Федеральное государственное образовательное бюджетное
учреждение высшего образования
**«Финансовый университет при Правительстве
Российской Федерации»**

Департамент **«Анализа данных, принятия решений и финансовых технологий»**

Курсовая работа

По дисциплине «**Технологии анализа данных и машинное обучение**»

на тему:

«Реализация и анализ прототипа алгоритма параллельного анализа графа»

Выполнила:
студентка гр. **ПМ3-3**
Степенко З. В.
Научный руководитель:
к. э. н., доцент
Гринева Н. В.

Москва 2020

Содержание

ВВЕДЕНИЕ

В настоящий момент алгоритмическое моделирование находит широкое практическое применение, что связано с возможностью описать теорией графов многие процессы и явления.

Так, при изучении Всемирной Паутины исследователи используют направленные графы для представления структуры веб-страниц во всей «Паутине»: вершина представляет собой конкретную веб-страницу, а наличие направленного ребра между двумя вершинами означает наличие ссылки с одной веб-страницы на другую.

В социальных науках графы также широко используются, например, для анализа социальных сетей. Исследователи могут извлечь интересную информацию из таких графов: измерить уровень популярности медийных лиц, исследовать пути распространения слухов и так далее.

При решении подобных реальных задач приходится иметь дело с графами большой размерности - с миллионами вершин и ребер - проводить с ними огромное количество действий (в процессах обхода в ширину и глубину, ранжирования, поиска кратчайшего пути и т.д.), на что уходит немало времени. Чтобы сократить длительность выполнения задач, прибегают к *параллельным вычислениям.*

Для выполнения параллельных алгоритмов применительно к большим графам и сложным сетям разработано множество моделей (BGL, Pregel, gunrock), которые, несмотря на возможность массового распараллеливания, обладают таким недостатком как нелокальность. Для решения данной проблемы С. В. Макрушиным и Е. В. Пановым [1] была предложена модель обхода сетевой структуры с применением пучков траектории – модификация метода частиц для решения задач о случайном блуждании по вершинам графа. Данная модель позволяет повысить производительность проводимых с графами вычислений и в перспективе может быть реализована с использованием CUDA-архитектуры.

Целью моей курсовой работы является рассмотрение концепции и реализации прототипа алгоритма обхода пучка траекторий для параллельного анализа графа, а

также определение его производительности и актуальности для дальнейшего использования в сфере анализа больших графов на примере решения задачи PageRank.

В ходе выполнения были решены следующие **задачи**:

- выявление текущих проблем, связанных с параллельными вычислениями на графах;
- ознакомление с системой параллельной обработки графов (на примере CUDA);
- рассмотрение задачи ссылочного ранжирования для сложной сети и существующих реализаций ее решения на языке Python;
- изучение реализации прототипа предложенного ОПГ-алгоритма (на языке Python);
- настройка PageRank-весов для вершин графа на базе инфраструктуры алгоритма ОПГ;
- оценка времени выполнения программ, составленных исходя из последовательной и параллельной моделей;
- проверка идентичности результатов вычислений весов вершин двумя способами

При написании работы были использованы как отечественные, так и зарубежные (англоязычные источники).

Глава 1. Параллельные вычисления на графах

1.1 Проблемы анализа графов на GPU

В обычных алгоритмах обработки графов на основе ЦП важно спроектировать компоновку данных для обеспечения непрерывного доступа к памяти для повышения скорости попадания TLB и кэша. Но в графическом процессоре существует глобальная память, общая для всех процессов графического процессора, и для каждого доступа к памяти полезно передавать данные более чем в один поток SIMD. Потоки в графическом процессоре выполняются в группах (warp в CUDA).

Графический процессор может достичь своей максимальной пропускной способности доступа к памяти только тогда, когда алгоритм имеет регулярную схему доступа к памяти, т. е. данные, к которым обращаются последовательные потоки деформации, занимают непрерывный сегмент. Однако графовые структуры данных и графовые алгоритмы часто приводят к нерегулярным обращениям к памяти.

Например, в алгоритме параллельного обхода графов, когда используется структура данных «список смежности», различные потоки будут обращаться к данным, разбросанным по разным ячейкам памяти, что требует от графического процессора выдавать несколько обращений к памяти для извлечения всех необходимых данных. Такая нерегулярная компоновка данных существенно ограничивает степень параллельности графического процессора и не способствует высвобождению его полной мощности.

Следовательно, для использования вычислительных мощностей видеокарты в полной мере следует использовать такие структуры данных, которые позволят скомпоновать данные на регулярной основе и исключить рассогласованность обращений к памяти при работе с ними.

Процессор обычно оснащен очень большой основной памятью, которой вполне достаточно для обработки большинства реальных графов. Кроме того, даже с графами, которые больше, чем размер основной памяти, системы на базе процессора

могут эффективно использовать вторичное хранилище для решения этой проблемы из-за относительно высокой пропускной способности.

Но графический процессор обычно оснащен высокоскоростной, но небольшой по размеру встроенной общей памятью, которая может использоваться для кэширования часто используемых данных, чтобы уменьшить необходимость доступа к глобальной памяти на устройстве. Однако если многие потоки одновременно обращаются к различным данным в общей памяти, это вызовет конфликты банка памяти и, следовательно, ограничит степень параллелизма.

Кроме того, доступ к глобальной памяти в современном графическом процессоре обычно осуществляется в единицах блоков. Размер блока обычно составляет 64 КБ, но также зависит от архитектуры графического процессора. Поэтому, если доступы к глобальной памяти, выдаваемые варпом, объединены и выровнены в пределах одного или нескольких блоков доступа к памяти, то это может значительно улучшить использование полосы пропускания памяти. Компоновка данных, тщательное проектирование шаблона доступа к памяти также является критической проблемой в вычислениях GPU.

Процессор имеет сильный и гибкий блок управления, который может легко изменять стратегию планирования во время выполнения. Но GPU работает в модели Multiple Threads (SIMT). И после того, как команда распределена контроллером, невозможно изменить стратегию планирования до следующей итерации. Появляется дисбаланс: различные вершины в графе часто имеют очень разные степени, что затрудняет балансировку рабочей нагрузки между параллельными задачами. Неравномерное распределение нагрузки между потоками внутри вызова ядра может значительно ухудшить производительность. Кроме того, поскольку CPU и GPU отдают предпочтение различным типам задач, разделение рабочей нагрузки между CPU и GPU для достижения хорошей общей производительности в такой гибридной системе также становится сложной задачей.

Реализация эффективных вычислений для графов на графических процессорах требует решения и других проблем, таких как дивергенция ветвей, вызовы ядра и конфигурация ядра. Благодаря своему гибкому блоку управления процессор хорошо

справляется с ветвями состояния. Но для графического процессора расхождение ветвей возникает, когда разные потоки принимают разные пути в условной ветви в одном и том же волновом фронте.

Расхождение ветвей является большой проблемой для программистов GPU, а значит и необходимость совмещать синхронный и асинхронный подход к обработке данных. Поскольку GPU является параллельным потоковым процессором, синхронные операции могут ограничить вычислительную мощность GPU. С другой стороны, трудно реализовать асинхронные операции обработки графов на графических процессорах, так как между вершинами графов проходит большое количество сообщений.

1.2 О CUDA-архитектуре

Compute Unified Device Architecture (CUDA) - программно-аппаратный комплекс для реализации параллельных вычислений, позволяющий кратно повысить производительность машинных вычислений вследствие перераспределения нагрузки с центрального процессора на графический процессор (видеокарту). При работе используют видеокарты производителя NVIDIA. На настоящий момент последняя версия – CUDA SDK 10.2.

Графические процессоры (GPU) обладают такими полезными характеристиками как: поддержка высокоуровневых языков, большая вычислительная мощность, высокая арифметическая плотность – т. е. отношение числа выполняемых арифметических операций к числу обращений к памяти, минимальные требования к flow control (управление исполнением программы). В парадигме CUDA графический процессор является вторым процессором (device) по отношению к центральному (host), при это обладает собственной памятью DRAM (device memory).

GPU использует архитектуру SIMD-based, которая получает высокую производительность благодаря массивному параллелизму. В графическом процессоре большая часть площади матрицы используется арифметико-логическими блоками (ALU), в то время как небольшая часть площади вносится в

блоки управления и кэши. Кроме того, GPU обычно имеет очень высокую пропускную способность доступа к памяти, но ограниченное пространство памяти. Эта архитектура позволяет графическому процессору выполнять регулярные вычисления с очень большой степенью параллелизма.

Программа с исполнением на GPU работает с несколькими параллельными потоками **(threads)**, а функция в данной программы носит название ядра **(kernel)**. Потоки комбинируются в блоки **(thread blocks)** равной длины (по количеству потоков), и исполнение ядра ведется с блоками, объединенными в единую сетку **(grid)**. Внутри блока потоков каждый поток взаимодействует другими при помощи конкретных контрольных точек (барьеров синхронизации) и разделяемой памяти **(shared memory)**. Каждый поток и блок потоков имеет свои координаты. В первом случае двумерные, в другом трехмерные.

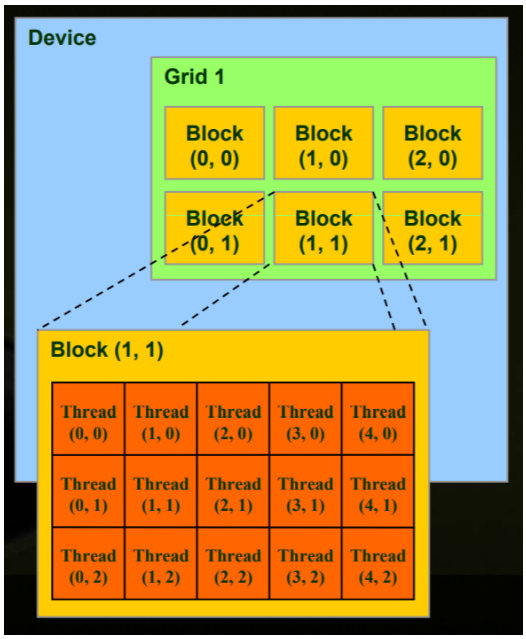


Рис. 1. Схема введения thread и thread-block координат

При CUDA-разработке предполагается, что каждый поток имеет доступ к локальным регистрам, а также к локальной, **глобальной**, **константной**, **текстурной** и разделяемой памяти. **Выделенные** типы памяти получают доступ к запросам и обновлениям от центрального процессора CPU.

