TME4

April 9, 2019

Exercice 1

In [ ]:
```
import random as rdm
import networkx as nx

COLORS = ['blue','red','yellow','green']
COULEUR = [COLORS[i//100] for i in range(400)]

option = {
    'node_size' : 5,
    'node_color' : COULEUR
    }

def benchmark(nombre_clusters,taille_cluster,p,q):


    nombre_nodes = nombre_clusters * taille_cluster
    map_sommet_voisin = dict()

    for i in range(nombre_nodes):
        map_sommet_voisin[i] = list()



    G = nx.Graph()
    G.add_nodes_from([i for i in range(nombre_nodes) ])

    for i in range(nombre_nodes):
        for j in range(i+1, nombre_nodes):
            if i//taille_cluster == j//taille_cluster and rdm.random() < p:
                G.add_edge(i,j)
                map_sommet_voisin[i].append(j)
                map_sommet_voisin[j].append(i)
            if i//taille_cluster != j//taille_cluster and rdm.random() < q:
                G.add_edge(i,j)
                map_sommet_voisin[i].append(j)
                map_sommet_voisin[j].append(i)
    return [map_sommet_voisin,G]
```

```
In [ ]: l = benchmark(4, 100,0.1, 0.0)
        nx.draw(l[1],**option)
```

Pour p = 0.1 et q = 0, on remarque qu'il n'y a pas de liens entre les communautés.

```
In [ ]: l = benchmark(4, 100,0.1, 0.0005)
        nx.draw(l[1],**option)
```

Pour p = 0.1 et q = 0.005, on remarque que les liens entre les communautés sont conséquents.

```
In [ ]: l = benchmark(4, 100,0.5, 0.05)
        nx.draw(l[1],**option)
```

0.0.1 Remarques

- Pour p = 0.1 et q tend vers 0 : le rapport tend vers l'infini
- Pour p = 0.1 et q = 0.0005 : le rapport est de : 200.0
- Pour p = 0.1 et q = 0.05 : le rapport est de : 10.0

Plus p/q augmente, plus les communautés sont moins connectés, dans le pire des cas on a un rapport qui tend vers l'infini, ce qui nous donne 4 communautés sans liens communs. Dans le cas moyen, le rapport nous donne un certain nombre fini >1, ce qui nous donne un certain nombre moyen de liens entre les communautés. Et dans le cas ou p/q tend vers 1, les communautés sont hyper-connectés.

0.1 Exercice 2

Fonctions utilitaires pour l'exercice 2

```
In [ ]: # liste de sommets x map sommet label -> liste de label
        def getLabels(liste,map_labels):
            l = []
            for i in liste:
                l.append(map_labels[i])
            return l
```

```
In [ ]: # Comparaison de deux map : l'anciennne et la courante
        # map x map -> boolean
        def compare_maps(map1,map2):
            values1 =  list(map1.values())
            values2 = list(map2.values())
            if values1 ==  values2:
                return True
            return False
```

0.1.1 1. Label propagation avec un graph de 400 noeuds

```
In [ ]: import copy
        import time
        # Liste d'adjacence représentant notre graphe
```

```
    def propagation_label(map_sommet_voisin):
        now = time.time()
        # Constitution de ordre
        ordre = []
        for k in map_sommet_voisin.keys():
            ordre.append(int(k))

        # Step 1
        map_label = dict()
        map_label_precedente = dict()
        for i in list(map_sommet_voisin.keys()):
            map_label[i]=int(i)

        while(True):
            # Step 2
            rdm.shuffle(ordre)

            # Step 3
            for s in ordre:
                map_nb_occ = dict()
                for v in map_sommet_voisin[s]:
                    if not(map_label[v] in map_nb_occ.keys()):
                        map_nb_occ[map_label[v]] = 0
                    map_nb_occ[map_label[v]] = map_nb_occ[map_label[v]] + 1

                label = max(map_nb_occ, key=map_nb_occ.get)
                map_label[s] = label
                label = None
            print("tour")
            # Step 4
            if(compare_maps(map_label,map_label_precedente)):
                break;
            else:
                map_label_precedente = copy.deepcopy(map_label)
        after = time.time()
        # map_community : label_community -> nombre de noeud
        map_community = dict()
        for s in list(map_label.values()):
            if(s in map_community.keys()):
                map_community[s] = map_community[s] + 1
            else:
                map_community[s] = 1
        return [len(map_community.keys()),after-now,map_label]

In [ ]: def tranform_to_community(map_label):
        communities = dict()
        l = list(map_label.values())
        for i in l :
```

```
            communities[i] =  list()
        for k,v in map_label.items():
            communities[v].append(str(k))
        return communities
```

```
In [ ]: l = benchmark(4, 100,0.1, 0.0005)
        propagation_label(l[0])
```

