# Лабораторная работа №5

## Спектральная теория графов

### Будем анализировать графы с помощью матриц

Предмет: Практическая линейная алгебра

*Автор: Made by Polyakov Anton, the part of R3236, suir family*

Преподаватель: Алексей Алексеевич Перегудин

Исходный код, репозиторий

## Навигация



## Вступление, приветствие

Всем привет, это моя лаба. Делал в VS code + jupyter notebook, экспортом файлов испытывал дикие проблемы, поэтому будьте осторожны, если захотите нативно отчёты в .ipynb делать

# Вспомогательные функции и библиотеки, их описание

## Задание 1

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import networkx as nx # для построения графов

from numpy import linalg as LA # пакет линейной алгебры
from sklearn.cluster import KMeans # Для машинного обучения, отсюда берём k-means
from random import randint

def printBeauty(matrix, Type):
    A = pd.DataFrame(matrix.astype(Type))
    A.columns = ['']*A.shape[1]
    print(A.to_string(index=False))

def findMinIdx(n,matrix):
    result = []
    for i in range(n):
        elem = np.amin(matrix)
        i = np.where(np.isclose(matrix, elem))
        result.append(i)
        matrix[i]= 100000
    return result

def generateRandomHexColor():
    r = lambda: randint(0,255)
    return '#%02X%02X%02X' % (r(),r(),r())

def plotBeautyGraph(k, G, claster_points):
    # Подготавливаем цвета для узлов
    color_map = []
    colors = dict()

    s = set(list(claster_points))
    for el in s:
```

```
        colors.update({el:generateRandomHexColor()}) # "уникальные зоны"
    # Красим и выводим
    for node in G:
        color_map.append(colors[claster_points[node]])

    plt.title(f"Friends graph with k={k}")
    nx.draw_spring(G, node_color=color_map, node_size=200, font_size=14, with_labels=True)
    plt.show()

def doTask1(k, G, minVectors):
    km = KMeans(n_clusters = k, random_state = 0, n_init='auto')
    model = km.fit(minVectors)
    groups = model.labels_
    plotBeautyGraph(k, G, groups)
```

## Задание 2

In [ ]:

```
def calculateM(G):
    M = []

    for i in range(0,15+1):
        for j in range(0,15+1):
            x = 0
            # ищем числитель из m_ij
            for edge in G.out_edges(j):
                if edge[1] == i:
                    x+=1
            # ищем знаменатель из m_ij
            y = len(G.out_edges(j))
            M.append(x/y)
    # округлим, чтобы не тащить тяжёлые дроби
    M = np.round(np.reshape(np.array(M), (16,16)),3)
    return M

# imported from wiki
def pagerank(M, num_iterations: int = 100, d: float = 1):
    """PageRank algorithm with explicit number of iterations. Returns ranking of nodes (pages) in the adjacency matrix.
```

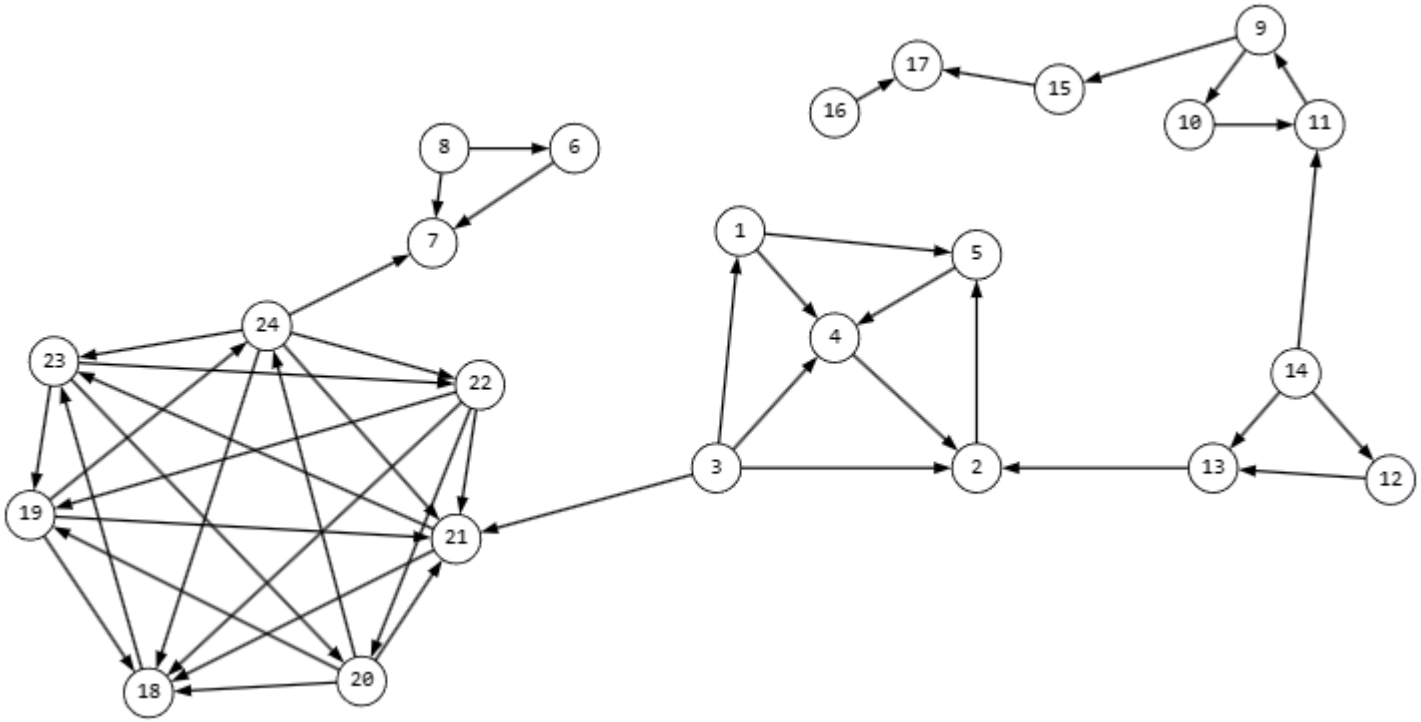
```
    Parameters
    ----------
    M : numpy array
        adjacency matrix where M_i,j represents the link from 'j' to 'i', such that for all 'j'
        sum(i, M_i,j) = 1
    num_iterations : int, optional
        number of iterations, by default 100
    d : float, optional
        damping factor, by default 0.85

    Returns
    -------
    numpy array
        a vector of ranks such that v_i is the i-th rank from [0, 1],
        v sums to 1