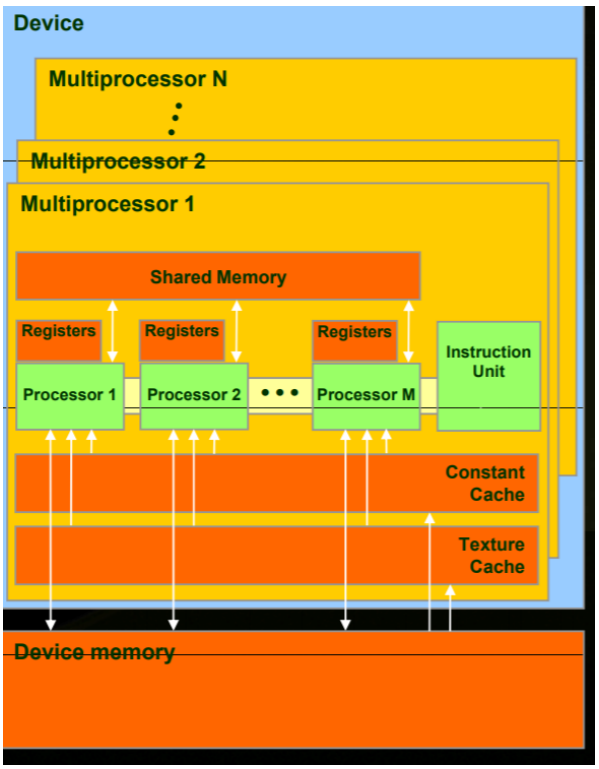


Рис. 2. Схематичное изображение архитектуры памяти

Потоки в модели CUDA также объединяют в варпы (warp) – подгруппы thread block, состоящие из группы скалярных потоков с обозначенными координатами. Исполняются они на одном мультипроцессоре, при этом имеет место разделение локального регистра с выделением его части каждому мультипроцессору. Размер регистра на поток и разделяемой памяти на блок уменьшается, а значит появляется возможности выполнять параллельно больше потоков => и алгоритмических действий, растет производительность.

1.3 Случайные блуждания и алгоритм PageRank

Пусть $G = (V, E)$ – связный граф (между любой парой вершин существует как минимум одна связь – ребро) с n вершинами и m ребрами. Рассмотрим случайное блуждание по G: начнем обход графа G с вершины v_0 и если на шаге с номером i в процессе обхода мы оказываемся в вершине v_i, то с вероятностью $1/\text{степень}(v_i)$ на шаге с номером (i + 1) мы переместимся в соседнюю с v_i вершину. Полученная последовательность случайно посещенных вершин $v_i: i = 0,1, \ldots$ представляет собой

цепь Маркова. Стартовая вершина v_0 может быть как зафиксирована, так и выбрана из некоторого распределения вероятностей P_0.

Определим следующее распределение вероятностей P_i для вершины v_i:

$$P_i(k) = P(v_i = k)$$

Также обозначим степень вершины v_i как $d(v_i)$, а матрицу вероятностей переходов нашей марковской цепи как $M = (p_{kl}), k, l \in V$:

$$p_{kl} = \begin{cases} \frac{1}{d(k)}, \text{если } k, l \in E \\ 0, \text{иначе} \end{cases}$$

Тогда можно выстроить правило случайного блуждания в матричной форме: $P_{i+1} = M^T * P_i$ и $P_i = (M^T)^i * P_0$ => вероятность p_{kl}^i того, что стартовав из вершины с номером k мы достигнем вершины с номером l за i шагов будет, по сути, представлять собой элемент m_{kl} матрицы M.

Распределения вероятностей $P_0, P_1, \ldots$ в общем случае различны. Если $P_1 = P_0$, а, следовательно, и $P_i = P_0 \; \forall \, i \geq 0$, то распределение P_0, как и все блуждание называют **стационарным**.

Алгоритм PageRank был создан Пейджем и Брином ([3]) в 1998 году и, по сути, положил начало поисковой системе Google. Причиной его создания была необходимость выявления уровня популярности веб-страницы, основываясь на вычислении важности других страниц Сети Интернет, имеющих ссылку на нее. Веб-страница, чей вес в ранжированном списке - наибольший попадает в топ выдачи информации по запросу в Google, затем идет веб-страница со вторым по величине весом и т.д. К настоящему времени были разработаны и иные методы ранжирования сайтов для выдачи по поисковым запросам, но, тем не менее, данный алгоритм по-прежнему играет не последнюю роль в определении значимости сайта для пользователя.

Базовая идея алгоритма PageRank заключается в нахождении стационарного распределения P_0 для некоторого графа G = (V, E). Связи между страницами (наличие ссылок) представлены графом, где вершины – непосредственно конкретные веб-страницы, а ребра – наличие ссылки с одной страницы на другую.

Пусть N – общее кол-во веб-сайтов (вершин графа, узлов в сложной сети), а d(N_i) – кол-во ссылок со страницы N_i с номером i на другие страницы (иными словами, исходящая степень узла N_i). Рассмотрим следующую функцию:

$$m_{ij} = \begin{cases} \frac{1}{d(N_i)}, \text{если есть ссылка с } N_i \text{ на } N_j \\ 0, \text{иначе} \end{cases}$$

Составим матрицу размера N x N:

$M = \begin{pmatrix} m_{11} & \dots & m_{1N} \\ \dots & \dots & \dots \\ m_{N1} & \dots & m_{NN} \end{pmatrix}$, отметим, что $\sum_{j=1}^{N} m_{ij} = 1$ и $m_{ij} \geq 0 \; \forall i, j \in G$

Заметим, что матрицу в таком виде получаем в случае, если каждая веб-страница имеет как минимум одну исходящую ссылку, т. е. $d(N_i) > 0$.

Создадим единичный вектор размерности N x 1:

$X = \begin{pmatrix} x_{N_1} \\ \dots \\ x_{N_i} \end{pmatrix} = \begin{pmatrix} 1 \\ \dots \\ 1 \end{pmatrix}$ и на каждом шаге алгоритма, до заданной точности будем переопределять его значение: $X_k = M * X_{k-1}$. Полученные на конечном шаге значения элементов вектора X и будут искомыми ранжированными весами для N веб-страниц.

Проблема возникает тогда, когда хотя бы один из анализируемых сайтов не имеет исходящей ссылки. В таком случае узел в сети, соответствующий данному сайту, называется «висячим» (dangling node). Если N-й узел – висячий, то матрица M примет вид:

$M = \begin{pmatrix} m_{11} & \dots & 0 \\ \dots & \dots & \dots \\ m_{N1} & \dots & 0 \end{pmatrix}$ и если анализировать сеть с таким узлом простым алгоритмом, описанным выше, то значения вектора X окажутся быстро сходящимися к нулю.

Одно из решений проблемы – заменить нули в проблемном столбце на 1/N:

$$M^* = \begin{pmatrix} m_{11} & \dots & \frac{1}{N} \\ \dots & \dots & \dots \\ m_{N1} & \dots & \frac{1}{N} \end{pmatrix} =>$$

$$m_{ij}^* = \begin{cases} \frac{1}{d(N_i)}, \text{если есть ссылка с } N_i \text{на } N_j \\ \frac{1}{N}, \text{если узел } N_i - \text{висячий} \\ 0, \text{иначе} \end{cases}$$

Далее можно воспользоваться той же формулой для отыскания ранжированных весов:

$X_k = M^* * X_{k-1}$

Еще одна проблема в составлении алгоритма PageRank, требующая внимания это – работа с особым видом сети - сводимым (reducible) графом. Граф называется несводимым, если для любой пары различных узлов мы можем стартовать обход графа (сети) с одного из них, перемещаться по ссылкам (ребрам) в нашем веб-графе и прийти к другому узлу, и наоборот. Граф, который не является несводимым, называется сводимым.

В таком случае рейтинг веб-страницы, представляющей для пользователя значимость, будет стремиться к нулю – т. е. ее pagerank будет определен неверно. Чтобы правильно рассчитать ранги страниц для сводимого веб-графа, Пейдж и Брин предложили взять средневзвешенное значение матрицы M^*с матрицей единичной матрицей размерности N × N.

Тогда окончательная матрица примет вид:

$$A = d * M^* + \frac{(1-d)}{N} * \begin{pmatrix} 1 & \dots & 1 \\ \dots & \dots & \dots \\ 1 & \dots & 1 \end{pmatrix} (1)$$

Здесь d – так называемый коэффициент или константа затухания (damping factor). В Google коэффициент затухания по умолчанию равен 0,85. Его можно интерпретировать как вероятность того, что пользователь, посетивший сайт, кликнет по одной из содержащихся на нем ссылок и именно этим, а не иным образом попадет на следующий сайт.

Проведем аналогичную процедуру с вектором X, пока не достигнем некой сходимости или некой заданной точности:

$X_k = A * X_{k-1} = P_0$ и получим итоговый результат. Также с помощью теоремы Перрона-Фробениуса было доказано, что для вычисленной по правилу (1) матрицы A задача взвешивания узлов сети разрешима, так как она всегда имеет собственный вектор $\lambda = 1$.

В популярной библиотеке для анализа графов и сетей NetworkX, написанной на языке программирования Python уже реализован алгоритм PageRank. PageRank был создан для работы с ориентированными графами, однако данная реализация позволяет работать и с неориентированными графами: в таком случае каждое ребро неориентированного графа «удваивается». Данная реализация не обеспечивает точную сходимость, поскольку точка остановки работы алгоритма определяется методом степенной итерации. Соответственно, алгоритм останавливается, когда достигается максимальное число итераций или определенная погрешность вычисления значений вектора X.

Также стоит отметить, что существуют и иные реализации алгоритма ссылочного ранжирования на ЯП Python: в библиотеках NumPy и SciPy. Разница заключается в способе вычисления собственных значений матрицы A, по которым сверяется сходимость, т. е. момент остановки вычислений. Реализация в NumPy использует средства LAPACK (Linear Algebra PACKage) для нахождения собственных значений и обеспечивает максимальную скорость и точность для графов небольшой размерности, реализация в SciPy использует для решения указанной проблемы с поиском собственных значений разреженную матрицу (sparce matrix). Обе указанные реализации могут работать с мультиграфами (имеют кратные ребра).

1.4 Концепция алгоритма обхода пучков траекторий

Алгоритм ОПТ относится к классу массово-параллельных алгоритмов для обработки больших графов и позволяет преодолеть важнейшую проблему, свойственную алгоритмам данного класса: чтобы реализовать их эффективно,

необходимо точно локализовать в памяти данные о вершинах и ребрах графа. Данная проблема решается вводом особой структуры данных – множества траекторий.

Суть алгоритма ОПГ сводится к следующему: по графу выполняется серия случайных блужданий (random walkings), причем каждая последовательность рандомизированных переходов от одного узла к другому или, иными словами, последовательность посещенных за итерацию блуждания вершин – это траектория. Таким образом, в структуре «множество траекторий» узел каждой траектории – это одна из вершин анализируемого сетевого графа, а непосредственно траектория связана с ребрами (связями) сети: каждое ребро сети сопоставляется с парой последовательных узлов в одной из траекторий.

Значит, используя предложенную структуру, можно однозначно восстановить исходную сеть. Тем не менее, следует учитывать, что не каждый узел получит однозначное представление. И чтобы не потерять эту однозначность, было решено провести массовое усреднение значения представлений вершин попарно.

Каждый вычислительный узел работает с неким подмножеством рассматриваемой структуры (множество траекторий), такое подмножество будем называть **пучком траекторий**. В рамках пучка издержки на передачу информации от одного потока последовательно к другому (от одной траектории к другой) низки, а также используются не произвольные, а последовательные ячейки памяти, что снижает нагрузку на ОЗУ.