0.1.2 2. Label propagation avec le graph Youtube

a. Construction de la liste d'adjacence

```
In [ ]: import copy
```

```
        # Structure de données: Liste d'adjacence
        def read_file(nom_fichier,separator):
            # Dans le graph, trouver le sommet d'indice maximum
            cleaned_graph = open(nom_fichier, "r")
            max_int = -1
            G = nx.Graph()
            for e in cleaned_graph:
                if e.startswith("#"):
                    continue
                e = e.rstrip("\n")
                indice_point = e.split(separator)
                if int(indice_point[0])>max_int:
                    max_int = int(indice_point[0])
                if int(indice_point[1])>max_int:
                    max_int = int(indice_point[1])
            cleaned_graph.close()


            cleaned_graph = open(nom_fichier, "r")
            map_sommet_voisin_no_filtred = dict()
            for i in range(max_int+1):
                map_sommet_voisin_no_filtred[i] = list()
            for e in cleaned_graph:
                if e.startswith("#"):
                    continue
                edge_str = e.rstrip("\n")
                edge = edge_str.split(separator)
                if(not(int(edge[1]) in map_sommet_voisin_no_filtred[int(edge[0])])):
                    G.add_edge(int(edge[0]),int(edge[1]))
                    map_sommet_voisin_no_filtred[int(edge[0])].append(int(edge[1]))
                if(not(int(edge[0]) in map_sommet_voisin_no_filtred[int(edge[1])])):
                    map_sommet_voisin_no_filtred[int(edge[1])].append(int(edge[0]))
                    G.add_edge(int(edge[1]),int(edge[0]))

            cleaned_graph.close()
```

```
    # Les sommets sans voisins ne sont pas comptés dans une communauté.
    # On filtre notre map
    map_sommet_voisin = dict()
    for k in map_sommet_voisin_no_filtred.keys():
        if len(map_sommet_voisin_no_filtred[k]) == 0:
            continue
        map_sommet_voisin[k] = map_sommet_voisin_no_filtred[k]

    print("Nombre de sommet : " + str(len(list(map_sommet_voisin.keys()))))
    return [map_sommet_voisin,G]
```

b. Lancement du label propagation Nous allons executer l'algorithme de propagation de label sur le graphe de Youtube et former un dictionnaire formé de numeros de communautés comme cléfs et pour valeur nombre de membres de cette derniere.

```
In [ ]: map_sommet_voisin = read_file("com-youtube.ungraph.txt/data.txt","\t")[0]
```

```
In [ ]: # Constitution de ordre
        ordre = []
        for k in map_sommet_voisin.keys():
            ordre.append(int(k))

        # Step 1
        map_label = dict()
        map_label_precedente = dict()
        for i in list(map_sommet_voisin.keys()):
            map_label[i]=int(i)

        while(True):
            # Step 2
            rdm.shuffle(ordre)

            # Step 3
            for s in ordre:
                map_nb_occ = dict()
                for v in map_sommet_voisin[s]:
                    if not(map_label[v] in map_nb_occ.keys()):
                        map_nb_occ[map_label[v]] = 0
                    map_nb_occ[map_label[v]] = map_nb_occ[map_label[v]] + 1

                label = max(map_nb_occ, key=map_nb_occ.get)
                map_label[s] = label
                label = None

            print("tour")
```

```
            # Step 4
            if(compare_maps(map_label,map_label_precedente)):
                break;
            else:
                map_label_precedente = copy.deepcopy(map_label)

        # map_community : label_community -> nombre de noeud
        map_community = dict()
        for s in map_label.values():
            if(s in map_community.keys()):
                map_community[s] = map_community[s] + 1
            else:
                map_community[s] = 1

        print("Nombre de communauté : " + str(len(set(map_community.keys()))))
```

