        p=np.array(np.cumsum(deg))/np.sum(deg)
        # on tire un nombre al atoire
        s=rd.random()
```

```
        # on regarde sa position j dans p
        j=-1
        for k in range(len(p)-1):
            if s>=p[k] and s<p[k+1]:
                j=k
        # on relie i avec j
        A[i][j]=1
        A[j][i]=1
        # on actualise deg
        deg[j+1]+=1
        deg.append(1)
    # on retourne la matrice d'adjacence ainsi que la liste des degres
    return A,deg[1:]
```

Listing 1 – Simulation de G_n

Quelques Exemples :

$n = 2$: $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

$n = 5$: $\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

Remarques :

Deux petites remarques concernant le script ci-dessus :

— Le choix d'une méthode itérative, qui est justifié par une perfermence supérieure en gestion de mémoire par rapport à la méthode récursive, qui prend beaucoup plus de temps pour n assez grand.

— Le choix de retourner la liste des degrés, qui permet directement de l'exploiter dans ce qui suit (la distribution de la loi des degrés)

2.3 Loi des degrés des sommets

Pour n et k assez grand (avec $k < n$), on peut montrer que :

$$\mathbb{P}(degre(s) = k) \approx ck^{-\alpha}$$

Pour une simulation de la loi des degrés des sommets et pour une bonne illustration, on peut utilisé la fonction seaborn.displot, qu'on peut installer avec la commande conda install seaborn.

Le script de la simulation est le suivant :

```python
import seaborn as sbn

def Degre_distribution(n,n_sim):
    L=np.zeros(n)
    for i in range(n_sim):  # on simule n_sim fois,
        _,A=Gr(n)           # pour n tres grand une simulation suffit
        L+=np.array(A)/n_sim  # A repr sente la liste des degr s
    sbn.distplot(L)
    plt.show()
    return L

L=Degre_distribution(1000,10)
```

Listing 2 – Simulation de la loi des degres des sommets

Pour $n = 1000$ et $n_sim = 10$ On retrouve le résultat suivant, où c'est assez clair que c'est loi de puissance.

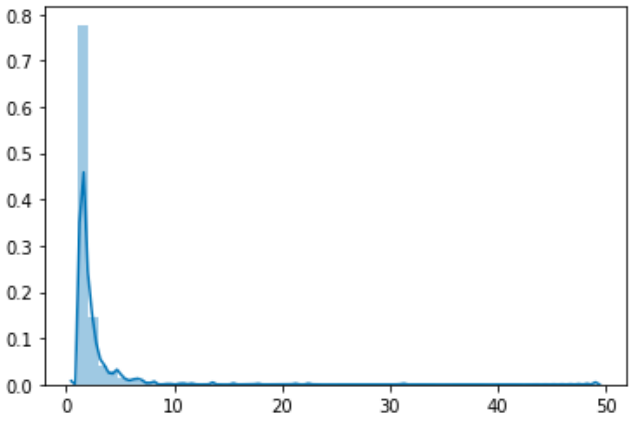

Figure 1 – Illustration de la loi de puissance

.

Reste a savoir comment approximer α.
Pour cela, trois méthodes sont envisageables :

- on pourrait représenter la loi de probabilité dans un repère log-log, dans ce cas, la courbe serait une droite de pente - α. Le probleme avec cette méthode c'est que meme pour des n assez grand, y a des degrés qui n'apparaissent pas dans la liste, et donc le log n'est pas défini
- Pour remédier à cela, on peut stocker dans une liste, le log du nombre des apparences non nul, et en utilisant la fonction optimize.leastsq pour trouver la meilleur droite qui pourra passer par ces points [1]
- ou plus simplement, utiliser la fonction powelow.Fit directement sur la liste des degrés

Une implémentation simple de la dernière méthode donne :