    """
    N = M.shape[1]
    v = np.ones(N) / N
    M_hat = (d * M + (1 - d) / N)
    for i in range(num_iterations):
        v = M_hat @ v
    return v
```

# Содержательная часть

## Задание 1. Кластеризация социальной сети

Задача этого задания состоит в том, чтобы выдеолить в неориентированной графе "социальных отношений" кластеры - сообществ, которые в большей степени дружат внутри себя, чем с другими людьми

Придумаем связный граф с помощью этого сервиса, в нём создадим неориентированный граф, потом с помощью "волшебной палочки" мы можем дать случайную ориентацию всем рёбрам, это нам ещё пригодится при создании Лапласиана графа. Там же можно получить матрицу инцидентности...




Получим Лапласиан неориетированного графа с помощью следующей формулы: $L = B * B^T$

In [ ]:
```
inc = np.matrix(np.loadtxt('graph1_data.txt')) # скопировали матрицу инцидентности с сайта, теперь обработаем её здесь
L = inc*(inc.transpose())
printBeauty(L, int)
```

```
 3  0 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  4 -1 -1 -1  0  0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0
-1 -1  4 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0  0
-1 -1 -1  4 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
-1 -1  0 -1  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  2 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0 -1  3 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1
 0  0  0  0  0 -1 -1  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  3 -1 -1  0  0  0 -1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0 -1  2 -1  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0 -1 -1  3  0  0 -1  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  2 -1 -1  0  0  0  0  0  0  0  0  0
 0 -1  0  0  0  0  0  0  0  0  0 -1  3 -1  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0 -1 -1 -1  3  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0 -1  0  0  0  0  0  2  0 -1  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
 0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1  2  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  6 -1 -1 -1 -1 -1
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  6 -1 -1 -1 -1
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1  6 -1 -1 -1
