Homework assignment 4

Author: Vendramin Nicolò

The task has been fulfilled with a python script containing the functions explained below. The implementation follows the steps illustrated by the paper.

A. Function - "import_graph":

```python
def import_graph(input_file):
    file = open(input_file, "rU")
    line = file.readline()
    first_edges = []
    second_edges = []
    affinity_scores = []
    edge_list = []
    affinity_dictionary = {}

    while(len(line)>0):
        infos = line.split(",")
        edge_list.append((int(infos[0]),int(infos[1])))

        first_edges.append(int(infos[0])-1)
        second_edges.append(int(infos[1])-1)
        if len(infos) > 2:
            affinity_scores.append(int(infos[2]))
        else:
            affinity_scores.append(1)

        line = file.readline()

    affinity_matrix = csr_matrix((affinity_scores, (first_edges, second_edges)))

    return edge_list, affinity_matrix
```

This function takes as input the path of an input file containing the affinity matrix of the graph to be analysed, stored as a multiple lines each one containing an edge and their affinity value. After opening the file and initialising the variables the content of the file is slid line by line and the values of the vertices of each edge along with their affinity is stored in three local lists. At the end of this procedure those three lists are used to initialise a sparse matrix containing the affinity of each pair of edges. The matrix would be symmetric and that's way only the upper right triangle is kept without storing also the symmetric double.

The function returns the list of the edges and the affinity matrix.

B. Function - "spectral_clusering":

This function embodies the main core of the algorithm and represent the python implementation of the six steps described in the section 2, "Algorithm" of the paper authored by Ng et Al..
The first step is to get the affinity matrix that in this function is provided as input, along with the number k of clusters that we want to identify.
We form the diagonal matrix D by reducing each row to the sum of its element, so that the only non zero element of the matrix D are the element d_{ii} = $\text{sum}_j (a_{ij})$.
After this step we build the matrix L as a result of the matrix multiplication between $D^{-1/2} A D^{-1/2}$ and the eigenvectors and eigenvalues of that matrix are extracted and the k eigenvectors with bigger eigenvalues are stacked in the matrix X that is right after normalised to have unit norm over each row to form the matrix Y.

Finally the KMeans clustering technique from Scikit Learn is applied to the matrix Y and to each row of the matrix Y, corresponding to an edge of the graph is assigned to a cluster according to the clustering outputted by the application of KMeans. For this reason the list of labels is returned by the method.

```python
def spectral_clustering(affinity_matrix, k=5):
    d_elements = np.ravel(affinity_matrix.sum(axis=1))
    d_matrix = csr_matrix((d_elements, (np.arange(0, d_elements.shape[0]), (np.arange(0, d_elements.shape[0])))))

    d_neg_sqrt = fractional_matrix_power(d_matrix.todense(), -0.5)
    L = d_neg_sqrt.dot(affinity_matrix.todense()).dot(d_neg_sqrt)

    w, v = np.linalg.eig(L)
    top_eigenvalues_indices = np.argsort(-w)[:k]

    x = w[top_eigenvalues_indices[0]]*v[:,top_eigenvalues_indices[0]]

    for index in top_eigenvalues_indices[1:]:
        x = np.append(x, w[index]*v[:,index], 1)

    factors = []
    for row in x:
        factor = np.power(row, 2).sum()
        factors.append(1/m.sqrt(factor))

    y = np.diag(np.asarray(factors)) * x

    kmeans = KMeans(n_clusters=k, random_state=0).fit(y)

    return kmeans.labels_
```

C. Function - “graph_plot”:

```python
def graph_plot(graph, (clusters, k)=(None, 1)):
    G = nx.from_edgelist(graph)
    plt.figure(1)
    values = []
    if clusters != None:
        for node in G.nodes():
            values.append(clusters.get(int(node), k+1))
        nx.draw(G, cmap=plt.get_cmap('jet'), node_color=values, node_size=80, with_labels=False)
    else:
        nx.draw_networkx(G,node_color='b', node_size=80, with_labels=False)
    plt.show()
```

This utility function is just a method that receives an edge list as an input and optionally a clustering dictionary along with the number of clusters, and prints the graph, using colours in case the clustering is provided. The library networkx and matplotlib are employed to represent and plot the graph.