```
import powerlaw
results = powerlaw.Fit(L)
print(results.power_law.alpha)
```

Listing 3 – Evaluation de α

On trouve : $\alpha \approx 3$.
Ce qui est en accord avec les résultats trouvés par Barabási et R.Albert.

Une généralisation :

On peut même généraliser ceci, en prenant :

$$\mathbb{P}(v_{k+1} = i) = \beta\frac{1}{k} + (1-\beta)\frac{degre(i)}{2k-1}, \beta \in [0,1] \setminus \{1\}$$

Et dans ce cas, la loi des degrés serait approximativement, une loi de puissance de parametre $\frac{3-\beta}{1-\beta}$.[2]
Ceci peut donc être une manière concevable pour simuler des lois de puissance.
D'autre part, ceci est aussi une particularité des graphes à attachement préférentiel, qui expliquent l'apparition de la loi de puissance observé aussi dans des modèles dans la nature(réseau sociaux, internet).

2.4 Rich-gets-richer Phenomenon

On a :

$$E(X_{k+1}|X_k) = p_k(X_{k+1}+1) + (1-p_k)X_k$$

avec :

$$p_k = \frac{Xk}{2k-1}$$

car soit on branche k+1 avec 1, et donc le nombre de degré devient $X_k + 1$, ou non et le nombre reste le meme.
En passsant à l'esperence, on trouve :

$$E(X_{k+1}) = E(X_K)\frac{2k}{2k-1}$$

On voit bien que la moyenne des sommets de 1 ne fait qu'augmenter (strictement croissante), et assez supérieur à la moyenne.
Ainsi, 1 qui commence déjà "riche", le devient de plus en plus.
C'est le Rich-gets-richer Phenomenon.

3 VISUALISATION DU GRAPHE

Dans cette section, il est question de représenter les graphes de manière harmonieuse. Une première approche simple et basique serait de les représenter d'une manière aléatoire.

3.1 Visualisation aléatoire

— On tire au sort n points uniformes indépendants $(X_1, Y_1), ..., (X_n, Y_n)$ dans le carré $[0, 1]^2$

— Si i, j sont voisins dans G_n, tracer un segment entre (X_i, Y_i) et (X_j, Y_j)

Le script de cette visualisation est le suivant :

```
def Graph(A,X,Y):
    n=len(A)                # A la matrice d'adjacence
    plt.plot(X,Y,'ob')      # X,Y la liste des absices et ordon es resp
    for i in range(n):
        for j in range(i,n):
            if A[i][j]==1:
                plt.plot((X[i],X[j]),(Y[i],Y[j]),'r')
    plt.show()

n=50
Graph(Gr(n)[0],rd.random(n),rd.random(n))
```

Listing 4 – Evaluation de alpha

Le résultat est une représentaton chaotique et désastreuse.

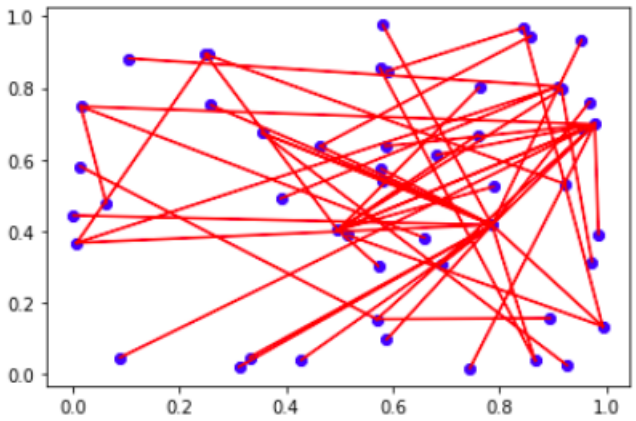

Figure 2 – Représentation aléatoire du graphe G_{50}

.

Pour remédier à cela, on définit l'enérgie suivante :

$$E = \sum_{i,j} \frac{\left(\frac{1}{\sqrt{2}}\|M_i - M_j\| - D^*_{i,j}\right)^2}{\left(D^*_{i,j}\right)^2}$$

où

$$D^*_{i,j} = \frac{Distance(i,j)}{max_{i,j} Distance(i,j)}$$

On cherche la configuration des M_i qui va minimiser cette énergie en utilisant l'algorithme "gradient descent". Mais avant, on introduit les fonctions intermediaires suivantes :

