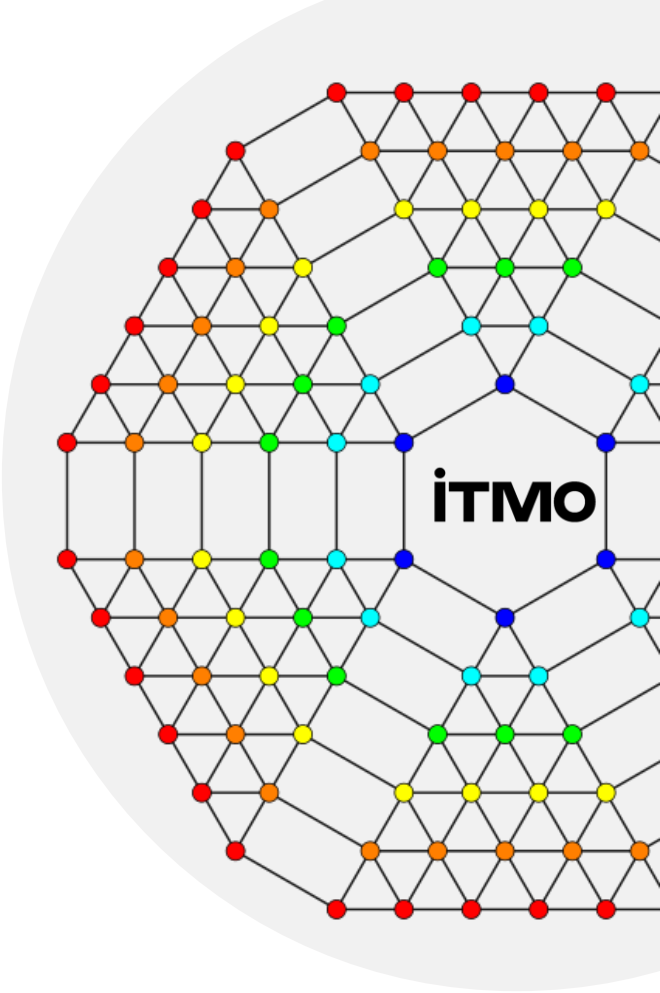
ЛАБОРАТОРНАЯ РАБОТА 5

Спектральная теория графов



Преподаватель:

Перегудин А. А.,

Ассистент фак. СУиР

Выполнила:

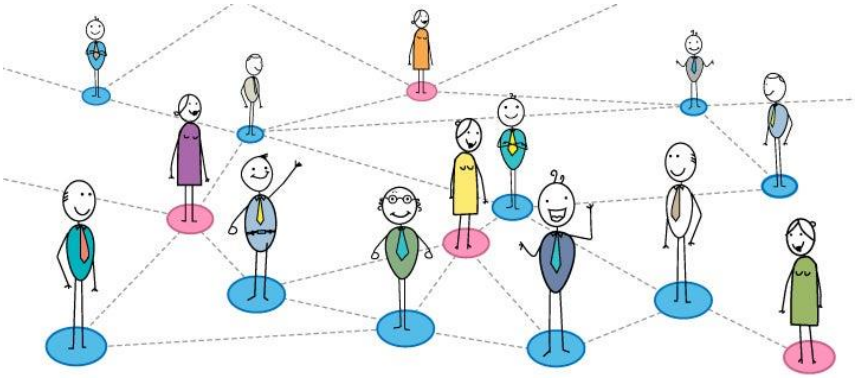
студентка гр. R3235

Нгуен Кхань Нгок

Санкт-Петербург 202

ВВЕДЕНИЕ

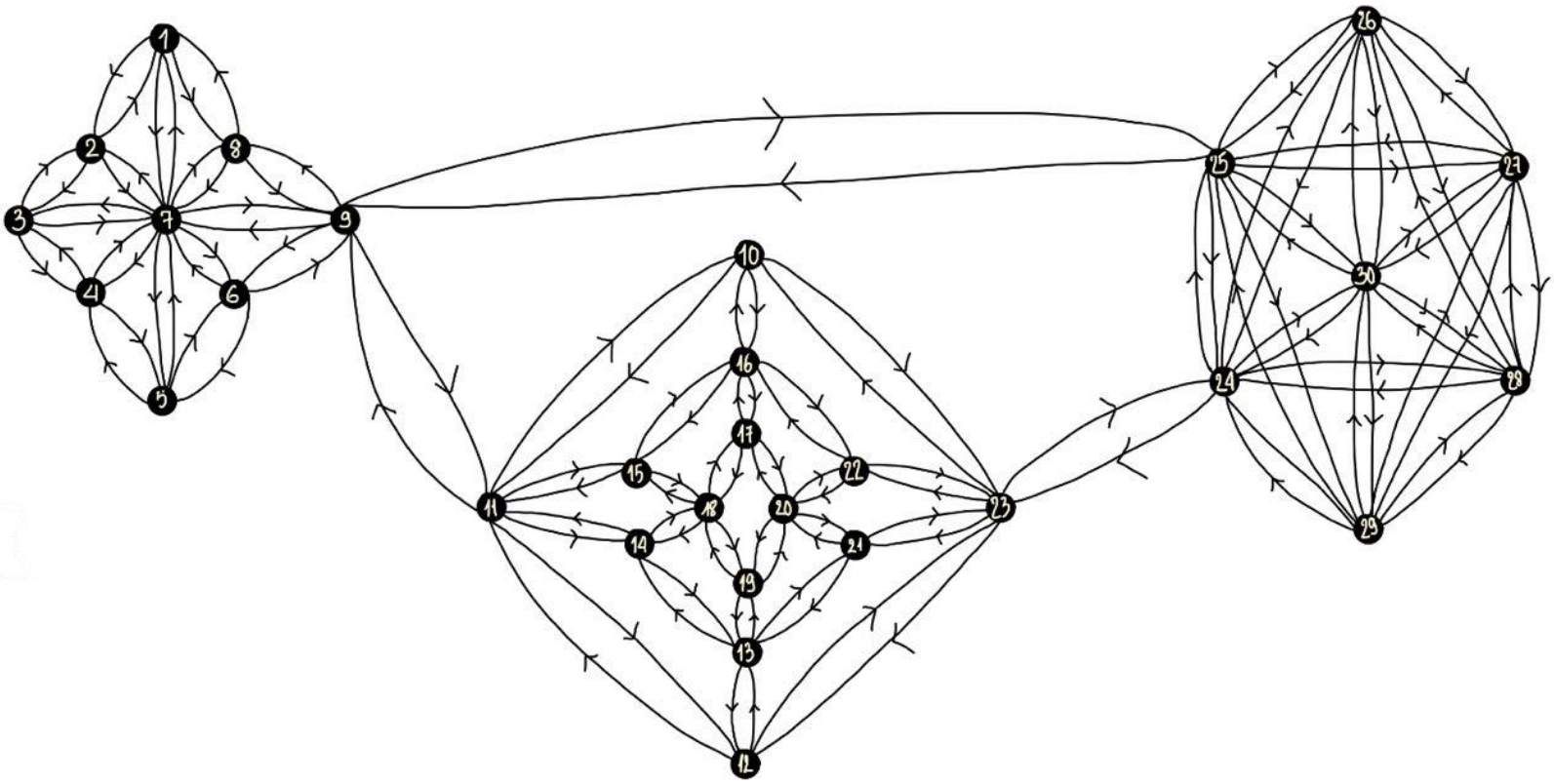
В этой лабораторной вы будете анализировать сети (графы) с помощью матриц.