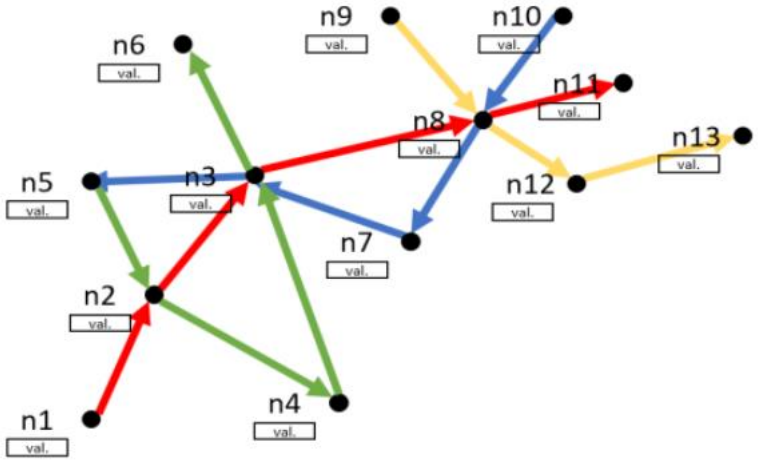


Рис. 3. Исходный ориентированный граф ([1])

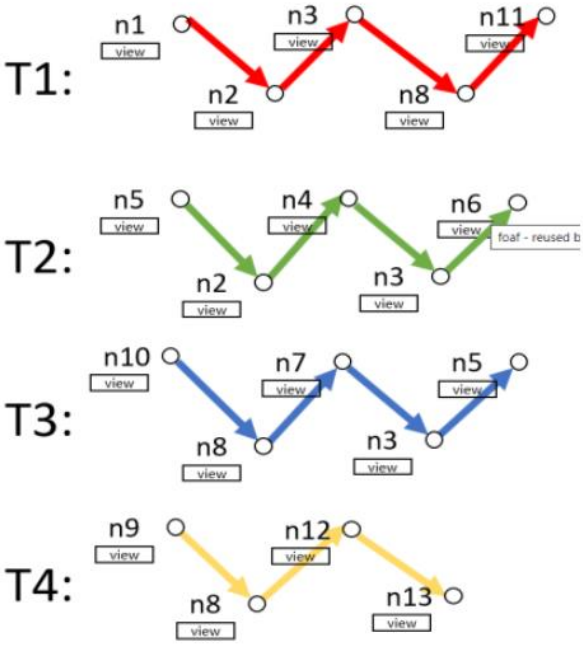


Рис. 4. Представление исходного графа в виде пучков траекторий ([1])

Алгоритм обхода пучков траекторий предоставляет инфраструктуру для улучшения качества вычислений на графе, проводимых иными, целевыми алгоритмами и, следовательно, требует некоторого усложнения при их реализации. В программе, содержащей исходный код для реализации целевого алгоритма, должны присутствовать 2 функции **U** – меняет значения вершин, которые посещаются при случайном обходе графа и **A** – попарно усредняет все значения, приписанные каждой вершине при каждом обходе. Они не будут совпадать для разных траекторий.

Что касается работы с новой структурой «траектория», то было предложено использовать 4 функции: **update**, **merge**, **exchange**, **read**.

Функция **update** служит для выполнения одного случайного обхода графа и изменения значений вершин (с применением упомянутой ранее функции **U**) после его завершения.

Функция **merge** удлиняет траекторию, производя слияние для двух траекторий с одинаковыми конечной и начальной вершиной, записанных в теле каждой траектории. Значение вершины, на которой произошло слияние меняем функцией **A**

– т. е. усредняем. Данный процесс позволяет с начала работы с большим графом представить его в виде множества траекторий и тем самым уменьшить расход ресурсов компьютера.

Функция **exchange** работает попарно с траекториями, где есть один и тот же узел, она также обращается к функции **A** с целью усреднить значение на «проблемной» вершине и генерирует новые, «перемешанные» траектории, которые, в свою очередь, уже обращаются к функции **update**.

Стоит отметить, что **merge** и **exchange** должны выполняться параллельно для каждого пучка траектории – на отдельном вычислительном узле. Чтобы повысить скорость выполнения этих функций, было принято решение использовать кэши (а именно алгоритм вытесняющих кэшей LRU Cache). Функция **read** добавляет в кэши новые значения вершин из новых траекторий, прошедших через выполнение функции **update,** не меняя их. Таким образом в кэше остаются данные о последних обойденных вершинах в пучке.

Главный недостаток алгоритма ОПГ – необходимость перевести привычные алгоритмы для анализа графов, которые будут базироваться на предложенной архитектуре памяти, на логику случайных блужданий. Примером подобного алгоритма может послужить алгоритм ранжирования ссылок PageRank. На его примере мы и будем сравнивать эффективность выполнения и простоту реализации привычных последовательных и параллельных алгоритмов. В частности, обратимся к последовательному прототипу алгоритма ОПГ, реализованному Е. В. Пановым.

Глава 2. Применение прототипа алгоритма ОПГ для PageRank

2.1 Текущие Python-реализации алгоритма PageRank

Проверим работу существующих в библиотеке NetworkX реализаций алгоритма ссылочного ранжирования PageRank. Действия по данному разделу практической части проведем в командной оболочке Jupyter Notebook.

Для этого воспользуемся представлением графа, приведенного в файле “src.edgelist”. Данный файл содержит в себе набор числовых строк, по два значения в каждой, разделенных пробелом. Каждое числовое значение из пары представляет собой номер вершины (узла, веб-страницы), а наличие строки, содержащий два определенных значения, говорит о том, что между данными узлами существует связь. Представление графа отсортировано по второму столбцу в порядке возрастания (по порядковому номеру второй вершины) => в первом столбце будут содержаться все соседние со второй вершины, а их количество укажет на степень узла.

Первым делом определим общую информацию о графе и визуализируем его. Используем функцию **view_graph()**, с помощью которой представим граф в должном для работы с библиотекой NetworkX формате. В качестве параметра передадим имя файла с представлением графа. Определим количество его вершин и ребер, а также количество изолированных вершин (имеющих нулевую степень). Затем отрисуем сеть, раскрасив вершины в разные цвета для наглядности.

Вывод функции:

Количество узлов в сети: 379

Количество связей в сети: 914

Количество изолированных узлов в сети: 0

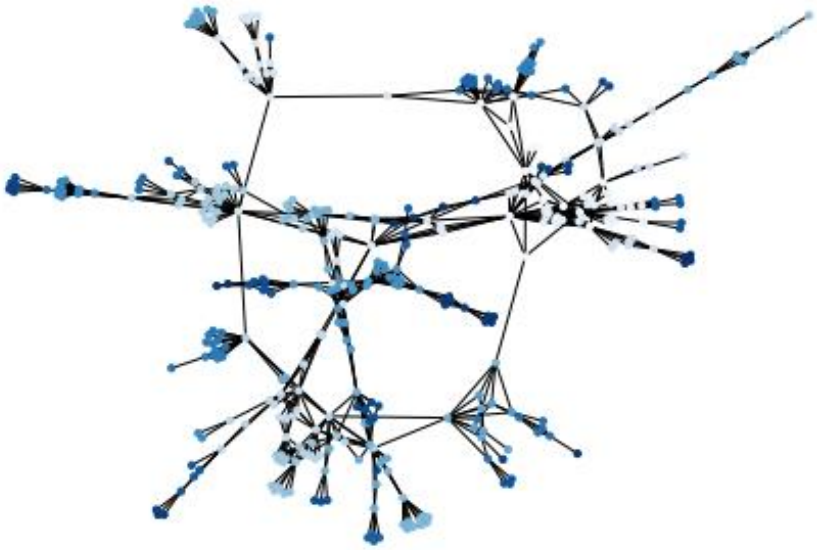


Рис. 5. Визуализация тестового графа

Видим, что большинство узлов плотно связаны со своими «соседями», а связи между дальними вершинами почти отсутствуют. При этом каждый узел имеет хотя бы по одной связи. Значит, скорее всего имеем дело с графом из модели «тесного мира». Также функция вернула нужное нам представление графа, которое мы сохраним в соответствующей переменной при вызове.

Далее вновь обратимся к пользовательской функции – **draw_nodes_distr(),** в качестве параметра передадим ей сохраненное в переменной G1 представление рассматриваемой сети. Получим распределение степеней графа, которое по форме близко к экспоненциальному.:

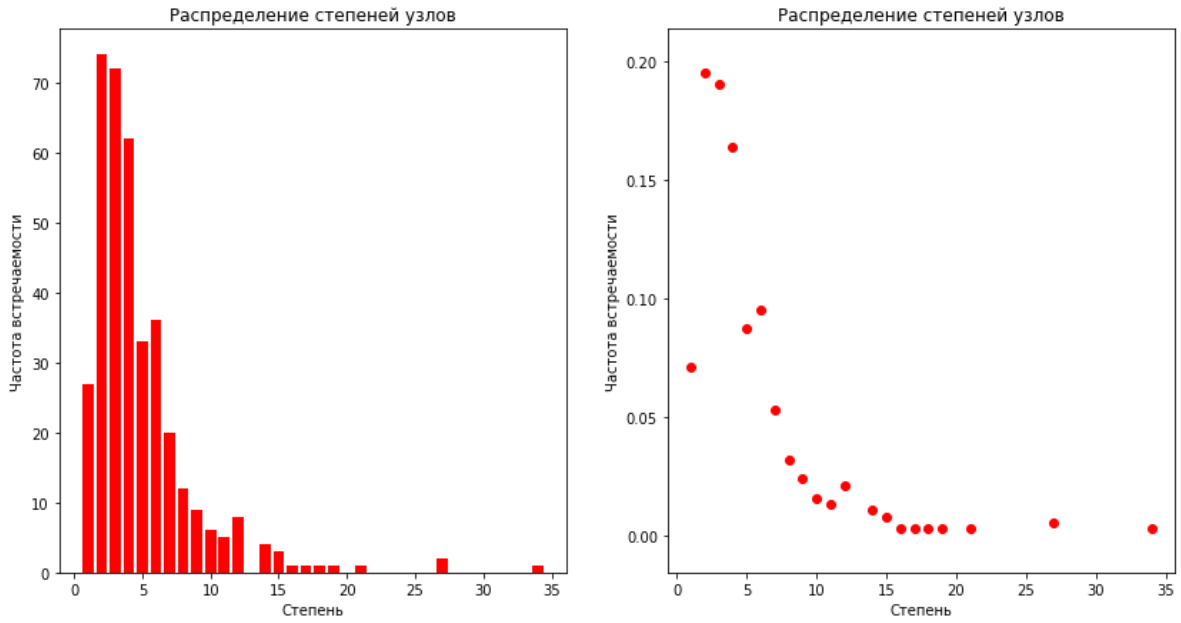


Рис. 6. Распределение степеней для тестового графа

После того, как получили базовое представление о том, с какой сетью имеем дело, подсчитаем веса узлов алгоритмом ссылочного ранжирования. В теоретической части были упомянуты три network-реализации для алгоритма PageRank: **nx.pagerank(), nx.pagerank_scipy()** и **nx.pagerank_numpy().** Их основное отличие заключается в методике определения точки сходимости алгоритма.

Применим все 3 реализации к графу G1 и отсортируем результаты работы каждой реализации в порядке убывания по значению искомого ранга. Эта процедура была отражена в функции **pageranking(),** возвращающей трехмерный массив, элементами которого являются вложенные списки с парами отсортированных по второму ключу значений «номер узла – pagerank-вес». Этот массив сохраним в переменной rank и обратимся к функции **visualize_pagerank(),** которая примет на вход переменную rank и отрисует графики с итогами выполнения каждой из трех nx-реализаций алгоритма pagerank.

