МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Допущен к защите

Заведующий кафедрой ПМИ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.В. Разова

**Разработка системы ссылочного ранжирования**

Курсовой проект по дисциплине  
«Теория алгоритмов»

Выполнил студент группы ФИб-3301-51-00     / Д. О. Ощепков/

Руководитель к.т.н.,

доцент кафедры ПМИ     / А. Г. Татаринова/

Работа защищена с оценкой     \_\_\_\_.\_\_\_\_.2024 г.

Члены комиссии:     / /

    / /

Киров 2024

Содержание

[Введение 4](#_Toc169132356)

[1 Обзор алгоритмов ссылочного ранжирования 5](#_Toc169132357)

[1.1 Постановка задачи 6](#_Toc169132358)

[1.2 Метрики оценки качества 6](#_Toc169132359)

[1.2.1 Pairwise orderedness 7](#_Toc169132360)

[1.2.2 Precision 7](#_Toc169132361)

[1.2.3 Recall 8](#_Toc169132362)

[1.2.4 метрика и ее частный случай метрика 8](#_Toc169132363)

[1.3 Обзор рассматриваемых алгоритмов 9](#_Toc169132364)

[1.3.1 Pagerank 9](#_Toc169132365)

[1.3.2 HITS 13](#_Toc169132366)

[1.4 Выводы по разделу 18](#_Toc169132367)

[2 Применение алгоритмов на практике 19](#_Toc169132368)

[2.1 Google Web Graph 20](#_Toc169132369)

[2.1.1 Pagerank 22](#_Toc169132370)

[2.1.2 HITS 24](#_Toc169132371)

[2.1.3 Сравнение производительности 25](#_Toc169132372)

[2.2 MovieLens 20M Dataset 26](#_Toc169132373)

[2.2.1 Pagerank 32](#_Toc169132374)

[2.2.2 HITS 34](#_Toc169132375)

[2.2.3 Сравнение производительности и метрик качества 35](#_Toc169132376)

[2.3 Выводы по разделу 37](#_Toc169132377)

[Заключение 38](#_Toc169132378)

[Библиографический список 39](#_Toc169132379)

[Приложения 40](#_Toc169132380)

[Приложение А. Таблицы со временем выполнения алгоритма 40](#_Toc169132381)

[Приложение B. Листинг программы 42](#_Toc169132382)

# Введение

Ранжирование – процесс упорядочивания элементов по значимости, релевантности или другим критериям. Самый простой пример ранжирования – это сортировка.

Однако, не всегда есть явный критерий, который позволит упорядочить множество объектов по наибольшей значимости для какой-либо задачи. В этом курсовом проекте мы рассмотрим алгоритмы, выполняющие присвоение ранга объектам из некоторого множества на основе связей между ними.

Рассмотрим более конкретный пример. Веб-страницы представляют в основном HTML-текст, в котором есть гиперссылки на другие веб-страницы, что может быть представлено ориентированным графом, в котором каждая веб-страница является вершиной, а гиперссылка – направленным ребром [1, стр. 389]. Задача ранжирования будет представляться в этом случае как определение значимости (важности) каждой веб-страницы относительно других.

Если сформулировать задачу для абстрактных взаимосвязанных объектов, то ссылочное ранжирования заключается в том, чтобы определить степень важности каждой вершины на основе структуры графа, образованного самими элементами и связями между ними.

Алгоритмы ссылочного ранжирования часто применяются в поисковых системах для ранжирования результатов поиска [2], в социальных сетях для определения влиятельных пользователей или контента [3], в анализе сетей для выявления ключевых узлов [4].

Актуальность курсового проекта обусловлена широким применением алгоритмов ссылочного ранжирования для решения большого спектра различных задач. Поэтому исследование производительности данных алгоритмов при различных условиях является важным.

Цель курсового проекта заключается в реализации известных алгоритмов ссылочного ранжирования и исследование данных алгоритмов по производительности на реальных данных, а также формирование рекомендаций по их выбору.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Выполнить обзор существующих алгоритмов ссылочного ранжирования, описать принципы их работы на примерах. Описание и анализ методов ранжирования.
2. Выбрать и проанализировать данные для проведения экспериментального исследования.
3. Реализовать алгоритмы ссылочного ранжирования.
4. Провести исследование реализованных алгоритмов по производительности, проанализировать полученные результаты и сформулировать выводы.

Курсовой проект включает два раздела. В первом разделе проведен обзор литературы и алгоритмов по предметной области. Во втором разделе был проведен анализ алгоритмов ранжирования.

# Обзор алгоритмов ссылочного ранжирования

Первым алгоритмом ссылочного ранжирования был Pagerank и его вариации, такие как personalized pagerank, weighted pagerank, personalized weighted pagerank. Они используют итеративный процесс для расчета весов страницы, учитывая количество и качество входящих ссылок [2]. Однако, эти алгоритмы оказались уязвимы к манипуляциям, так как спамеры научились искусственно наращивать ссылочную массу сайтов.

В ответ на эту проблему был разработан алгоритм TrustRank. В отличие от PageRank, который рассматривает все ссылки равнозначно, TrustRank отдает предпочтение ссылкам с доверенных сайтов.

Помимо PageRank и TrustRank, существует еще один важный алгоритм ссылочного ранжирования. Используется для определения важности веб-страниц на основе их характеристик как "хабов" и "авторитетов". Хабы (Hubs) – это страницы, которые содержат много ссылок на другие страницы. Чем больше ссылок у страницы, тем она более "хабовая". Авторитеты (Authorities) – это страницы, на которые много ссылается. Такие страницы считаются важными и качественными и обладают высоким "авторитетом" [5, стр. 607].

## Постановка задачи

Дано:

* Множество объектов , где - количество объектов.
* Ориентированный граф , представляющий связи между объектами. Ребро означает, что объект "ссылается" на объект .

Алгоритм должен присвоить каждому объекту числовой ранг , отражающий его относительную "важность" в контексте графа .

## Метрики оценки качества

В дальнейшем нам понадобится следующее:

Таблица 1 The Contingency Table for Category . [7, p. 38]

|  |  |  |  |
| --- | --- | --- | --- |
| Category | |  | |
| YES | NO |
|  | YES |  |  |
| NO |  |  |

Таблица адаптирована под курсовой проект.

Здесь – Истинно положительный (True Positive), – Ложно Положительный (False Positive), – Ложно Отрицательный (False Negative), – Истинно Отрицательный (True Negative).

Метрики ниже взяты из статьи [6, 578].

Формализуем человеческую оценку в функциональный вид

Где , а – множество вершин графа страниц

Хорошая страница или плохая можно определить по-разному на практике. Этой оценкой может быть, например, то, что значение, даваемое экспертом, выше, чем среднее значение по всем оценкам страниц.

### Pairwise orderedness

Для данной метрики понадобиться вспомогательная функция, которая будет сигнализировать, если плохая страница получила равный или более высокий рейтинг.

– функция, возвращающая ранг вершины согласно алгоритму, а – это вершины графа.

Теперь можем сформулировать pairwise orderedness

Если pairord равна 1, не существует случаев, когда T неверно оценил пару. Наоборот, если pairord равна нулю, то T неверно оценил все пары.

### Precision

Показывает долю высоко оцененных вершин, которые действительно являются хорошими, от всех высоко оцененных вершин.

Precision показывает способность модели точно предсказывать положительный класс. Если вы хотите избегать ложных срабатываний и быть уверенными в том, что предсказанный положительный класс действительно является положительным, вам нужно стремиться к высокому значению precision.

Формулу можно преобразовать в следующее:

### Recall

Показывает долю хороших вершин, которые высоко оценены алгоритмом, от всех хороших вершин.

Recall показывает способность модели обнаруживать все положительные примеры. Если важно, чтобы модель обнаруживала как можно больше положительных случаев, даже если это приводит к большему количеству ложных срабатываний, то стоит стремиться к высокому значению recall.

Формулу можно преобразовать в следующее.

### метрика и ее частный случай метрика

Является объединением Precision и Recall, показывает приоритет одной метрики над другой. При метрики учитываются в равной степени.

-мера полезна если данные несбалансированны, так как она учитывает и точность, и полноту, что позволяет получить общую оценку производительности модели, не учитывая смещение в сторону преобладающего класса.

## Обзор рассматриваемых алгоритмов

На практике матрица смежности, построенная на основе графа, часто бывает очень разряженной. Хранение таких матриц в стандартных массивах делает умножение и хранение матриц очень затратными задачами. Поэтому используются специальные структуры данных, такие как сжатый строчный или столбцовый формат, в которых нулевые элементы не хранятся. Все операции оптимизированы так, чтобы избегать ненужных вычислений, таких как умножение на ноль.

### Pagerank

Представьте себе случайного пользователя, который начинает с веб-страницы (узла веб-графа) и выполняет случайное блуждание по сети следующим образом. На каждом временном шаге пользователь переходит с текущей страницы A на случайно выбранную веб-страницу, на которую A ссылается гиперссылкой. На рисунке 1‑1 показан пользователь в узле A, из которого есть три гиперссылки на узлы B, C и D; на следующем временном шаге пользователь переходит к одному из этих трех узлов с равными вероятностями 1/3.

По мере того, как пользователь переходит от узла к узлу в этом случайном блуждании, он посещает некоторые узлы чаще, чем другие; интуитивно понятно, что это узлы, на которые ссылается множество других часто посещаемых узлов. Идея, лежащая в основе PageRank, заключается в том, что страницы, которые посещаются чаще в этом блуждании, являются более важными [1, стр. 424].

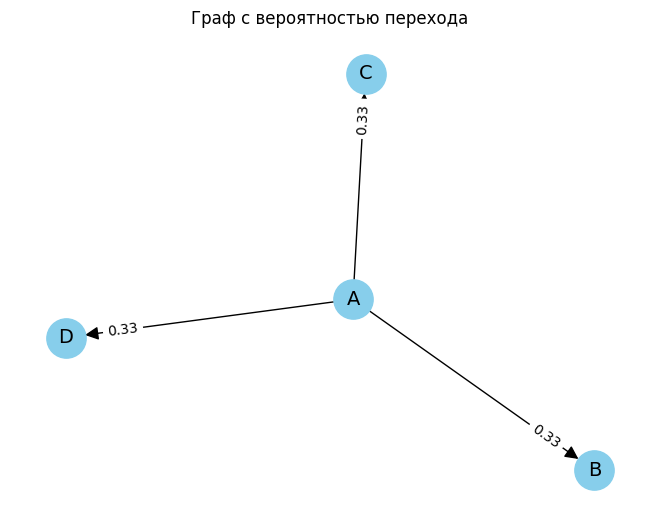


Рисунок 1‑2 Иллюстрация из книги[1] для объяснения идеи Pagerank

Естественным образом из базового алгоритма вытекает weighted pagerank. Если известно не только наличие ссылок между вершинами графа, а еще вероятности перехода по ним, то вместо (n – количество исходящих ссылок) в качестве вероятности перехода от вершины к вершине, мы можем взять сами вероятности переходов [11, стр. 4].