Задание 1. Кластеризация социальной сети

Представьте, что вам нужно проанализировать социальную сеть – множество людей, часть из которых являются “друзьями” друг друга. Отношения между людьми всегда взаимные

- Придумайте связный граф из 15-30 вершин. Каждая вершина – это человек, а ребро – отношение дружбы. Пронумеруйте вершины графа от 1 до n



× Составьте матрицу Лапласа для вашего графа в соответствии с нумерацией вершин. Найдите (не руками) её собственные числа и соответствующие им собственные вектора.
У Нас ориентированный граф. Из-за этого мы можем составить матрицу Лапласиан ориентированого графа L_{nxn} (где n – вершины)
Чтобы создать матрицу Лапласа ориентированного графа, нам необходимо следовать ее правилам:

l_{ij} = степень i – той вершины (число выходящих рёбер)
$l_{ij} = -1$, если $i \neq j$ и есть стрелка из i в j
$l_{ij} = 0$, если $i \neq j$ и нет стрелки из i в j

матрицу Лапласа L

3	-1	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	0
-1	3	-1	0	0	0	-1	0	0	0	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	-1	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	3	-1	0	-1	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	3	-1	-1	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	3	-1	0	-1	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	8	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	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	0
0	0	0	0	0	-1	-1	-1	5	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	3	-1	0	0	0	0	-1	0	0	0	0	0	0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	-1	-1	5	-1	0	-1	-1	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	3	-1	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	-1	4	-1	0	0	0	0	-1	0	-1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	-1	0	-1	3	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	-1	0	0	0	3	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	-1	0	0	0	0	-1	4	-1	0	0	0	0	-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	3	-1	0	-1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1	0	-1	4	-1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	-1	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	0	0	0	0	0	0	-1	0	-1	4	-1	-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	3	0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	-1	0	3	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	-1	0	-1	0	0	0	0	0	0	0	0	-1	-1	5	-1	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	6	-1	-1	0	-1	-1	-1
0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	6	-1	-1	0	-1	-1
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	5	-1	-1	0	-1
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	5	-1	-1	-1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-1	-1	5	-1	-1
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	0	-1	-1	5	-1
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	6

30 x 30

Найти собственные значения и собственные векторы для матрицы Лапласа можно с помощью функции **sparse.linalg.eigsh()** в библиотеке **scipy** на Python. И Возьмем k собственных векторов $v_1, \ldots, v_k$ матрицы Лапласа, соответствующих самым маленьким собственным числам

```
import scipy as sp

l,v = sp.sparse.linalg.eigsh(L, k = 3, which='SM')
print('eigenvalue: ', np.round(l,3))
print('eigenvectors: ', np.round(v,3))
```

Это команда для вычисления собственных значений l и собственных векторов v разреженной матрицы **L** методом Крылова-Шура.

Для выполнения этого расчета используется функция **eigsh** в модуле **scipy.sparse.linalg**.

Эта функция находит k собственных значений (l) и собственных векторов(v) симметричной квадратной матрицы или комплексной эрмитовой матрицы A.

k: Количество желаемых собственных значений и собственных векторов. k должно быть меньше N. Невозможно вычислить все собственные векторы матрицы.

which='SM': наименьшие (по величине) собственные значения.

Возвращаться:

l - Массив k собственных значений.

v - Массив, представляющий k собственных векторов. Столбец v [:, i] — это собственный вектор, соответствующий собственному значению l [i].

Первое собственное значение имеет значение, практически равное 0, его можно считать = 0. А его собственными векторами являются векторы $[c\ c\ \ldots\ c]^T$, эквивалентные v= $c[1\ 1 \ldots 1\ ]^T$. Это удовлетворяет условию матрицы Лапласа

$$\mathbf{0} = \boldsymbol{\lambda_1} \leq \boldsymbol{\lambda_2} \leq \cdots \leq \boldsymbol{\lambda_n}$$

np.round(l,3): Функция округления библиотеки numpy

Мы можем переписать матрицу V_{30xk} из них следующим образом

K = 3

V =

1,00	-261,88	-0,33
1,00	-276,84	-0,36
1,00	-280,73	-0,37
1,00	-276,84	-0,36
1,00	-261,88	-0,33
1,00	-223,13	-0,24
1,00	-246,16	-0,29
1,00	-223,13	-0,24
1,00	-127,70	-0,04
1,00	135,01	-0,22
1,00	91,45	-0,26
1,00	135,01	-0,22
1,00	164,73	-0,33
1,00	150,57	-0,36
1,00	150,57	-0,36
1,00	164,73	-0,33
1,00	182,03	-0,40
1,00	172,82	-0,41
1,00	182,03	-0,40
1,00	181,08	-0,35
1,00	166,47	-0,26
1,00	166,47	-0,26
1,00	128,49	-0,03
1,00	20,09	0,84
1,00	-18,27	0,84
1,00	1,00	1,00
1,00	-2,26	1,03
1,00	4,30	1,03
1,00	1,00	1,00
1,00	1,00	1,00

$\lambda_1 = 0$ $\lambda_9 \approx 0,151$ $\lambda_{10} \approx 0,26$

K = 2

V=

1,00	-261,88
1,00	-276,84
1,00	-280,73
1,00	-276,84
1,00	-261,88
1,00	-223,13
1,00	-246,16
1,00	-223,13
1,00	-127,70
1,00	135,01
1,00	91,45
1,00	135,01
1,00	164,73
1,00	150,57
1,00	150,57
1,00	164,73
1,00	182,03
1,00	172,82
1,00	182,03
1,00	181,08
1,00	166,47
1,00	166,47
1,00	128,49
1,00	20,09
1,00	-18,27
1,00	1,00
1,00	-2,26
1,00	4,30
1,00	1,00
1,00	1,00

$\lambda_1 = 0$ $\lambda_9 \approx 0,151$

K = 6

V=

1	-261,88	-0,33	0,00	0,62	1,41
1	-276,84	-0,36	0,00	9,28	1,00
1	-280,73	-0,37	0,00	12,79	0,00
1	-276,84	-0,36	0,00	9,28	-1,00
1	-261,88	-0,33	0,00	0,62	-1,41
1	-223,13	-0,24	0,00	-8,64	-1,00
1	-246,16	-0,29	0,00	0,26	0,00
1	-223,13	-0,24	0,00	-8,64	1,00
1	-127,70	-0,04	0,00	-13,60	0,00
1	135,01	-0,22	1,00	-16,47	0,00
1	91,45	-0,26	0,00	-28,42	0,00
1	135,01	-0,22	-1,00	-16,47	0,00
1	164,73	-0,33	-1,56	-1,10	0,00
1	150,57	-0,36	-1,00	-26,07	0,00
1	150,57	-0,36	1,00	-26,07	0,00
1	164,73	-0,33	1,56	-1,10	0,00
1	182,03	-0,40	1,00	15,11	0,00
1	172,82	-0,41	0,00	-8,87	0,00
1	182,03	-0,40	-1,00	15,11	0,00
1	181,08	-0,35	0,00	32,21	0,00
1	166,47	-0,26	-1,00	24,71	0,00
1	166,47	-0,26	1,00	24,71	0,00
1	128,49	-0,03	0,00	5,27	0,00
1	20,09	0,84	0,00	1,81	0,00
1	-18,27	0,84	0,00	-1,78	0,00
1	1,00	1,00	0,00	1,00	0,00
1	-2,26	1,03	0,00	0,82	0,00
1	4,30	1,03	0,00	1,62	0,00
1	1,00	1,00	0,00	1,00	0,00
1	1,00	1,00	0,00	1,00	0,00

$\lambda = 0$ $\lambda \approx 0,151$ $\lambda \approx 0,26$ $\lambda \approx 1,44$ $\lambda \approx 1,53$ $\lambda \approx 1,59$

× Рассмотрите строки составленной матрицы V как точки пространства R^k . Примените к этим точкам любой метод кластеризации (например, метод k-means, реализованный как в MATLAB, так и на Python)

ЧТО ТАКОЕ МЕТОД КЛАСТЕРИЗАЦИИ K-MEANS?