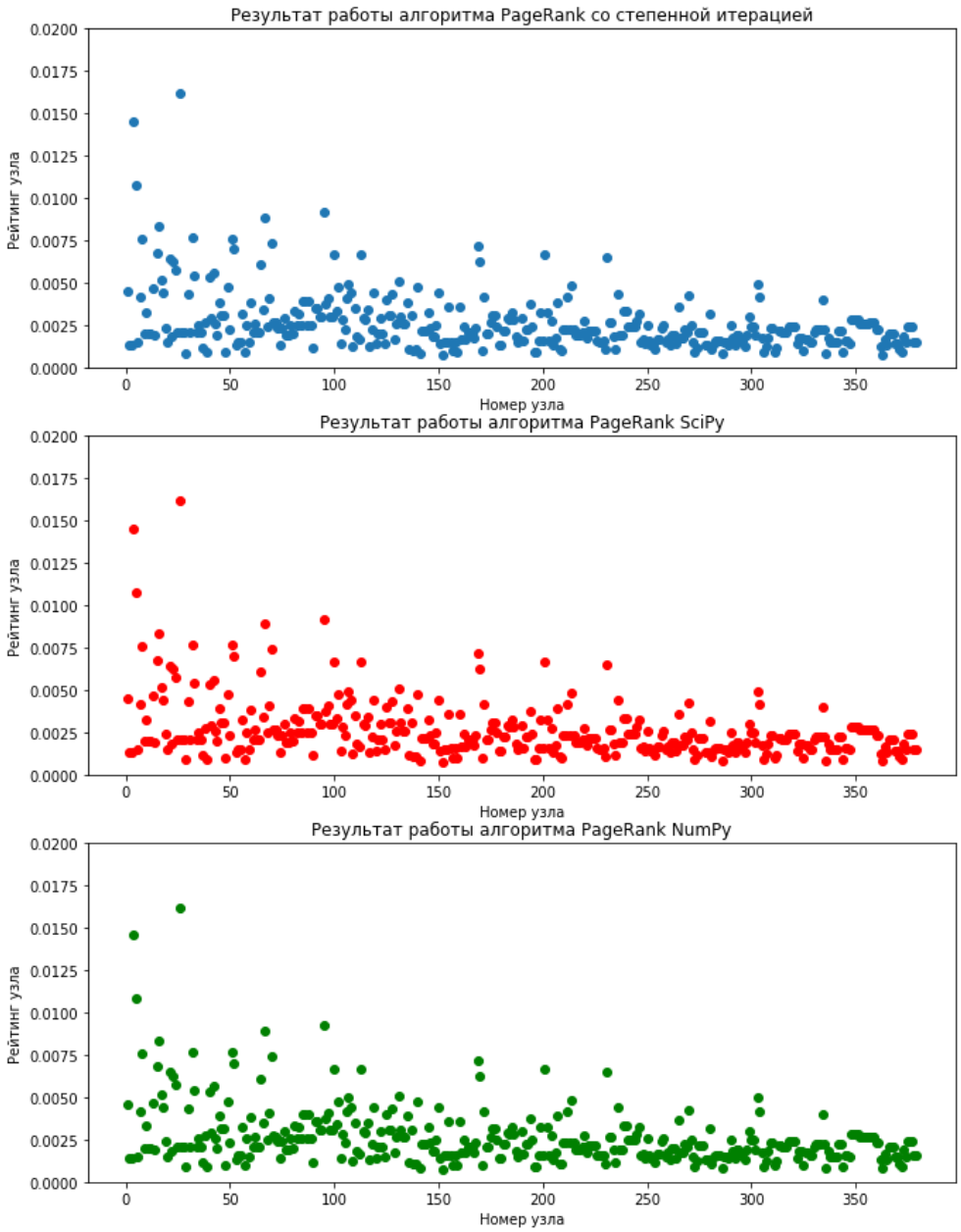


Рис. 7. Визуализация итогов выполнения pagerank из networkx

Видим, что визуально вычисленные рейтинги страниц-узлов (в терминах оригинальной задачи об определении весов веб-страниц во Всемирной Паутине) не отличаются. Возвращаемые функцией **visualize_pagerank(),** массивы вершин и их весов сохраним в переменной weights.

Теперь оценим расхождения в итогах работы различных реализациях алгоритма PageRank при помощи таких метрик как MAE (Mean Absolute Error) и MSE (Mean Squared Error) и выведем результаты на экран. Значения метрик найдем встроенными в пакет **sklearn.metrics** функциями, в качестве аргументов передадим

им рейтинги страниц-узлов, найденные различными способами и сохраненные в переменной weights.

Результат выполнения функции **compare_pageranks():**

MAE для PageRank и PageRank Scipy: 4.711559308927772e-19

MSE для PageRank и PageRank Scipy: 5.261810464828524e-37

MAE для PageRank и PageRank Numpy: 3.4070607923808104e-06

MSE для PageRank и PageRank Numpy: 2.559679564147556e-11

MAE для PageRank SciPy и PageRank Numpy: 3.4070607923808887e-06

MSE для PageRank SciPy и PageRank Numpy: 2.5596795641475834e-11

Убеждаемся в том, что расхождения в значениях pagerank незначительны от реализации к реализации, причем у PageRank со степенной итерацией («классического») и PageRank, использующего инструментарий пакета scipy они наименьшие.

Наконец, оценим время исполнения каждой из используемых PageRank-реализаций, воспользовавшись пользовательской функцией **algorithm_timer()** – аргументом является представление графа в виде списка смежных вершин (в нашем случае это G1), где проведем выполнение алгоритма в трех версиях, зафиксируем время начала и время окончания их работы. Результат выведем на экран в секундах:

Время выполнения PageRank для графа с 379 вершинами и 914 ребрами:

Для реализации со степенной итерацией: 0.09278 сек.

Для scipy-реализации: 0.003991 сек.

Для numpy-реализации: 0.051451 сек.

Таким образом, самой быстрой оказалась SciPy реализация алгоритма PageRank, а учитывая размер погрешности, найденный выше, можно сделать вывод о том, что для данного «хорошего» графа целесообразнее использовать именно эту реализацию. Помимо прочего: наш граф имеет всего 379 вершин и 914 ребер, но разница во времени выполнения уже наглядна, и с ростом размерности графа она будет все существеннее. Потому необходимо проверить, как изменится скорость

работы алгоритма PageRank, если использовать для его представления архитектуру пучков траекторий.

Однако, прежде чем рассматривать прототип алгоритма ОПГ, построим столбчатую диаграмму с 10-ю вершинами, имеющими максимальный вес. Это нужно для того, чтобы проверять правильность работу PageRank с использованием обхода пучков траекторий визуально.

При построении графика пользовались функцией **most_valuable_pages()** (аргумент – представление графа в виде списка смежности graph и нужное кол-во страниц из топа по весам tops_number, значение по умолчанию – 10) При выборе значений весов можем опираться на любую реализацию из networkx, так как погрешности не существенны, здесь представлены веса, вычисленные стандартным **nx.pagerank()**.

2.2 Последовательный прототип алгоритма ОПГ

Рассмотрим реализацию последовательного алгоритма ОПГ на ЯП Python, начатую Е. В. Пановым. Реализация проведена с использованием парадигм объектно-ориентированного программирования (ООП). Система, описывающая принцип работы обхода пучков траекторий, была запущена, изучена и протестирована нами в среде программирования PyCharm от JetBrains.

Программа, содержащая в себе описание параллельного прототипа ОПГ, состоит из следующих файлов:

- cache.py
- data_1.pickle
- data.py
- funcs.py
- handler.py
- main.py
- node.py
- trajectory_dumb.py

- trajectory.py
- values.py

Файл **cache.py** предназначен для использования кэша “least recently used”: для повышения эффективности работы с пучками траекторий необходимо использовать такую модель хранения данных, которая позволила бы обращаться к определенному количеству последних рассмотрений в пучке траектории узлов, в то время как «устаревшие» узлы очищались бы из памяти, оставляя место для новых узлов. Он содержит 3 класса: **CacheStruct**, **QNode**, **LRUCache**.

Файл **data_1.pickle** включает в себя сгенерированные в ходе работы программы последовательности траекторий из множества траекторий. Заполняется **data_1.pickle** в функции **save_trajectories(),** расположенной в исполняемом Python-файле **trajectory_dumb.py,** где также происходит чтение графа в виде списка смежности из файла, выделяются его вершины и ребра, происходит считывание существующих траекторий и в целом проводится основная работа по превращению сырых данных в структуры, используемые в архитектуре обхода пучков траекторий.

Файлы **handler.py**, **node.py**, **trajectory.py** описывают соответствующие классы для объектов типов Пучок, Узел и Траектория (**Handler**, **Node**, **Trajectory**). Каждый класс имеет как стандартные конструкторы и «магические» методы перегрузки, так и пользовательские. Так, операции слияния траекторий с одним и тем же головным (head) и хвостовым (tail) узлами, а также перемешивания траекторий, когда происходит перемена местами головных (head) узлов в паре траекторий выполняются в функциях **merge()** и **exchange()** класса **Handle** соответственно.

В **data.py** происходит работа с загруженными из файла траекториями, их перевод на язык ООП – превращение в объекты, а также аналогичные действия с узлами, в них содержащимися. Кроме того, здесь же делается вывод результатов формирования траекторий, передаваемый в главную функцию **main().** Инициализация начальных значений для переменных, предназначенных для хранения данных кэша с последовательностями последних «рабочих» узлов, кэша с

узловыми «хвостами», списка пучков траекторий, количества слияний и перемешиваний траекторий задается в файле **values.py**.

Файл **funcs.py**, в котором хранятся функции first_average() (первое попарное усреднение значений узла) и first_update() (первое обновление значения узла после усреднения) будем использовать как хранилище для реализации алгоритма ссылочного ранжирования PageRank в контексте обхода пучков траекторий.

Также построим UML-диаграммы для каждого класса, участвующего в обработке графа, используя встроенный плагин среды разработки PyCharm (см. Приложение 2).

2.3 PageRank в инфраструктуре обхода пучков траекторий

Прежде чем приступить к настройке алгоритма PageRank, необходимо создать «эталонную» функцию, вычисляющую значения весов для вершин рассматриваемого графа, по методике SciPy из NetworkX. Эту функцию назовем **pageranking()** и поместим в файл **funcs.py**. Ее возвращаемым значением будет двумерный список вида [[node1, pagerank1], [node2, pagerank2], …, [node379, pagerank379]], отсортированный по убыванию номеров вершин, приведенных к типу integer.

Далее создадим в этом же файле функцию **OPG_pagerank()**, принимающую на вход множество сформированных траекторий и возвращающую двумерный список аналогичного двумерному списку, упомянутому ранее, вида. Поскольку множество траекторий содержит в себе все траектории, созданные программой с начала случайного блуждания по графу, которое, в свою очередь, подразумевает рандомизированный выбор следующей вершины для посещения в рамках формирования траектории, то вершины, имеющие больше всего ссылок на себя, будут появляться чаще своих соседей при любом старте обхода графа. К примеру, вершина, имеющая 50 ссылок из узлов с большим количеством ссылок на себя, будет присутствовать в большем числе траекторий (попадет в нее с большей вероятностью), чем узел с одной ссылкой.