Существует персонализированная версия pagerank, которая учитывает уникальные вероятности переходов от вершины к вершине для каждой вершины, путем добавления специального вектора смещения [10, стр. 163].

В работе мы объединим все подходы и реализуем взвешенный персонализированный pagerank как наиболее гибкий вариант.

Входные данные:

* – матрица переходов графа, где - вероятность перехода от узла к узлу .
* – вектор персонализации, отражающий предпочтения пользователя к определенным страницам.
* – коэффициент, определяющий вес матрицы переходов.
* – коэффициент, определяющий вес вектора персонализации.
* – коэффициент, необходимый для чтобы каждая веб-страница имела самоцикл с ненулевой вероятностью, а также каждая пара веб-страниц была связана.
* – количество итераций.
* Алгоритм:

1. Инициализация:  
   Создаем вектор PageRank и устанавливаем начальные значения:
2. Итеративный процесс:

Для от 1 до :

Выходные данные:

* вектор PageRank, где представляет рейтинг узла .

Замечания:

* Коэффициенты , и должны удовлетворять условию

.

* Вектор персонализации должен быть нормализован, так что сумма его элементов равна 1.
* Количество итераций iterations влияет на точность результатов.

Использование разреженных матриц для представления графа значительно улучшает эффективность алгоритма PageRank, особенно для больших графов с небольшим количеством связей.

Временная сложность:

Цикл выполняется iterations раз.

Умножение разреженной матрицы на вектор: - поскольку мы перебираем только ненулевые элементы матрицы.

Умножение вектора на вектор: .

Сложение векторов:

Итого: Временная сложность алгоритма PageRank с использованием разреженных матриц составляет . Если количество связей намного больше, чем количество вершин, то можно упростить до .

Приведем простой пример.

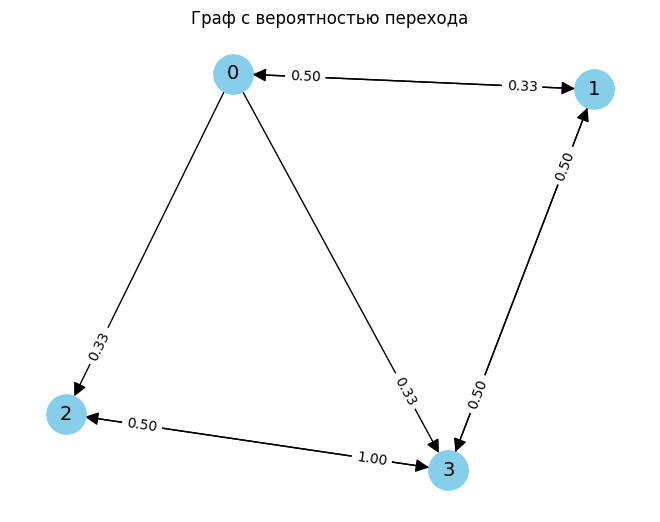


Рисунок 1‑3 Демонстрационный граф

Сделаем несколько итераций алгоритма с параметрами a=0.85, b=0 (нет персонализации), c=0.15. Матрица переходов имеет вид (каждое значение равно , если существует связь)

Инициализируем вектор .

Итерация 1:

Итерация 2:

Итерация 3:

Получаем, что узел 3 является наиболее важным в этом графе, что легко интерпретировать, потому что к нему идет больше всех сильных связей.

### HITS

Предназначен для поиска по широкой теме. Для любого запроса мы вычисляем два ранжированных списка результатов вместо одного. Ранжирование одного списка определяется оценками концентрации, а другого – оценками авторитетности.

Этот подход основан на особом понимании процесса создания веб-страниц, а именно на том, что существует два основных типа веб-страниц, полезных в качестве результатов поиска по широким темам. Под поиском по широкой теме мы подразумеваем информационный запрос, такой как «Я хочу узнать о лейкемии». Существуют авторитетные источники информации по этой теме; в данном случае страницей такого рода была бы страница Национального института рака о лейкемии. Мы будем называть такие страницы авторитетами; в вычислениях, которые мы собираемся описать, это страницы, которые будут иметь высокие показатели авторитетности [1, стр. 433].

Входные данные:

* : Разреженная матрица, представляющая ориентированный граф, где ненулевое, если есть ребро от узла к узлу .
* iterations: Число итераций алгоритма.

Алгоритм:

1. Инициализация:
   * Создать векторы hubs и authorities размера n (число узлов в графе).
2. Итеративный процесс:
   * Повторить iterations раз:
     + Обновление authorities:
       - Для каждого узла j:
     + Обновление hubs:
       - Для каждого узла j:
     + Нормализация:
       - Разделить все элементы hubs и authorities на их евклидову норму.

Где ­ мера авторитетности узла , — мера хабности узла . и ­ обновленные значения и соответственно [8, стр. 23].

Выходные данные:

* hubs: Вектор, содержащий hub оценки для каждого узла.
* authorities: Вектор, содержащий authority оценки для каждого узла.

Замечания:

* Использование разреженных матриц для представления графа значительно улучшает эффективность алгоритма HITS, особенно для больших графов с небольшим количеством связей.
* Нормализация предотвращает экспоненциальный рост значений hubs и authorities в процессе итераций.

Временная сложность:

Цикл выполняется iterations раз.

Умножение разреженной матрицы на вектор: - поскольку мы перебираем только ненулевые элементы матрицы.

Нормализация: – нахождение суммы вектора и деление каждого элемента на эту сумму.

Итого: Временная сложность алгоритма PageRank с использованием разреженных матриц составляет . Если количество связей намного больше, чем количество вершин, то можно упростить до .

Приведем простой пример на графе из рисунка 1‑4.

Сделаем несколько итераций алгоритма. Матрица переходов имеет вид (каждое значение равно , если существует связь)

Инициализируем вектор .

В матричном виде формулы обновления имеют следующий вид:

.

Итерация 1:

Итерация 2:

Итерация 3:

Наибольший авторитет также имеет узел 3.

Существует еще один подход нахождения hits, который вытекает из следующей теоремы.

Теорема 3.2 – это главный собственный вектор матрицы , а – главный собственный вектор матрицы [9, стр. 11].

Согласно выводу выше достаточно найти главный собственный вектор – это будет , затем по формуле итеративного нахождения hubness найти .

Существует много способов найти собственный вектор матрицы, давайте воспользуемся сингулярным разложением матрицы для этой цели. Любую матрицу M можно разложить следующим образом:

При этом – это диагональная матрица, по диагонали которой расположены собственные числа. При этом одно из свойств разложения, говорит, что столбцы матрицы V – это собственные вектора матрицы Остается только найти главный собственный вектор. Стоит отметить, что существуют библиотеки, которые позволяют найти, только один собственный вектор при сингулярном разложении, что довольно эффективно по времени.

Приведем пример для матрицы M.

Разложим матрицу с помощью какого-либо математического пакета.

Главный собственный вектор – это вектор, соответствующий максимальному собственному значению (в нашем случае столбец 3). Получаем

## Выводы по разделу

В данной главе были рассмотрены теоретические основы ссылочного ранжирования и два его ключевых алгоритма: PageRank и HITS.

PageRank рассматривает всю сеть как единое целое и оценивает важность страницы, исходя из количества и "веса" входящих ссылок, интерпретируя ссылку как "голос" одной страницы за другую. Метод использует итеративный алгоритм, базирующийся на матрице переходов, для вычисления стационарного распределения PageRank в сети. Важным преимуществом PageRank является его устойчивость к случайным изменениям в структуре сети и способность выделять авторитетные узлы в масштабных графах.

Алгоритм HITS, в отличие от PageRank, ориентирован на контекст конкретного запроса и анализирует только подграф страниц, релевантных этому запросу. Он вычисляет две метрики: авторитетность (authority) и концентрацию (hub) страницы. Авторитетность отражает важность информации на самой странице, а концентрация – ее роль как "путеводителя" по другим важным ресурсам. HITS позволяет эффективно находить как авторитетные страницы, так и узлы-хабы, релевантные заданной теме.

# Применение алгоритмов на практике

В начале работы алгоритмы были реализованы на языке Python с использованием массивов numpy и библиотеки scipy. Но так как в ходе исследования итеративные версии алгоритмов не показали хорошей производительности, то алгоритмы были также реализованы на языке C++.

Для реализации HITS с помощью сингулярного разложения использовался Python версии 3.12. Выбор был обусловлен высокопроизводительными алгоритмами из библиотеки scipy.

В работе были использованы следующие библиотеки языка Python:

* NumPy обеспечивает эффективную работу с матрицами смежности графов.
* SciPy предоставляет алгоритмы линейной алгебры
* Pandas упрощает работу с таблицами
* Matplotlib позволяет визуализировать результаты работы алгоритмов, например

В работе были использованы следующие библиотеки языка C++:

* Eigen обеспечивает эффективную работу с матрицами, что важно для представления графов и выполнения операций линейной алгебры
* RapidCSV упрощает чтение и запись данных в формате CSV, позволяя легко загружать и сохранять информацию о графах и результатах работы алгоритмов

Результаты, рассмотренные далее, сравнивалиь с известной библиотекой Python NetworkX[[1]](#footnote-1). Эта библиотека позволяет удобно создавать граф из различных наборов данных, предоставляет готовые реализации алгоритмов PageRank и HITS, обладает гибкостью для собственных исследований, интегрируется с Matplotlib для визуализации и органично вписывается в экосистему Python для обработки данных и машинного обучения.

Замеры времени в работе были получены путем многократного запуска и усреднения результатов для каждого подмножества графа.

## Google Web Graph[[2]](#footnote-2)

Этот набор данных представляет собой граф веб-страниц, где узлы соответствуют веб-страницам, а ориентированные ребра – гиперссылкам между ними. Данные были опубликованы Google в 2002 году в рамках конкурса Google Programming Contest.

Количество узлов: 875 713

Количество ребер: 5 105 039

Проведем небольшой дополнительный анализ количества входящих и исходящих ребер у верши.

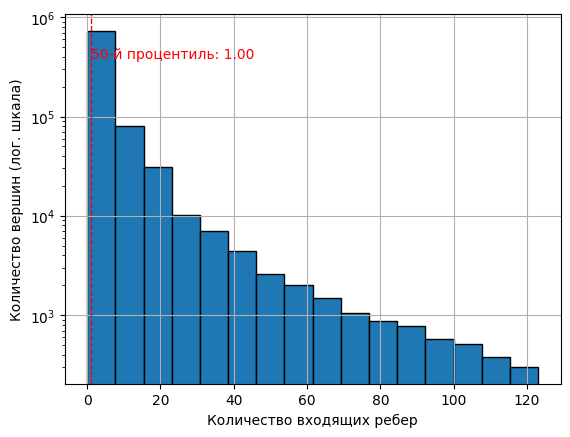


Рисунок 2‑1 Гистограмма распределения входящих ребер из вершин.

По оси x находится количество входящих ребер, а по оси y количество вершин, имеющих такое число входящих ребер.