```
# cette fonction retourne la matrice D_etoile defini ci-dessus
def distance_etoile(G):         #prend en parametre la matrice d'adjacence
    n=len(G)
    A=np.zeros((n,n))
    A[0][1],A[1][0]=1,1         #initialisation
    for i in range(2,n):
        for j in range(i):
            if G[i][j]==1:      #pour chaque it ration, on cherche l'element
     branch
                for k in range(i):
                    A[k][i],A[i][k]=A[k][j]+1,A[k][j]+1  #on actualise la
    distance
                break
    return A/np.max(A)
#cette fonction retourne la distance entre Mi et Mj
def dist_sqr2(X,Y,i,j):
    return np.sqrt(((X[i]-X[j])**2+(Y[i]-Y[j])**2)/2)
def E(G,X,Y,D):
    n=len(G)
    S=0
    for i in range(n):
        for j in range(i):
            d=dist_sqr2(X,Y,i,j)
            S+=((d-D[i][j])**2)/(D[i][j])**2
    return S
#cette fonction retourne le gradient de E
def derive_E(G,X,Y,D):
    n=len(G)
    X_der=np.zeros(n)
    Y_der=np.zeros(n)
    for i in range(n):
        S,T=0,0
        for j in range(n):
            if j!=i:
                d=dist_sqr2(X,Y,i,j)
                S+=((d-D[i][j])*(X[i]-X[j]))/(d*(D[i][j])**2)
                T+=((d-D[i][j])*(Y[i]-Y[j]))/(d*(D[i][j])**2)
        X_der[i]=S
        Y_der[i]=T
    return X_der,Y_der
```

```
#cette fonction calcule la variation totale entre deux configurations des Mi
def var(X,Y,M,N):
    return np.sum([np.sqrt((X[i]-M[i])**2+(Y[i]-N[i])**2)  for i in range(
   len(X))])
```

Listing 5 – Fonctions intermédiaires

Le script du "gradient descent" est le suivant :

```
#epsilon repr sente la limite de convergence, et alpha le pas de la
   descente
def Gradient_descent(G,epsilon,alpha):
    n=len(G)
    X=rd.random(n)
    Y=rd.random(n)
    D=distance_etoile(G)
    v=epsilon+1
    while(v>epsilon):
        X_der,Y_der=derive_E(G,X,Y,D)
        v=var(X,Y,X-alpha*X_der,Y-alpha*Y_der)
        X,Y=X-alpha*X_der,Y-alpha*Y_der
    return X,Y

X,Y=Gradient_descent(G,0.01,0.001)
Graph(G,X,Y)
```

Listing 6 – Gradient descent

Pour le même graphe que tout à l'heure, on trouve cette fois une représentation plus harmonieuse avec epsilon=0.01 et alpha=0.001 : Le choix de epsilon controle la convergence du modèle.

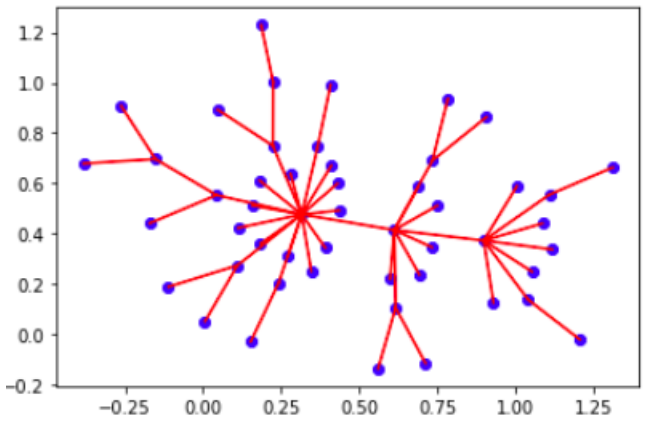

Figure 3 – Représentation harmonieuse du graphe G_{50}

.

Le choix du pas est assez cruciale. Pour un assez grand, le modele diverge rapidement.
D'autre part, on peut aussi représenter ce graphe en 3D, il suffit d'ajouter Z aux fonctions précédente.

3.2 Attachement δ-préferentiel

On peut généraliser le modèle du graphe à attachement préférentiel de la façon suivante. Soit $\delta > -1$ un paramètre fixé, on remplace l'équation précedente par

$$\mathbb{P}(v_{k+1} = i) = \frac{degre(i)+\delta}{\sum_j (degre(j)+\delta)} = \frac{degre(i)+\delta}{2k-1+k\delta}$$

Pour $\delta = 0$, on retrouve le modèle initial. Le script de simulation est le suivant :