Исходя из этого предлагается следующий алгоритм PageRank в инфраструктуре алгоритма обхода пучков траекторий:

1) Создается список кортежей **nod** для записи пары (суммарное значение узла, подсчитанное по всем траекториям; число траекторий, содержащих данный узел).
2) Выполняется обход по каждой траектории из множества траекторий, по всем узлам из конкретной траектории.
3) Каждый узел либо добавляется в кортеж **nod,** если не был обойден раннее, либо его значения обновляются: к первому – прибавляется текущее значение узла, ко второму – единица.
4) Создается переменная **total_n** для хранения усредненных значений всех узлов графа, в цикле выполняется расчет усредненного значения для рассматриваемого узла, также оно добавляется к переменной **total_n**.
5) Усредненное значение каждого узла делится на **total_n –** это и будет решение задачи PageRank для данной вершины
6) Полученный итоговый кортеж сортируется по номеру вершины по убыванию и конвертируется в двумерный список для простоты сравнения с результатом выполнения функции **pageranking().**

Для сравнения результатов решения задачи ссылочного ранжирования при двух разных подходах к обходу и представлению графа будем пользоваться функцией **pagerank_deviations().** В ней вычислим и выведем на экран значения метрик MAE и MSE:

$$MAE = \frac{1}{number\ of\ nodes} \sum_{i=1}^{number\ of\ nodes} |pagerank_{nx\_i} - pagerank_{OPG\_i}|$$

$$MSE = \frac{1}{number\ of\ nodes} \sum_{i=1}^{number\ of\ nodes} (pagerank_{nx\_i} - pagerank_{OPG\_i})^2$$

Вызовем описанные выше функции в main(), предварительно представив граф из файла в надлежащем виде:

```
# двумерные списки вида [номер узла, ранжированный вес узла]
```

```
pg = pageranking()
pagerank = OPG_pagerank(t)
# отклонения в значениях, вычисленных двумя методами
pagerank_deviations(pagerank)
```

Получим в консоли отклонения значений pagerank для вершин графа:

```
Сравнение ОПГ
Средняя абсолютная ошибка:
0.0004793672567496542
Среднеквадратичная ошибка:
4.5820716553957985e-07
```

Заключаем, что в среднем отклонения, полученные по метрикам MAE и MSE невелики.

Однако при применении алгоритма ссылочного ранжирования, даже малейшие отличия в знаках после запятой могут привести к серьезным последствиям: возможны не только численные отклонения, но и составления неверного рейтинга веб-сайтов, если наш алгоритм будет использован поисковой системой.

Создадим отдельный исполняемый файл **compare_algos.py,** в который поместим вспомогательные функции, позволяющие сравнивать итоги решения задачи ссылочного ранжирования на одном и том же графе, но двумя разными методами.

Чтобы проверить, какие сайты-узлы попали в топ-10, воспользуемся функцией **most_valuable_pages()**. На вход подаем переменную с результатами выполнения функции **OPG_pagerank()**, формируем отдельно список «страниц» и их «рейтингов», сортируем по убыванию рейтинга и отображаем результаты на столбчатой диаграмме, аналогичной той, что строили при первичном анализе графа:

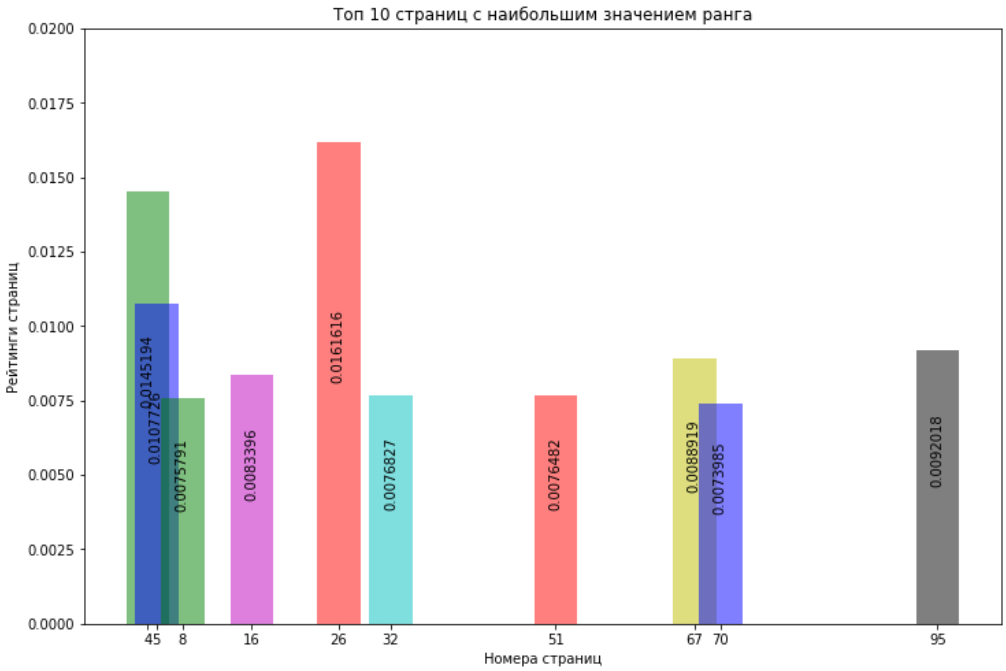


Рис. 8. Узлы с наибольшими весами по networkx

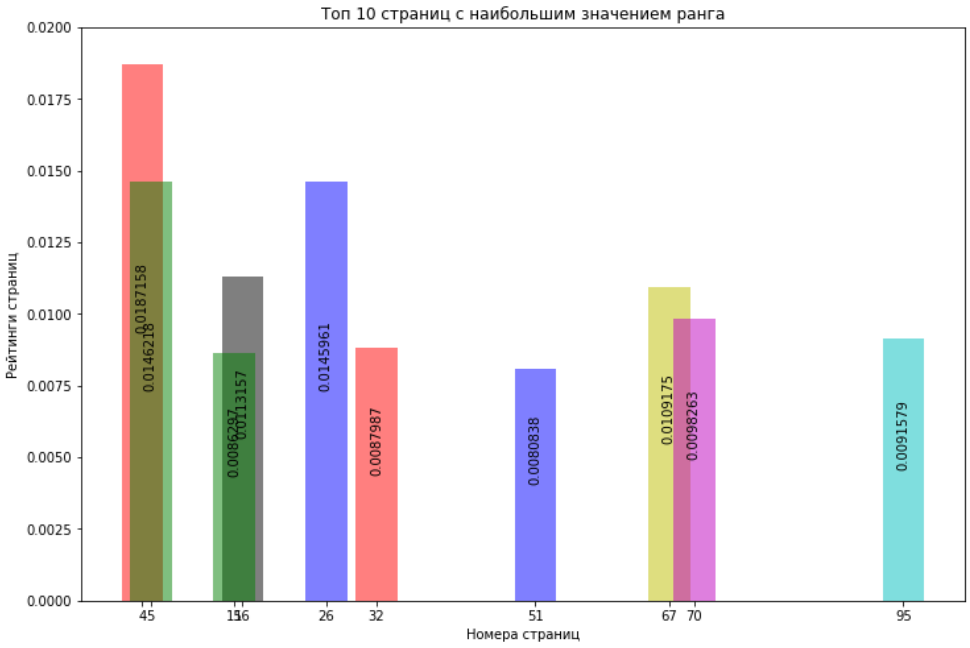


Рис. 9. Узлы с наибольшими весами по ОПГ

Отмечаем, что при работе алгоритма PageRank Scipy в топ-10 попал узел с номером 8, в то время как алгоритм PageRank OPG вместо узла с номером 8 добавил в топ-10 узел с номером 15. И, тем не менее, прочие узлы совпали, и значения их весов находятся близко друг к другу.

В таком случае, проверим, какое количество вершин графа получило неправильное место в рейтинге и отобразим данные различия графически.

Вызываем из главного метода **main()** метод **find_not_right():** в отсортированных по убываний пейджранков списках проверяется соответствие номеров узлов по индексу места в топе. Затем строим график соответствия узлов в двух реализациях PageRank. При идентичности результатов линия должна представлять собой 45-градусную прямую:

Рис. 10. Проверка идентичности ранжирования вершин

Количество узлов с "неправильными" местами в рейтинге: 372

Таким образом, незначительные расхождения в величинах пейджранков повлекли за собой колоссальные расхождения в местах веб-страниц в рейтинге поисковика. Подавляющее большинство точек находится на значительном расстоянии от идеальной прямой, расположенной под углом 45 градусов по отношении к осям.

Проиллюстрируем итоги работы двух версий алгоритма pagerank:

Рис. 11. Сравнение итогов ранжирования ссылок

Здесь синим цветом отмечены веса вершин, найденные с применением инфраструктуры алгоритма ОПГ, а зеленым, соответственно, реализации из библиотеки для анализа больших графов и сложных сетей NetworkX. В целом наблюдаем тенденцию к небольшому расхождению весов, исходя из наложения линий.

Несмотря на то, что работу алгоритма PageRank в данной инфраструктуре необходимо улучшить, исходя из проведенного анализа, мы, тем не менее, можем убедиться в том, что даже при запуске прототипа программы с использованием ОПГ, задача ссылочного ранжирования решается эффективнее.

Сравним время исполнения команд для **nx.pagerank_scipy()** (для рассматриваемого графа - самой быстрой «пакетной» реализации на Python) и новым алгоритмом. Объявим в **compare_algos.py** функцию **time_compare(),** где зафиксируем время старта и окончания работы каждого алгоритма, найдем разницу между ними и повторим процедуру 100 раз и определим среднюю разницу в работе. Выведем итоги на экран через **main():**

Время выполнения nx.pagerank_scipy():

0.007979 сек.

Время выполнения PageRank на ОПГ:

0.0 сек.

Средняя разница во времени выполнения:

0.00833762 сек.

Таким образом, на графе с 379 вершинами в среднем были сэкономлены доли секунды. Однако разница в скорости обработки данных составляет кратное число раз и в перспективе, при увеличении размерности анализируемой сети, подобное качество алгоритма ОПГ позволит существенно сэкономить вычислительные ресурсы. Кроме того, не стоит забывать о том, что проведенные нами действия выполнялись исходя из наличия прототипа алгоритма ОПГ, а не его полномасштабной программно-аппаратной реализации с переносом вычислений на графический процессор NVIDIA.

ЗАКЛЮЧЕНИЕ

Исследование графов представляет особый интерес в современном мире в связи с возникновением множества прикладных задач, связанных с их обработкой. Поэтому возникает необходимость оптимизировать эту обработку, максимально высвободив вычислительные мощности технических устройств. Один из наиболее перспективных способов добиться нужного результата – использовать параллельные вычисления.

Несмотря на наличие некоторого количества реализованных решений для параллелизма применительно к работе с большими графами, все еще имеют место проблемы с нерегулярными доступами к памяти, компоновкой данных, эффективностью перераспределения задач между CPU и GPU. Создатели архитектуры CUDA приблизились к разрешению данных проблем, однако в полной мере оно невозможно без создания принципиально новой алгоритмической парадигмы.

Предлагаемый С. В. Макрушиным и Е. В. Пановым алгоритм обхода пучков траекторий способствует избавлению от существенного недостатка многих методов работы с массивным параллелизмом на графах: не локальностью представления данных. Основанный на использовании особой структуры данных – множества траекторий случайных блужданий по вершинам графа, сохраняемых в памяти последовательно, но в неявном виде, он увеличивает скорость взаимодействия с оперативной памятью и позволяет применять на практике более оптимальное представление графа.

Рассмотрев классическую задачу ссылочного ранжирования (PageRank), мы определили веса вершин графа размерности (V, E) = (379, 914), используя стандартные средства библиотеки NetworkX и реализацию в контексте инфраструктуры алгоритма обхода пучков траекторий. Получив достаточно близкие в численном смысле значения весов для вершин, мы, тем не менее, были вынуждены отметить, что алгоритм PageRank нуждается в более тщательной доработке при условии его применения с ОПГ. Если представить исследуемый граф как сеть из

веб-страниц, которые выдаются пользователю поисковой системой, то при запуске разных версий PageRank, система выдала бы ему абсолютно разные, несмотря на близость расположения мест, рейтинги сайтов.