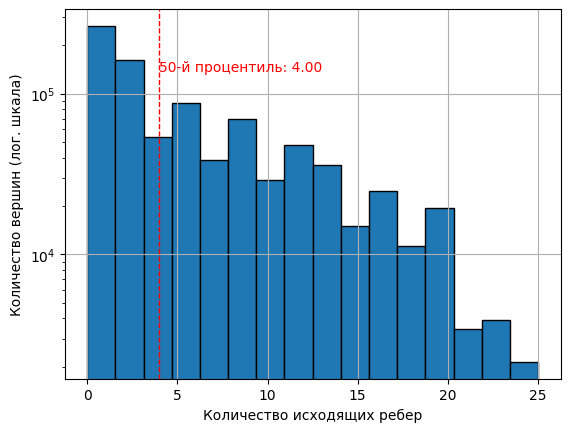


Рисунок 2‑2 Гистограмма распределения исходящих ребер из вершин.

По оси x находится количество исходящих ребер, а по оси y количество вершин, имеющих такое число входящих ребер.

Графики демонстрируют высокую разреженность полученного графа.

Для тестирования скорости работы алгоритма случайным образом выбиралась доля ребер для формирования подграфа, на котором проводились замеры скорости.

### Pagerank

На рисунке ниже представлена зависимость времени выполнения PageRank от размера графа для различных реализаций алгоритма (в частях от полного набора данных).

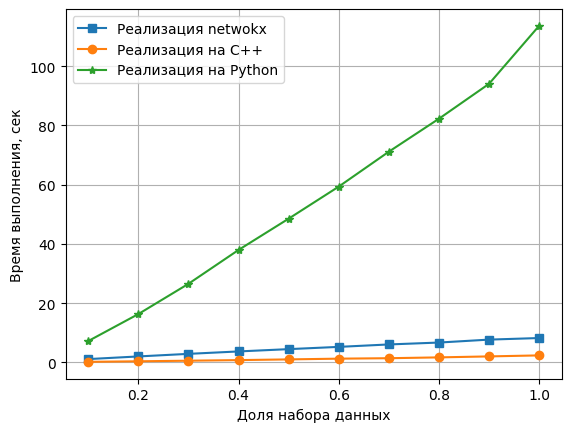
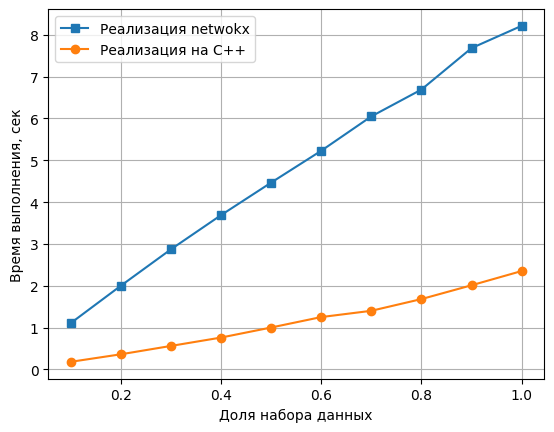


Рисунок 2‑3 Скорость работы реализации pagerank на C++, Python и реализации networkx в зависимости от доли датасета Google Web Graph.

Этот рисунок подтверждает теоретическую линейную сложность алгоритма, зависящую от количества ребер графа.

Реализация на С++ показывает свое превосходство по времени благодаря тому, что граф хранится сразу в нужном формате и низкоуровневой природе языка. Благодаря этим преимуществом она быстрее, чем аналог из networkx. Это демонстрирует рисунок ниже.



Графики был сформирован по таблице 5 из приложения A.

### HITS

На рисунке представлена зависимость времени выполнения рассматриваемых реализаций HITS от размера графа (в частях от полного набора данных).

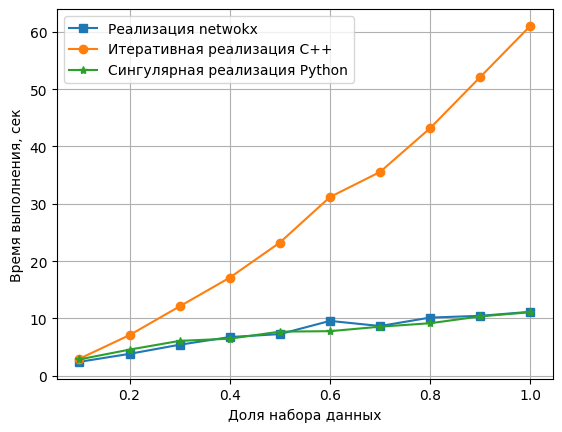


Рисунок 2‑4 Скорость работы итеративного и сингулярного варианта HITS зависимости от доли датасета Google Web Graph.

Вариант с сингулярным разложением значительно превосходит по скорости итеративную вариацию.

Реализация network и Python идут очень близко на иллюстрации, поэтому рассмотрим их на более крупном масштабе.

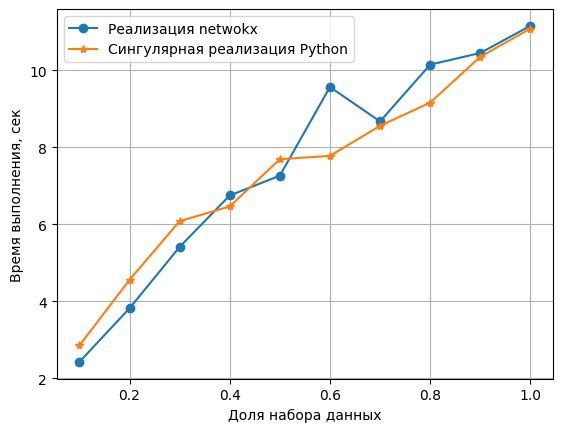


Рисунок 2‑5 Сравнение сингулярной и network реализаций HITS зависимости от доли датасета Google Web Graph.

Учитывая, что NetworkX производит преобразование данных в нужный формат, то самостоятельная реализация работает медленнее. Это связано с тем, что в библиотечной реализации указано максимальное количество итераций при сингулярном разложении. Рисунок и таблица ниже демонстрируют скорость работы.

Рисунки построены по таблице 6 из приложения A.

### Сравнение производительности

В данном случае 100 итераций pagerank оказались быстрее, чем сингулярное разложение алгоритма HITS.

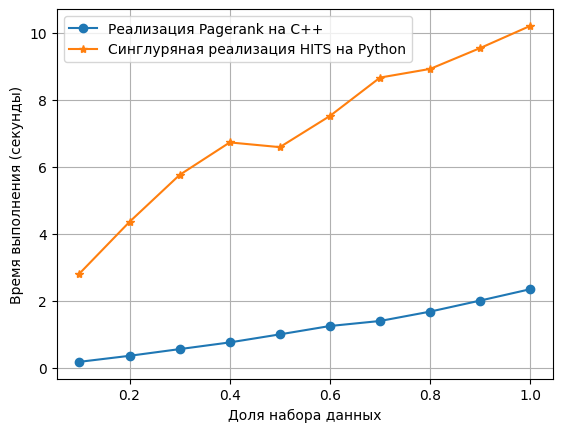


Рисунок 2‑6 Зависимость времени выполнения от размера выборки алгоритмов PageRank и HITS на датасете Google Web Graph.

Рисунок был построен по таблице 7 из приложения А.

## MovieLens 20M Dataset[[3]](#footnote-3)

Набор данных описывает рейтинги и теги, присвоенные пользователями сервиса рекомендаций фильмов MovieLens. Он содержит 20 000 263 рейтинга и 465 564 тега, относящихся к 27 278 фильмам. Эти данные были созданы 138 493 пользователями в период с 9 января 1995 года по 31 марта 2015 года. Сам набор данных был сгенерирован 17 октября 2016 года.

Пользователи для включения в набор данных выбирались случайным образом. Все выбранные пользователи оценили не менее 20 фильмов.

Содержание:

Демографическая информация не включена. Каждый пользователь представлен идентификатором, и никакой другой информации не предоставляется.

Данные содержатся в шести файлах:

tag.csv содержит теги, примененные пользователями к фильмам (не использовался):

* userId
* movieId
* tag
* timestamp

rating.csv содержит рейтинги фильмов, выставленные пользователями:

* userId (идентификатор пользователя)
* movieId (идентификатор фильма)
* rating (оценка, данная пользователем фильму)
* timestamp (временная метка, не использовалась)

movie.csv содержит информацию о фильмах:

* movieId (идентификатор фильма)
* title (полное название)
* genres (жанр)

link.csv содержит идентификаторы, которые можно использовать для связи с другими источниками (не использовался):

* movieId
* imdbId
* tmbdId

genome\_scores.csv содержит данные о релевантности тегов к фильмам (не использовался):

* movieId
* tagId
* relevance

genome\_tags.csv содержит описания тегов (не использовался):

* tagId
* tag

В рамках исследования алгоритмов ссылочного ранжирования был проведён эксперимент их применения на данных для рекомендательных систем, в котором оценивалась важность фильмов в зависимости от пользовательских отзывов. Для этого строился граф на основе данных о рейтингах пользователей. Вершинами графа являются пользователи сервиса, а фильмами – объекты рекомендаций. Ребро между пользователем и фильмом проводится, если пользователь поставил оценку этому фильму. Вес ребра соответствует значению оценки.

Приведем пример. Если rating.csv имеет следующий вид

Таблица 2 Пример rating.csv

|  |  |  |
| --- | --- | --- |
| userId | movieId | rating |
| user1 | movie1 | 5 |
| user1 | movie2 | 3 |
| user2 | movie2 | 4.5 |
| user3 | movie1 | 4 |
| user3 | movie2 | 3.5 |

Тогда граф будет иметь вид

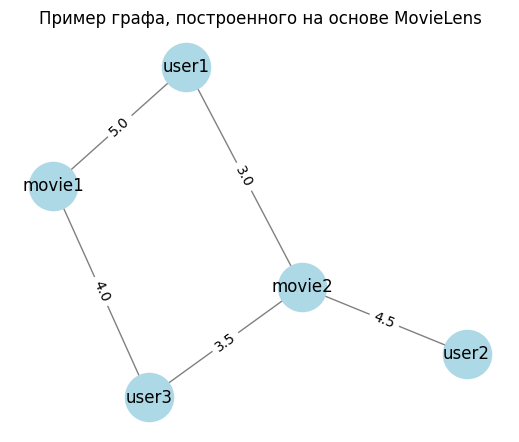


Рисунок 2‑7 Иллюстрация преобразования оценок пользователей в граф.

Проведем анализ описанного датасета.

