Projet final : Graphes (modèles et algorithmes)

Ali MAMMADOV

Novembre-Décembre 2020

1 Simulation de graphes aléatoires : graphe par attachement préférentiel

Question 1.1

Mon implémentation du modèle de Barabasi pour la construction de graphes se compose de 3 fonctions:
La **première** consiste à trouver des probabilités cumulatives de degrés de sommets.

```
def cum_prob( G ):
    degrees = G.degree
    probs = degrees / np.sum(degrees)
    for i in range(1,len(probs)):
        probs[i] += probs[i-1]
    return probs
```

La première étape est le calcul des probabilités comme mentionné dans l'énoncé. Après, probabilité cumulative obtenue simplement en ajoutant itérativement la valeur précédente à la valeur actuelle des probabilités. *Par exemple:*
Probs = [0,3 0,2 0,5]
Cum_probs = [0,3 0,2 + 0,3 0,5 + 0,2 + 0,3] = [0,3 0,5 1,0]

La **seconde** est pour le tirage un sommet.

```
def tire_sommet(G):
    cum_probs = cum_prob(G)
    _random = np.random.random()
    for i in range(len(cum_probs)):
        if _random <= cum_probs[i]:
            return i
```

Je prends une variable flottante aléatoire entre 0 et 1. Après cela, le sommet sera sélectionné par détection quelle probabilité cumulative correspond à notre variable aléatoire. *Par exemple:*
Cum_probs = [0,3 0,5 1,0], Rand = 0.4, Rand $\leq$ Cum_probs[1].
Rand correspond à deuxième sommet.

La **troisième** fonction est pour la construction du graphe.

```
def Barabasi(n):
    G = Graph()
    G.add_node(0)
    G.add_edge(0,0)
    for i in range(1,n):
        G.add_node(i)
        G.add_edge( tire_sommet(G),i)
    return G
```

Comme dit, la fonction commence par ajouter un sommet et une boucle au premier sommet. Ensuite, il ajoute un nouveau sommet et une nouvelle arête qui connecte le nouveau sommet avec un autre sélectionné de la manière expliquée précédemment. Selon lequel, le sommet avec un degré élevé a plus de chance d'être sélectionné.

Listing 1: Matrice d'adjacence de G_{10}

```
1 1 1 1 0 1 0 0 0 0
1 0 0 0 1 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 1 1 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
```

```
Matrice d'adjacence de graph de 10 sommets,
cree par attachement preferentiel.
Les boucles sont sur diagonal
et que le sommet 1 qui en a.
```

Question 1.2

Loi de puissance:

$$\begin{aligned} P(k) &= ck^{-\alpha} \\ log(P(k)) &= log(ck^{-\alpha}) \\ log(P(k)) &= log(k^{-\alpha}) + log(c) \\ log(P(k)) &= -\alpha log(k) + log(c) \\ y = log(P(k)), x &= log(k), b = log(c), a = -\alpha \\ y &= a * x + b; \end{aligned} \tag{1}$$

Ainsi, en faisant une simple régression linéaire, nous pouvons obtenir les valeurs de α et c.

```
def reg_lin(G):
    X = np.log(prob(G)) # prob, simple function returns [k, P(k)]
    Y = np.copy(X[:,1])
    X[:,1] = np.ones_like(X[:,1])
    XTX = np.dot(X.T,X)
    XTY = np.dot(X.T,Y)
    a, b = np.dot( np.linalg.inv(XTX), XTY)
    alpha, c = -a, (np.e)**b
    return alpha, c
```

Pour le graphe avec 1000 sommets - G_{1000}, j'ai obtenu: $\alpha = 0.4$, $c = 0.008$.

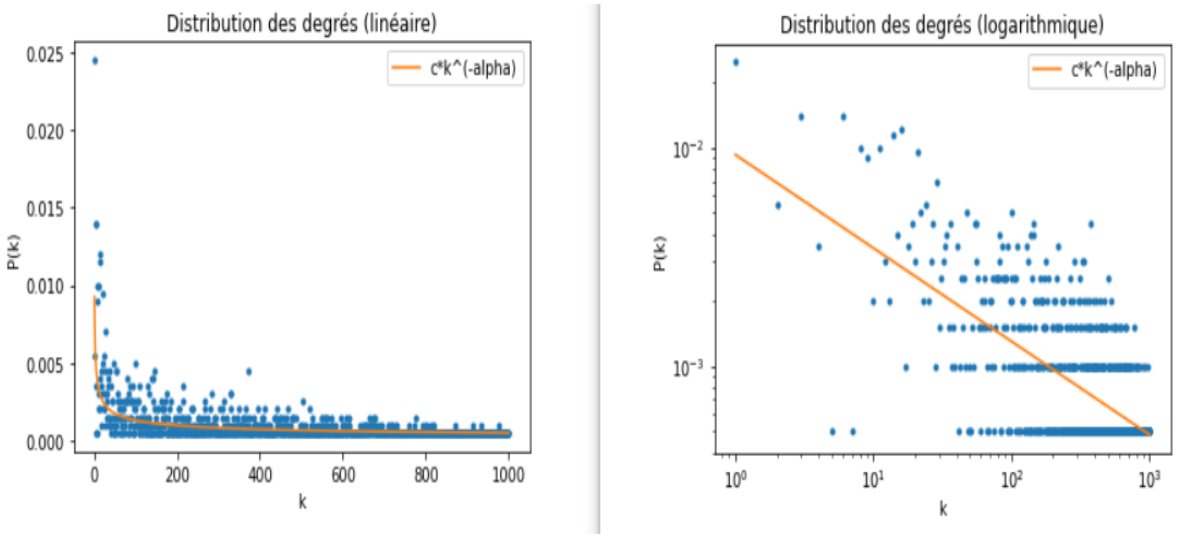

Figure 1: G_{1000}, $\alpha = 0.4$, $c = 0.008$

Question 1.3

Example su la matrice dans Question 1.2

$E[X1] = \frac{1+1+1+1+0+1+0+0+0+0}{10} = 1/2$; $E[Xm] < 1/2$; pour tous m > 1

2 Visualisation d'un graphe : un problème d'optimisation

Question 2.1

```
def draw_graph(g):
    X = np.zeros( g.size() ) # g.size() est nombre de sommets
    Y = np.zeros( g.size() )
    m = matrice_adjacence(g)
    for i in range( g.size() ):
        X[i] = np.random.random()
        Y[i] = np.random.random()
    plt.plot(X, Y, 'o') # dessiner des sommets
    for i in range( g.size() ):
        for j in range( i, g.size() ):
            if m[i][j] : # si i, j sont voisins, tracer l'arete
                plt.plot([X[i], X[j]],[Y[i], Y[j]], 'red')
    plt.show()
```