In [ ]:
```
        import matplotlib.pyplot as plt
        x = list(map_community.values())
        bin_edges =range(101)
        plt.hist(x,bin_edges)
        plt.show()
```

In [ ]:
```
        # Formation d'un dictionnaire avec 10 decompositions de communautés
        # Constitution de ordre
        ordre = []
        for k in map_sommet_voisin.keys():
            ordre.append(int(k))
        map_longueur = dict()
        j = 0
        while (j!= 10):
            # Step 1
            map_label = dict()
            map_label_precedente = dict()
            for i in list(map_sommet_voisin.keys()):
                map_label[i]=int(i)

            while(True):
                # Step 2
                rdm.shuffle(ordre)

                # Step 3
                for s in ordre:
                    map_nb_occ = dict()
                    for v in map_sommet_voisin[s]:
                        if not(map_label[v] in map_nb_occ.keys()):
                            map_nb_occ[map_label[v]] = 0
                        map_nb_occ[map_label[v]] = map_nb_occ[map_label[v]] + 1
```

```
                label = max(map_nb_occ, key=map_nb_occ.get)
                map_label[s] = label
                label = None

            print("tour")

            # Step 4
            if(compare_maps(map_label,map_label_precedente)):
                break;
            else:
                map_label_precedente = copy.deepcopy(map_label)

        # map_community : label_community -> nombre de noeud
        map_community = dict()
        for s in map_label.values():
            if(s in map_community.keys()):
                map_community[s] = map_community[s] + 1
            else:
                map_community[s] = 1
        j = j + 1

        map_longueur[j] =  len(set(map_community.keys()))
        print(j)
```

```
In [ ]: import matplotlib.pyplot as plt
        x = list(map_longueur.values())
        bin_edges = x
        plt.hist(x)
        plt.show()
```

On a malheureusement pas pu executé 1000 fois l'algorithme donc on l'a seulement fait 10 fois, on obtient donc 10 nombres de communautés. L'intervalle 38000 - 42000 est la plus large donc c'est celle qui comporte le nombre de communautés en moyenne, le graphe de youtube etant un graphe de terrain et comme expliqué en tme, le nombre de communautés qu'il a réellement est different que celui calculé ici, le nombre de communauté dépend de la qualité du shuffle qu'on fait à chaque fois.

0.2 Exercice 3

0.2.1 1. Alogrithme de Louvain

a. Exemple de l'énoncé

```
In [ ]: import community
        import networkx as nx
        import matplotlib.pyplot as plt
        import time

        def louvain(G):
```

```
        now = time.time()
        #first compute the best partition
        partition = community.best_partition(G)
        after = time.time()
        #print("haha")

        #drawing
        size = float(len(set(partition.values())))
        pos = nx.spring_layout(G)
        count = 0.
        communities = dict()
        for com in set(partition.values()) :

            list_nodes = [str(nodes) for nodes in partition.keys()
                                        if partition[nodes] == com]
            communities[count] =  list_nodes

            count = count + 1.
            print(count)


        return [[count,after-now],communities]
```

0.3 Evaluation de la rapidité d'execution des 2 algorithmes:

0.3.1 Etude expérimentale:

Nous allons tester les implémentations de l'algorithme de louvain et de propagation de label sur différents jeux de données generées grace à notre fonction de benchmark utilisée à l'exercice 1. [400,1000,1500,2000] Les 4 communautés ne se chevauchent pas et on peut les detecter intuitivement. Sachant que la métrique utilisée pour l'algorithme de louvain est la modularité de partition.

```
In [ ]: #lecture des graphes en mode dicos
        l_1 =  benchmark(4,500,0.1, 0.0005)
        map_sommet_voisin_1 = l_1[0]
        G_1 =  l_1[1]

In [ ]: print(propagation_label(map_sommet_voisin_1))

In [ ]: print(louvain(G_1))

In [ ]: [4, 0.07597064971923828] [4, 0.1759326457977295] [4, 0.4238443374633789] [4, 0.4918131
        [4.0, 3.848557472229004] [4.0, 30.23245906829834] [4.0, 53.572322368621826] [4.0, 85.5

In [ ]: x = [400,1000,1500,2000]
        y_label_propagation =  [0.07597064971923828,0.1759326457977295,0.4238443374633789,0.49
        y_louvain = [3.848557472229004, 30.23245906829834,53.572322368621826,85.50057578086853
        plt.plot(x,y_label_propagation)
        plt.plot(x,y_louvain)
```

```
In [ ]: plt.show()
```

La courbe bleu représente le temps d'execution en fonction de la taille du graph en utilisant l'algorithme de propagation de label et la courbe orange représente l'execution en utilisant l'algorithme de louvain. On remarque très bien que l'algorithme de louvain est plus lent à s'exécuter que celui de propagation de label.