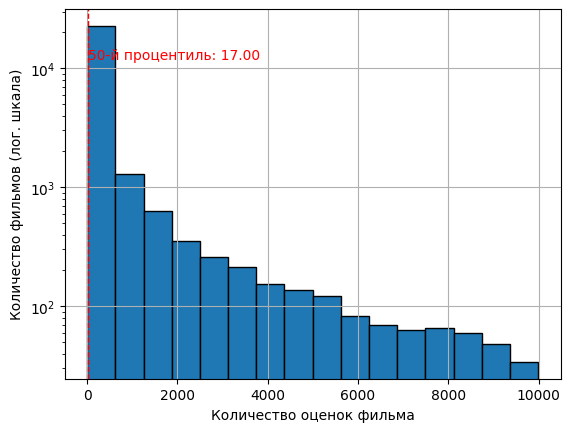


Рисунок 2‑8 Гистограмма распределения количества оценок у фильмов.

По оси x на рисунке количество оценок фильма, а по оси y количество фильмов с таким числом оценок.

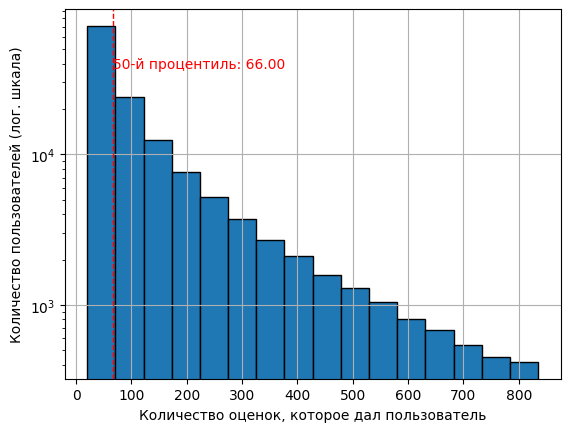


Рисунок 2‑9 Гистограмма распределения количества оценок, которое дал пользователь.

По оси x на рисунке количеств оцененных фильмов пользователем, а по оси y количество пользователей с таким количеством оцененных фильмов.

Полученный граф имеет следующее распределение количества ребер по вершинам.

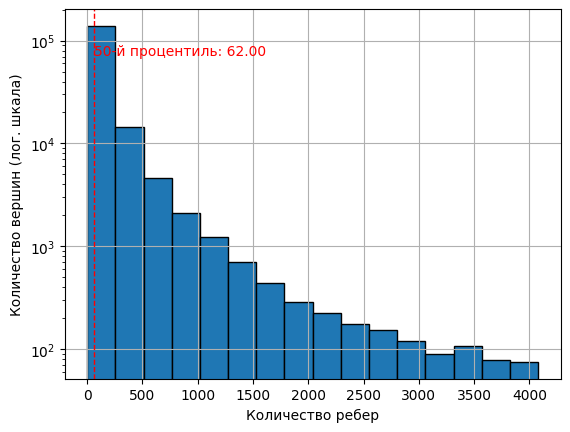


Рисунок 2‑10 Гистограмма распределения количества ребер у вершин для полученного графа.

Графики демонстрируют высокую разреженность полученного графа.

Для тестирования скорости работы алгоритма случайным образом выбиралась доля ребер для формирования подграфа, на котором проводились замеры скорости.

### Pagerank

На рисунке ниже представлена зависимость времени выполнения реализаций PageRank от размера графа (в частях от полного набора данных).

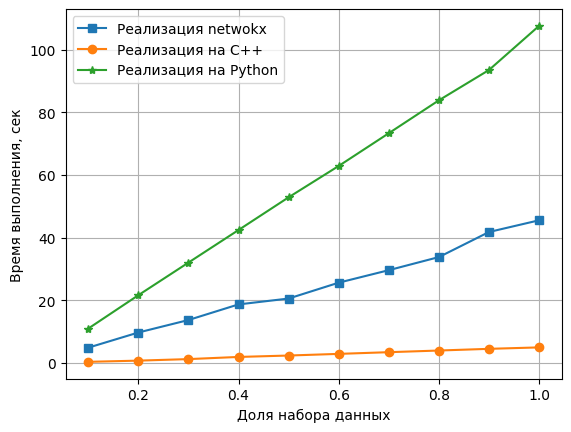


Рисунок 2‑11 Время выполнения 100 итераций pagerank в зависимости от доли датасета MovieLens.

Этот рисунок подтверждает линейную зависимость времени выполнения алгоритма в зависимости от m, что согласуется с теоретической оценкой.

Благодаря тому, что код написан на C++ и отсутствию промежуточного представления графа (в самостоятельной реализации граф сразу хранится в матрице смежности) мы получаем производительность в несколько раз превосходящую реализацию из NetworkX.

На данном датасете возможно оценить качество полученных результатов. Воспользуемся метриками из главы 1.2. Для этого нужно определить функцию Пусть она будет иметь вид

То есть, мы считаем фильм «хорошим», если пользователь оценил фильм выше, чем данная им средняя оценка всем фильмам.

Тогда усреднено для всех пользователей получаем следующие значения:

Accuracy: 0.5340

Precision: 0.5250

Recall: 0.9333

F1 Score: 0.6720

Pairorder: 0.6701

Рисунок в этом параграфе построен по таблице 8 из приложения A.

### HITS

На рисунке ниже представлена зависимость времени выполнения реализаций HITS от размера графа (в частях от полного набора данных).

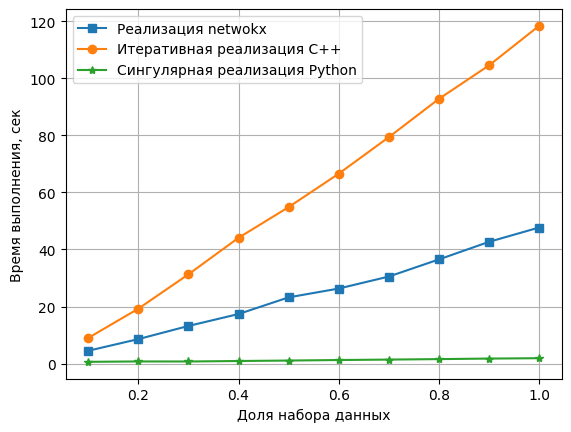


Рисунок 2‑12 Время выполнения реализаций HITS в зависимости от размера датасета MovieLens.

Этот рисунок подтверждает линейную зависимость времени выполнения алгоритма в зависимости от m, что согласуется с теоретической оценкой.

Второй вариант подсчета рейтинга HITS (с помощью сингулярного разложения) показал гораздо более хорошее время и при этом, он имеет такие же показатели метрик.

В алгоритме HITS самостоятельная реализация также показывает свое превосходство. Но стоит отметить, что в такая скорость достигается только за счет того, что в самостоятельной реализации граф сразу хранится необходимой для алгоритма матрице смежности.

На рассматриваемом датасете возможно оценить качество полученных результатов. Воспользуемся метриками из главы 1.2. Для этого нужно определить функцию Пусть она будет иметь вид

Тогда усреднено для всех пользователей получаем следующие значения:

Accuracy: 0.5681

Precision: 0.5492

Recall: 0.8666

F1 Score: 0.6724

Pairorder: 0.6712

Рисунок в этом параграфе построен по таблице 9 из приложения A.

### Сравнение производительности и метрик качества

PageRank демонстрирует более высокую скорость работы по сравнению с итеративным HITS. Это связано с тем, что в HITS на каждую итерацию приходится 2 умножения матрицы на вектор, в то время как в PageRank только одно, что является менее ресурсоемкой задачей. Однако, HITS, рассчитываемый с помощью сингулярного разложения, имеет скорость лучше, чем pagerank. Скорость работы хорошо демонстрирует рисунок ниже.

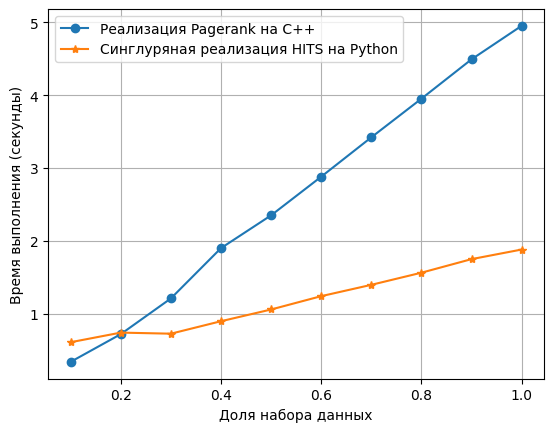


Рисунок 2‑13 Зависимость времени выполнения от размера выборки алгоритмов PageRank и HITS на датасете MovieLens.

Анализ результатов применения алгоритмов PageRank и HITS к набору данных MovieLens демонстрирует различия в их способности ранжировать и рекомендовать фильмы (таблица 4)

PageRank, продемонстрировав более высокий показатель Recall, эффективнее находит релевантные фильмы, то есть, с большей вероятностью включит в рекомендации все фильмы, которые пользователь потенциально оценит высоко. Это делает его предпочтительным выбором для рекомендательных систем, нацеленных на охват максимального количества подходящих вариантов.

HITS, в свою очередь, выделяется более высоким показателем Precision, что говорит о его точности в отборе фильмов для рекомендаций. Алгоритм сфокусирован на минимизации ложных срабатываний, то есть, включения в рекомендации фильмов, которые пользователю не понравятся. HITS оптимален в сценариях, где критически важно рекомендовать исключительно те фильмы, которые гарантированно понравятся пользователю, пусть даже ценой пропуска некоторых релевантных вариантов.

|  |  |  |
| --- | --- | --- |
| Метрика | HITS | PageRank |
| Accuracy | 0.5681 | 0.5340 |
| Precision | 0.5492 | 0.5250 |
| Recall | 0.8666 | 0.9333 |
| F1 Score | 0.6724 | 0.6720 |
| Pairorder | 0.6712 | 0.6701 |

Таблица 4 Сравнение метрик на датасете MovieLens.

## Выводы по разделу

Выполнение обоих алгоритмов линейно зависит от количества ссылок в сети. С увеличением количества ссылок, время выполнения алгоритмов возрастает прямо пропорционально.

Сфера применения: HITS лучше подходит для поиска информации по конкретным темам, так как позволяет выявлять страницы-концентраторы. PageRank же более универсален и применим для общего ранжирования веб-страниц.

PageRank продемонстрировал более высокий recall, в то время как HITS продемонстрировал более высокий precision. Но стоит отметить, что существуют более подходящие для персональных рекомендаций.

Преобразования данных в подходящий формат, может занимать даже больше времени, чем выполнение алгоритма.

Алгоритм, который будет работать быстрее, определяется структурой графа.

# Заключение

В данном курсовом проекте было проведено комплексное исследование алгоритмов ссылочного ранжирования.

PageRank, рассматривая сеть как единое целое, эффективно ранжирует страницы по глобальной важности, в то время как HITS концентрируется на поиске авторитетных источников информации и узлов-хабов в рамках конкретного тематического запроса.

Используйте библиотеки, которые работают с графом в нужном формате или реализуйте алгоритмы и хранение данных вручную так, чтобы избежать конвертацию данных. Это значительно ускорит процесс ранжирования.

Предпочитайте способ с сингулярным разложением методу простых итераций в алгоритме HITS, если только нет предпосылок, что алгоритм сойдется за малое количество итераций.