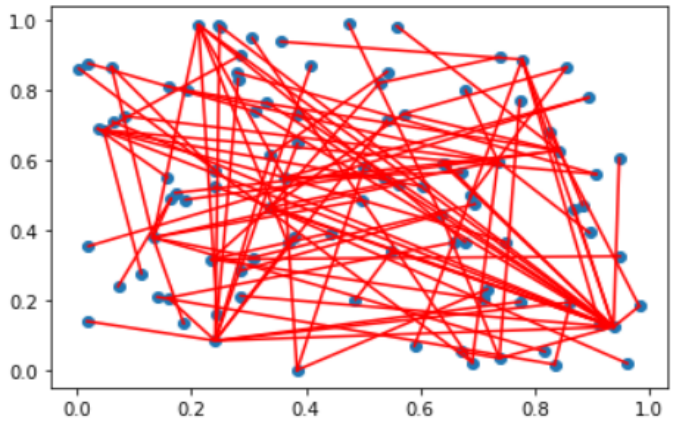

Figure 2: G_{100},

Question 2.2

Pour dessiner un graphe élégante, nous devons d'abord calculer la matrice de distance. Qui composent longueur du plus court chemin entre tous les sommets. Il y a quelques algorithmes pour cela comme Floyd-Warshall etc. Mais si nous ignorons la boucle sur le premier sommet, nous aurons un arbre. Et pour calculer la matrice de distance, nous appliquerons l'algorithme BFS (Breadth First Search).

```
dist = np.zeros([N,N]) # Matrice de distances
for n in range(N): # pour chaque sommets
    # tous les sommets visites
    explored = []
    # sommets a verifier
    queue = [n]
    # continuez a boucler jusqu'a ce qu'il reste pas des sommets
    while queue:
        # pop (premier sommet) de la file d'attente
        node = queue.pop(0)
        if node not in explored:
            # ajouter le sommet a la liste des sommets verifies
            explored.append(node)
            neighbours = list(graph.adj[node])
            for i in neighbours:
                if i not in explored:
                # calculation de longueur du chemin des
```

```
                dist[n][i] += (1+dist[n][node])
            # ajouter des voisins du sommet a la file d'attente
            queue.append(i)
```

Il commence à la racine(un sommets arbitraire de graphe) de l'arbre, explore tous les sommets voisins à la profondeur actuelle et calcule la longueur du chemin des voisins à partir de la racine avant de passer aux sommets au niveau de profondeur suivant.

Obtention de la fonction de descente de gradient à partir de l'énergie:

$$\begin{aligned}
\frac{\delta E}{\delta M_k} &= \sum_{i,j} \frac{(\frac{1}{\sqrt{2}}\|M_i - M_j\| - D^*_{i,j})^2}{(D^*_{i,j})^2} = \\
&= \sum_{i,j} \left[\frac{2(\frac{1}{\sqrt{2}}\|M_i - M_j\| - D^*_{i,j})}{(D^*_{i,j})^2} \frac{1}{\sqrt{2}} \frac{\|M_i - M_j\|}{\delta M_k} \right] = \\
&= \sum_{i,j} \left[\frac{(\|M_i - M_j\| - \sqrt{2}D^*_{i,j})}{(D^*_{i,j})^2} \frac{M_i - M_j}{\|M_i - M_j\|} \right] = \delta E \qquad (2) \\
\frac{\delta E}{\delta M_k} &= \delta E; k = i \\
\frac{\delta E}{\delta M_k} &= -\delta E; k = j
\end{aligned}$$

Algorithm 1: Minimisation de l'énergie E

```
d(i,j) – matrice_distance(G);
M – random_points(N,2) ;
while E < seuil do
    for each i,j do
        M_i = M_i − δE;
        M_j = M_j + δE;
    end
    Mise à jour E;
end
```

Listing 2: "Version Python"

```
# continuez a boucler jusqu'a ce qu'il
# difference des energies est moins de tolerence
while( (p_E - E) > tol ):
  p_E = E # Energie precedente
  E = 0    # Energie
  for i in range(G.size()): # pour chaque i < j
    for j in range(i+1,G.size()):
      dE = (np.linalg.norm( M[i] - M[j]) - D(i,j) * np.sqrt(2)) * (  M[i] - M[j] )
      dE /= ( (D(i,j)**2) * np.linalg.norm( M[i] - M[j]) )
      M[i] = M[i] - eta*dE # k = i, dE > 0
      M[j] = M[j] + eta*dE # k = j, dE < 0
      # Mise a jour de l'energie E
      E += ((np.linalg.norm( M[i] - M[j])/np.sqrt(2) - D(i,j))**2 / (D(i,j)**2))
```

Question 2.3

À partir des images, nous pouvons voir que quand δ est supérieur à zéro, le graphe se propage davantage. Cela brise également le phénomène «Rich Gets Richer». Parce que les probabilités des sommets sont devenues plus équiprobables et la chance d'obtenir plus d'arêtes pour les premiers sommets diminue voir sur la (Fig. 8). Mais lorsque δ est inférieur à zéro, le phénomène «Rich Gets Richer» devient plus fort et le graphe ne se propage pas beaucoup. Comme on peut le voir sur la (Fig. 5), le degré du premier sommet est beaucoup plus que les autres. Car à chaque itération, sa probabilité devient de plus en plus par rapport aux autres.