0.4 Evaluation de la précision des 2 algorithmes :

0.4.1 Utilisation du benchmark du 1er exercice :

Nous allons maintenant tester nos deux algorithmes sur des graphes à partir de notre benchmark du 1er exercice afin d'évaluer leur précision et les comparer en utilisant la metrique NMI abordée en cours.

```
In [ ]: l_1  = benchmark(4,210,0.1, 0.0005)
```

```
In [ ]: coo = propagation_label(l_1[0])
        go = tranform_to_community(coo[2])
```

0.4.2 Metrique NMI

```
In [ ]: network = read_file("package1/binary_networks/network.dat","\t")
        #louvaned_graph = louvain(network[1])
        propaged_graph = tranform_to_community(propagation_label(network[0])[2])
```

```
In [ ]: f2 = open("file2_1","w+")
        for k,v in propaged_graph.items():
            ch = " ".join(v)
            f2.write(ch+'\n')
        f2.close()
```

```
In [ ]: def convert_to_communities(file,separator):
            cleaned_graph = open(file, "r")
            communities  = dict()
            for e in cleaned_graph:
                e = e.rstrip("\n")
                indice_point = e.split(separator)
                deuxieme_partie = indice_point[1].split("\t")
                #deuxieme_partie.remove("") #seuelemnt pour LFR Benchmark
                if len(deuxieme_partie) > 1:
                    for v in deuxieme_partie:
                        #print(int(v))
                        v  =  int(v)
                        if not(v in list(communities.keys())):
                            communities[v]  = list()
                        communities[v].append(indice_point[0])
                else:
                        deuxieme_partie = int(deuxieme_partie[0])
```

```
                    if not(deuxieme_partie in list(communities.keys())):
                        communities[deuxieme_partie]  = list()
                    communities[deuxieme_partie].append(indice_point[0])
        return communities
```

```
In [ ]: oups = convert_to_communities("package1/binary_networks/community.dat","\t")
```

```
In [ ]: f = open("file1","w+")
        for k,v in oups.items():
            #print(v)
            ch = " ".join(v)
            f.write(ch+'\n')
        f.close()
```

```
In [ ]: print(propagation_label(network[0]))
```

NMI entre fichier LFR benchmark et fichier louvain: NMI: 0.426303 Other measures: lfkNMI: 0.548187 NMI: 0.513461 NMI entre fichier LFR benchmark et fichier label propagation: NMI: 0.474214 Other measures: lfkNMI: 0.585301 NMI: 0.563418

0.4.3 Utilisation du fichier Youtube

```
In [ ]: f = open("modisco_louvain-master/graph_youtube.tree","r")
        map_community_louvain_youtube = dict()
        for e in f:
            e = e.rstrip("\n")
            coordonnee = e.split("\t")
            map_community_louvain_youtube[coordonnee[0]] = coordonnee[1]
        f.close()
```

```
In [ ]: oups1 = tranform_to_community(map_community_louvain_youtube)
```

```
In [ ]: # le fichier genere de communautes avec louvain
        f_youtube = open("youtube_louvain","w+")
        for k,v in oups1.items():
            ch = " ".join(v)
            f_youtube.write(ch+'\n')
        f_youtube.close()
```

```
In [ ]: youtube = read_file("modisco_louvain-master/data.txt","\t" )[0]
```

```
In [ ]: map_label_youtube = propagation_label(youtube)[2]
```

```
In [ ]: oups2 = tranform_to_community(map_label_youtube)
        f_youtube_label_propagation = open("youtube_label_propagation","w+")
        for k,v in oups2.items():
            ch = " ".join(v)
            f_youtube_label_propagation.write(ch+'\n')
        f_youtube_label_propagation.close()
```

```
In [ ]: community_youtube_truth = convert_to_communities("com-youtube.all.cmty.txt","r")
```