Чтобы ответить на этот вопрос, давайте выясним

× Что такое кластеризация

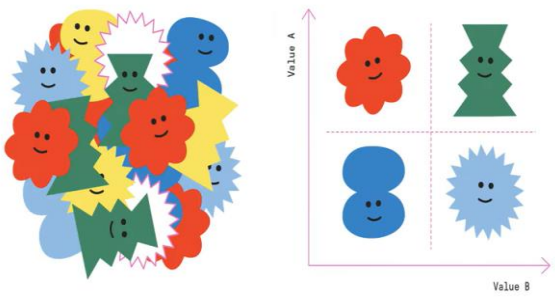
Кластеризация — основная проблема машинного обучения. При этой проблеме установленные модели автоматически разделят данные на разные кластеры. В частности, кластеризация данных (или Clustering; Labelless Learning, Eng: Data Clustering) — это процесс разделения набора объектов на группы или классы, кластеры объектов данных со схожими свойствами.

Кластер, или класс (cluster) — это совокупность объектов данных, в которой объекты одного кластера похожи друг на друга и менее похожи (или различны), чем объекты других кластеров. Подобие определяется по определенному стандарту, в зависимости от каждого конкретного применения и задано заранее.

В отличие от процесса **классификации**, часто необходимо заранее знать свойства или характеристики объектов одного и того же класса (кластера) и на основании этого относить новый объект к тому или иному классу. Вместо этого в процессе **кластеризации** мы заранее не знаем свойств кластеров, а должны полагаться на отношения между объектами, чтобы найти характерное сходство для каждого кластера между объектами по определенной мере. И цель алгоритмов кластеризации состоит в том, чтобы получить доступ к этому огромному набору данных. Как мы узнаем, какие конкретные группы данных там содержатся? К какой группе относятся все данные там? Это то, на что наш алгоритм кластеризации должен найти ответ.

Из-за этого свойства алгоритмы **кластеризации** — задачи, называемые обучением без присмотра *(*Unsupervised learning*)*, что означает, что данные для обучения модели ранее не маркировали

Например, мы даем ребенку много кусочков пазла разной формы и цвета, например треугольники, квадраты, круги синего и красного цветов, а затем просим ребенка разделить их на группы. Даже не сообщая детям, какая деталь какой формы или цвета соответствует, они, скорее всего, смогут сортировать детали по цвету или форме.

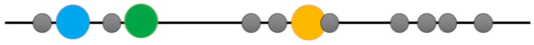


В частности, в нашей лаборатории под определением кластеризации понимают тип кластеризации графа, который представляет собой область кластерного анализа, целью которой является поиск групп (clusters) связанных вершин в графе. Кластеризация графов дает такие результаты, что внутри каждого кластера есть вершины со многими тесно связанными ребрами, тогда как между кластерами имеется только несколько связанных ребер.

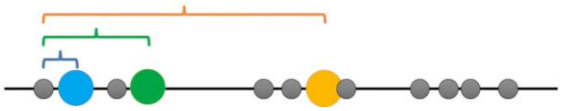
Кластеризация может быть решена с помощью ряда алгоритмов, таких как алгоритм KMeans, алгоритм Hierarchical Clustering, DBSCAN (Density-Based Spatial Clustering of Applications with Noise), Gaussian Mixture Models (GMM), спектральная кластеризация (Spectral Clustering) и т. д. Анализируя, что такое кластеризация, мы можем ответить, что такое алгоритм KMeans.

- **KMeans**

Основная идея алгоритма **K-Means** — найти способ сгруппировать данные объекты в K кластеров (K — количество заданных кластеров, k — целое положительное число). Основная идея алгоритма K-Means — найти способ сгруппировать заданные объекты в K кластеров (K — заданное количество кластеров, k — целое положительное число).
Сначала определите значение k (количество кластеров) по желанию.
Затем случайным образом выберите k случайных центральных точек.



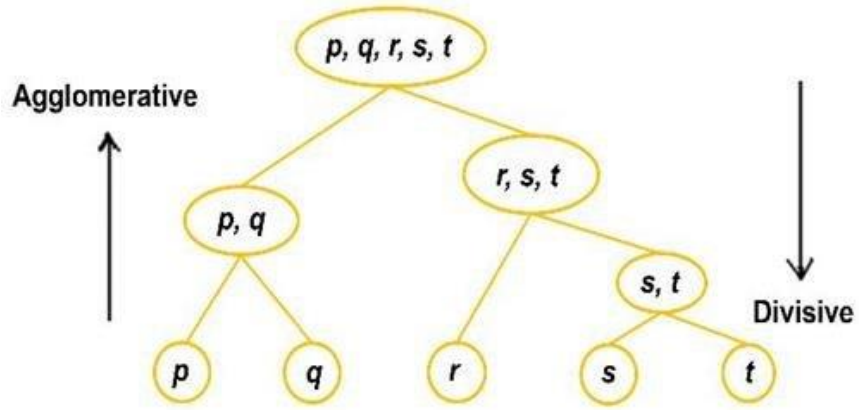
Далее измерьте расстояние между точками относительно центральных точек.



Затем назначьте точки ближайшей центральнх точек, чтобы сформировать кластер.

затем вычислите среднее значение кластера
Это повторяется до тех пор, пока кластеры не будут сформированы оптимально.

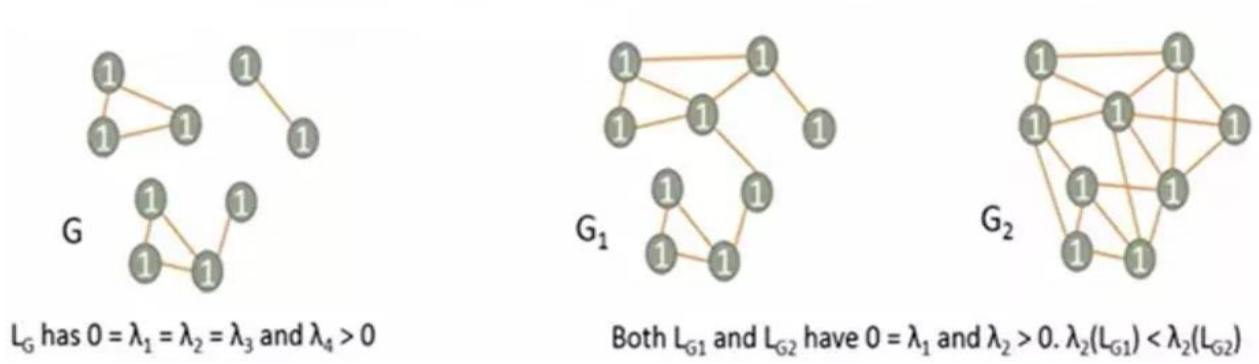
- **Hierarchical Clustering**



Agglomerative hierarchical clustering: начиная с каждой точки идет кластер, кластеризация – это объединение малых кластеров в более крупные (снизу вверх).