 0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1  7 -1 -1
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1  6 -1
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1  6 -1
 0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1 -1  7
```

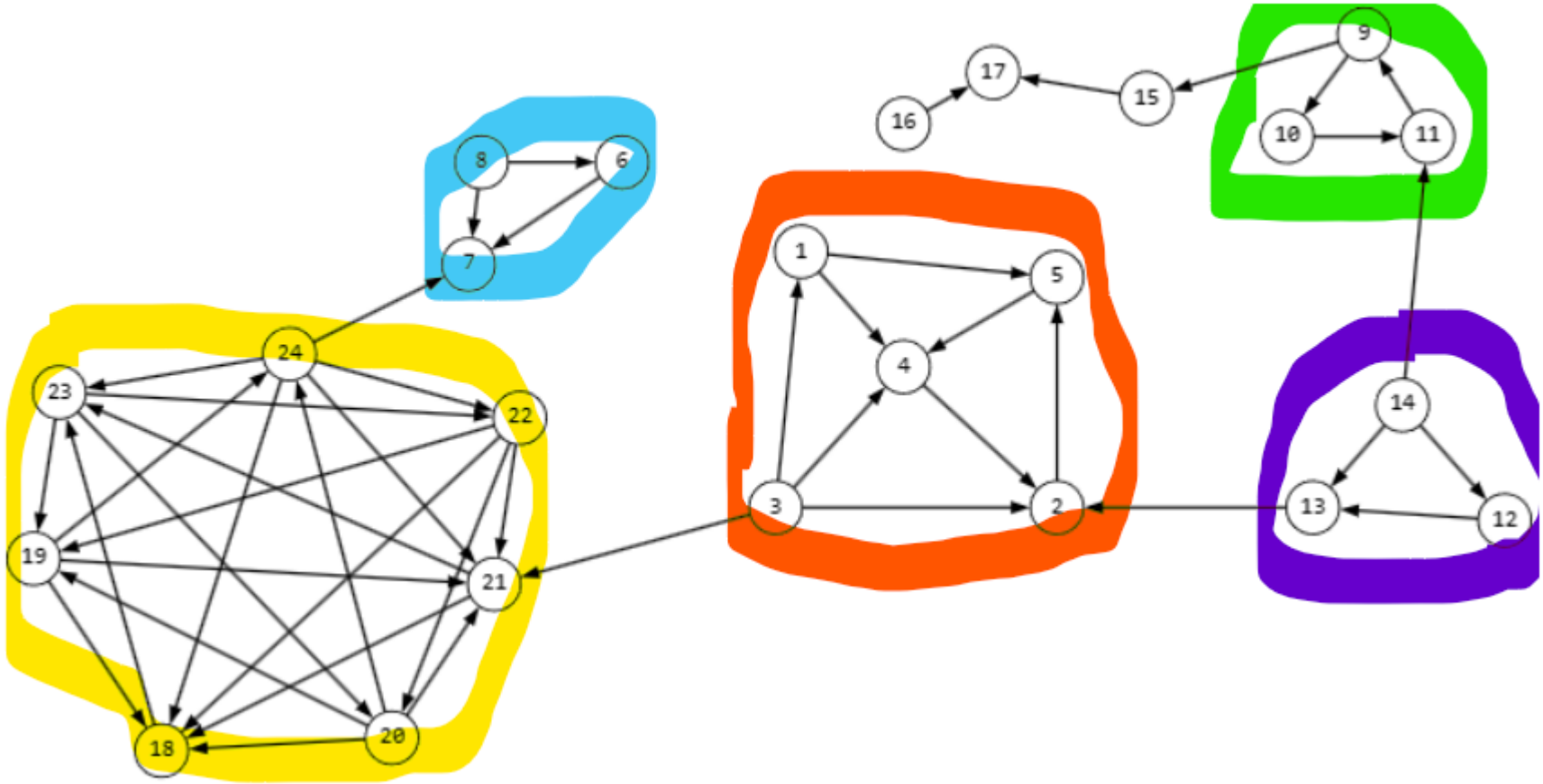
Определим собственные числа и векторы матрицы выше, их вышло довольно много (матрица 24x24 всё-таки), поэтому ограничимся выводом только собственных чисел. Заметим, что второе собственное число лапласиана далеко от нуля, что свидетельствует о хорошей связности графа, ведь в нашем случае он связный

BTW, $\lambda_2 > 0$ - отвечает за алгебраическую связность

In [ ]:
```
eigValuesL, eigVectorsL = LA.eig(L)
list(np.ndarray.round(eigValuesL,2))
```

Out[ ]:
```
[8.24,
 7.94,
 0.0,
 0.05,
 0.18,
 0.34,
 0.44,
 0.72,
 1.51,
 5.73,
 5.13,
 4.71,
 4.13,
 3.9,
 3.37,
 3.6,
 3.0,
 7.0,
 7.0,
 3.0,
 3.0,
 3.0,
 7.0,
 7.0]
```

Теперь выберем число k желаемых компонент кластеризации графа - так как граф небольшой, то на глаз можно понять, что должно быть около 5 кластеров




In [ ]:
```
k=3
```

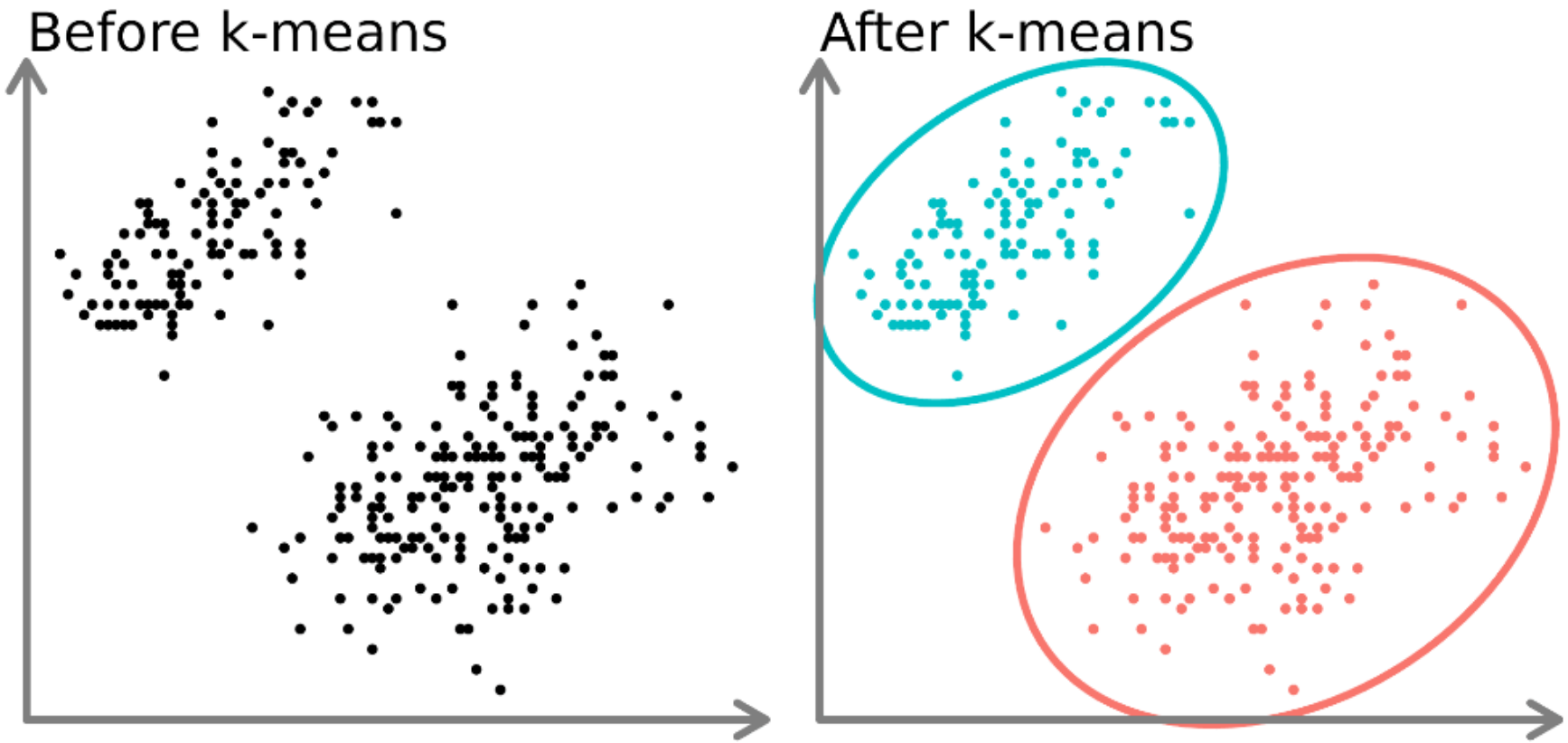
Возьмём k собственных векторов $v_1, \ldots, v_k$ матрицы Лапласа, соответствующих самым маленьким собственным числам, и составим из них матрицу:

$$\begin{bmatrix} | & & | \\ v_1 & \ldots & v_k \\ | & & | \end{bmatrix}$$

Ширина этой матрицы будет равна количеству компонент кластеризации $k$, а высота – числу вершин графа $n$.

In [ ]:
```
# находим пять минимальных собственных значений
min_indices = np.argpartition(eigValuesL, 5)[:5]
# матрица из 5 собственных векторов
# asarray используем, чтобы под аргументы функции k-means подогнать тип данных
min5Vectors = np.asarray(np.column_stack([eigVectorsL[:,[i]] for i in min_indices]))
```

Рассмотрим строки составленной матрицы $V$ как точки пространства $\mathbb{R}^k$. Применим к этим точкам метод кластеризации k-means для разбиения их на k кластеров, простой графический пример что должно произойти ниже...

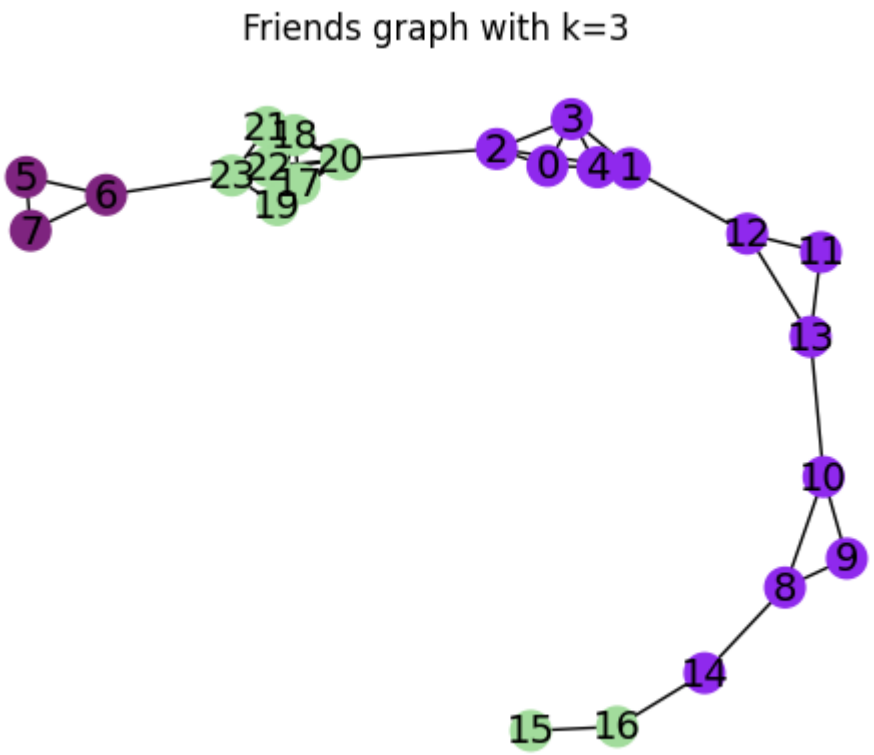



In [ ]:
```
# init - какие начальные центроиды выбирать
# init='k-means++', по-умолчанию, тогда используется метод k-means++,
# у которого уже свой алгоритм выбора центроид
```