На датасете MovieLens PageRank продемонстрировал более высокий recall, в то время как HITS продемонстрировал более высокий precision. Выбирайте тот или иной алгоритм в зависимости от ваших требований по этим метрикам. Учитывайте, что метрики отличаются не слишком значительно, и что алгоритмы ранжируют фильмы по важности для группы пользователей (плохо подходит для персональной рекомендации, но может использоваться в совокупности с другими алгоритмами).

Код реализация на Python (https://github.com/DimaOshchepkov/PageRank-Course-Project/tree/main) находится в приложении Б, на С++ (https://github.com/DimaOshchepkov/LinkRanking) в приложении С.

Дальнейшие исследования в данной области могут быть направлены на:

Разработку гибридных алгоритмов, сочетающих сильные стороны PageRank и HITS.

Адаптацию алгоритмов к анализу динамических сетей с учётом временного фактора.

Исследование эффективности алгоритмов на различных типах данных.

Изучение возможности распараллеливания алгоритмов при помощи видеокарты.

# Библиографический список

1. Schütze, Hinrich, Christopher D. Manning, and Prabhakar Raghavan. Introduction to information retrieval. Vol. 39. Cambridge: Cambridge University Press, 2008.
2. Brin, Sergey, and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine." *Computer networks and ISDN systems* 30.1-7 (1998): 107-117.
3. Shahriari, Moshen, and Mahdi Jalili. "Ranking nodes in signed social networks." *Social network analysis and mining* 4 (2014): 1-12.
4. Zhu, Darui, et al. "Identification of key nodes in a power grid based on modified PageRank algorithm." *Energies* 15.3 (2022): 797.
5. Kleinberg, Jon M. "Authoritative sources in a hyperlinked environment." *Journal of the ACM (JACM)* 46.5 (1999): 604-632.
6. Gyöngyi, Zoltán, Hector Garcia-Molina, and Jan Pedersen. "Combating web spam with trustrank." *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004.
7. Sebastiani, Fabrizio. "Machine learning in automated text categorization." *ACM computing surveys (CSUR)* 34.1 (2002): 1-47.
8. Tanudjaja, Francisco. *Using web graph structures to personalize search*. Diss. Massachusetts Institute of Technology, 2001.
9. Kleinberg, Jon M. "Authoritative sources in a hyperlinked environment." *Journal of the ACM (JACM)* 46.5 (1999): 604-632.
10. Lofgren P., Banerjee S., Goel A. Personalized pagerank estimation and search: A bidirectional approach //Proceedings of the Ninth ACM International Conference on Web Search and Data Mining. – 2016. – С. 163-172.
11. Xing W., Ghorbani A. Weighted pagerank algorithm //Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004. – IEEE, 2004. – С. 305-314.

# Приложения

## Приложение А. Таблицы со временем выполнения алгоритма

Таблица 5 Скорость работы pagerank реализаций на C++ и Python в зависимости от доли датасета Google Web Graph.

|  |  |  |  |
| --- | --- | --- | --- |
| Часть от целого датасета | С++ (секунды) | Python (секунды) | NetworkX (секунды) |
| 0,1 | 0,18 | 7,18 | 1,10 |
| 0,2 | 0,36 | 16,29 | 2,00 |
| 0,3 | 0,56 | 26,50 | 2,87 |
| 0,4 | 0,76 | 37,93 | 3,69 |
| 0,5 | 1 | 48,49 | 4,46 |
| 0,6 | 1,25 | 59,32 | 5,23 |
| 0,7 | 1,4 | 71,10 | 6,05 |
| 0,8 | 1,68 | 82,23 | 6,69 |
| 0,9 | 2,01 | 94,01 | 7,68 |
| 1,0 | 2,35 | 113,60 | 8,21 |

Таблица 6 Скорость работы итеративного и сингулярного варианта HITS зависимости от доли датасета Google Web Graph.

|  |  |  |  |
| --- | --- | --- | --- |
| Часть от целого датасета | Простые итерации C++ (секунды) | Сингулярное разложение (секунды) | NetworkX (секунды) |
| 0,1 | 2,97 | 2,81 | 1,10 |
| 0,2 | 7,09 | 4,36 | 2,00 |
| 0,3 | 12,1 | 5,76 | 2,87 |
| 0,4 | 17,11 | 6,73 | 3,69 |
| 0,5 | 23,23 | 6,59 | 4,46 |
| 0,6 | 31,17 | 7,52 | 5,23 |
| 0,7 | 35,55 | 8,66 | 6,05 |
| 0,8 | 43,19 | 8,92 | 6,69 |
| 0,9 | 52,11 | 9,55 | 7,68 |
| 1,0 | 61,04 | 10,21 | 8,21 |

Таблица 7 Зависимость времени выполнения от размера выборки алгоритмов PageRank и HITS на датасете Google Web Graph.

|  |  |  |
| --- | --- | --- |
| Доля датасета | Pagerank (секунды) | HITS (секунды) |
| 0.1 | 0.18 | 2.81 |
| 0.2 | 0.36 | 4.36 |
| 0.3 | 0.56 | 5.76 |
| 0.4 | 0.76 | 6.73 |
| 0.5 | 1.00 | 6.59 |
| 0.6 | 1.25 | 7.52 |
| 0.7 | 1.40 | 8.66 |
| 0.8 | 1.68 | 8.92 |
| 0.9 | 2.01 | 9.55 |
| 1.0 | 2.35 | 10.21 |

Таблица 8 Время выполнения Pagerank и преобразования данных для него на части датасета MovieLens.

|  |  |  |  |
| --- | --- | --- | --- |
| Часть от целого датасета | Время выполнения | Время считывания | NetworkX (секунды) |
| 0,1 | 0,34 | 2,02 | 4,85 |
| 0,2 | 0,72 | 4,00 | 9,70 |
| 0,3 | 1,21 | 6,10 | 13,68 |
| 0,4 | 1,90 | 7,93 | 18,67 |
| 0,5 | 2,35 | 10,35 | 20,55 |
| 0,6 | 2,88 | 12,09 | 25,61 |
| 0,7 | 3,42 | 13,76 | 29,61 |
| 0,8 | 3,95 | 15,91 | 33,79 |
| 0,9 | 4,49 | 18,44 | 41,80 |
| 1,0 | 4,95 | 19,79 | 45,54 |

с

Таблица 9 Скорость работы реализаций HITS в зависимости от доли датасета MovieLens.

|  |  |  |  |
| --- | --- | --- | --- |
| Часть от целого датасета | Простые итерации (секунды) | Сингулярное разложение (секунды) | NetworkX (секунды) |
| 0,1 | 8,98 | 0,61 | 4,48 |
| 0,2 | 19,23 | 0,71 | 8,57 |
| 0,3 | 31,28 | 0,71 | 13,18 |
| 0,4 | 44,09 | 0,86 | 17,36 |
| 0,5 | 54,86 | 1,05 | 23,21 |
| 0,6 | 66,57 | 1,20 | 26,30 |
| 0,7 | 79,39 | 1,37 | 30,51 |
| 0,8 | 92,81 | 1,51 | 36,51 |
| 0,9 | 104,55 | 1,75 | 42,68 |
| 1,0 | 118,43 | 1,87 | 47,70 |

## Приложение B. Листинг программы Python

//ranking.common.py

import numpy as np

import pandas as pd

from scipy import sparse

from typing import Tuple, Mapping, Any

from scipy.sparse import coo\_matrix

from enum import Enum

from typing import Optional

from sklearn.model\_selection import train\_test\_split

from pathlib import Path

class BaseGraphContext:

""" Базовый класс, определяющий константы для типов узлов графа. """

class NodeType(Enum):

SOURCE = "source"

TARGET = "target"

MappingIntToObject = Mapping[int, Tuple[Any, BaseGraphContext.NodeType]]

class ReadFromPandasToTransition():

def \_read\_pd(self,

data: pd.DataFrame,

source: str, target: str,

edge\_attr: str,

personalization: Optional[Any] = None,

directional: bool = False) -> Tuple[sparse.csr\_matrix,

sparse.csr\_matrix,

MappingIntToObject]:

""" Создает объект Tuple[sparse.csr\_matrix,sparse.csr\_matrix,MappingIntToObject]

из данных Pandas DataFrame.

Args:

data: DataFrame, содержащий ребра графа.

source: Название столбца с исходными вершинами ребер.

target: Название столбца с целевыми вершинами ребер.

edge\_attr: Название столбца с весами ребер.

personalization: Опциональный объект, для которого будет

персонализирован PageRank. Если None персонализации не будет

Returns:

Объект Tuple[sparse.csr\_matrix,sparse.csr\_matrix,MappingIntToObject],

инициализированный на основе данных DataFrame.

"""

source\_uniq = data[source].unique()

target\_uniq = data[target].unique()

obj\_source\_to\_N = dict(zip(source\_uniq, range(len(source\_uniq))))

obj\_target\_to\_N = dict(zip(target\_uniq,

range(len(source\_uniq), len(source\_uniq) + len(target\_uniq))))

N\_to\_obj = {}

N\_to\_obj.update({i: (obj, BaseGraphContext.NodeType.SOURCE) for i, obj in enumerate(source\_uniq)})

N\_to\_obj.update({i: (obj, BaseGraphContext.NodeType.TARGET) for i, obj in enumerate(target\_uniq, start=len(source\_uniq))})

count\_top = len(source\_uniq) + len(target\_uniq)

start\_node = data[source].apply(lambda x: obj\_source\_to\_N[x])

end\_node = data[target].apply(lambda x: obj\_target\_to\_N[x])

transition\_matrix = coo\_matrix((data[edge\_attr], (start\_node, end\_node)),

shape=(count\_top, count\_top))

# Для неориентированного графа

if directional:

transition\_matrix += transition\_matrix.transpose()

transition\_matrix\_csr = transition\_matrix.tocsr()

row\_sums = transition\_matrix\_csr.sum(axis=1)

transition\_matrix\_csr /= row\_sums

if personalization is None:

return (

transition\_matrix\_csr,

None,

N\_to\_obj)

return (

transition\_matrix\_csr,

transition\_matrix\_csr\

.getrow(obj\_source\_to\_N[personalization])\

.transpose(),

N\_to\_obj)

def split\_by\_target\_column\_value(df: pd.DataFrame,

test\_size: float,

target\_column: str,

target\_value: Any,

random\_state: int = 42) -> Tuple[pd.DataFrame,

pd.DataFrame,

pd.DataFrame]:

"""

Разделяет DataFrame на обучающий и тестовый наборы на основе определенного значения целевого столбца.

Выдает в качестве тестового набора test\_size от набора, в котором значение в target\_column == target\_value

В тренировочном все остальное

Параметры:

df (pd.DataFrame): DataFrame для разделения.

test\_size: Пропорция данных, которые следует включить в тестовый набор.

target\_column (str): Название целевого столбца для разделения.

target\_value (Any): Значение целевого столбца, используемое для разделения.

random\_state (int): Зерно для генератора случайных чисел.