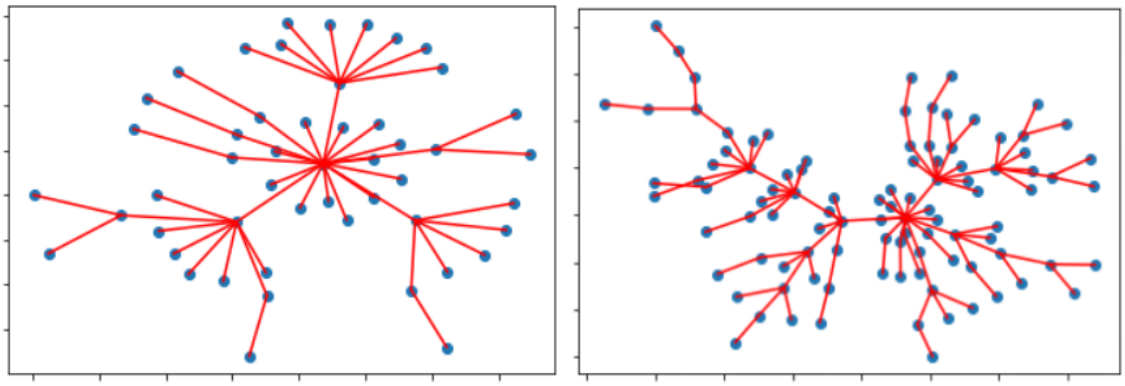

Figure 3: G_{50} et G_{100} avec descent gradient en 2D

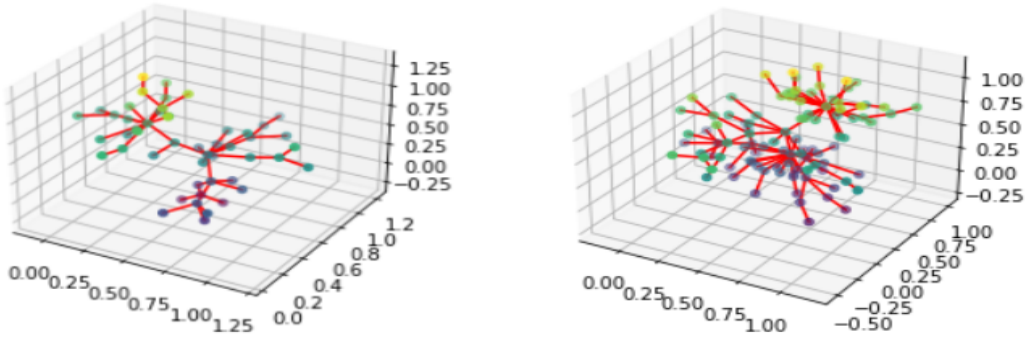

Figure 4: G_{50} et G_{100} avec descent gradient en 3D

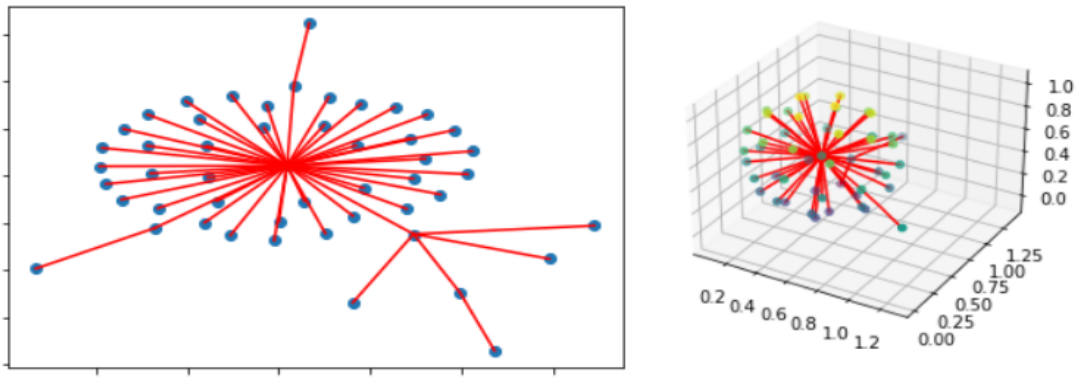

Figure 5: G_{50} avec $\delta = -0.9$ en 2D et 3D

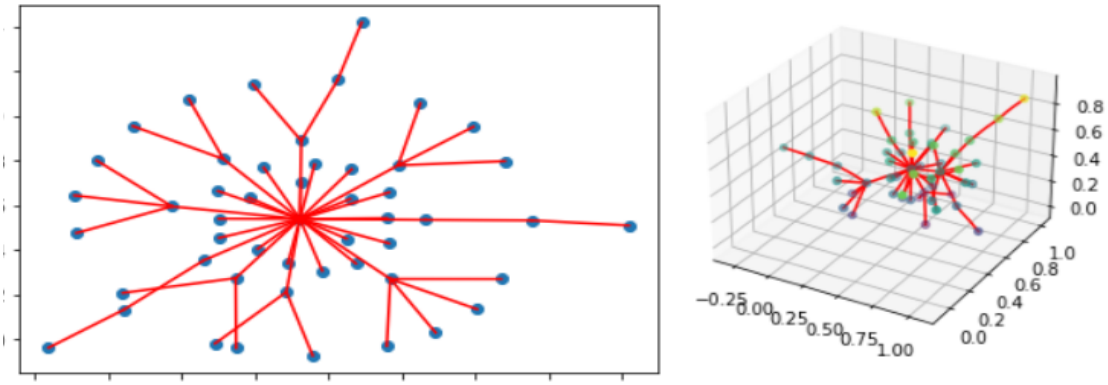

Figure 6: G_{50} avec $\delta = -0.3$ en 2D et 3D

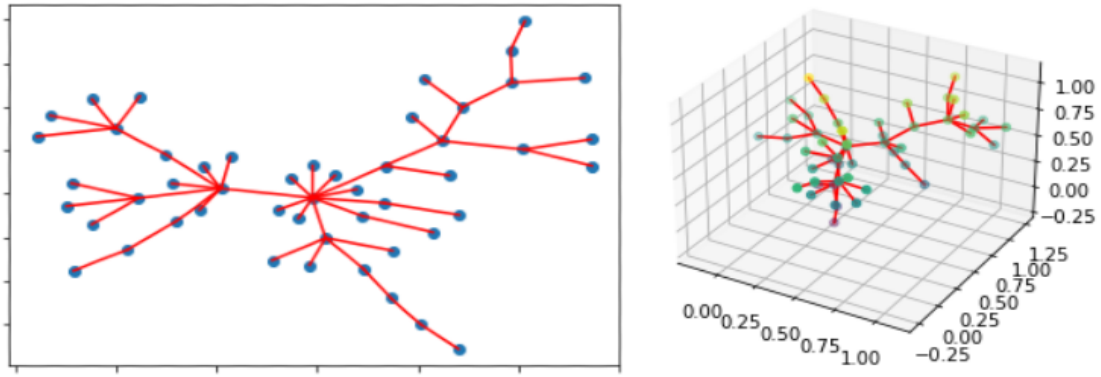

Figure 7: G_{50} avec $\delta = 0.3$ en 2D et 3D

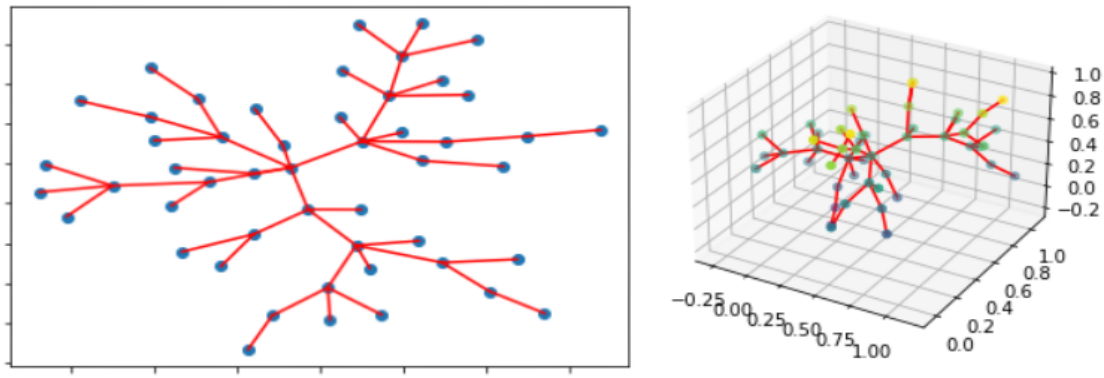

Figure 8: G_{50} avec $\delta = 5$ en 2D et 3D

3 Clustering et détection de communautés : utilisation de la décomposition spectrale

Question 3.1

Soit $\{x_1, \ldots, x_n\}$ un ensemble de n points de données. L'algorithme K-means vise à regrouper les points de données en K clusters, où le nombre de clusters K est une entrée de l'algorithme. La fonction objectif K-means est égale à,

$$\underset{(C_1,\ldots,C_K)}{\operatorname{argmin}} \sum_{j=1}^{K} \sum_{i \in C_j} \|x_i - c_j\|^2, \tag{3}$$

où $\{c_1, \ldots, c_K\}$ sont les centroïdes des classes K notés $\{C_1, \ldots, C_K\}$. Cette fonction objectif est résolue à l'aide d'un algorithme itératif, où les deux suivants des étapes interviennent à chaque itération: (i) prendre chaque instance appartenant à l'ensemble de données et