Divisive hierarchical clustering: все объекты/точки представляют собой кластер, кластеризация заключается в разбиении большого кластера на более мелкие кластеры (сверху вниз).

- **Спектральная кластеризация (Spectral Clustering)**



Spectral Clustering — алгоритм кластеризации графов, основанный на матрице смежности (матрица смежности A), матрице степеней, матрице Лапласа, их спектрах (собственных значениях и собственных векторах) и т.д. Этот алгоритм аналогичен алгоритму кластеризации KMeans.

В эту лабораторную работу я использовала Python для кластеризации.

И реализовала метод кластеризации **k-means**.
Здесь я использую алгоритм кластеризации **k-means** из библиотеки **scikit-learn (sklearn.cluster.KMeans)** на Python.

```
from sklearn.cluster import Kmeans
import scipy as sp

l,v = sp.sparse.linalg.eigsh(L, k = 6, which='SM')
print('eigenvalue: ', np.round(l,5))
print('eigenvectors: ', np.round(v,5))

X = v*l # X представляет особенности графика.
kmeans = KMeans(init='k-means++', n_clusters=3, n_init=10)
kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_
print(np.round(centroids,16))
labels = kmeans.labels_
print(labels)
error = kmeans.inertia_
```

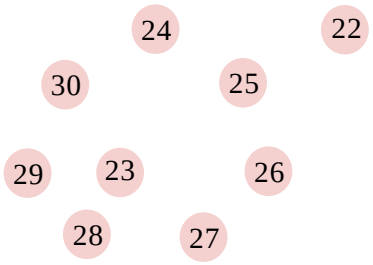
`init = 'k-means++'` в алгоритме K-means относится к методу инициализации начального центра кластера. По умолчанию в алгоритме K-means начальные центры кластеров выбираются случайным образом из входных данных.
`n_clusters=3`: это желаемое количество кластеров.
`n_init=10`: В алгоритме K-средних начальный процесс инициализации центра кластера может повлиять на результаты кластеризации. Запустив алгоритм с множеством различных инициализаций (n_init = количество раз), у нас есть шанс найти лучший результат по инерции.
После получения **X** алгоритм **k-means** применяется для кластеризации вершин графа.
X используется в качестве входных данных для метода fit_predict в алгоритме **k-means** (`fit_predict(X)`).
Назначьте вершины ближайшему кластеру на основе расстояния этой точки от центра кластера и пометьте вершины, принадлежащие этому кластеру
Результатом процесса кластеризации являются метки и соответствующие центры кластеров.
`centroids = kmeans.cluster_centers_` возвращает информацию о центрах кластеров.
`labels = kmeans.labels_` возвращает информацию о метках каждой точки данных.

Возвращаемый результат:

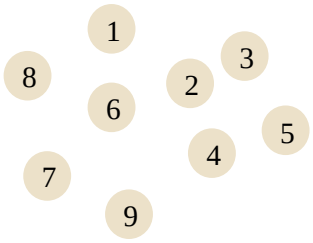
На основе графа и idx мы можем кластеризовать вершины следующим образом:

$$V = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1]^T$$

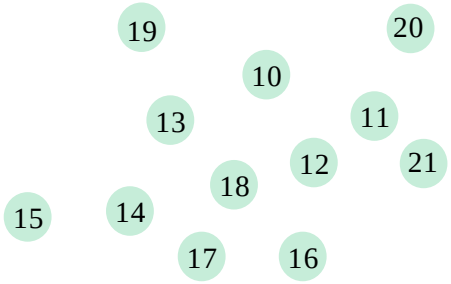
$$C = \begin{bmatrix} 0 & -0.03866087 & -0.02506351 \\ 0 & 0.00015629 & 0.08462067 \\ 0 & 0.02477527 & -0.02619808 \end{bmatrix}$$



Вершины с 22 по 30 имеют метку кластера 3.

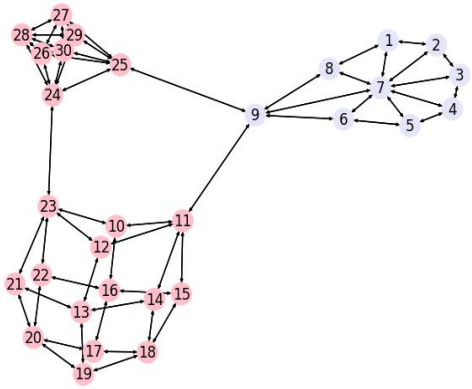
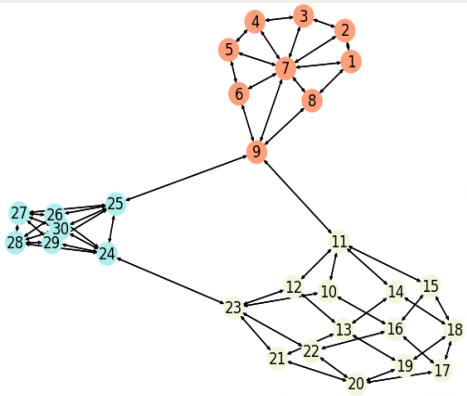
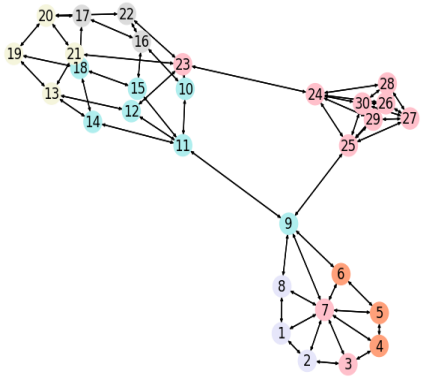


Вершины с 1 по 9 имеют метку кластера 1



Вершины с 10 по 21 имеют метку кластера 2 .

× Представьте результат кластеризации в виде раскрашенного графа.

K=2	K=3	K=6
		

После выполнения кластеризации с разными значениями k (k=2, k=3, k=6), мы видим

× Если мы разделим общество на 2 уровня (k=2), мы увидим, что кластеризация не кажется оптимизированной.

× Если разделить общество на 6 уровней (k=6). Здесь есть несколько вершин другого кластера, расположенных в области другого кластера и далеко от их центра кластера. Это показывает, что кластеры не отражают четкую социальную структуру.

× А при k = 3 видно, что классификация становится более четкой, кластеры разделяются на основе связей между вершинами, близкими друг к другу.

КАК Я ЭТО СДЕЛАТЬ?

При кластеризации социальных графов, помимо понятий матрицы Лапласа, нам необходимо знать некоторые из следующих терминалов.