D. Function - “main”:

```python
def main():
    edge_list, affinity_matrix = import_graph(args.input_file)
    clusters = spectral_clustering(affinity_matrix, args.clusters)
    print(clusters)
    cl = {i+1: clusters[i] for i in range(0, clusters.shape[0])}
    print(cl)
    if args.plot == "normal":
        graph_plot(edge_list)
    elif args.plot == "clustered":
        graph_plot(edge_list, cl, args.clusters)
    elif args.plot == "all":
        graph_plot(edge_list)
        graph_plot(edge_list, (cl, args.clusters))
    else:
        print("Plot option not supported. Try [\"normal\", \"clustered\", \"all\"].")
```

This function just calls the different methods according to the input parameters to import the affinity matrix, perform the clustering applying the spectral clustering algorithm and finally plotting the graph, before, after or both the application of the clustering, in order to be able to visually check the results and to make sure that the algorithm correctly identified at least the clusters that are easy to spot at first sight.

E. Imports and usage:

```python
import numpy as np
from progressbar import ProgressBar
from scipy.sparse import csr_matrix
from scipy.linalg import fractional_matrix_power
import time
import math as m
import argparse
from sklearn.cluster import KMeans
import networkx as nx
import matplotlib.pyplot as plt

parser = argparse.ArgumentParser()
parser.add_argument('--clusters', type=int, default=4)
parser.add_argument('--input_file', type=str, default="real_graph.txt")
parser.add_argument('--plot', type=str, default="all")

args = parser.parse_args()
```

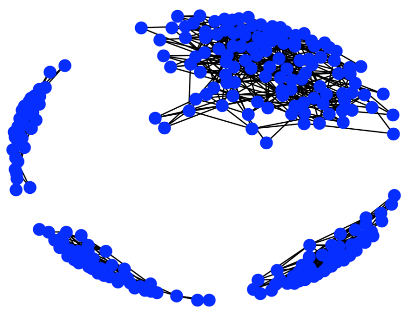
To run the code is just necessary to set up an environment with Python 3.5, install the progressbar, argparse, numpy, scikit-learn, networkx, matplotlib and scipy external libraries, open a console in the folder of the project and run the command:

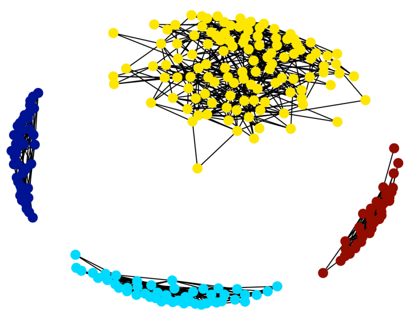
"python spectral_clustering.py"

With the following argument options:
- —clusters: is an integer value expressing the desired number of clusters to identify in the graph. Default value is 4.
- —input_file: is a string indicating the relative path to the input file. Default is "real_graph.txt".
- —plot: is s string value expressing the plot options desired to show the obtained results. Three values are available ["all", "clustered", "normal"]. Default is "all".

F. Analysis of the results:

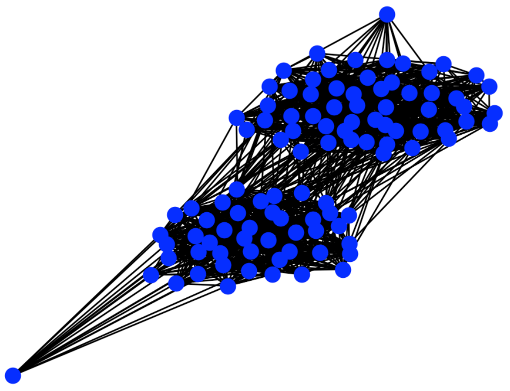
Below I attach the image of the application of the technique to the two graphs that have been provided together with the assignment. It's clear that the techniques work properly, being able to identify correctly the main clusters present in the graph.

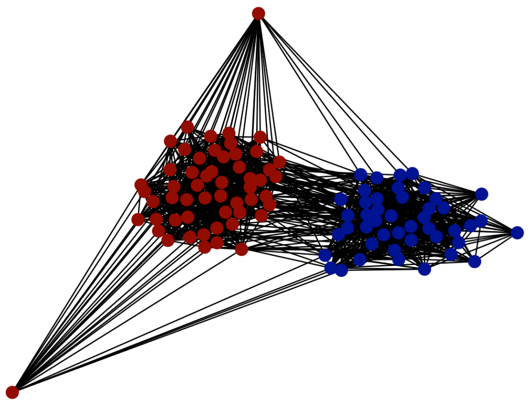




The two plots above have been obtained by running the script on the real graph (—input_file = "real_graph.txt") and isolating 4 clusters (—clusters = 4). It's evident that the technique effectively identifies the 4 clusters and separates them in a really clear and distinct way.

The two representations reported below instead show the application of the script on the syntethic graph stored in the “syntethic_graph.txt” file with a cluster number of 2.





Again the plots seem to confirm the goodness and the precision of the clustering technique exposed in the paper.