l'assigner au centroïde le plus proche, et (ii) recalculer les centroïdes de chacun des clusters K. Ainsi, les centroïdes K changent leur emplacement étape par étape jusqu'à ce que plus aucun changement ne soit effectué.

Listing 3: Kmeans

```
def K_means(X, K):
    E=10**(-8) # Tolerance
    indices = np.random.choice(np.arange(N),size=K,replace=False)
    c = X[indices,:]
    c_temp = 100
    while(np.abs(np.mean(c-c_temp))>E):
        dists = np.zeros((N,K))
        for k in range(K):
            dists[:,k] = np.linalg.norm(X-c[k,:], axis=1)
        #step(1)
        # matrice de centroids
        cluster_membership = np.argmin(dists, axis=1)
        # S - matrice binaire contenant les classes de clusters
        S = np.zeros((N,K))
        for i in range(N):
            S[i,cluster_membership[i]] = 1
        #step(2)
        c_temp = c
        c = S.T@X/ np.sum(S,0).reshape(-1,1)
```

Listing 4: Spectral Clustering

```
def Spectral_Clustering( G, M, k ):
    D = np.array(G.degree,dtype='float')[:,1]
    A = matrice_adjacence(G)
    # Laplacian matrix
    L = np.diag(D)-A
    eigval, eigvec = np.linalg.eig(L)
    eigvec = np.real( eigvec )
    eigval = np.real( eigval )
    kfirst = np.argsort(eigval)[:k]
    Uk = eigvec[:,kfirst]
    c,S = K_means(Uk,k)
```

Résultats de partie 3.1 et 3.3:

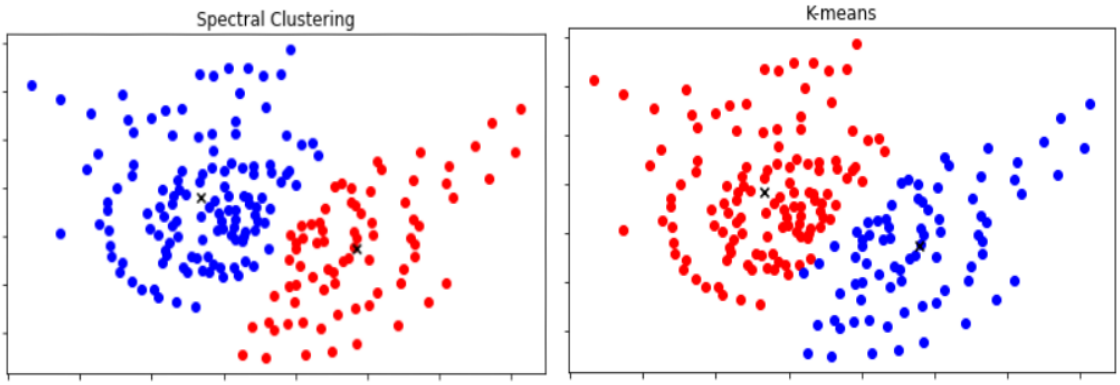

Figure 9: Graphe G_{200} par attachement préférentiel avec $K = 2$

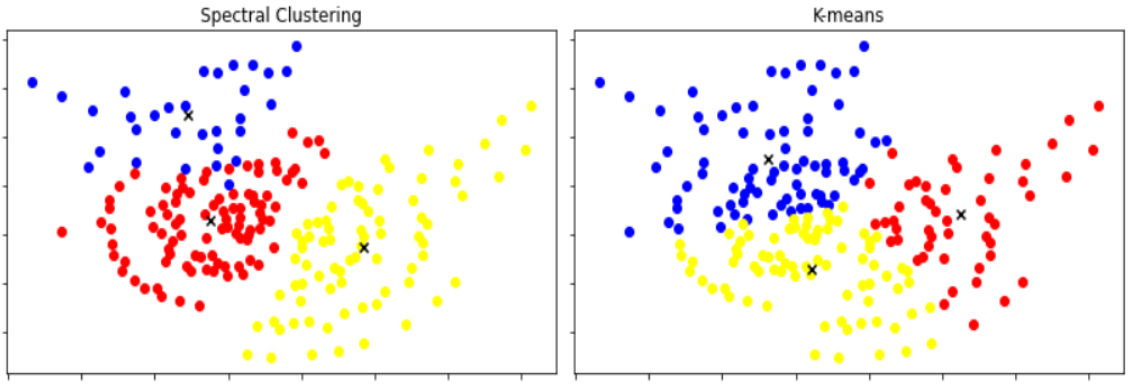

Figure 10: Graphe G_{200} par attachement préférentiel avec $K = 3$

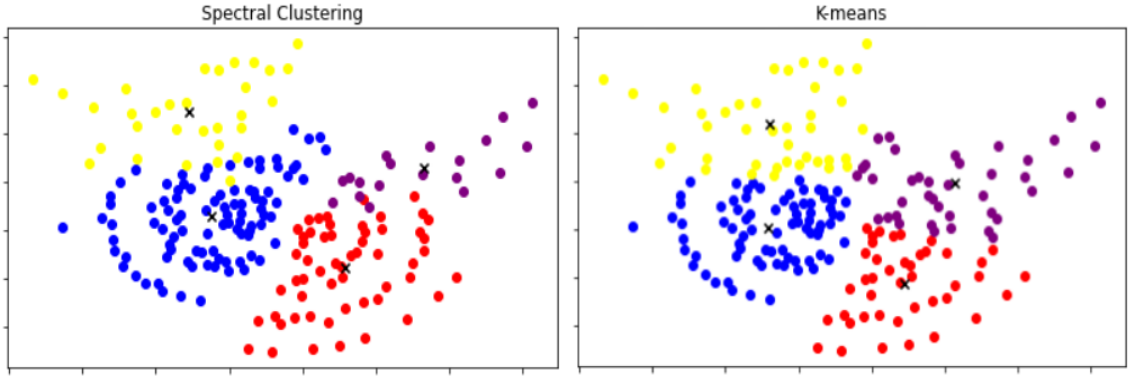

Figure 11: Graphe G_{200} par attachement préférentiel avec $K = 4$

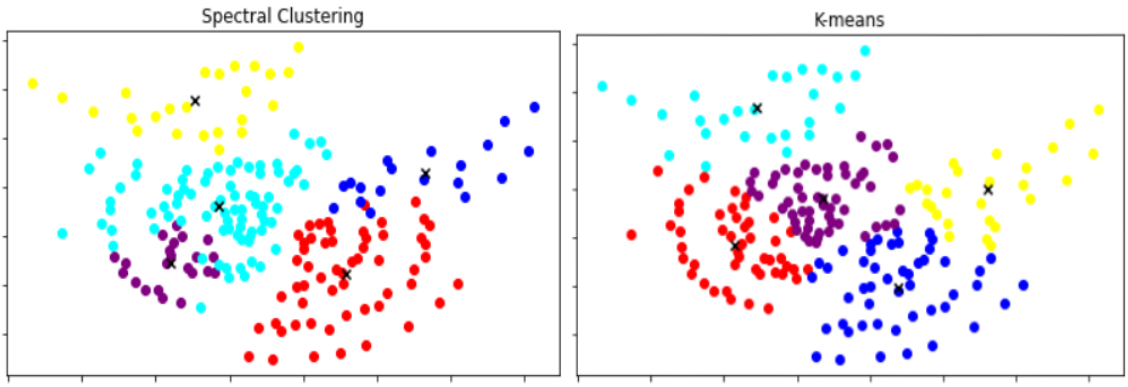

Figure 12: Graphe G_{200} par attachement préférentiel avec $K = 5$

Question 3.2

Listing 5: Stochastic Block Model

```
def SBM(N, K, p=.8, q=.05):
    G = nx.Graph()
    G.add_nodes_from(range(N))
    for i in range(N):
        for j in range(i + 1, N):
            theta = p if (i % K == j % K) else q
            if np.random.random() < theta:
                G.add_edge(i, j)
    return G
```

q - est la probabilité entre deux classes différentes. p - est la probabilité dans la même classe. On tire un sommet avec la probabilité p si deux sommets appartiennent à la même classe. Sinon avec q.

Résultats de partie 3.2 et 3.3:

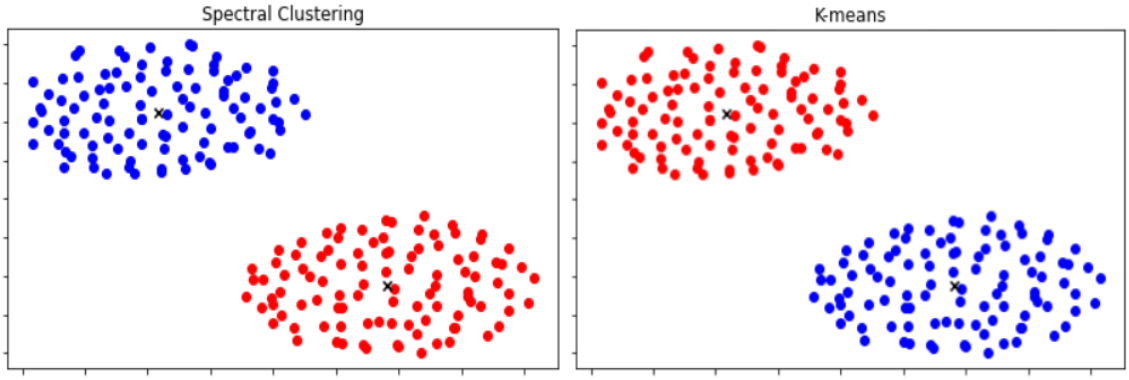

Figure 13: Graphe G_{200} par Stochastic Block Model (SBM) avec $K = 2$

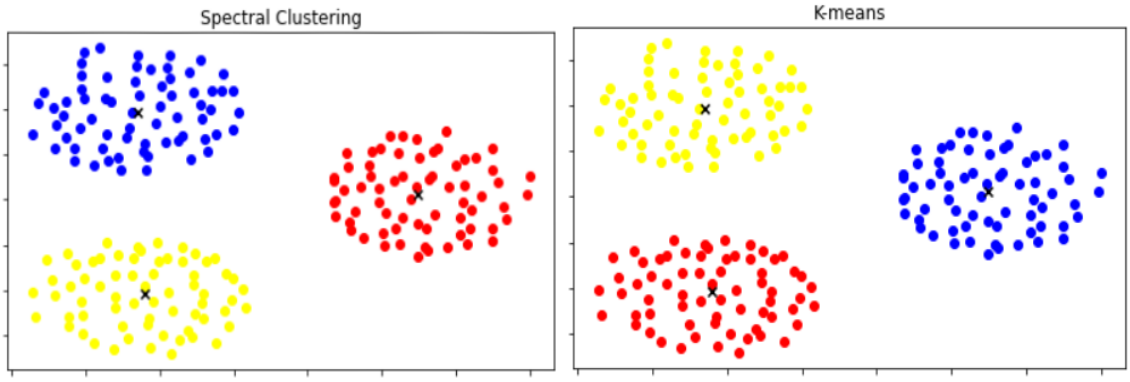

Figure 14: Graphe G_{200} par Stochastic Block Model (SBM) avec $K = 3$

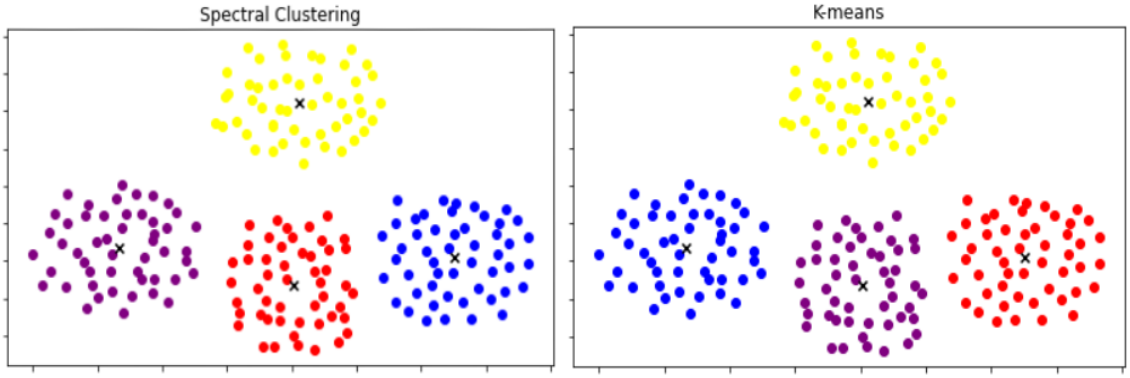

Figure 15: Graphe G_{200} par Stochastic Block Model (SBM) avec $K = 4$

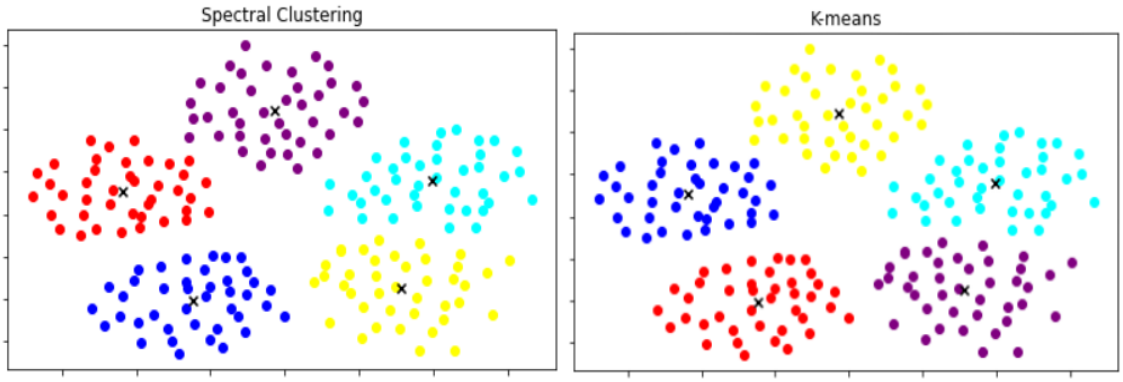

Figure 16: Graphe G_{200} par Stochastic Block Model (SBM) avec $K = 5$

Question 3.3: est couvert dans les questions 3.1 et 3.2!!!