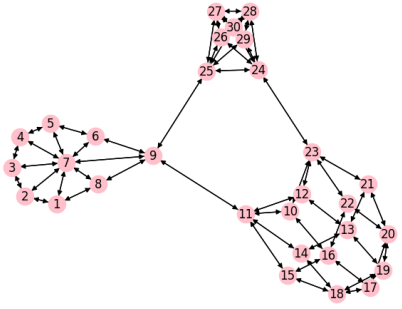
- **Матрица смежности A (Matrix Adjacentry A)**
  При представлении графа с помощью матрицы смежности мы используем квадратную матрицу nxn (где n — количество вершин в графе). В котором каждая строка представляет вершину, каждый столбец представляет собой место, где вершина имеет соединение (например, в строке 5 столбец 10 = 1, то есть 5-я вершина соединена с 10-й вершиной).
- **Степень вершины $u_i \in V$**
  Для вершины u_i в графе определите степень, символ $\deg(u_i)$ называется количеством ребер, принадлежащих u. В графе сумма количества ребер, принадлежащих u, также равно сумма количества вершин, смежных с u
- **Степень матрицы D (Matrix Degree D)**
  Определяется как диагональная матрица со степенями deg(u1),...,deg(un) на диагонали.

Теперь давайте нарисуем социальный граф

```
edges = [(1, 2), (1, 7), (1, 8), (2, 1), ...(25, 9)] #Объявление ребер графа
nodes = list(range(1, 31)) #Создание списка 30 nodes

G_dir = nx.DiGraph()#  ориентированный граф
# График с использованием библиотеки networkx. Здесь я рисовала ориентированный
граф, похожий на желаемый граф. Но при расчете я преобразую его в неориентированный
граф.
G_und = nx.Graph(G_dir) # неориентированный граф

# добавить в граф ранее объявленные ребра и узлы
G_dir.add_nodes_from(nodes)
G_dir.add_edges_from(edges)

num_nodes = G_dir.number_of_nodes()
num_edges = G_dir.number_of_edges()

#рисование графа с помеченными вершинами
nx.draw(G_dir, with_labels='True', node_color='pink')
plt.show()
```

OUTPUT:

10



× **Сначала нам нужно определить и распознать центральные вершины или вершины со многими смежными ребрами.**

Для этого нам нужно вычислить отношение степени вершины u к общему количеству вешин в графе минус 1. Знаменатель – это нормализующая часть, которая приведет все значения в значение [0,1] .

$$C_D(u_i) = \frac{\deg(u_i)}{n-1}$$

Почему n-1? Потому что каждая вершина может быть соединена с всеми количества вершин минус она сама, каждой соединяемой вершине будет присвоено значение 1.

В этой формуле также есть формула для ориентированного графа. Нам просто нужно изменить степень на внутреннюю степень (in-degree) вершины. Однако, поскольку 2 смежные вершины будут иметь 2 смежных ребра в противоположных направлениях, соединяющих их, внутренняя степень Вершины по сравнению с ориентированным графом в этом случае будут равны степени направленности по сравнению с неориентированным графом.

```
deg_cen = {}
graph=[G]

list_nodes=list(graph[0].nodes)
num_nodes = graph[0].number_of_nodes() # = 30
num_neighbors = graph[0].degree(list_nodes[8])
print('degree of node', list_nodes[8], 'is ', num_neighbors )
print('Degree of Centrality:', round(num_neighbors/(num_nodes-1),3))
```

OUTPUT: Degree of node 9 is 5
Degree of Centrality: 0.172

× **Далее давайте рассчитаем Closeness Centrality:**

Closeness Centrality— это тип расчета центральности, основанный на том, насколько близко вершина находится к другим вершинам. Близость каждой вершины к другим вершинам сети. Если вершина имеет более низкую степень близости, для связи с другими вершинами она должна пройти через множество других вершин в сети. Closeness Centrality помогает нам определить, какие люди (вершины) лучше всего могут влиять на сеть.
Closeness Centrality равно отношению n-1 к суммарному кратчайшему расстоянию u_i до u_j

$$C_c(u_i) = \frac{n-1}{\sum_{j=1}^{n} d(u_i, u_j)}$$

```
s = 0
node_clos = 6

for node in list_nodes:
    if nodes!=node_clos:
        shortest_path_length =  nx.shortest_path_length(graphs[0],node_clos,node)
        s+=shortest_path_length
```

OUTPUT: The closeness centrality distance of node 6 is 0.32222222222222224

× **Затем вычислите Betweenness centrality of nodes**

Betweenness centrality of nodes *p*ассчитает влияние вершины на распространение информации в сети. Он определяет, какие вершины сети действуют как «мосты» между другими вершинами. Он находит кратчайшие пути между всеми парами вершин и определяет центральность вершины по посредничеству на основе того, как часто она появляется в кратчайшем пути из двух вершин.
Вершина с большей центральностью посредничества играет важную роль в социальной сети, поскольку он может быть мостом, соединяющим две группы, и в случае его потери обмен информацией между двумя группами будет невозможен.Betweenness centrality of nodes рассчитывается как сумма кратчайших путей (включая ui), деленная на сумма количества кратчайших путей от всех вершин i до вершины j.

$$C_B(u_i) = \sum_{j<k} \frac{p_{jk}(u_i)}{p_{jk}}$$

```
betw_cen = {}
for g in graphs:
    betw_cen[g] = nx.betweenness_centrality(g)
```

OUTPUT: (5, {5: [5], 4: [5, 4], 7: [5, 7], 6: [5, 6], 3: [5, 4, 3], 1: [5, 7, 1], 2: [5, 7, 2], 8: [5, 7, 8], 9: [5, 7, 9], 11: [5, 7, 9, 11], 25: [5, 7, 9, 25], 10: [5, 7, 9, 11, 10], 12: [5, 7, 9, 11, 12], 14: [5, 7, 9, 11, 14], 15: [5, 7, 9, 11, 15], 24: [5, 7, 9, 25, 24], 26: [5, 7, 9, 25, 26], 27: [5, 7, 9, 25, 27], 29: [5, 7, 9, 25, 29], 30: [5, 7, 9, 25, 30], 16: [5, 7, 9, 11, 10, 16], 23: [5, 7, 9, 11, 10, 23], 13: [5, 7, 9, 11, 12, 13], 18: [5, 7, 9, 11, 14, 18], 28: [5, 7, 9, 25, 24, 28], 17: [5, 7, 9, 11, 10, 16, 17], 22: [5, 7, 9, 11, 10, 16, 22], 21: [5, 7, 9, 11, 10, 23, 21], 19: [5, 7, 9, 11, 12, 13, 19], 20: [5, 7, 9, 11, 10, 16, 17, 20]})
из вершины 5 в вершину 5 = [5]
из вершины 5 в вершину 4 = [5, 4]
из вершины 5 в вершину 3 = [5, 4, 3].....