Однако так же были замечены существенные улучшения в скорости обработки информации в зависимости от концепции алгоритма, применяемого при решении задачи ссылочного ранжирования PageRank, даже относительно графа не столь большой размерности.

Подводя итоги, стоит отметить, что доработку алгоритма ссылочного ранжирования, впрочем как и любого другого алгоритма анализа сложной сети, под инфраструктуру обхода пучков траекторий следует считать перспективной, а возможности, которые открывает применение архитектуры CUDA при воплощении его идеи на практике поистине революционными.

ЛИТЕРАТУРА

1. Макрушин С. В. Параллельный алгоритм анализа сетей на основе обхода пучка траекторий / С. В. Макрушин, Е. В. Панов // ВШЭ, 2020
2. Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens Gunrock: GPU Graph Analytics. // ACM Trans. Parallel Comput. 9, 4, Article 39, 2017
3. L. Page, S. Brin, et al. The PageRank Citation Ranking: Bringing Order to the Web // Technical report, Stanford Digital Library Technologies Project, 1998
4. Xuanhua Shi, Hai Jin, Yongluan Zhou Graph Processing on GPUs // ACM Computing Surveys, 2017
5. Random Walks on Graphs [Электронный ресурс] – Режим доступа: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-spring-2015/readings/MIT6_042JS15_Session35.pdf, свободный (01.05.2020)
6. PageRank and Random Walks on Directed Graphs [Электронный ресурс] – Режим доступа: http://www.cs.yale.edu/homes/spielman/462/2010/lect16-10.pdf, свободный (02.05.2020)
7. Networkx Tutorial [Электронный ресурс] – Режим доступа: https://networkx.github.io/documentation/stable/tutorial.html#, свободный (08.05.2020)
8. KONECT Networks DataBase [Электронный ресурс] – Режим доступа: http://konect.uni-koblenz.de/networks/, свободный (08.05.2020)
9. Stack Overflow [Электронный ресурс] – Режим доступа: https://stackoverflow.com/, свободный (12.05.2020)
10. Py Charm [Электронный ресурс] – Режим доступа:

https://www.jetbrains.com/ru-ru/pycharm/, свободный (12.05.2020)

ПРИЛОЖЕНИЕ

Приложение 1.1 Листинг файла compare_algos.py

```
import networkx as nx
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import datetime
from funcs import *
from statistics import mean
def most_valuable_pages(p_r, tops_number = 10):
    p_r = (sorted(p_r, key = lambda item:item[1], reverse = True))
    print(p_r)
    nodes = []
    ranks = []
    for i in range(len(p_r)):
        nodes.append(p_r[i][0])
        ranks.append(p_r[i][1])
    nodes_classic = np.array(nodes[0:tops_number])
    ranks_classic = np.array(ranks[0:tops_number])
    lal = list(map(str, nodes_classic))
    bar_lab = [round(rank, 7) for rank in ranks_classic]
    title = "Топ " + str(tops_number) + " страниц с наибольшим значением ранга"
    my_colors = 'rgbkymc'
    f, ax = plt.subplots(figsize = (12, 8))
    br = plt.bar(nodes_classic, height = ranks_classic, width = 5, alpha = 0.5, color = my_colors, tick_label
= lal)
    title = "Топ " + str(tops_number) + " страниц с наибольшим значением ранга"
    plt.xlabel("Номера страниц")
    plt.ylabel("Рейтинги страниц")
    plt.title(title)
    def autolabel(rects):
```

```
        for idx, rect in enumerate(br):
            height = rect.get_height()
            ax.text(rect.get_x() + rect.get_width()/2., 0.5*height,
                    bar_lab[idx],
                    ha = 'center', va = 'bottom', rotation = 90)
    autolabel(br)
    plt.ylim(0,0.02)
    plt.show()
    return p_r
def find_not_right(PR_classic, PR_OPG):
    PR_classic = (sorted(PR_classic, key = lambda item:item[1], reverse = True))
    PR_OPG = (sorted(PR_OPG, key = lambda item:item[1], reverse = True))
    comp = []
    y1 = []
    y2 = []
    for i in range(len(PR_OPG)):
        if (PR_OPG[i][0] != PR_classic[i][0]):
            comp.append(1)
        y1.append(PR_OPG[i][0])
        y2.append(PR_classic[i][0])
    plt.scatter(y1, y2, color = 'green', marker = 'o')
    plt.plot(y1, y1, color = 'red')
    plt.title('График несовпадения мест в рейтинге страниц')
    plt.show(
    print('Количество узлов с "неправильными" местами в рейтинге:', len(comp))
def plot_PR_values(PR_classic, PR_OPG):
    PR_classic_rates = []
    PR_OPG_rates = []
    PR_nodes = []
    for i in range(len(PR_classic)):
        PR_classic_rates.append(PR_classic[i][1])
        PR_OPG_rates.append(PR_OPG[i][1])
        PR_nodes.append(PR_classic[i][0])
    plt.plot(PR_nodes, PR_classic_rates, '-g', label = 'SciPy из NetworkX')
    plt.plot(PR_nodes, PR_OPG_rates, '--b', label = 'Метод ОПГ')
```

```
    plt.xlabel('Порядковый номер страницы')
    plt.ylabel('PageRank-значение')
    plt.title('PageRank в зависимости от номера узла')
    plt.show()
def time_compare(t):
      times = []
      for i in range(100):
        start = datetime.datetime.now()
        pg = pageranking()
        end = datetime.datetime.now()
        work_time1 = (end - start).total_seconds()
        start = datetime.datetime.now()
        pagerank = OPG_pagerank(t)
        end = datetime.datetime.now()
        work_time2 = (end - start).total_seconds()

        time_dif = work_time1 - work_time2
        times.append(time_dif)
      print("Время выполнения nx.pagerank_scipy():")
      print(work_time1, 'сек.')
      print("Время выполнения PageRank на ОПГ:")
      print(work_time2, 'сек.')
      print("Средняя разница во времени выполнения:")
      print(mean(times), 'сек.')
      return work_time1, work_time2, mean(times)
```

Приложение 1.2 Листинг функций из funcs.py

```
    def OPG_pagerank(trajectories):
        nod = {}
        for nodes in trajectories:
            #print(nodes)
            for node in nodes:
                #print(node)
```

```
            if node.id in nod:
                # меняем значение узла
                # а также увеличиваем частоту появления во множестве траекторий
                # если уже рассматривали данный узел
                nod[node.id] = [nod[node.id][0] + node.value, nod[node.id][1] + 1]
            else:
                # если рассматриваем узел впервые,
                # то добавляем его в nod
                nod[node.id] = [node.value, 1]
    total_n = 0
    for i in node.keys():
        # вычисляем для каждого узла его итоговое среднее значение
        node[i] = node[i][0] / node[i][1]
        # и сумму средних значений всех узлов
        total_n = total_n + node[i]
    for i in node.keys():
        # значение PageRank для i-го узла:
        node[i] = node[i]/total_n
    # сортировка по ключу
    so = sorted(n.items(), key = operator.itemgetter(0), reverse = True)
    pages = []
    for i in range(len(res)):
        pages.append(list(so[i]))
    return pages
def pagerank_deviations(OPG_pagerank):
    errors = []
    true_pr = pageranking()
    for i in range(len(true_pr)):
        errors.append([true_pr[i][0], (true_pr[i][1] - OPG_pagerank[i][1])])
    MAE = 0
    MSE = 0
    for i in errors:
        MAE = MAE + abs(i[1])
        MSE = MSE + i[1]**2
    MAE = MAE/len(errors)
```

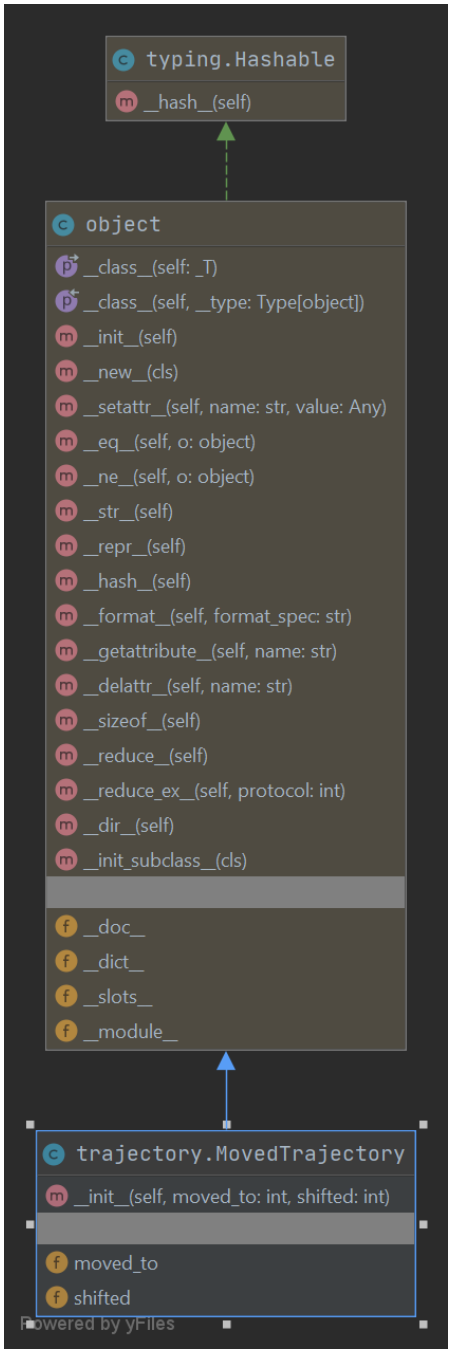
```
    MSE = MSE/len(errors)
    print('Сравнение ОПГ ')
    print('Средняя абсолютная ошибка:')
    print(MAE)
    print('Среднеквадратичная ошибка:')
    print(MSE)
    return MAE, MSE
```