Bonus: MatriceAdjacence(600*600) sur moodle

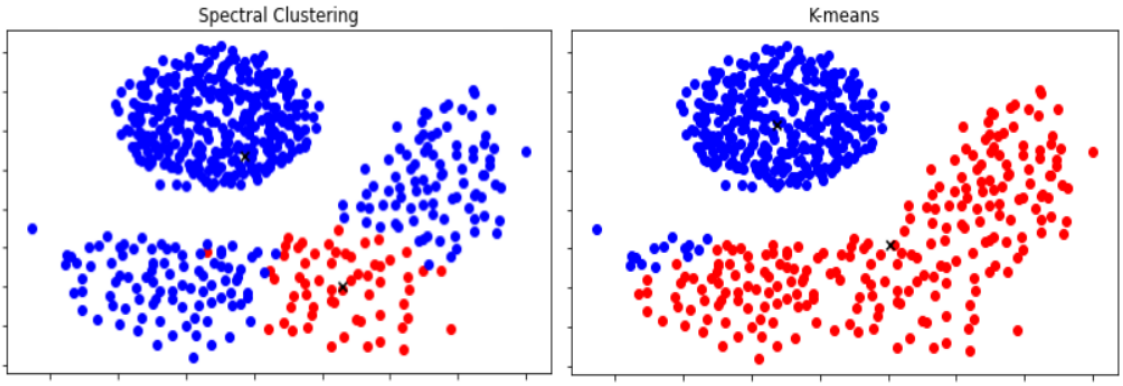

Figure 17: Graphe G_{600} par (SBM) sur Moodle avec $K = 2$

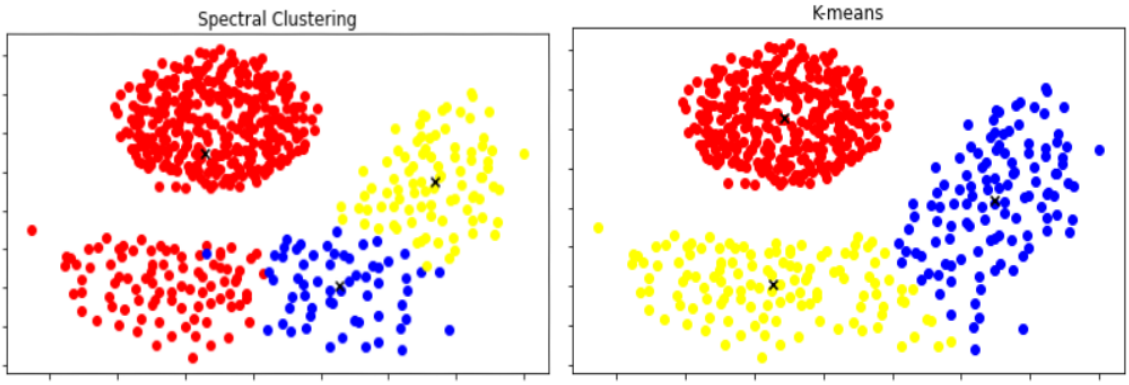

Figure 18: Graphe G_{600} par (SBM) sur Moodle avec $K = 3$

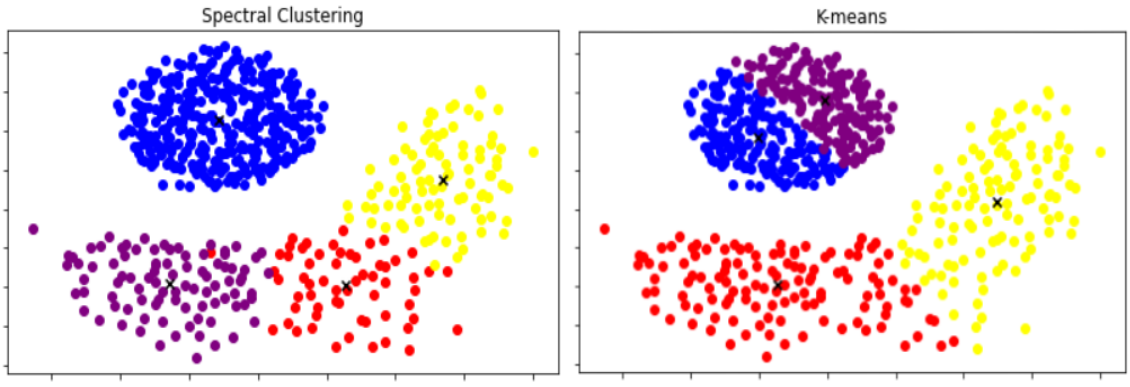

Figure 19: Graphe G_{600} par (SBM) sur Moodle avec $K = 4$

D'après les figures (17-19), nous pouvons remarquer que le clustering spectral essaie à chaque fois de diviser les sommets en quatre communautés. Et les clusters de la figure 19 montrent comment notre graphe convient à 4 communautés.

4 Popularité dans un graphe : une approche par valeurs propres (PageRank)

Question 4.1

J'ai utilisé un simulateur en ligne [1] pour PageRank pour vérifier l'exactitude de ma mise en œuvre. Sur les images, des rectangles jaunes sont les résultats du simulateur en ligne. Et si vous regardez le tableau juste au-dessus de chaque image, vous pouvez voir que le PageRank de mon implémentation (lorsque $\epsilon = 0,15$) est le même que celui du simulateur. Et à partir des tableaux, nous pouvons voir que lorsque nous augmentons la valeur de ϵ, PageRank devient presque la même pour tous les sommets. Donc, à partir de là, nous pouvons conclure que si ϵ est plus élevé, les PageRank des sommets sont devenus équiprobables. Enfin, si $\epsilon = 1$, tous les PageRanks seront identiques

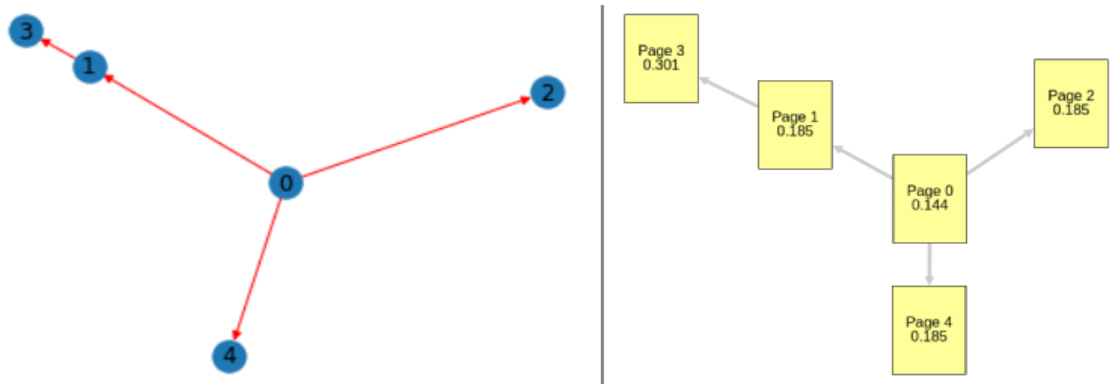

Figure 20: Graphe G_5 avec PageRank: $\epsilon = 0.15$

ϵ	PageRank				
0.05	0.139	0.183	0.183	0.313	0.183
0.15	0.144	0.185	0.185	0.301	0.185
0.30	0.152	0.188	0.188	0.284	0.188
0.60	0.171	0.194	0.194	0.248	0.194
0.90	0.192	0.199	0.199	0.212	0.199

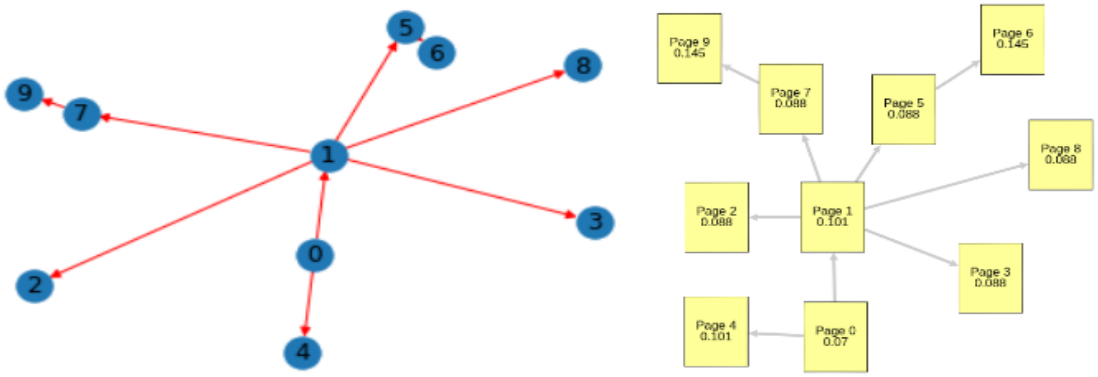

Figure 21: Graphe G_{10} avec PageRank: $\epsilon = 0.15$

ϵ	PageRank									
0.05	0.068	0.1	0.087	0.087	0.1	0.087	0.15	0.087	0.087	0.15
0.15	0.071	0.101	0.088	0.088	0.101	0.088	0.145	0.088	0.088	0.145
0.30	0.075	0.101	0.089	0.089	0.101	0.089	0.138	0.089	0.089	0.138
0.60	0.085	0.102	0.093	0.093	0.102	0.093	0.122	0.093	0.093	0.122
0.90	0.096	0.101	0.098	0.098	0.101	0.098	0.106	0.098	0.098	0.106

[1] http://computerscience.chemeketa.edu/cs160Reader/_static/pageRankApp/index.html