```
# можно также сделать init = 'random', тогда центроиды - произвольные

# random_state - параметр для генерации случайных чисел внутри библиотеки

# n_init = 'auto' - при  таком параметре количество запусков опять будет
# определяться k-means методом, каждый запуск будет с разными  "centroid seeds"
# Лучший запуск определяется "инерционностью"
km = KMeans(n_clusters = k, random_state = 0, n_init='auto')
model = km.fit(min5Vectors)
groups = model.labels_
```

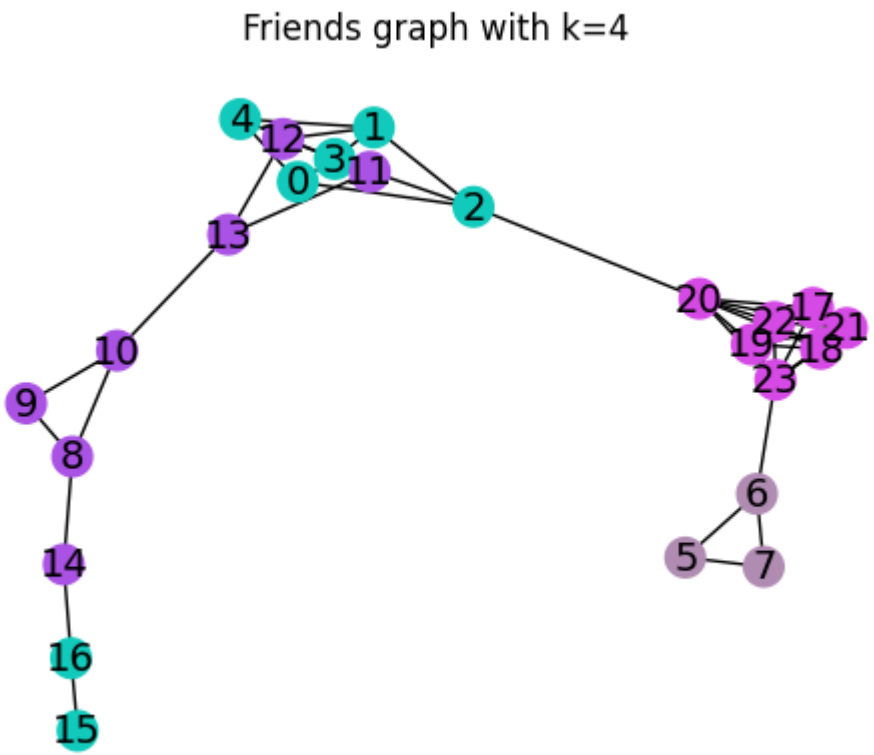
In [ ]:

```
# С помощью таблицы смежности создадим изначальный граф для библиотеки-визуализатора
data = np.loadtxt("adjadency_matrix.txt")
G= nx.from_numpy_array(data) # G = graph