- **Затем вычислите Eigenvector Centrality**

Eigenvector Centrality, Эта мера аналогична степени (degree), но вместо подсчета количества связей с этой вершиной она учитывает степень связанной с ней вершины. В сети, когда две вершины имеют одинаковую степень, эта величина будет указывать, какая вершина соединена с более важными вершинами в сети.

```
eig_cen = {}
for g in graphs:
    eig_cen[g] = {node: np.round(value, 3) for node, value in
nx.eigenvector_centrality(g).items()}
```

OUTPUT:
{1: 0.018, 2: 0.015, 3: 0.014, 4: 0.015, 5: 0.018, 6: 0.031, 7: 0.046, 8: 0.031, 9: 0.099, 10: 0.026, 11: 0.032, 12: 0.026, 13: 0.012, 14: 0.009, 15: 0.009, 16: 0.012, 17: 0.005, 18: 0.006, 19: 0.005, 20: 0.01, 21: 0.021, 22: 0.021, 23: 0.09, 24: 0.375, 25: 0.377, 26: 0.36, 27: 0.358, 28: 0.358, 29: 0.36, 30: 0.418}}

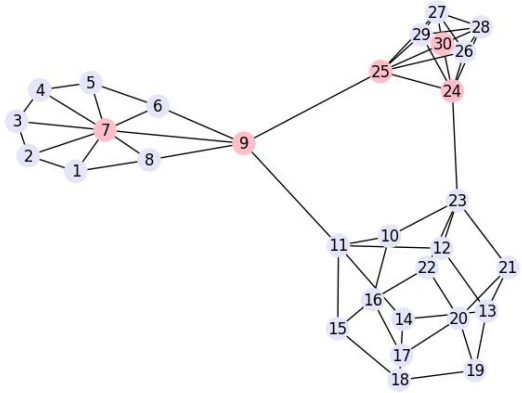
Итак, все необходимые значения рассчитаны. Как видите, все это время мы искали только центр и больше ничего не искали. Так насколько это важно? Центральность можно использовать как меру для определения важности пика. Центральность по близости — это тип расчета центральности, основанный на том, насколько близко вершина находится к другим вершинам.

Теперь определим, какие вершины в графе наиболее важны.

```
for ix, g in enumerate(graphs):
    temp_dict = {}
    for w in sorted(deg_cen[g], key=deg_cen[g].get, reverse=True):
        temp_dict[w] = deg_cen[g][w]
```

`sorted(deg_cen[g], key=deg_cen[g].get, reverse=True)` сортировка элементов deg_cen[g] по убыванию значения центральности.

OUTPUT: Отсортированная важность вершин с точки зрения степени центральности для графа равна [7, 24, 25, 30, 9]



Наконец, начинается работа по кластеризации.
Вот я использовала алгоритм **спектральной кластеризации**, чтобы воспользоваться свойствами матрицы Лапласиан.

Этапы спектральной кластеризации включают в себя:

1) Построить матрицу Лапласиан графа.

2) Вычислять собственные значения и собственные векторы матрицы Лапласиан, затем нам нужно сопоставить каждую точку с представлением меньшей размерности.

3) Алгоритм K-Means: создайте группы кластеров, используя приведенные выше данные.

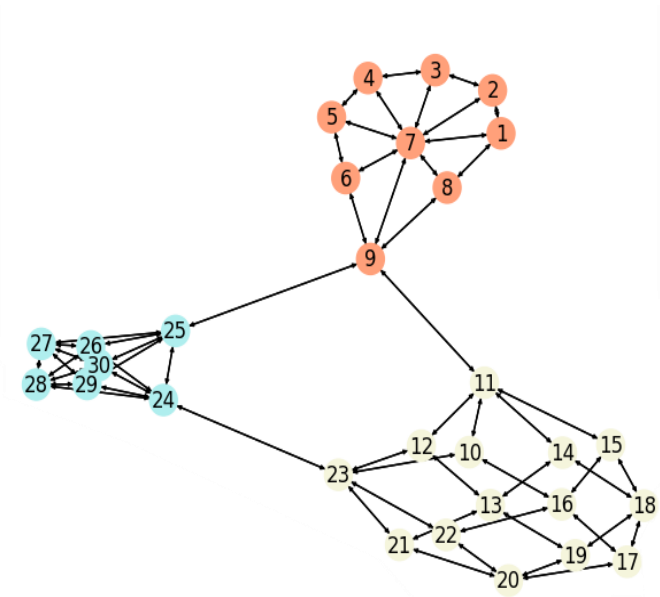
Мы получили матрицу Лапласа, аналогичную первой матрице в задание 1.

Поскольку мы нашли собственный вектор и собственное значение ранее, теперь мы начнем их кластеризовать.

```
from sklearn.cluster import KMeans
X = v*l # X представляет особенности графика.
kmeans = KMeans(init='k-means++', n_clusters=6, n_init=10)
kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_
print(np.round(centroids,16))
labels = kmeans.labels_
print(labels)
error = kmeans.inertia_
colors = ('pink','lavender','lightsalmon', 'beige', 'paleturquoise', 'gainsboro')
node_colors = [colors[label] for label in labels]

G = nx.DiGraph(G_p1)


nx.draw(G, node_color=node_colors, with_labels=True,arrows=True, arrowsize=5)

plt.show()
```

Эта кластеризация также объяснялась ранее, просто добавляя дополнительный цвет к кластерам.

Итак, мы получили результат, продолжаем заменять k = 2 и k = 6, получим результат, как указано выше.



Задание 2. Google PageRank алгоритм.

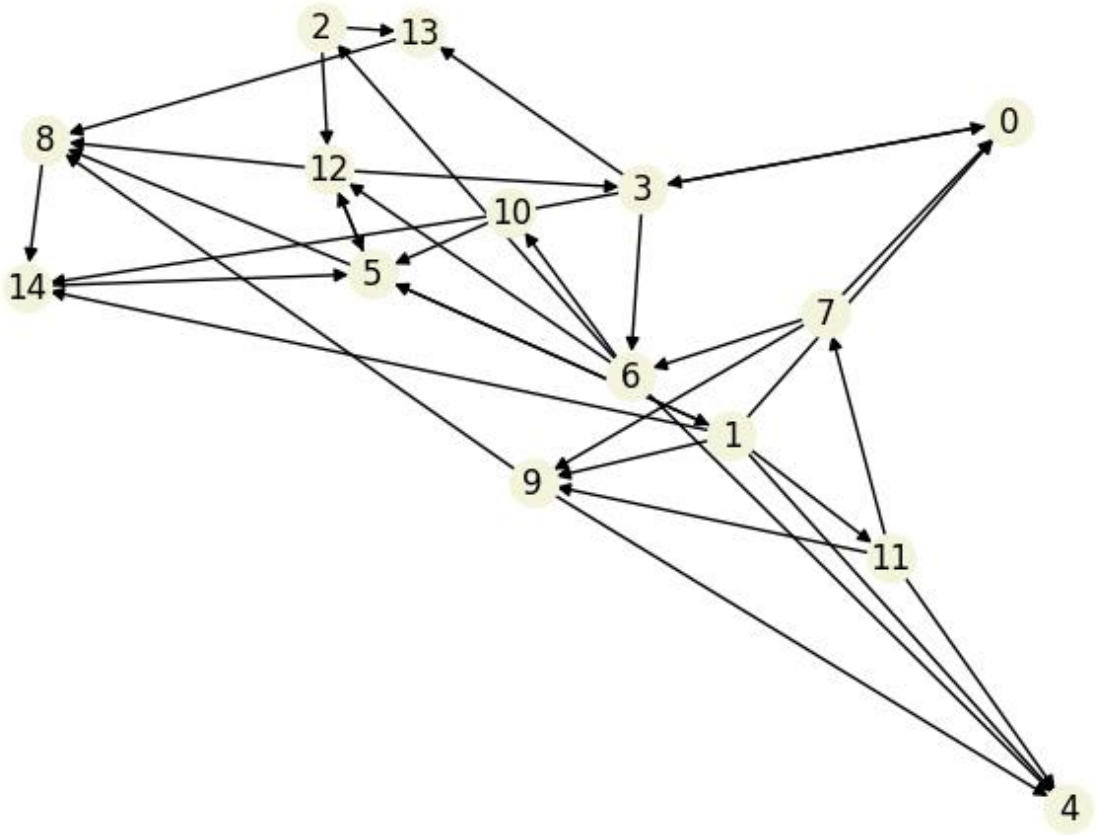
В этом задании вы попробуете применить самый первый и исторически важный алгоритм сортировки страниц, который использовался поисковой системой Google – PageRank.