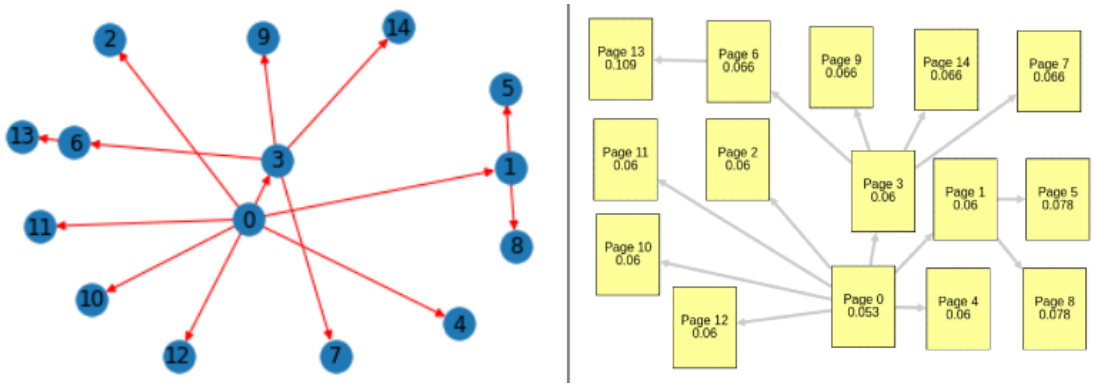

Figure 22: Graphe G_{15} avec PageRank: $\epsilon = 0.15$

ϵ	PageRank												
0.05	0.052	0.059	0.059	0.059	0.059	0.08	0.066	0.066	0.08	0.066	0.059	0.059	0.059
0.15	0.053	0.06	0.06	0.06	0.06	0.079	0.066	0.066	0.079	0.066	0.06	0.06	0.06
0.30	0.055	0.061	0.061	0.061	0.061	0.077	0.066	0.066	0.077	0.066	0.061	0.061	0.061
0.60	0.06	0.063	0.063	0.063	0.063	0.073	0.066	0.066	0.073	0.066	0.063	0.063	0.063
0.90	0.065	0.066	0.066	0.066	0.066	0.068	0.067	0.067	0.068	0.067	0.066	0.066	0.066

Question 4.2

Pour trouver le vecteur propre, j'ai utilisé la méthode de la puissance qui a été conseillée par Brin & Page. Et cela semble performant car pour le graphe avec 10 000 sommets, cela prend environ une seconde.

Implémentation de la méthode de la puissance avec λ (valeur propre) égale 1.

Listing 6: Power

```
def power(A, N_max, TOL):
    n = A.shape[0]
    V = np.random.rand(n) # initialize to random vector
    V /= V.sum() # make v sum to 1
    err = 1
    for _ in range(N_max):
        if( err < TOL ) : break
        prev_V = V
        V = np.dot(A,V)
        err = np.linalg.norm( V - prev_V )
    return V
```

Listing 7: PageRank

```
def PageRank( G, e = 0.15, tol = 1e-6 ):
    n = len(G.nodes)
    M = matrice_adjacence(G)
    # Renormalisation de matrice d'adjacence
    M[0][0] = 0
    for i in range(n):
        tmp = np.sum(M[i,:])
        if tmp > 0:
            M[i] /= tmp
        else :
            M[i] = np.ones(n)/n
    P = (1-e)*M.T + (e/n)*np.ones_like(M)
    return power(P, N_max=1000, TOL=tol)
```

Rappelons que l'algorithme de PageRank classe les sommets en fonction des éléments correspondants de V - vecteur des scores de PageRank. Sur la Figure 20, la quatrième composante

(sommet 3) de V est la plus grande, alors sommet 3 a le rang le plus élevé. L'interprétation est qu'un internaute aléatoire cliquait au hasard sur des liens pendant un temps infini, puis la page qu'il visiterait le plus fréquemment est 3. Cela a du sens: la page 3 est la plus importante car elle est accessible à partir du plus grand nombre de pages. Dans notre cas, c'est égal à 2 depuis 0 et 1, mais les autres n'en ont que 1.

Question 4.3

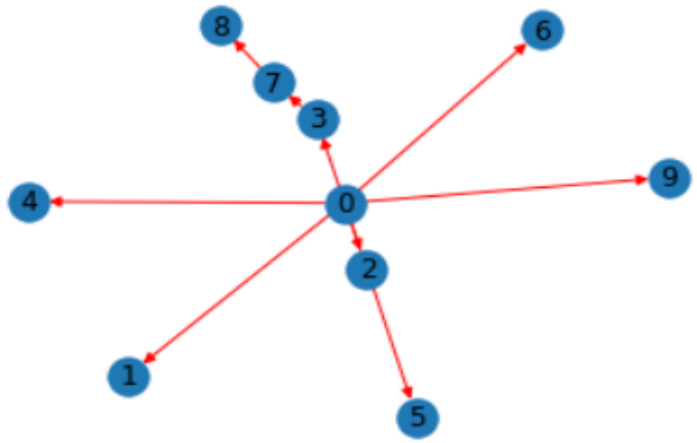

Figure 23: Graphe G_{10}

PageRank authentique									
0.066	0.078	0.078	0.099	0.078	0.099	0.078	0.151	0.195	0.078

PageRank avec la triche sur la sommet 0									
0.265	0.089	0.089	0.069	0.089	0.069	0.089	0.074	0.076	0.089

PageRank avec la triche sur la sommet 1									
0.061	**0.296**	0.071	0.081	0.071	0.081	0.071	0.095	0.101	0.071

PageRank avec la triche sur la sommet 2									
0.026	0.030	**0.343**	0.172	0.030	0.172	0.030	0.099	0.068	0.030

PageRank avec la triche sur la sommet 3									
0.035	0.041	0.041	**0.283**	0.041	0.052	0.041	0.275	0.152	0.042

PageRank avec la triche sur la sommet 4									
0.056	0.065	0.065	0.074	**0.309**	0.074	0.065	0.119	0.106	0.065

PageRank avec la triche sur la sommet 5									
0.055	0.065	0.065	0.083	0.065	**0.302**	0.065	0.126	0.109	0.065

PageRank avec la triche sur la sommet 6									
0.059	0.069	0.069	0.079	0.069	0.079	**0.313**	0.093	0.099	0.069

PageRank avec la triche sur la sommet 7									
0.021	0.025	0.025	0.028	0.025	0.028	0.025	**0.419**	0.378	0.025

PageRank avec la triche sur la sommet 8									
0.061	0.071	0.071	0.081	0.071	0.081	0.071	0.095	**0.326**	0.071

PageRank avec la triche sur la sommet 9									
0.061	0.071	0.071	0.081	0.071	0.081	0.071	0.095	0.101	**0.296**

À partir des tableaux, nous pouvons voir qu'il est bien possible de tricher pour augmenter le PageRank d'un sommet en ajoutant des sommets artificiels qui ne pointent que vers ce sommet.