Приложение 1.3 Листинг из main.py

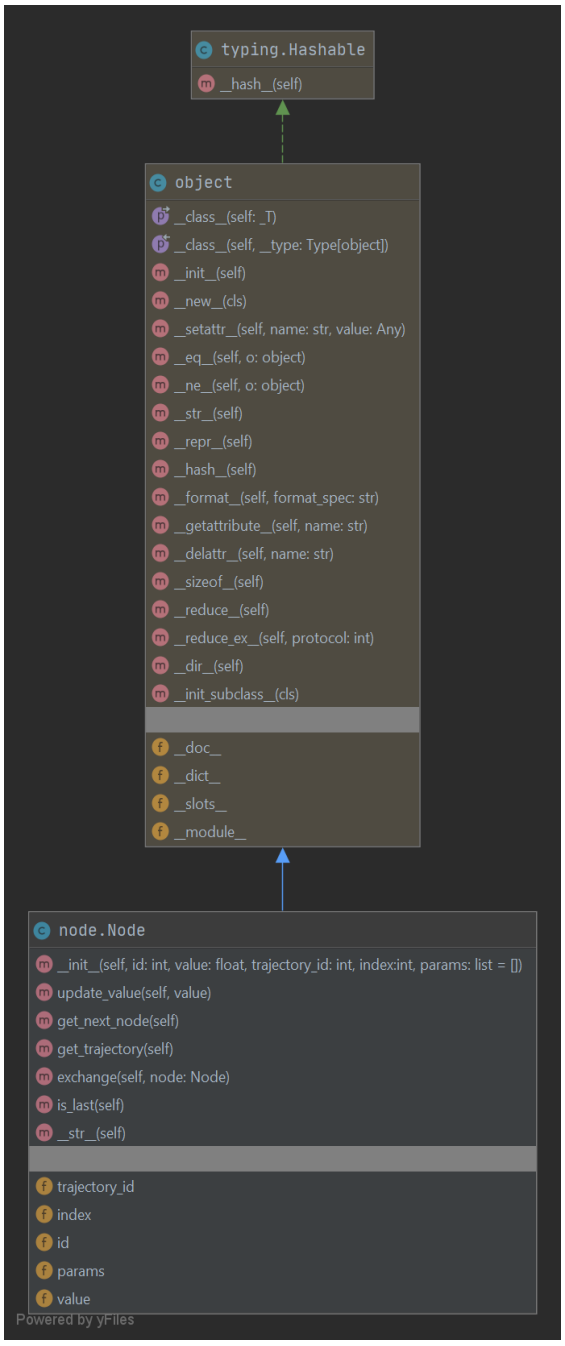
```
# двумерные списки вида [номер узла, ранжированный вес узла]
pg = pageranking()
pagerank = OPG_pagerank(t)
# отклонения в значениях, вычисленных двумя методами
pagerank_deviations(pagerank)
# топ-n страниц (узлов) по значению весов
# рисует график
most_valuable_pages(pagerank)
# визуализация мест в "рейтингах" всех узлов
# вычисленных двумя способами
find_not_right(pg, pagerank)
# ранг в зависимости от номера узла
plot_PR_values(pg, pagerank)
time_compare(t)
```

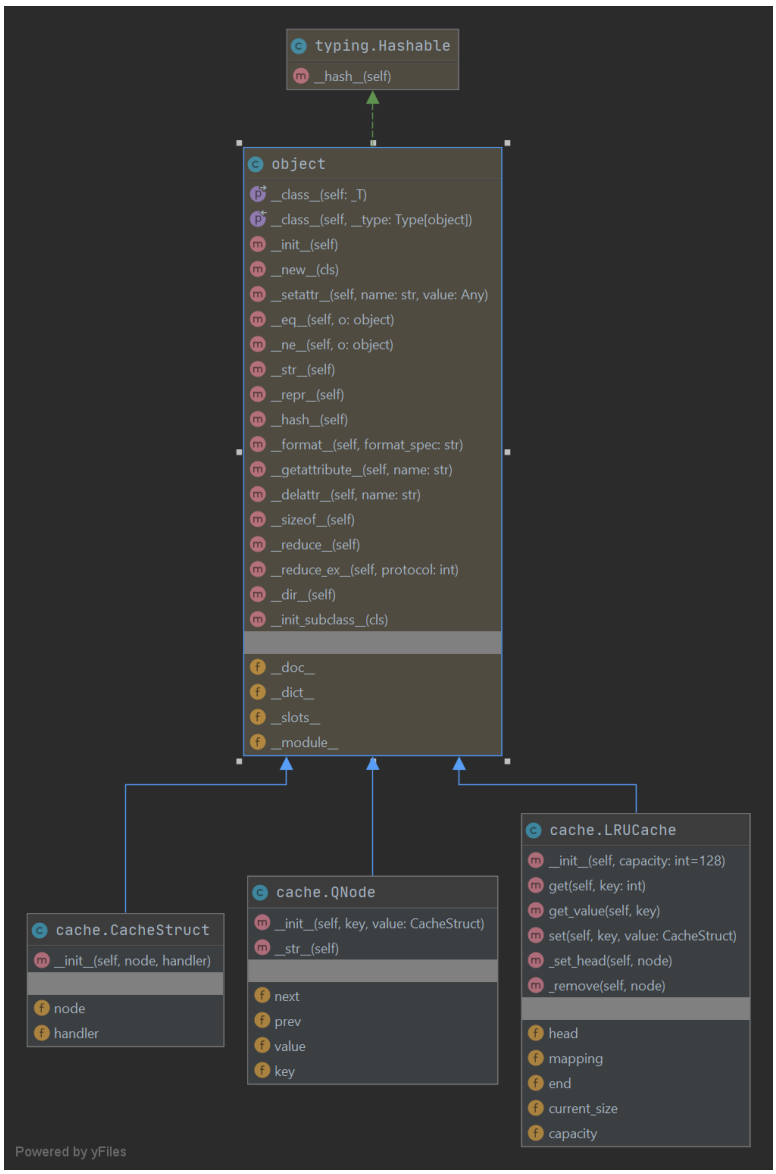
Приложение 2. UML-диаграммы для классов базовой реализации алгоритма ОПГ

Класс Moved Trajectory

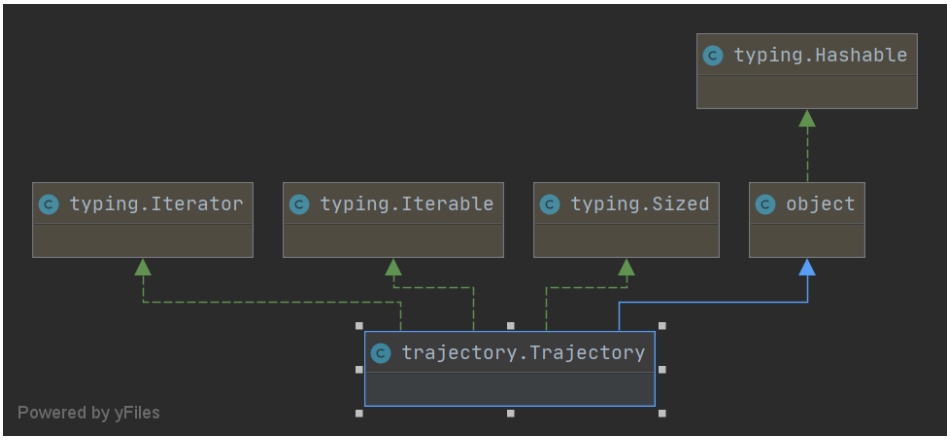


Класс Node





Классы CacheStruct, QNode, LRUCache



Класс Trajectory

Приложение 3. Листинг IPYNB-файла последовательного анализа графа

In [2]:

```
import matplotlib
import matplotlib.pyplot as plt
from collections import Counter
```

In [3]:

```
import networkx as nx
```

In [246]:

```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [4]:

```
import datetime
```

In [5]:

```
import warnings
warnings.filterwarnings('ignore')
```

In [247]:

```
import scipy as sc
#%matplotlib inline
```

In [58]:

```
import sklearn.metrics as mr
```

In [195]:

```
import matplotlib.colors as mcolors
import numpy as np
```

In [8]:

```
# Первичная работа с графом
def view_graph(filename):

    # вытягиваем список смежности из файла
    G = nx.read_edgelist(filename, comments='%', nodetype=int)
    V = int(G.number_of_nodes()) # кол-во вершин
    E = int(G.number_of_edges()) # кол-во ребер

    # вывод результатов
    print('Количество узлов в сети:', V)
    print('Количество связей в сети:', E)
    print('Количество изолированных узлов в сети:', len(list(nx.isolates(G))))

    # визуализация графа
    pos = nx.spring_layout(G, iterations = 200)
    nx.draw(G, pos, node_color = range(V), node_size = 10, cmap = plt.cm.Blues)
    return G
```

In [267]:

```
def draw_nodes_distr(graph):

    dgr_seq = sorted([d for n, d in graph.degree()], reverse=True)
    ndeg_cntr = Counter(dgr_seq)
    deg, cnt = zip(*ndeg_cntr.items())
    cnt_p = [c/graph.number_of_nodes() for c in cnt]
```

```
    fig, ax = plt.subplots(1,2, figsize = (14,7))
    ax[0].bar(deg, cnt, color='r')
    ax[1].scatter(deg, cnt_p, color='r')
    ax[0].set_title("Распределение степеней узлов")
    ax[1].set_title("Распределение степеней узлов")
    ax[0].set_ylabel("Частота встречаемости")
    ax[0].set_xlabel("Степень")
    ax[1].set_ylabel("Частота встречаемости")
    ax[1].set_xlabel("Степень")
    plt.show()
```

In [143]:

```
# Применение PageRank к графу в трех вариациях
def pageranking(graph):

    # словарь с итогами вычислений
    realizations = {
        'power iteration': nx.pagerank(graph),
        'scipy': nx.pagerank_scipy(graph),
        'numpy': nx.pagerank_numpy(graph)
    }

    page_ranks = []
    for realization in realizations:
        #print(realizations[realization])
        page_ranks.append(list(realizations[realization].items()))
    #print(page_ranks)

    p_r = []
    ranks = []
    for j in range(3):
        p_r.append([])
        ranks.append([])
        for i in range(len(page_ranks[j])):
                p_r[j].append(list(page_ranks[j][i]))
                p_r[j][i][0] = int(p_r[j][i][0])
        ranks[j] = (sorted(p_r[j], key = lambda item:item[1], reverse = True)) # сортиро
вка списков по значениям весов
    return ranks
```

In [153]:

```
def vizualize_pagerank(p_r):
    nodes = []
    ranks = []
    for i in range(len(p_r)):
        nodes.append([])
        ranks.append([])
        for j in range(len(p_r[i])):
            nodes[i].append(p_r[i][j][0])
            ranks[i].append(p_r[i][j][1])
    #print(nodes)

    f = plt.figure(figsize = (11,15))
    ax1 = f.add_subplot(311)
    #plt.subplot(2, 1, 1)
    ax1.scatter(nodes[0], ranks[0])
    ax1.set_ylim(0, 0.02)
    ax1.set_title('Результат работы алгоритма PageRank со степенной итерацией')
    ax1.set_xlabel('Номер узла')
    ax1.set_ylabel('Рейтинг узла')

    ax2 = f.add_subplot(312)
    ax2.scatter(nodes[1], ranks[1], color='r')
    ax2.set_ylim(0, 0.02)
    ax2.set_title('Результат работы алгоритма PageRank SciPy')
    ax2.set_xlabel('Номер узла')
    ax2.set_ylabel('Рейтинг узла')
```

```
    ax2 = f.add_subplot(313)
    ax2.scatter(nodes[2], ranks[2], color='g')
    ax2.set_ylim(0, 0.02)
    ax2.set_title('Результат работы алгоритма PageRank NumPy')
    ax2.set_xlabel('Номер узла')
    ax2.set_ylabel('Рейтинг узла')
    return nodes, ranks
```

In [167]:

```
def compare_pageranks(weights):
    weighted_classic = weights[1][0]
    weighted_scipy = weights[1][1]
    weighted_numpy = weights[1][2]
    m_ab_er1 = mr.mean_absolute_error(weighted_classic, weighted_scipy)
    m_sq_er1 = mr.mean_squared_error(weighted_classic, weighted_scipy)
    m_ab_er2 = mr.mean_absolute_error(weighted_classic, weighted_numpy)
    m_sq_er2 = mr.mean_squared_error(weighted_classic, weighted_numpy)
    m_ab_er3 = mr.mean_absolute_error(weighted_scipy, weighted_numpy)
    m_sq_er3 = mr.mean_squared_error(weighted_scipy, weighted_numpy)
    print('MAE для PageRank и PageRank Scipy:', m_ab_er1)
    print('MSE для PageRank и PageRank Scipy:', m_sq_er1)
    print()
    print('MAE для PageRank и PageRank Numpy:', m_ab_er2)
    print('MSE для PageRank и PageRank Numpy:', m_sq_er2)
    print()
    print('MAE для PageRank SciPy и PageRank Numpy:', m_ab_er3)
    print('MSE для PageRank SciPy и PageRank Numpy:', m_sq_er3)
```

In [269]:

```
def algorithm_timer(graph):
    v = graph.number_of_nodes()
    e = graph.number_of_edges()
    print('Время выполнения PageRank для графа с', v, 'вершинами и', e, 'ребрами:')
    start = datetime.datetime.now()
    nx.pagerank(graph)
    end = datetime.datetime.now()
    work_time = (end - start).total_seconds()
    print('Для реализации со степенной итерацией:', work_time, 'сек.')
    start = datetime.datetime.now()
    nx.pagerank_scipy(graph)
    end = datetime.datetime.now()
    work_time = (end - start).total_seconds()
    print('Для scipy-реализации:', work_time, 'сек.')
    start = datetime.datetime.now()
    nx.pagerank_numpy(graph)
    end = datetime.datetime.now()
    work_time = (end - start).total_seconds()
    print('Для numpy-реализации:', work_time, 'сек.')
```

In [265]:

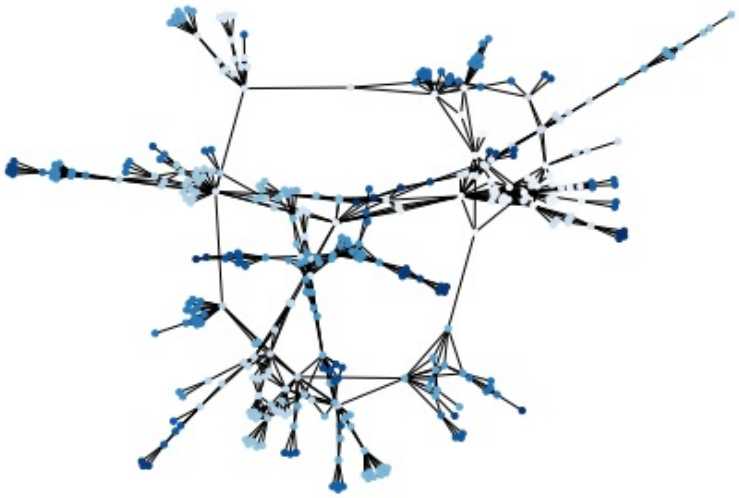
```
def most_valuable_pages(graph, tops_number = 10):

    p_r = pageranking(graph)

    nodes = []
    ranks = []
    for i in range(len(p_r)):
        nodes.append([])
        ranks.append([])
        for j in range(len(p_r[i])):
            nodes[i].append(p_r[i][j][0])
            ranks[i].append(p_r[i][j][1])

    nodes_classic = np.array(nodes[0][0:tops_number])
    ranks_classic = np.array(ranks[0][0:tops_number])
    lal = list(map(str, nodes_classic))
    bar_lab = [round(rank, 7) for rank in ranks_classic]
    title = "Топ " + str(tops_number) + " страниц с наибольшим значением ранга"
```

```
    #clist = [(0, "red"), (0.125, "red"), (0.25, "orange"), (0.5, "green"),
         #(0.7, "green"), (0.75, "blue"), (1, "blue")]
    #rvb = mcolors.LinearSegmentedColormap.from_list("", clist)

    my_colors = 'rgbkymc'
    f, ax = plt.subplots(figsize = (12, 8))
    br = plt.bar(nodes_classic, height = ranks_classic, width = 5, alpha = 0.5, color =
my_colors, tick_label = lal)
    title = "Топ " + str(tops_number) + " страниц с наибольшим значением ранга"
    plt.xlabel("Номера страниц")
    plt.ylabel("Рейтинги страниц")
    plt.title(title)
    def autolabel(rects):
        for idx, rect in enumerate(br):
            height = rect.get_height()
            ax.text(rect.get_x() + rect.get_width()/2., 0.5*height,
                    bar_lab[idx],
                    ha = 'center', va = 'bottom', rotation = 90)

    autolabel(br)

    plt.ylim(0,0.02)
    plt.show()
```

In [13]:

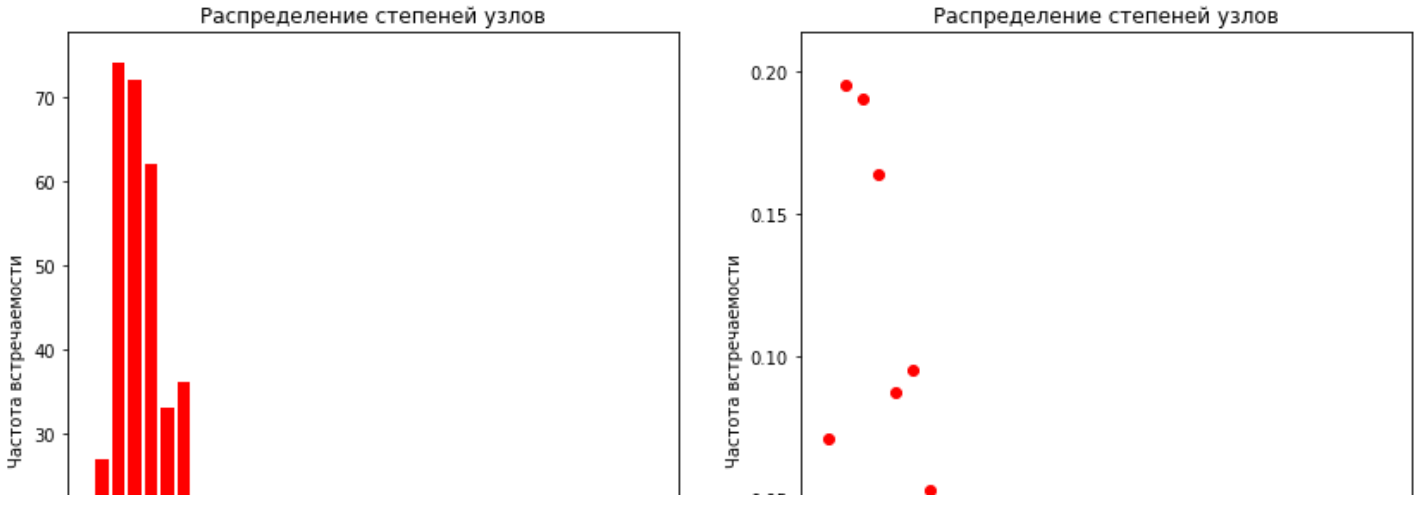
```
G1 = view_graph("src.edgelist")
```

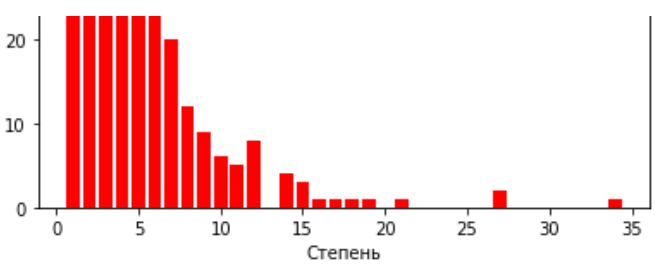
```
Количество узлов в сети: 379
Количество связей в сети: 914
Количество изолированных узлов в сети: 0
```

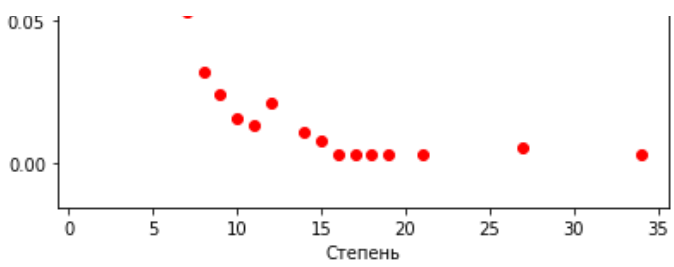


In [268]:

```
draw_nodes_distr(G1)
```





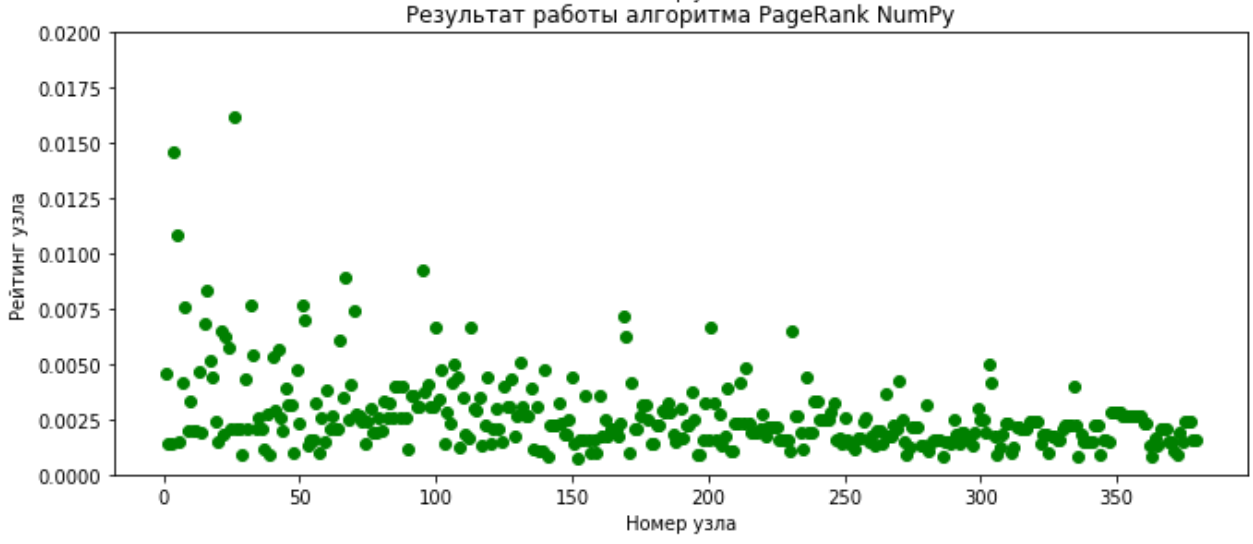


In [154]:

```
rank = pageranking(G1)
weights = vizualize_pagerank(rank)
#len(weights[0])
```

Out[154]:

3



In [168]:

```
compare_pageranks(weights)
```

```
MAE для PageRank и PageRank Scipy: 4.711559308927772e-19
MSE для PageRank и PageRank Scipy: 5.261810464828524e-37

MAE для PageRank и PageRank Numpy: 3.4070607923808104e-06
MSE для PageRank и PageRank Numpy: 2.559679564147556e-11

MAE для PageRank SciPy и PageRank Numpy: 3.4070607923808887e-06
MSE для PageRank SciPy и PageRank Numpy: 2.5596795641475834e-11
```

In [271]:

```
algorithm_timer(G1)
```

```
Время выполнения PageRank для графа с 379 вершинами и 914 ребрами:
Для реализации со степенной итерацией: 0.09278 сек.
Для scipy-реализации: 0.003991 сек.
Для numpy-реализации: 0.051451 сек.
```

In [266]:

```
most_valuable_pages(G1)
```