```
def Gr_delta(n,d):
    if n==1:
        return np.array([[1]])
    deg=[0,1+d]
    A=np.zeros((n,n))
    A[0][0]=1
    for i in range(1,n):
        p=np.array(np.cumsum((deg))/np.sum(deg))
        a=rd.random()
        j=-1
        for k in range(len(p)-1):
            if a >=p[k] and a <p[k+1]:
                j=k
        deg[j+1]+=1
        deg.append(1+d)
        A[i][j]=1
        A[j][i]=1
    return A
```

Listing 7 – Gradient descent

Quand δ tend vers l'infini, on choisit v_{k+1} de manière uniforme dans $\{1, ..., k\}$
Ainsi, δ permet de controler la préference par rapport aux sommets.
Pour $\delta = 0$, on a vu qu'il y a une tendence de "Rich gets richer".
Pour δ plus grand, le graph est plus filaire et homogène (moins de concentration de richesse).
Pour $n = 50$ et $= 10000$, on trouve :

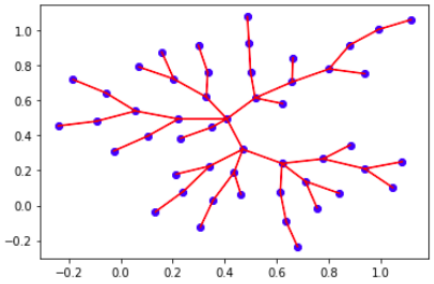

Figure 4 – Représentation du graphe δ préferentiel G_{50}

.

4 SPECTRAL CLUSTERING

4.1 L'algorithme

Soit k le nombre de clusters, et n la taille du graphe.

- On calcule le laplacien L=D-A, avec D une matrice diagonale tel que $D_{i,i} = degre(i)$ et A la matrice d'adjacence
- On calcule les k premiers vecteurs propres de L
- Soit $U \in R^{nxk}$ la matrice qui contient ces vecteurs en colonnes
- Pour $i = 1, ..., n$ soit $y_i \in R^k$, on "cluster" ces points en k clusters en utilisant l'algorithme k-means

Le script correspedant est le suivant :