plotBeautyGraph(k, G, groups) # все подробности функции в описании, в начале...
```

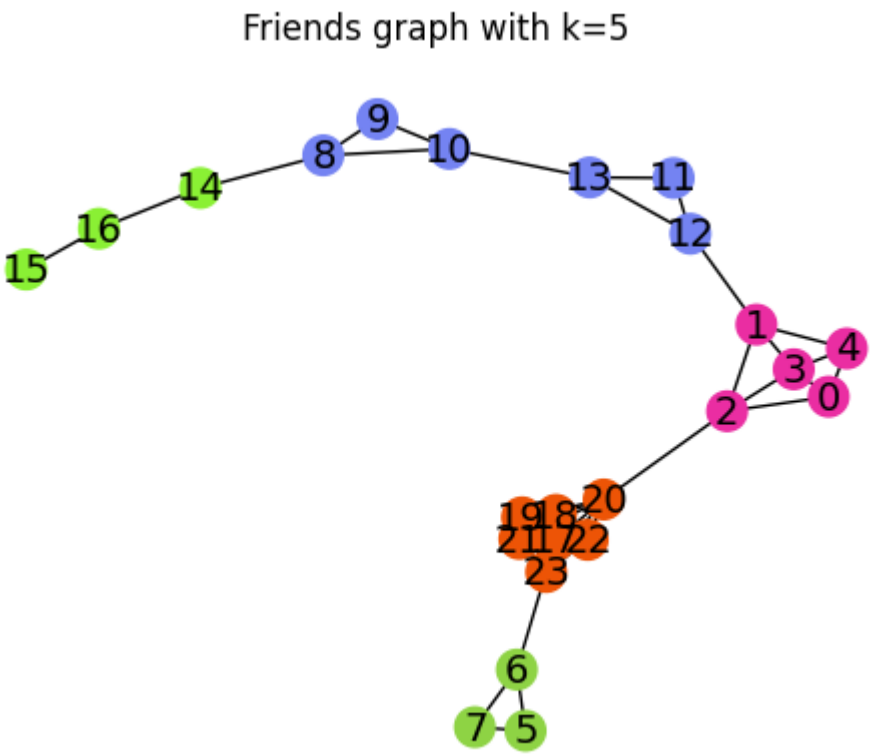



Попробуем проделать то же самое, но уже для других $k$. Упакуем все действия, связанные с методом $k-means$ и рисованием графа в общую функцию, результаты ниже...

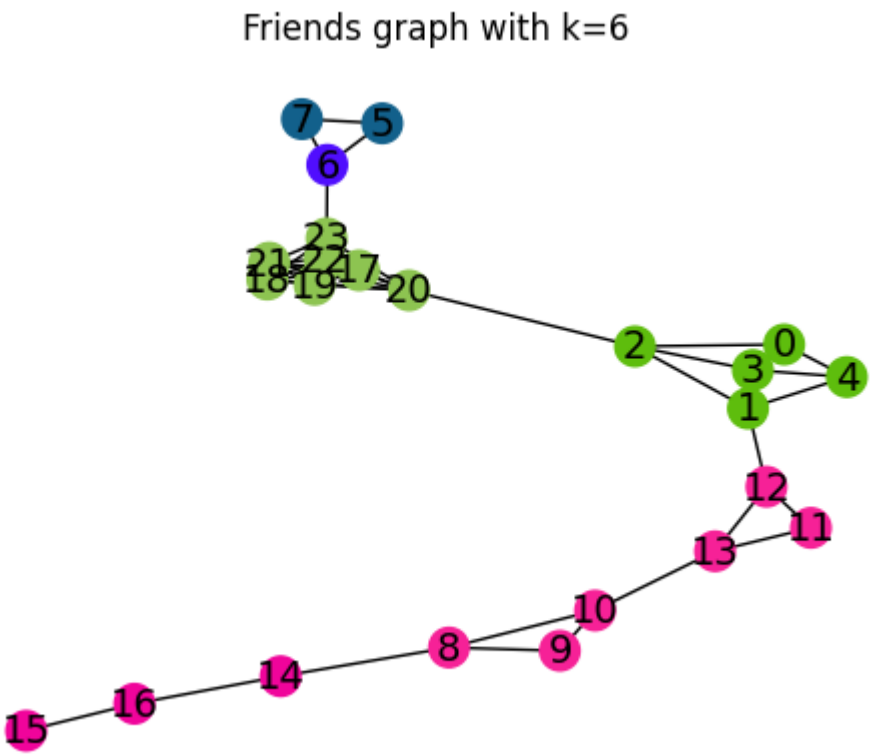
In [ ]:
```
doTask1(4, G, min5Vectors)
```




In [ ]:
```
doTask1(5, G, min5Vectors)
```

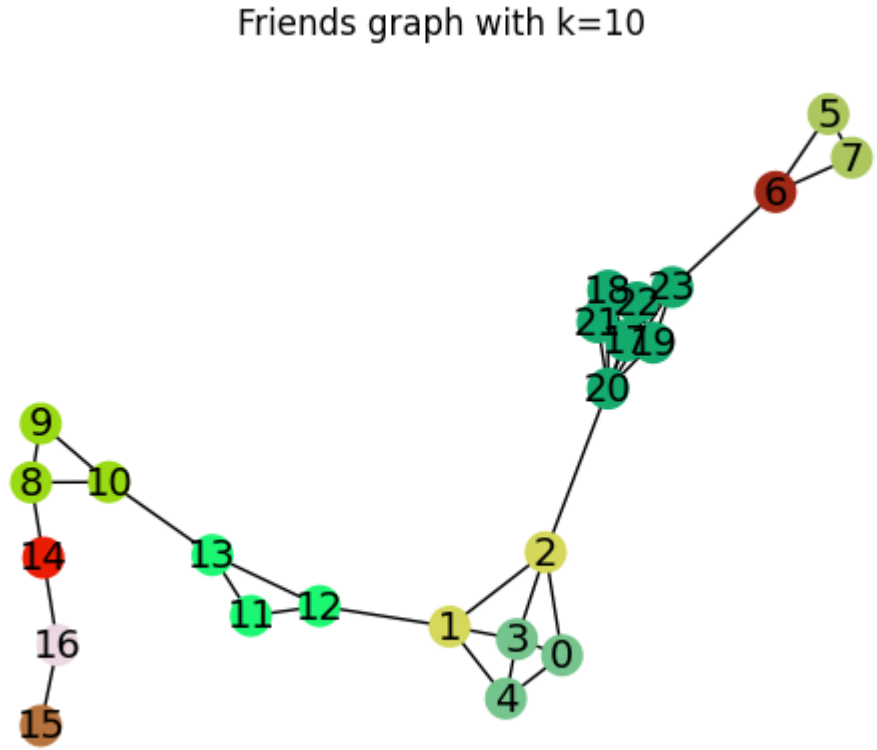



In [ ]:
```
doTask1(6, G, min5Vectors)
```

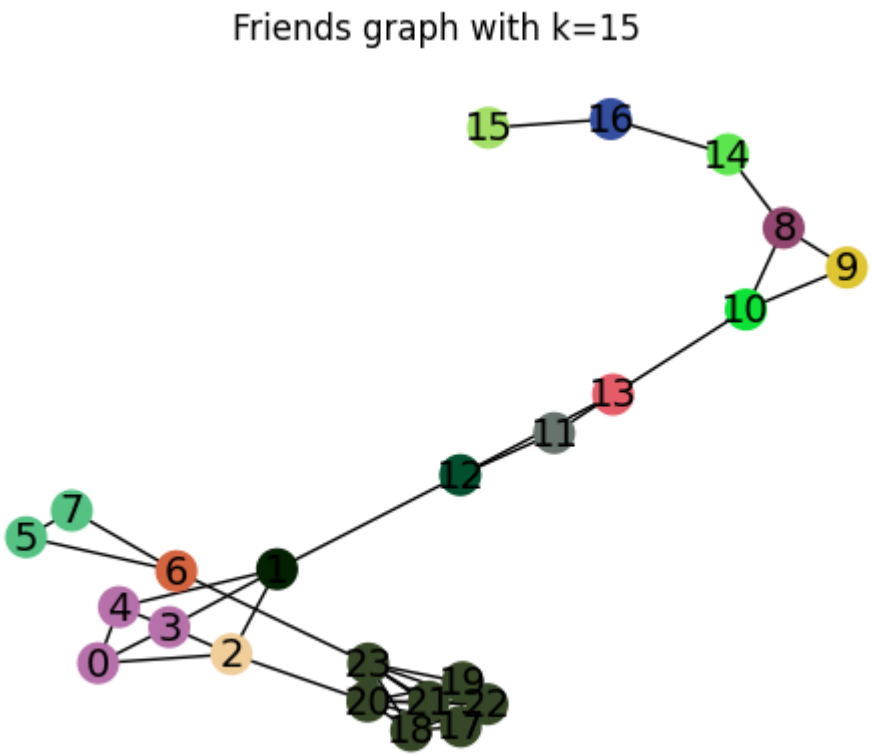



Для небольших $k$ алгоритм делит граф вполне очевидно - ближайшие связи становятся товарищами по группе, но что будет при $k$ побольше?

In [ ]:
```
doTask1(10, G, min5Vectors)
```




In [ ]:
```
doTask1(15, G, min5Vectors)
```




Вот тут и начинаются проблемы, потому что алгоритм уже непонятными путями выбирает кластеры, давайте копнём немного в теорию...

### Почему k-means сработал в нашем случае?

Мы положили в этот метод матрицу некоторых собственных векторов лапласиана, причём эти вектора соответствовали min собственным числам, и брали мы такое их количество, равное количеству кластеров, на которое мы хотели поделить наш граф. Это важно тоже учитывать... Сработал этот метод потому, что собственные вектора несли в себе характеристику некоторых локальных "центров" графа, а метод уже довёл их до кластеров, не будем слишком сильно углубляться в особенности метода *k-means*, но поймём, что этот ролик неплохо визуализирует общий подход семейства таких методов, а также - алгоритм зависит от входного *k*, под которое он пытается подобрать лучшую по его мнению кластеризацию...

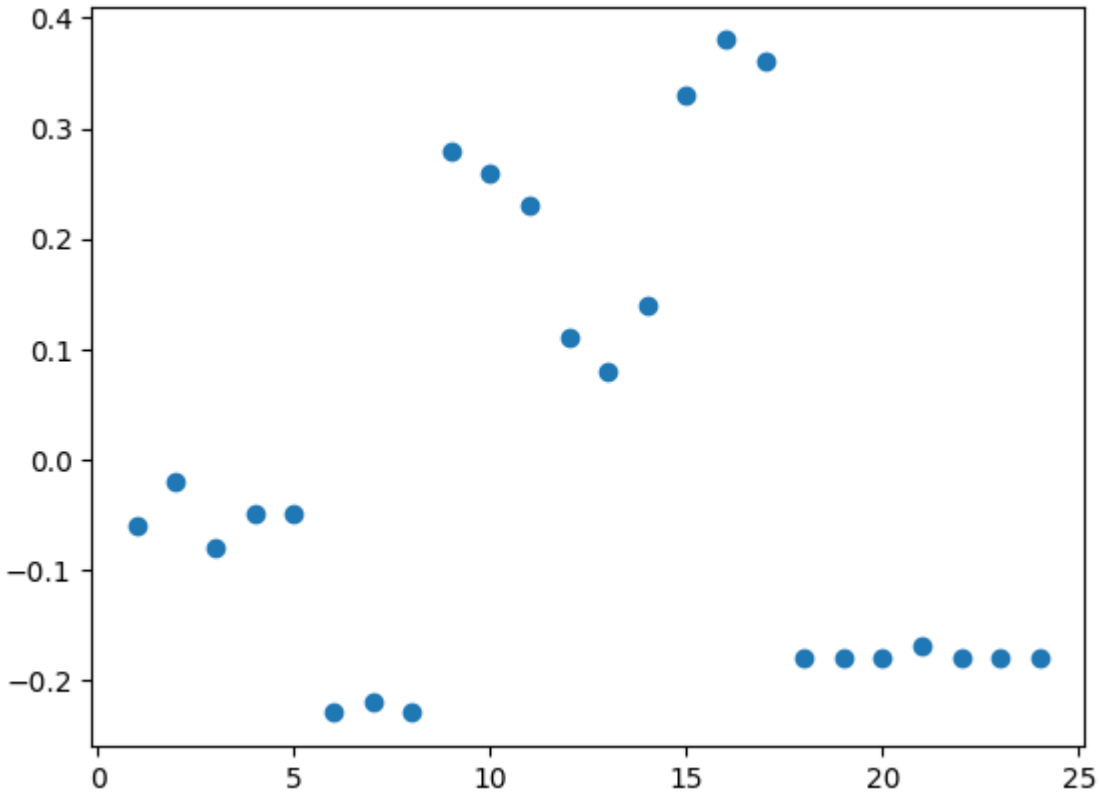