Возвращает:

Tuple[pd.DataFrame, pd.Series, pd.DataFrame, pd.Series]: Данные для обучения; тестовые данные для целевой колонки, которые не попали в data,

тренировочные данные для целевой колоники, которые попали в data

"""

grouped = df.groupby(by=target\_column)

train\_df, test\_df = train\_test\_split(grouped.get\_group(target\_value), test\_size=test\_size,

random\_state=random\_state)

data = pd.concat([group for ind, group in grouped if ind != target\_value], ignore\_index=True)

data = pd.concat([train\_df, data], ignore\_index=True)

return data, test\_df, train\_df

// raning.hits.py

from pathlib import Path

from typing import Any, Optional, Tuple

import pandas as pd

from scipy import sparse

import scipy

from .common import (ReadFromTxtToTransition, split\_by\_target\_column\_value,

BaseGraphContext,

ReadFromPandasToTransition,

MappingIntToObject)

from tqdm import tqdm

import numpy as np

# https://dspace.mit.edu/bitstream/handle/1721.1/86737/49317146-MIT.pdf?sequence=2&isAllowed=y

class HITS():

def \_\_init\_\_(self,

transition\_matrix\_csr: sparse.csr\_matrix,

personalized\_vec: Optional[sparse.csr\_matrix],

index\_to\_object: MappingIntToObject) -> None:

# Запоминаем матрицу переходов, вероятности переходов для персонализируемой вершины,

# и отображение целых чисел (индексов) в исходные объекты

(self.\_\_transition\_matrix,

self.\_\_user\_person\_vec,

self.\_\_N\_to\_obj) = (transition\_matrix\_csr,

personalized\_vec,

index\_to\_object)

def get\_rank(self,

max\_iter: int = 100) -> pd.DataFrame:

authoritativeness, hubness = self.\_\_get\_rank(self.\_\_transition\_matrix,

user\_rank\_vec=self.\_\_user\_person\_vec,

max\_iter=max\_iter)

target\_column = []

rank\_column\_auth = []

rank\_column\_hub = []

for ind, (a, h) in enumerate(zip(authoritativeness, hubness)):

if self.\_\_N\_to\_obj[ind][1] == BaseGraphContext.NodeType.TARGET:

target\_column.append(self.\_\_N\_to\_obj[ind][0])

rank\_column\_auth.append(a)

rank\_column\_hub.append(h)

return pd.DataFrame({BaseGraphContext.NodeType.TARGET.value: target\_column,

'authoritativeness': rank\_column\_auth,

'hubness': rank\_column\_hub})

def \_\_get\_rank(self,

transition\_matrix: sparse.csr\_matrix,

user\_rank\_vec: sparse.csr\_matrix,

max\_iter: int) -> Tuple[np.ndarray, np.ndarray]:

# transition\_T = transition\_matrix.transpose()

# hub = user\_rank\_vec.copy()

# auth = user\_rank\_vec.copy()

# # Update the PageRank vector until convergence! Our convergence criterion is to see whether

# # the magnitude of the difference of the PageRank vector after an update is 0

# for i in tqdm(range(max\_iter)):

# # (deep) copy the current PageRank vector to compare with the updated vector

# old\_hub = hub.copy()

# old\_auth = auth.copy()

# old\_hub /= old\_hub.sum()

# old\_auth /= old\_auth.sum()

# # Update!

# hub = sparse.csr\_matrix(transition\_matrix.multiply(old\_auth.T).sum(axis=1))

# auth = sparse.csr\_matrix(transition\_T.multiply(old\_hub.T).sum(axis=1))

\_, \_, vt = scipy.sparse.linalg.svds(transition\_matrix, k=1, maxiter=max\_iter)

a = vt.flatten().real

h = transition\_matrix @ a

h /= h.sum()

a /= a.sum()

return a.flatten(), h.flatten()

class HITSFactory():

def read\_pd(self,

data: pd.DataFrame,

source: str, target: str,

edge\_attr: str,

personalization: Optional[Any] = None,

directional: bool = False) -> HITS:

""" Создает объект HITS из данных Pandas DataFrame.

Args:

data: DataFrame, содержащий ребра графа.

source: Название столбца с исходными вершинами ребер.

target: Название столбца с целевыми вершинами ребер.

edge\_attr: Название столбца с весами ребер.

personalization: Опциональный объект, для которого будет

персонализирован HITS. Если None персонализации не будет

Returns:

Объект HITS, инициализированный на основе данных DataFrame.

"""

(transition\_matrix\_csr,

pers\_vec,

N\_to\_obj) = ReadFromPandasToTransition().\_read\_pd(

data,

source,

target,

edge\_attr,

personalization,

directional)

return HITS(

transition\_matrix\_csr,

pers\_vec,

N\_to\_obj)

def read\_txt(self,

file\_name: Path,

personalization: Optional[Any] = None):

(transition\_matrix\_csr,

pers\_vec,

N\_to\_obj) = ReadFromTxtToTransition().\_read\_txt(file\_name, personalization)

return HITS(

transition\_matrix\_csr,

pers\_vec,

N\_to\_obj)

def main():

data\_test = pd.DataFrame({

'source': ['A', 'B', 'B', 'B'],

'target': ['X', 'X', 'Z', 'C'],

'edge\_attr': [4, 4, 5, 3]

})

pg = HITSFactory().read\_pd(data\_test, 'source', 'target', 'edge\_attr', 'A')

print(pg.get\_rank())

if \_\_name\_\_ == '\_\_main\_\_':

main()

// ranking.pagerank.py

from pathlib import Path

from typing import Any, Optional

import pandas as pd

from scipy import sparse

from .common import (ReadFromTxtToTransition, split\_by\_target\_column\_value,

BaseGraphContext,

ReadFromPandasToTransition,

MappingIntToObject)

import numpy as np

from tqdm import tqdm

class PageRank():

"""

Код класса является переработкой для конкретной задачи кода из

https://github.com/danieljunhee/Tutorial-on-Personalized-PageRank/blob/master/Personalized\_PageRank\_Tutorial.ipynb

"""

def \_\_init\_\_(self,

transition\_matrix\_csr: sparse.csr\_matrix,

personalized\_vec: Optional[sparse.csr\_matrix],

index\_to\_object: MappingIntToObject) -> None:

# Запоминаем матрицу переходов, вероятности переходов для персонализируемой вершины,

# и отображение целых чисел (индексов) в исходные объекты

(self.\_\_transition\_matrix,

self.\_\_user\_person\_vec,

self.\_\_N\_to\_obj) = (transition\_matrix\_csr,

personalized\_vec,

index\_to\_object)

def get\_pagerank(self,

a: float = 0.8,

b: float = 0.15,

c: float = 0.05,

max\_iter: int = 100) -> pd.DataFrame:

""" Вычисляет PageRank для графа.

Метод итеративно вычисляет персонализированный PageRank с учетом

вектора предпочтений `user\_person\_vec` (если он есть), пока не будет достигнуто

максимальное количество итераций (`max\_iter`).

Важно: если нет персонализации, то коэфициент при телепортирующей матрице равен b + c

Args:

transition\_matrix: Матрица переходов графа в формате

разреженной матрицы CSR.

user\_person\_vec: Вектор предпочтений пользователя,

представленный разреженной матрицей CSR.

max\_iter: Максимальное количество итераций. По умолчанию 100.

a: Параметр затухания (damping factor). По умолчанию 0.8.

b: Параметр телепортации для персонализированного PageRank.

По умолчанию 0.15.

c: Дополнительный параметр, используемый для выравнивания

вероятностей. По умолчанию 0.05.

Returns:

DataFrame, содержащий две колонки:

- target: Целевые вершины графа.

- pagerank: Значения PageRank, соответствующие

целевым вершинам.

"""

if self.\_\_user\_person\_vec is not None:

pagerank = self.\_\_get\_personalized\_pagerank(

self.\_\_transition\_matrix,

self.\_\_user\_person\_vec,

max\_iter,

a, b, c

)

else:

pagerank = self.\_\_get\_pagerank(

self.\_\_transition\_matrix,

max\_iter,

a, b + c

)

target\_column = []

pagerank\_column = []

for ind, el\_pagerank in enumerate(pagerank):

if self.\_\_N\_to\_obj[ind][1] == BaseGraphContext.NodeType.TARGET:

target\_column.append(self.\_\_N\_to\_obj[ind][0])

pagerank\_column.append(el\_pagerank)

return pd.DataFrame({BaseGraphContext.NodeType.TARGET.value: target\_column , 'pagerank': pagerank\_column})

def \_\_get\_personalized\_pagerank(self,

transition\_matrix: sparse.csr\_matrix,

user\_person\_vec: sparse.csr\_matrix,

max\_iter: int = 100,

a: float = 0.8,

b: float = 0.15,

c: float = 0.05) -> np.ndarray:

""" Вычисляет персонализированный PageRank для графа.

Метод итеративно вычисляет персонализированный PageRank с учетом

вектора предпочтений `user\_person\_vec`, пока не будет достигнуто

максимальное количество итераций (`max\_iter`).

Args:

transition\_matrix: Матрица переходов графа в формате

разреженной матрицы CSR.

user\_person\_vec: Вектор предпочтений пользователя,

представленный разреженной матрицей CSR.

max\_iter: Максимальное количество итераций. По умолчанию 100.

a: Параметр затухания (damping factor). По умолчанию 0.8.

b: Параметр телепортации для персонализированного PageRank.

По умолчанию 0.15.

c: Дополнительный параметр, используемый для выравнивания

вероятностей. По умолчанию 0.05.

Returns:

Одномерный массив NumPy, содержащий значения

персонализированного PageRank для каждой вершины графа.

"""

# Initialize PageRank vector: height x 1 vector of all 1/height 's

# We also represent the PageRank vector as a sparse csr matrix so that every arithmetic operation result

# is stored as a sparse csr matrix throughout the algorithm... again dealing with sparsity issue & memory issue

transition\_matrix\_T\_mult\_a = a \* transition\_matrix.transpose()

user\_person\_vec\_mult\_b = b \* user\_person\_vec

height = transition\_matrix.shape[0]

user\_pagerank\_vec = sparse.csr\_matrix(np.ones((height,1)) / height)

sup\_vec = c\*sparse.csr\_matrix((1/height)\*np.ones((height,1)))

for i in tqdm(range(max\_iter)):

# Update!

user\_pagerank\_vec = transition\_matrix\_T\_mult\_a.dot(user\_pagerank\_vec) \

+ user\_person\_vec\_mult\_b.multiply(user\_pagerank\_vec) \

+ sup\_vec

if i % 10 == 0:

user\_pagerank\_vec /= user\_pagerank\_vec.sum()

return np.concatenate(user\_pagerank\_vec.toarray())

def \_\_get\_pagerank(self,

transition\_matrix: sparse.csr\_matrix,

max\_iter: int = 100,

a: float = 0.85,

b: float = 0.15) -> np.ndarray:

""" Вычисляет PageRank для графа, представленного матрицей переходов.