× Придумайте связный ориентированный (направленный) граф из 10-15 вершин и 25-50 стрелок (дуг, рёбер). Каждая вершина – это веб-страница, а стрелка – наличие ссылки, которая позволяет пользователю перейти с одной страницы на другую. Одна вершина может быть соединена с другой сразу несколькими стрелками.

Чтобы сэкономить время, я случайным образом сгенерировал граф с 15 вершинами и 35 ребрами на Python, используя библиотеку networkX.

```
G = nx.gnm_random_graph(15,35,directed=True)

nx.draw(G, with_labels='True', node_color='Beige')
plt.show()
```

OUTPUT:



× Составьте матрицу M

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

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

Иными словами,

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

Видно, что матрица M является матрицей смежности графа G. А библиотека NetworkX предоставляет конструктор массива NumPy, который представляет матрицу смежности графа.

```
n = int(G.number_of_nodes()) # Get the total number of nodes in the graph

A = nx.to_numpy_array(G) # Get the adjacency matrix of graph G
M = np.zeros((n, n))

for i in range(n):
    out_deg = G.out_degree(i)
    for j in range(n):
        if A[i,j] == 1:
            M[i,j] = np.round(A[i,j] / out_deg,1)
```

OUTPUT:

M =

0,00	0,00	0,50	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,50	0,00	0,00
0,00	0,00	0,00	0,00	0,00	0,00	0,50	0,00	0,00	0,50	0,00	0,00	0,00	0,00	0,00
0,00	0,00	0,00	0,00	0,20	0,00	0,00	0,00	0,20	0,20	0,20	0,20	0,00	0,00	0,00
0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
0,33	0,00	0,00	0,00	0,00	0,00	0,00	0,33	0,00	0,00	0,00	0,00	0,00	0,00	0,33
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
0,00	0,00	0,00	0,00	0,50	0,00	0,00	0,00	0,00	0,00	0,50	0,00	0,00	0,00	0,00
0,00	0,00	0,25	0,25	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,25	0,25
0,00	0,00	0,00	0,50	0,50	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
0,00	0,00	0,00	0,00	0,00	0,00	0,50	0,00	0,00	0,00	0,00	0,50	0,00	0,00	0,00
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,33	0,00	0,33	0,33	0,00	0,00
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00
0,50	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,50	0,00
0,00	0,00	0,00	0,00	0,25	0,00	0,25	0,00	0,25	0,00	0,25	0,00	0,00	0,00	0,00
0,00	0,00	0,00	0,00	0,50	0,50	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

× Найдите собственный вектор матрицы M, соответствующий наибольшему собственному числу.

```
eigenvalues, eigenvectors = np.linalg.eig(M.T)
max_eigenvalue = np.argmax(eigenvalues)
largest_eigenvector = eigenvectors[:, max_eigenvalue]
print(max_eigenvalue)
print(largest_eigenvector)
```

OUTPUT:

$\lambda_{max} = 2$
→ $v_{largest}$ = [-0.338+0.j ; 0.+0.j; -0.239+0.j; -0.111+0.j-0.578+0.j ; -0.241+0.j ; -0.11+0.j ; -0.208+0.j ; -0.102+0.j ; -0.11+0.j ; -0.161+0.j ; -0.169+0.j ; -0.241+0.j ; -0.186+0.j ; -0.447+0.j]

Результат содержит комплексные числа, но все мнимые части равны 0, поэтому мы можем получить только действительную часть.

```
Largest_eigvc = largest_eigenvector.real
largest_eigvc
```

OUTPUT:

([-0,33808, 0, -0,23884629, -0,11126099, -0,57769495, -0,24133358, -0,10952852, -0,20806324, -0,1019151-0, 10963114, -0,16108696, -0,16885844 , -0,24066133, -0,18621748, -0,44671416])

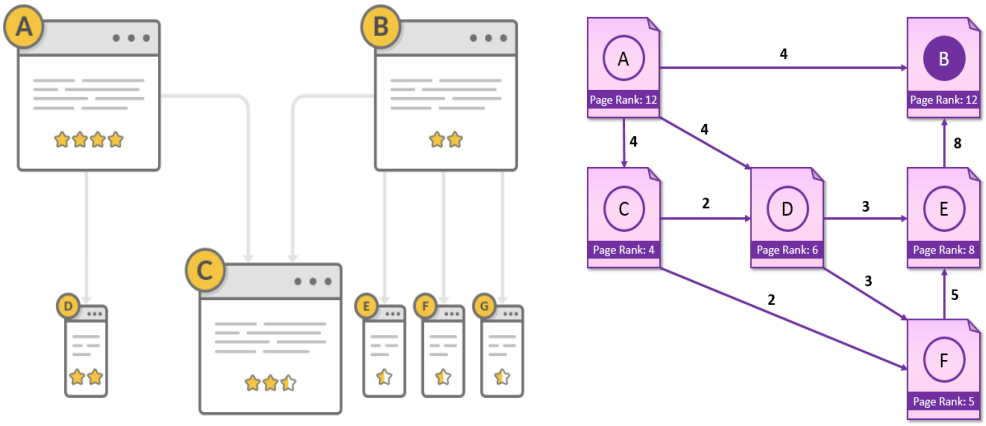
РАНЖИРУЙТЕ ВЕБ-СТРАНИЦЫ (ВЕРШИНЫ ВАШЕГО ГРАФА) В СООТВЕТСТВИИ С PAGERANK-АЛГОРИТМОМ ПРИ ОТСУТСТВИИ ЗАТУХАНИЯ (ТО ЕСТЬ, ПРИ D = 1).

- **Идея алгоритма PageRank**

Предположим, у нас есть набор из n веб-страниц с номерами от 1 до...n, PageRank веб страницы i рассчитывается на основе других веб-страниц, ссылающихся на нее (веб сайт j ссылается на i), но не каждая ссылка получает одинаковую оценку, нам нужен честный алгоритм! Алгоритм PageRank построен на основе двух основных идей:

- Веб-сайт A ссылается на B. Если сайт A имеет высокий рейтинг на доске, то он должен помочь B занять более высокий рейтинг.
- Веб-сайт A ссылается на B, количество веб-сайтов, на которые указывает A, обратно пропорционально рейтингу B, другими словами, чем больше страниц, на которые указывает A, тем меньше это помогает B повысить свой рейтинг.

Обычно, чем выше PageRank страницы, тем приоритетнее ее положение на странице результатов поиска.