## Кратко про *k-means* ?

Выбираем *k* центров искомых кластеров (случайные точки, не обязательно из набора вершин, можно и по-другому выбирать), затем для каждой точки определяется номер центра ближайшего(по метрике смотрим, в нашем случае это евклидова 2) к ней, это и будет номер кластера к которому она принадлежит. Потом начинается итерационный процесс: от новых координат центров кластеров вновь пересчитываются все расстояния, элементы по новой переспределяются, алгоритм заканчивается при окончании распределения всех вершин между всеми кластерами. Но согласитесь, что возможно рано заканчивать по такому условию? Поэтому просто придумали запускать алгоритм при начальных расположениях *центроид* и смотреть, какое соотношение захваченных вершин будет теперь. Если поделенное между всеми будет примерно поровну, значит кластеризацию провели верно. Например, если, при $k = 10$ и 200 вершинах каждый кластер имеет ~20 вершин, то всё хорошо. Очевидно, что из такого подхода вытекают некоторые минусы алгоритма...

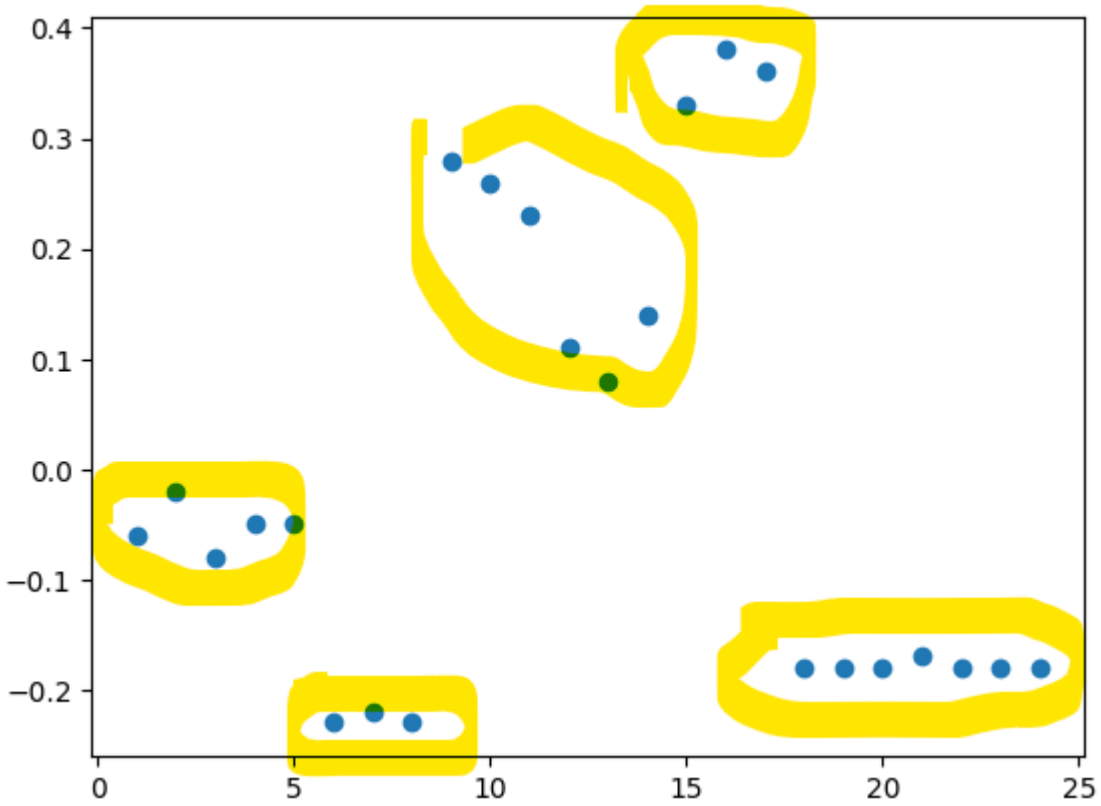
## Какой смысл несёт в себе собственный вектор лапласиана ?

Каждый вектор содержит кусочек информации о "локальной связности" вершин, поэтому нам недостаточно будет одного такого вектора, чтобы судить о кластеризации в целом. Чем больше вектооров мы имеем, тем больше поделить мы можем. Именно поэтому количество кластеров должно равняться количеству собственных векторов, которые мы берём для метода. Я лично понял такой вектор, если его развернуть в 2D график, небольшой пример думаю вам поможет...

In [ ]:
```
test_vector = np.ndarray.round(min5Vectors[:,[1]],2) # округлили для красоты графика
plt.scatter([i for i in range(1, len(test_vector)+1)],test_vector) # график
plt.show()
```




По oX - порядковый номер компоненты в вектора, по oY - её значение. Если приглядется, то можно выделить следующие $k$ кластеров, и такое совпадение должно примерно получаться у каждого собственного вектора:



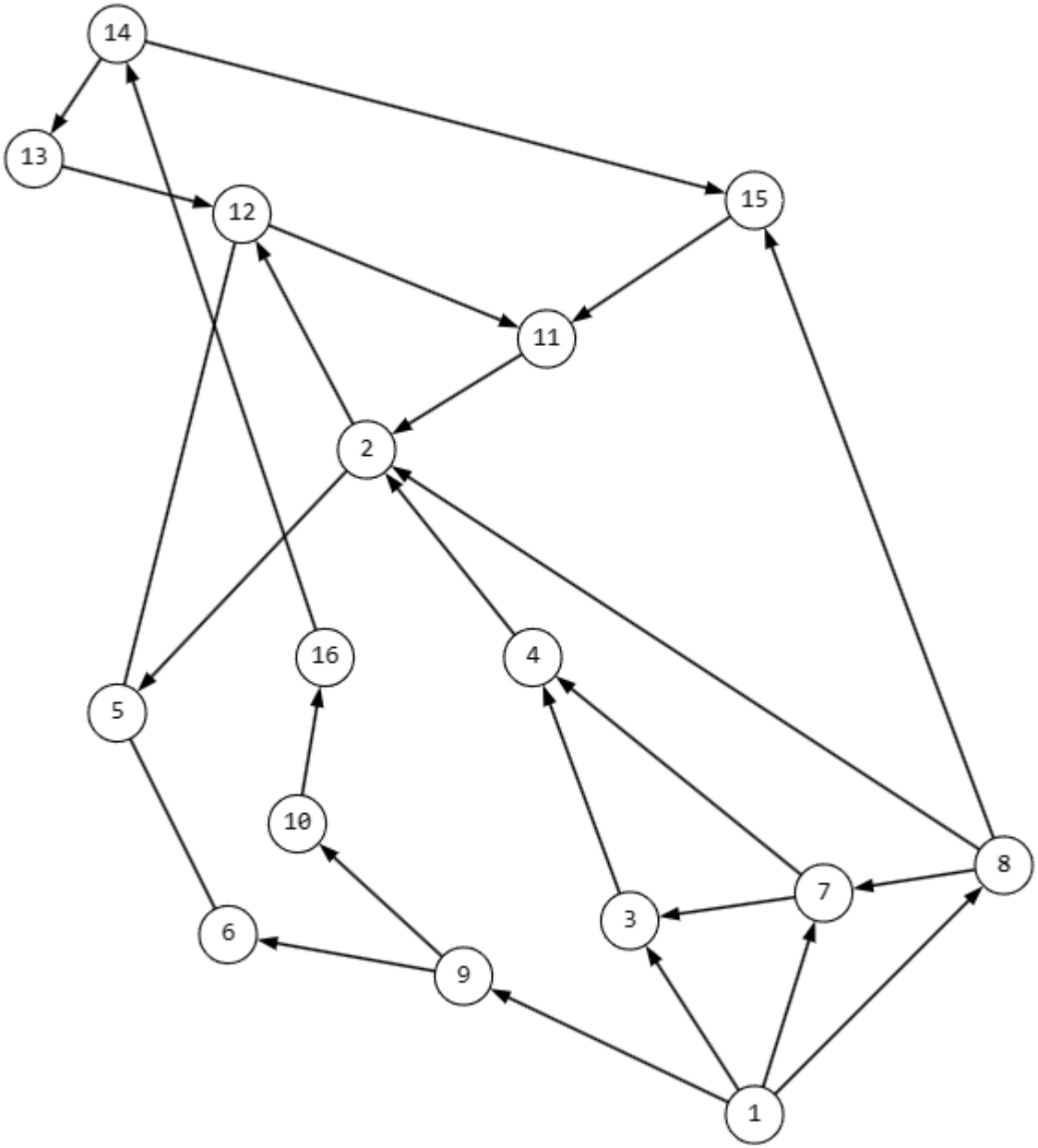

### Почему собственные вектора должны соответствовать именно *минимальным* собственным значениям, а не, например, *максимальным*? или каким-либо другим?

Из лекции по графам: чем меньше второе собственное число Лапласиана *связного графа*, тем этот граф **ближе к несвязному**, "есть узкое место внутри такого графа". В этом есть смысл, учитывая то, что каждый кластер по сути ограничем так называемым "узким местом", которое мы и должны задетектить.

## Задание 2. Google PageRank алгоритм

PageRank можно перевести с английского языка как «ранг страницы», однако Google Inc. связывает слово Page в названии алгоритма не с английским словом «страница», а с именем Ларри Пейджа

Создадим связный ориентированный граф из 16 вершин и 25 рёбер. Каждая вершина – это веб-страница, а стрелка – наличие ссылки, которая позволяет пользователю перейти с одной страницы на другую. Получим следующий граф...



Составим матрицу...

$$M = \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1n} \\ m_{21} & m_{22} & \dots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \dots & m_{nn} \end{bmatrix}$$

где $m_{ij}$ – это отношение числа ссылок на $j$-й странице, которые ведут на $i$-ю страницу, к общему числу ссылок на $j$-й странице.

Иными словами,

$$mij = \frac{\text{число стрелочек, выходящих из } j-\text{й вершины и входящих в } i-\text{ю вершину}}{\text{общее число стрелочек, выходящих из } j-\text{й вершины}}$$

Импортируем матрицу смежности с нашего любимого сайта и воспользуемся библиотекой *networkx* для построения графа, а после в цикле пройдёмся с условием выше и составим матрицу M

Получается таким способом мы создали матрицу смежности с весами для нашего графа

In [ ]:
```
data = np.loadtxt("test2.txt")
G= nx.DiGraph(data)
M = calculateM(G)
printBeauty(M, float)
```