Метод итеративно вычисляет PageRank, пока не будет достигнуто

максимальное количество итераций (`max\_iter`).

Args:

transition\_matrix: Матрица переходов графа в формате

разреженной матрицы CSR.

max\_iter: Максимальное количество итераций. По умолчанию 100.

a: Параметр затухания (damping factor). По умолчанию 0.85.

b: Параметр телепортации для персонализированного PageRank.

По умолчанию 0.15.

Returns:

Одномерный массив NumPy, содержащий значения PageRank

для каждой вершины графа.

"""

# Initialize PageRank vector: height x 1 vector of all 1/height 's

# We also represent the PageRank vector as a sparse csr matrix so that every arithmetic operation result

# is stored as a sparse csr matrix throughout the algorithm... again dealing with sparsity issue & memory issue

transition\_matrix\_T\_mult\_a = a \* transition\_matrix.transpose()

height = transition\_matrix.shape[0]

user\_pagerank\_vec = sparse.csr\_matrix(np.ones((height,1)) / height)

sup\_vec = b \* sparse.csr\_matrix((1/height)\*np.ones((height,1)))

for i in tqdm(range(max\_iter)):

# Update!

user\_pagerank\_vec = transition\_matrix\_T\_mult\_a.dot(user\_pagerank\_vec) \

+ sup\_vec

if i % 10 == 0:

user\_pagerank\_vec /= user\_pagerank\_vec.sum()

return np.concatenate(user\_pagerank\_vec.toarray())

class PagerankFactory():

def read\_pd(self,

data: pd.DataFrame,

source: str, target: str,

edge\_attr: str,

personalization: Optional[Any] = None,

directional: bool = False) -> PageRank:

""" Создает объект PageRank из данных Pandas DataFrame.

Args:

data: DataFrame, содержащий ребра графа.

source: Название столбца с исходными вершинами ребер.

target: Название столбца с целевыми вершинами ребер.

edge\_attr: Название столбца с весами ребер.

personalization: Опциональный объект, для которого будет

персонализирован PageRank. Если None персонализации не будет

Returns:

Объект PageRank, инициализированный на основе данных DataFrame.

"""

(transition\_matrix\_csr,

pers\_vec,

N\_to\_obj) = ReadFromPandasToTransition().\_read\_pd(

data,

source,

target,

edge\_attr,

personalization,

directional)

return PageRank(

transition\_matrix\_csr,

pers\_vec,

N\_to\_obj)

def read\_txt(self,

file\_name: Path,

personalization: Optional[Any] = None):

(transition\_matrix\_csr,

pers\_vec,

N\_to\_obj) = ReadFromTxtToTransition().\_read\_txt(file\_name, personalization)

return PageRank(

transition\_matrix\_csr,

pers\_vec,

N\_to\_obj)

if \_\_name\_\_ == '\_\_main\_\_':

data\_test = pd.DataFrame({

'source': ['A', 'B', 'B', 'B'],

'target': ['X', 'X', 'Z', 'C'],

'edge\_attr': [4, 4, 5, 3]

})

pg = PagerankFactory().read\_pd(data=data\_test, source='source', target='target',

edge\_attr='edge\_attr', personalization='A')

print(pg.get\_pagerank())

## Приложение B. Листинг программы C++

// LinkRanking.h

#pragma once

#include "pch.h"

#include <rapidcsv.h>

namespace ranking

{

enum Type {

SOURCE,

TARGET,

};

class TransitionMatrixBuilder {

public:

TransitionMatrixBuilder() = default;

TransitionMatrixBuilder& SetPath(const std::string& path);

TransitionMatrixBuilder& SetSourceName(const std::string& source\_name);

TransitionMatrixBuilder& SetTargetName(const std::string& target\_name);

TransitionMatrixBuilder& SetEdgeAttributeName(const std::string& edge\_attr\_name);

TransitionMatrixBuilder& SetDirectional(bool directional);

TransitionMatrixBuilder& SetLabelParams(const rapidcsv::LabelParams& label\_params);

TransitionMatrixBuilder& SetSample(double sample);

TransitionMatrixBuilder& SetSeparator(rapidcsv::SeparatorParams sep);

std::tuple<Eigen::SparseMatrix<double>, std::vector<std::pair<std::string, Type>>>

Build() const;

private:

std::string path\_ = "";

std::string source\_name\_ = "source";

std::string target\_name\_ = "target";

std::string edge\_attr\_name\_ = "edge\_attr";

bool directional\_ = false;

rapidcsv::LabelParams label\_params\_ = rapidcsv::LabelParams(0, -1);

double sample = 1;

rapidcsv::SeparatorParams sep = rapidcsv::SeparatorParams(',');

};

class TransitionMatrixFromTxtBuilder {

public:

TransitionMatrixFromTxtBuilder() = default;

TransitionMatrixFromTxtBuilder& SetPath(const std::string& path);

TransitionMatrixFromTxtBuilder& SetSourceName(int source\_name);

TransitionMatrixFromTxtBuilder& SetTargetName(int target\_name);

TransitionMatrixFromTxtBuilder& SetEdgeAttributeName(int edge\_attr\_name);

TransitionMatrixFromTxtBuilder& SetDirectional(bool directional);

TransitionMatrixFromTxtBuilder& SetLabelParams(const rapidcsv::LabelParams& label\_params);

TransitionMatrixFromTxtBuilder& SetSample(double sample);

TransitionMatrixFromTxtBuilder& SetSeparator(rapidcsv::SeparatorParams sep);

std::tuple<Eigen::SparseMatrix<double>, std::vector<std::pair<std::string, Type>>>

Build() const;

private:

std::string path\_ = "";

int source = 0;

int target = 1;

int edge = -1;

bool directional\_ = false;

rapidcsv::LabelParams label\_params\_ = rapidcsv::LabelParams(-1, -1);

double sample = 1;

rapidcsv::SeparatorParams sep = rapidcsv::SeparatorParams('\t');

};

Eigen::VectorXd get\_pagerank(const Eigen::SparseMatrix<double>& trasition,

const Eigen::VectorXd& person\_vec,

double a, double b, double c, int max\_iter);

class PageRankFactory {

public:

PageRankFactory() = default;

PageRankFactory& SetTransitionMatrix(const Eigen::SparseMatrix<double>& matrix);

PageRankFactory& SetPersonalizationVector(const Eigen::VectorXd& vector);

PageRankFactory& SetAlpha(double alpha);

PageRankFactory& SetBeta(double beta);

PageRankFactory& SetGamma(double gamma);

PageRankFactory& SetMaxIterations(int max\_iterations);

Eigen::VectorXd Build() const;

private:

const Eigen::SparseMatrix<double>\* transition\_matrix\_;

const Eigen::VectorXd\* personalization\_vector\_;

double alpha\_ = 0.80;

double beta\_ = 0.15;

double gamma\_ = 0.05;

int max\_iterations\_ = 100;

};

Eigen::VectorXd get\_personalization\_from\_matrix\_row(

const Eigen::SparseMatrix<double>& matrix,

int index);

class HITSFactory {

public:

HITSFactory() = default;

HITSFactory& SetTransitionMatrix(const Eigen::SparseMatrix<double>& matrix);

HITSFactory& SetMaxIter(int iterations);

std::tuple<Eigen::VectorXd, Eigen::VectorXd> Build() const;

private:

const Eigen::SparseMatrix<double>\* trasition;

int max\_iter = 100;

};

}

// LinkRanking.cpp : Определяет функции для статической библиотеки.

//

#include "pch.h"

#include "framework.h"

#include "LinkRanking.h"

#include <set>

#include <boost/container/flat\_map.hpp>

#include <boost/unordered\_set.hpp>

#include <boost/unordered\_map.hpp>

#include <tuple>

#include <rapidcsv.h>

#include <algorithm>

#include <optional>

#include <assert.h>

using namespace std;

namespace ranking

{

using namespace Eigen;

void \_norm\_matrix\_rows(SparseMatrix<double>& matrix, const Eigen::VectorXd& values) {

// Нормализуем строки

for (int i = 0; i < matrix.outerSize(); ++i) {

for (Eigen::SparseMatrix<double>::InnerIterator it(matrix, i); it; ++it) {

if (values(it.row()) != 0.0) {

it.valueRef() /= values(it.row());

}

}

}

}

Eigen::VectorXd \_get\_sum\_matrix\_rows(const SparseMatrix<double>& matrix) {

// Вектор для хранения сумм строк

Eigen::VectorXd rowSums(matrix.rows());

rowSums.setZero();

// Вычисляем сумму элементов в каждой строке

for (int i = 0; i < matrix.outerSize(); ++i) {

for (Eigen::SparseMatrix<double>::InnerIterator it(matrix, i); it; ++it) {

rowSums(it.row()) += it.value();

}

}

return rowSums;

}

std::tuple<

boost::unordered\_map<std::string, int>,

boost::unordered\_map<std::string, int>,

std::vector<std::pair<std::string, Type>>>

\_get\_mapping(

const std::vector<std::string>& source,

const std::vector<std::string>& target) {

boost::unordered\_set<std::string> source\_set(source.begin(), source.end());

boost::unordered\_set<std::string> target\_set(target.begin(), target.end());

int index\_map = 0;

boost::unordered\_map<std::string, int> object\_source\_to\_integer(source\_set.size());

boost::unordered\_map<std::string, int> object\_target\_to\_integer(target\_set.size());

std::vector<std::pair<std::string, Type>> integer\_to\_object;

integer\_to\_object.reserve(source\_set.size() + target\_set.size());

for (auto el : source\_set) {

object\_source\_to\_integer.insert({ el, index\_map });

integer\_to\_object.push\_back({ el, Type::SOURCE });

index\_map++;

}

for (auto el : target\_set) {

object\_target\_to\_integer.insert({ el, index\_map });

integer\_to\_object.push\_back({ el, Type::SOURCE });

index\_map++;

}

return { object\_source\_to\_integer , object\_target\_to\_integer, integer\_to\_object };

}

std::vector<Eigen::Triplet<double>>

\_get\_triplets(

const boost::unordered\_map<std::string, int>& object\_source\_to\_integer,

const boost::unordered\_map<std::string, int>& object\_target\_to\_integer,

const std::vector<std::string>& source,

const std::vector<std::string>& target,

const std::vector<double>& edge\_attr,

bool directional)

{

int count\_row = edge\_attr.size();

std::vector<Eigen::Triplet<double>> triplets;

triplets.reserve(count\_row);

for (int i = 0; i < count\_row; i++) {

auto from = object\_source\_to\_integer.at(source[i]);

auto to = object\_target\_to\_integer.at(target[i]);

triplets.emplace\_back(from, to, edge\_attr[i]);

if (!directional) {

triplets.emplace\_back(to, from, edge\_attr[i]);

}

}

return triplets;

}

std::tuple<

std::vector<std::string>,

std::vector<std::string>,

std::vector<double>>

\_extract\_column(

const std::string& path,

const std::string& source\_name,

const std::string& target\_name,

const std::string& edge\_attr\_name,

rapidcsv::LabelParams label\_params,

double sample,

rapidcsv::SeparatorParams sep)