```
from sklearn.cluster import KMeans
def spectral_cluster(G,k):
    D = np.diag(G.sum(axis=1))
    L=D-G
    vals,vecs = np.linalg.eig(L)
    vecs = vecs[:,np.argsort(vals)]
    vals = vals[np.argsort(vals)]
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(vecs[:,1:k])
    label = kmeans.labels_
    colors=["blue","red","green","cyan" ,"magenta","yellow","black","white"]
    for i in range(len(G)):
            plt.scatter(X[i],Y[i],c=colors[label[i]])
    for i in range(len(G)):
        for j in range(i,len(G)):
            if G[i][j]==1:
                plt.plot((X[i],X[j]),(Y[i],Y[j]),'black')
    plt.xlabel("k={}".format(k))
    plt.show()
```

Listing 8 – Spectral Clustring

Les sorties de cette algorithme pour différentes valeurs de k

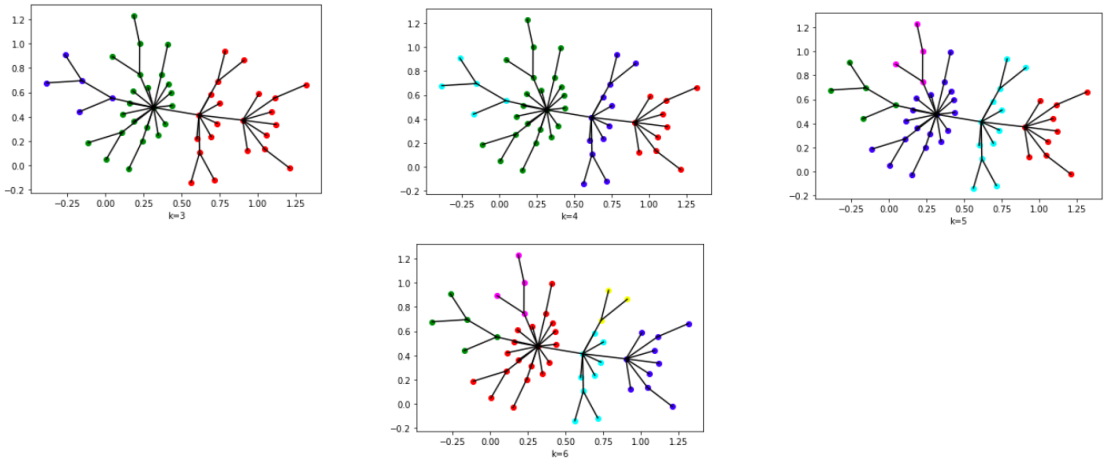

Figure 5 – Spectral clustering pour différents k

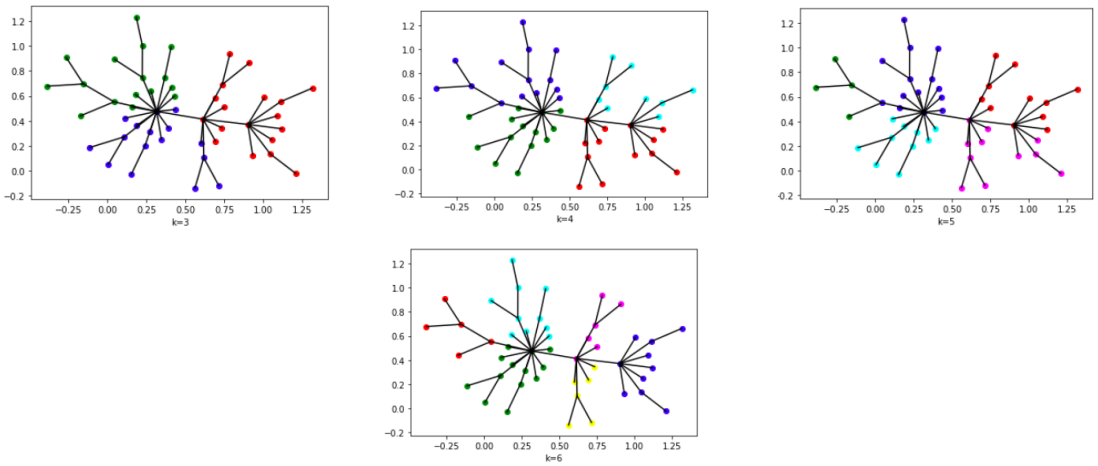

Figure 6 – K-means clustering pour différents k

On pourra le comparer avec un K-Means directement appliqué sur les points du graphe, comme dans la Figure 6 ci-dessus.
On voit clairement la particularité du Spectral Clustering.
Le K-means permet de trouver les clusters en minimisant la distance total, sans prendre en compte la structure du graphe.
D'autre part, le Spectral Clustering combine les deux contraintes.

4.2 Stochastic Block Model

On va maintenant s'interesser à implémenter le Spectral Clustring sur un autre modèle de graph.
Soit $E_1, ..., E_k$ une partition de $\{1, ..., n\}$ en K clusters, et $(q_{r,s})_{r,s \leqslant K}$ la matrice de probabilité inter-classe.
Si $i \in E_r$ et $j \in E_s$, on met une arrête entre i et j avec probabilité $q_{r,s}$

Supposons qu'un graphe est déjà simulé avec le modèle SBM, le but est de retrouver les clusters avec le Spectral Clustering.

On le test pour des petites valeurs de n.