```
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00 0.0 0.0 1.0 0.0 0.0 0.0 0.333 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.25 0.0 0.0 0.0 0.0 0.0 0.5 0.000 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00 0.0 1.0 0.0 0.0 0.0 0.5 0.000 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00 0.5 0.0 0.0 0.0 1.0 0.0 0.000 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0
0.00 0.0 0.0 0.0 0.5 0.0 0.0 0.000 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.25 0.0 0.0 0.0 0.0 0.0 0.0 0.333 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.25 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.25 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 0.0 0.0 0.5 0.0 0.0 1.0 0.0
0.00 0.5 0.0 0.0 0.5 0.0 0.0 0.000 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.333 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0
0.00 0.0 0.0 0.0 0.0 0.0 0.0 0.000 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Найдём собственный вектор матрицы $M$, соответствующий наибольшему собственному числу...

Ниже вывели собственное число, а после - вектор, который ему соответствует

In [ ]:
```
eigValuesM, eigVectorsM = LA.eig(M)
eigValuesM = list(np.real(np.ndarray.round(eigValuesM,3)))
max_eig_value = max(eigValuesM)
v = np.real(eigVectorsM[eigValuesM.index(max_eig_value)])

print(eigValuesM)
printBeauty(v, float)
```

```
[-0.895, 1.0, -0.25, -0.25, 0.395, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

-2.041355e-01
 2.407717e-01
-7.071068e-01
-7.071068e-01
 6.051620e-01
-5.585013e-16
-8.164966e-01
 5.345225e-01
-5.345225e-01
 9.355196e-17
 5.345225e-01
-9.355196e-17
 9.355196e-17
-9.355196e-17
-5.345225e-01
 9.355196e-17
```

Применим реализацию PageRank алгоритма, позаимствованную с википедии, при $d = 1$.

Результатом будет вектор, соответствующий вершинам нашего графа, а значения уже будут соответствовать вероятности случайного попадания спустя $iterations\ count$ итераций. Очевидно, что как и в марковских процессах, вектор должен рано или поздно сходиться

In [ ]:
```
d = 1; iterations_count = 1000
r2 = pagerank(M, iterations_count, d)
printBeauty(r2, float)
```

```
0.000000
0.117638
0.000000
0.000000
0.352914
0.176457
0.000000
0.000000
0.000000
0.000000
0.117638
0.235276
0.000000
0.000000
0.000000
0.000000
```

## Немного объяснений произошедшему

Получается, что мы столкнулись с марковским процессом, потому что сама википедия об этом говорит (CTRL+F -> Markov Chain). Доказательством этого будет также служить алгоритм построения матрицы вероятностей M и многое другое ( сумма столбцов такой матрицы всегда = 1 )

### Какой смысл имеет матрица M, почему она составлена именно так, и что она показывает?

Это матрица вероятности, составленная по школьному(привет ЕГЭ!!!) определению вероятности:

$$P = \frac{m - \text{благоприятные исходы}}{n - \text{все исходы}}$$

Так как нас интересовало только количество стрелочек из $j \to i$, то как раз мы их и принимали за *благоприятные* исходы. Таком образом благоприятные исходы значили вероятность кликнуть так, чтобы удачно переадресесоваться именно на $i$-ую страницу. А потом уже итеративно считали все возможные клики-переходы. В самой википедии используют подобную терминологию с "рандомными кликами"

### Какую роль играет параметр d ?

Это Damping factor - коэффициент демпфирования - "смягчать нежелательное, вредное воздействие чего-либо". Это вероятность, что на $n$-ом шаге юзер продолжит переходить рандомно по страницам графа. Также они добавляют, что $1 - d$ - вероятность на $n$-ом шаге перейти уже на произвольную случайную страницу, игнорирую графовую структуру. В нашем случау $d = 1$, поэтому такие *случайности* исключены, а также юзер продолжит кликать все итерации, которые мы зададим...

### Как можно интерпретировать собственный вектор этой матрицы, соответствующий наибольшему собственному числу?

Если принять это за марковский процесс, то это вектор сходимости, т.е. куда чаще всего попадёт юзер при очень долгом кликанье между страничками. Однако, сходимость может формировать не только один вектор, а сразу несколько, что и произошло в моём случае, потому что результат PageRank $\neq$ собственному вектору, соответствующему максимальному собственному числу

### Почему важен именно этот собственный вектор, а не какой-то другой?

Может оказаться и не один, тогда уже не предскажешь сходимость без моделирования в долгий срок. В моём случае процесс перехода по страницам *не сошёлся* с марковским собственным вектором по неизвестной мне причине :( . Почему важно ответил в вопросе выше

## Концовка, выводы

Спасибо всем, кто прочитал и оценил мою работу справедливо, держите картинку. И всем остальным тоже обязательно глянуть

Это я докупил к своему обеду сосиску в тесте за 70 рублей и теперь стал толще и беднее

А это тоже я, но не купивший сосиску в тесте за 70 рублей, а отложил их на будущую пенсию(так сказали делать в интернете)