{

try {

rapidcsv::Document doc(path, label\_params, sep); // Инициализация Document, открытие файла

auto count = (float)(doc.GetRowCount() \* sample);

// Чтение колонок во временные переменные

auto full\_source = doc.GetColumn<std::string>(source\_name);

auto full\_target = doc.GetColumn<std::string>(target\_name);

auto full\_edge\_attr = doc.GetColumn<double>(edge\_attr\_name);

if (sample == 1.0) {

return { full\_source, full\_target, full\_edge\_attr };

}

// Создание срезов

std::vector<std::string> source(full\_source.begin(), full\_source.begin() + count);

std::vector<std::string> target(full\_target.begin(), full\_target.begin() + count);

std::vector<double> edge\_attr(full\_edge\_attr.begin(), full\_edge\_attr.begin() + count);

return { source, target, edge\_attr };

}

catch (const std::ios\_base::failure& e) {

std::cerr << "I/O Error: " << e.what() << std::endl;

throw;

}

catch (const std::exception& ex) {

std::cerr << "Error: " << ex.what() << std::endl;

throw;

}

}

std::tuple<

std::vector<std::string>,

std::vector<std::string>,

std::vector<double>>

\_extract\_column(

const std::string& path,

int source\_name,

int target\_name,

int edge,

rapidcsv::LabelParams label\_params,

double sample,

rapidcsv::SeparatorParams sep)

{

try {

rapidcsv::Document doc(path, label\_params, sep); // Инициализация Document, открытие файла

auto count = (float)(doc.GetRowCount() \* sample);

// Чтение колонок во временные переменные

auto full\_source = doc.GetColumn<std::string>(source\_name);

auto full\_target = doc.GetColumn<std::string>(target\_name);

std::vector<double> full\_edge\_attr(full\_source.size(), 1.0);

if (sample == 1.0) {

return { full\_source, full\_target, full\_edge\_attr };

}

// Создание срезов

std::vector<std::string> source(full\_source.begin(), full\_source.begin() + count);

std::vector<std::string> target(full\_target.begin(), full\_target.begin() + count);

std::vector<double> edge\_attr(full\_edge\_attr.begin(), full\_edge\_attr.begin() + count);

return { source, target, edge\_attr };

}

catch (const std::ios\_base::failure& e) {

std::cerr << "I/O Error: " << e.what() << std::endl;

throw;

}

catch (const std::exception& ex) {

std::cerr << "Error: " << ex.what() << std::endl;

throw;

}

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetPath(const std::string& path) {

path\_ = path;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetSourceName(const std::string& source\_name) {

source\_name\_ = source\_name;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetTargetName(const std::string& target\_name) {

target\_name\_ = target\_name;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetEdgeAttributeName(

const std::string& edge\_attr\_name) {

edge\_attr\_name\_ = edge\_attr\_name;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetDirectional(bool directional) {

directional\_ = directional;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetLabelParams(

const rapidcsv::LabelParams& label\_params) {

label\_params\_ = label\_params;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetSample(double sample) {

this->sample = sample;

return \*this;

}

TransitionMatrixBuilder& TransitionMatrixBuilder::SetSeparator(rapidcsv::SeparatorParams sep)

{

this->sep = sep;

return \*this;

}

std::tuple<SparseMatrix<double>, std::vector<std::pair<std::string, Type>>>

TransitionMatrixBuilder::Build() const {

auto [source, target, edge\_attr] = \_extract\_column(

path\_, source\_name\_,

target\_name\_, edge\_attr\_name\_,

label\_params\_, sample, sep);

auto [object\_source\_to\_integer, object\_target\_to\_integer, integer\_to\_object] =

\_get\_mapping(source, target);

auto x = \_get\_mapping(source, target);

auto triplets = \_get\_triplets(object\_source\_to\_integer,

object\_target\_to\_integer,

source, target, edge\_attr, false);

SparseMatrix<double> A(integer\_to\_object.size(), integer\_to\_object.size());

A.setFromTriplets(triplets.begin(), triplets.end());

auto rowSums = \_get\_sum\_matrix\_rows(A);

\_norm\_matrix\_rows(A, rowSums);

return { A, integer\_to\_object };

}

Eigen::VectorXd get\_pagerank(const SparseMatrix<double>& trasition,

const Eigen::VectorXd& person\_vec,

double a, double b, double c, int max\_iter)

{

assert(a + b + c == 1.0);

auto transtion\_T\_mult\_a = (trasition.transpose() \* a).eval();

auto person\_vec\_mult\_b = (person\_vec \* b).eval();

auto sup\_vec = (Eigen::VectorXd::Ones(transtion\_T\_mult\_a.rows()) /

transtion\_T\_mult\_a.rows() \* c).eval();

auto pagerank = (Eigen::VectorXd::Ones(transtion\_T\_mult\_a.rows()) /

transtion\_T\_mult\_a.rows()).eval();

for (int i = 0; i < max\_iter; i++) {

pagerank = transtion\_T\_mult\_a \* pagerank + person\_vec\_mult\_b \* pagerank + sup\_vec;

}

return pagerank;

}

Eigen::VectorXd get\_pagerank(const SparseMatrix<double>& trasition,

const Eigen::VectorXd& person\_vec,

double a, double b, double c, int max\_iter);

PageRankFactory& PageRankFactory::SetTransitionMatrix(const SparseMatrix<double>& matrix) {

transition\_matrix\_ = &matrix;

return \*this;

}

PageRankFactory& PageRankFactory::SetPersonalizationVector(const Eigen::VectorXd& vector) {

personalization\_vector\_ = &vector;

return \*this;

}

PageRankFactory& PageRankFactory::SetAlpha(double alpha) {

alpha\_ = alpha;

return \*this;

}

PageRankFactory& PageRankFactory::SetBeta(double beta) {

beta\_ = beta;

return \*this;

}

PageRankFactory& PageRankFactory::SetGamma(double gamma) {

gamma\_ = gamma;

return \*this;

}

PageRankFactory& PageRankFactory::SetMaxIterations(int max\_iterations) {

max\_iterations\_ = max\_iterations;

return \*this;

}

Eigen::VectorXd PageRankFactory::Build() const {

assert(alpha\_ + beta\_ + gamma\_ == 1.0);

return get\_pagerank(

\*transition\_matrix\_,

\*personalization\_vector\_,

alpha\_, beta\_, gamma\_,

max\_iterations\_);

}

Eigen::VectorXd get\_personalization\_from\_matrix\_row(

const SparseMatrix<double>& matrix,

int index)

{

// Создаем плотный вектор той же размерности, что и строка

Eigen::VectorXd dense\_vec(matrix.cols());

// Копируем ненулевые элементы из строки в плотный вектор

for (Eigen::SparseMatrix<double>::InnerIterator it(matrix, index); it; ++it) {

dense\_vec[it.col()] = it.value();

}

return dense\_vec;

}

Eigen::SparseMatrix<double> multiply\_rows\_by\_scalar\_from\_vector(

const Eigen::SparseMatrix<double>& matrix,

const Eigen::VectorXd& scalars) {

assert(matrix.rows() == scalars.size());

// Создаем копию матрицы для результата

Eigen::SparseMatrix<double> result = matrix;

// Итерируемся по строкам разреженной матрицы

for (int i = 0; i < result.rows(); ++i) {

// Умножаем ненулевые элементы строки на соответствующий скаляр

for (Eigen::SparseMatrix<double>::InnerIterator it(result, i); it; ++it) {

it.valueRef() \*= scalars(i);

}

}

return result;

}

std::tuple<Eigen::VectorXd, Eigen::VectorXd>

get\_hits(const SparseMatrix<double>& trasition, int max\_iter)

{

auto transtion\_T = trasition.transpose().eval();

auto hub = (Eigen::VectorXd::Ones(transtion\_T.rows()) /

transtion\_T.rows()).eval();

auto auth = (Eigen::VectorXd::Ones(trasition.rows()) /

trasition.rows()).eval();

for (int i = 0; i < max\_iter; i++) {

auto old\_hub = hub / hub.sum();

auto old\_auth = auth / auth.sum();

hub = \_get\_sum\_matrix\_rows(

multiply\_rows\_by\_scalar\_from\_vector(trasition, old\_auth));

auth = \_get\_sum\_matrix\_rows(

multiply\_rows\_by\_scalar\_from\_vector(transtion\_T, old\_hub));

}

return { auth, hub };

}

HITSFactory& HITSFactory::SetTransitionMatrix(const Eigen::SparseMatrix<double>& matrix) {

trasition = &matrix;

return \*this;

}

HITSFactory& HITSFactory::SetMaxIter(int iterations) {

max\_iter = iterations;

return \*this;

}

std::tuple<Eigen::VectorXd, Eigen::VectorXd> HITSFactory::Build() const {

return get\_hits(\*trasition, max\_iter);

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetPath(const std::string& path)

{

this->path\_ = path;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetSourceName(int source\_name)

{

this->source = source\_name;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetTargetName(int target\_name)

{

this->target = target\_name;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetEdgeAttributeName(int edge\_attr\_name)

{

this->edge = edge\_attr\_name;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetDirectional(bool directional)

{

this->directional\_ = directional;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetLabelParams(const rapidcsv::LabelParams& label\_params)

{

this->label\_params\_ = label\_params;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetSample(double sample)

{

this->sample = sample;

return \*this;

}

TransitionMatrixFromTxtBuilder& TransitionMatrixFromTxtBuilder::SetSeparator(rapidcsv::SeparatorParams sep)

{

this->sep = sep;

return \*this;

}

std::tuple<Eigen::SparseMatrix<double>, std::vector<std::pair<std::string, Type>>> TransitionMatrixFromTxtBuilder::Build() const

{

auto [source, target, edge\_attr] = \_extract\_column(

path\_, this->source,

this->target, this->edge,

label\_params\_, sample, sep);

auto [object\_source\_to\_integer, object\_target\_to\_integer, integer\_to\_object] =

\_get\_mapping(source, target);

auto x = \_get\_mapping(source, target);

auto triplets = \_get\_triplets(object\_source\_to\_integer,

object\_target\_to\_integer,

source, target, edge\_attr, false);

SparseMatrix<double> A(integer\_to\_object.size(), integer\_to\_object.size());

A.setFromTriplets(triplets.begin(), triplets.end());

auto rowSums = \_get\_sum\_matrix\_rows(A);

\_norm\_matrix\_rows(A, rowSums);

return { A, integer\_to\_object };

return std::tuple<Eigen::SparseMatrix<double>, std::vector<std::pair<std::string, Type>>>();

}

}

1. URL: <https://github.com/networkx/networkx> (дата обращения 27.05.2024) [↑](#footnote-ref-1)
2. URL: <https://www.kaggle.com/datasets/pappukrjha/google-web-graph> (дата обращения 27.05.2024) [↑](#footnote-ref-2)
3. URL: <https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset> (дата обращения 27.05.2024) [↑](#footnote-ref-3)