Изображение для алгоритма PageRank
Источник: Интернет

Оригинальная формула ***PageRank***:

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots\right)$$

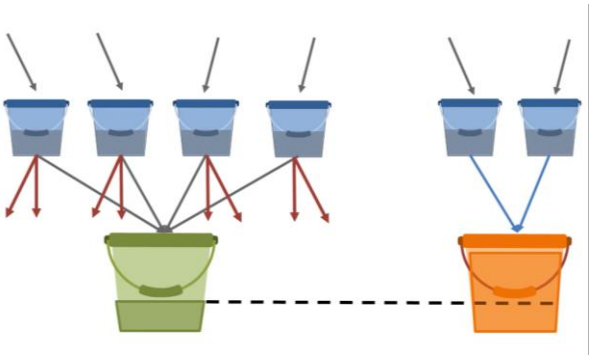
Где A, B, C и D — количество страниц,

L — количество ссылок, исходящих с каждой страницы, а N — общее количество страниц.

Коэффициента демпфирования d: он составляет примерно 0 < d < 1. Его можно понимать как вероятность нажатия на страницу.

Например

Здесь у нас есть синие ведра с равным количеством воды (другими словами, их рейтинг страницы или сила аналогичны).



Для случая А зеленого ведра, в котором имеют 4 синие ведра, в каждом ведре есть 3 исходящих ссылок, из которых 2 красных ссылок выходят на другую сторону, а только 1 ссылку в зеленое ведро

А в случае В с оранжевым ведром имеют только 2 синие ведра, и из каждого ведра выходит только одна ссылка, и все они входят в оранжевом ведро.

Видно, что хотя в случае А синих ведер больше, но поскольку у них слишком много исходящих связей, количество воды, поступающей в зеленое ведро, меньше, чем в оранжевое ведро в случае Б, где всего 2 синих ведра.

Отсюда мы можем сделать вывод, что если мы хотим, чтобы наш веб-сайт стал сильнее или сила, которую мы хотим передать, была большей, мы можем уменьшить количество ссылок на моей странице, чтобы поток силы не разделялся слишком сильно.

- **Ранжируйте веб-страницы соответствии с PageRank-алгоритмом**

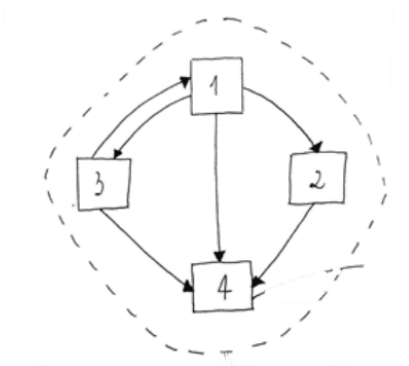
Рейтинг страниц на основе случайных пользователей. Допустим, мы путешествуем по Интернету, нажимая на случайные ссылки. Это можно понимать как цепь Маркова.
На предыдущем уроке, говоря о цепях Маркова, мы знали, что:
Цепи Маркова помогают предсказать поведение системы при переходе из одного состояния в другое, учитывая только текущее состояние. В каждый момент времени «t» пользователь переходит из состояния «i» в состояние «j» с вероятностью Pij (Pij называется вероятностью перехода).
Вероятность перехода помогает определить, каким будет следующее состояние объекта, рассматривая только текущее состояние, а не предыдущие состояния.

Чтобы определить рейтинг страниц веб-сайта, мы можем использовать матрицу смежности, чтобы представить ссылки от каждого вершины к каждому другой вершине и вычислить собственные векторы этой матрицы.

$$A_{ij} = \begin{cases} 1 & \text{если } \exists E(i,j), i \to j \\ 0 & \text{если } \nexists E(i,j) \end{cases}$$

В случае сайтов нет исходящих ссылок, поэтому все переходы в состояние никогда не будут выходить из состояния. Проблема здесь в том, что сложно точно определить

рейтинг страниц, поскольку в конечном итоге все пользователи, проходящие через эту сеть, будут лишь переведены в 4 состояние.



В зависимости от престижности ранга «важность» страницы i будет рассчитываться как сумма значений PageRank всех страниц, указывающих на страницу i. Поскольку страница может ссылаться на множество других страниц, ее престиж также должен быть разделен с этими страницами. В случае, если эта страница не ссылается ни на одну страницу, для справедливого ранжирования ее значение будет разделено поровну между всеми остальными вершинами.

В матрице смежности A графа G, если мы нормализуем каждую строку так, чтобы ее сумма составляла единицу, то каждый элемент результата будет представлять вероятность перехода с одного веб-сайта на другой.

$$A_{ij} = \begin{cases} \text{если } \exists E(i,j), i \to j \\ 0 \quad \text{если } \nexists E(i,j) \end{cases}$$

На основе нормализованной матрицы смежности A собственный вектор, соответствующий наибольшему собственному значению матрицы A, часто используется для определения рейтинга веб-сайта.

Наибольшее собственное значение соответствует важности веб-сайта в сети ссылок. Если собственное значение велико, веб-сайт имеет высокий рейтинг.

Собственный вектор, соответствующий наибольшему собственному значению, часто называют доминирующим собственным вектором (dominant eigenvector), и он во многом определяет рейтинг веб-сайта.

Алгоритм Pagerank, основанный на цепях Маркова, аналогичен: каждый веб-сайт или вершина в графе считается состоянием. На этом этапе ссылка считается кнопкой перехода состояний, то есть на основе ссылки одно состояние может перейти в другое состояние в зависимости от вероятности. Например, люди, которые случайно просматривают Интернет по первоначальным ссылкам, могут переключаться с одной ссылки на другую, но через некоторое время им «наскучили одни и те же сайты», и вдруг переход на другую не имеет ничего общего с первой ссылкой. На этом этапе в графе будут висячие вершины, что является частным случаем матрицы, содержащей элемент 0, как упоминалось выше.

```python
def markov_chain(G,d=1):

    L = np.asmatrix(nx.to_numpy_array(G))
    n = len(L)
    p = np.repeat(1.0 / n, n) # Personalization vector

    # In case the graph has websites that do not link to any website
    # even though there are links to other websites pointing to it.
    dangling_weight = p
    dangling_node = np.where(L.sum(axis=1)==0)[0]

    for node in dangling_node:
        L[node] = dangling_weight
    L /= L.sum(axis=1)
    L = d * L + (1 - d) * p
    return L
L = markov_chain(G)

def pagerank(G,d=1):
    if len(G) == 0: return {}
    L = markov_chain(G,d=1)

    eigvls, eigvcs = np.linalg.eig(L.T)
    max_eigvls = np.argmax(eigvls)
    print(max_eigenvalue)

    largest_eigvcs = np.array(eigvcs[:,max_eigvls]).flatten().real
    print(largest_eigvcs)
    norm = float(largest_eigvcs.sum())
    return dict(zip(G, largest_eigvcs / norm))
pagerank(G)
```

OUTPUT:

```
{0: 0.09836066501703916,
 1: 0.0047841649348219824,
 2: 0.06959737335080886,
 3: 0.036799635923463574,
 4: 0.1732420160851413,
 5: 0.07176247402232978,
 6: 0.04063257166129448,
 7: 0.06253150362986903,
 8: 0.032765190162348565,
 9: 0.03878954740339336,
 10: 0.05308147599299586,
 11: 0.05579223863767908,
 12: 0.07165832277434,
 13: 0.05624620222945957,
 14: 0.13395661817501525}
```