```
from graspy.simulations import sbm
K=2
n_SBM = [50, 50]
P_SBM = [[0.8, 0.2],
         [0.2, 0.5]]
G_SBM = sbm(n=n_SBM, p=P_SBM)
D_SBM = np.diag(G_SBM.sum(axis=1))
L_SBM=D_SBM-G_SBM
vals_SBM,vecs_SBM = np.linalg.eig(L_SBM)
vecs_SBM = vecs_SBM[:,np.argsort(vals_SBM)]
vals_SBM = vals_SBM[np.argsort(vals_SBM)]
kmeans_SBM = KMeans(n_clusters=K)
kmeans_SBM.fit(vecs_SBM[:,1:K])
label_SBM = kmeans_SBM.labels_
print(label_SBM)
```

Listing 9 – Spectral Clustring sur le SBM

La sortie est de : [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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
On récupère exactement les deux clusters initiaux, car on l'applique avec un nombre de K déjà connus, et la probabilité de rester dans le même cluster est assez grande (les éléments de la diagonale de la matrice de probabilité).
Si ces éléments sont assez petits, il est plus difficile de récupérer ces clusters.
Il est même possible de retrouver des conditions plus exactes entre ces deux probabilités pour retrouver le modèle.

```
MatriceAdjacence=np.loadtxt('StochasticBlockModel.txt')
D2 = np.diag(MatriceAdjacence.sum(axis=1))
L2=D2-MatriceAdjacence
vals2,vecs2 = np.linalg.eig(L2)
vecs2 = vecs2[:,np.argsort(vals2)]
vals2 = vals2[np.argsort(vals2)]
kmeans_ADJ = KMeans(n_clusters=4)
kmeans_ADJ.fit(vecs2[:,1:4])
label_ADJ = kmeans_ADJ.labels_
```

```
print(label_ADJ)
```

Listing 10 – Spectral Clustring sur l'exemple

Pour la matrice proposé, on trouve pour K=4 la partition suivante : [360 90 60 90]

5 PAGERANK

5.1 L'algorithme

Afin de trouver le score de popularité d'un graphe, on cherche le vecteur propre à gauche de la matrice suivante :

$$P_\varepsilon = (1-\varepsilon)\widetilde{A} + \frac{\varepsilon}{n}\begin{pmatrix} 1 & \dots & 1 \\ & . & \\ 1 & \dots & 1 \end{pmatrix}$$

avec $\widetilde{A}$ est la matrice d'adjacence renormalisé.

5.2 L'implémentation

Le script Python est le suivant :

```
from scipy.linalg import eig
def P(e,G):
    return (1-e)*(G/np.sum(G,axis=1)) + e*np.ones((len(G),len(G)))/(len(G))
def pagerank(e,G):
    n=len(G)
    S=np.zeros((n,n))
    # on trinagularise
    for i in range(n):
        for j in range(i+1):
            S[i][j]=G[i][j]
    Pe=P(e,S)
    values, vecs = eig(Pe, right = False, left = True)
    vecs = vecs[:,np.argsort(values)]
    values = values[np.argsort(values)]
    for i in range(n):
        if round(values[i].real,2)==1.00 :
            return vecs[:,i].real/np.sum(vecs[:,i].real)
    return np.zeros(n)
```

Listing 11 – Pagerank

Remarques :

— Sans triangulariser, l'algorithme ne fournit pas des résultats correctes

— pour epsilon=0, on trouve [1,0,...,0]

— pour epsilon=1, on trouve [1,1,...,1] normalisé
— epsilon représente donc le degré d'exploration du graphe
— prendre epsilon=0.15 permets d'avoir un bon critère de popularité
— il est quand mème toujour possible d'augmenter son score en créant artificiellement quelques sommets qui pointent vers un sommet fixé

5.3 Tricher

```
def tricher(e,G):
    n=len(G)
    S=np.zeros((n+3,n+3))
    for i in range(n):
        for j in range(n):
            S[i][j]=G[i][j]
    S[n][0],S[n+1][0],S[n+2][0]=1,1,1
    return pagerank(e,S)[0]-pagerank(e,G)[0]
tricher(0.15,F)
```

Listing 12 – Augmenter le score PageRank

Dans cet exemple, on ajoute manuellement 3 sommets avec 1, et on remarque que son score augmente, ce qui est normal.

RÉFÉRENCES

[1] Fitiing data in python. https ://scipy-cookbook.readthedocs.io/items/FittingData.html.

[2] Power-law random graphs. https ://www.di.ens.fr/ lelarge/X15/cours/slides8.pdf.