Санкт–Петербургский государственный университет

Криворучко Денис Игоревич

Отчёт по домашней работе 2
Графовые алгоритмы на основе линейной алгебры

Санкт-Петербург
2021 г.

Содержание

Глава 1. Введение

В настоящее время все чаще приходится иметь дело с данными, которые представлены в виде графа. Однако решения задач по взаимодествию с подобного рода информацией весьма сложно оптимизировать, поскольку она имеет сложную нелинейную структуру. В то же время, алгоритмы обработки графов можно свести к линейной алгебре и базовым операциям над матрицами и векторами, что может их существенно ускорить. А модель хранения данных в виде разреженной матрицы позволяет минимизировать значение расходуемой памяти.

В данной работе проведен анализ производительности некоторых графовых алгоритмов на основе линейной алгебры и предствлено сравнение их реализаций с использованием библиотек PyGraphBlas [1], SciPy [2] и IGraph [3]. Для этого были выбраны поиск в ширину, подсчет треугольников и поиск кратчайших путей из одной вершины.

Глава 2. Детали реализации

2.1 Обход в ширину

Первым параметром алгоритма обхода в ширину является информация о графе. Однако, в разных реализациях она представляется поразному. В PyGraphBlas граф представляется в виде списка из трех списков, в которых для каждого ребра содержится индекс вершины-начала (в первом списке), индекс вершины-конца (во втором списке), а также его вес (в третьем списке). В библиотеках SciPy и IGraph данные передаются в виде матрицы смежности, которая представлена списком списков. Элементом с номером m в списке с номером n, является вес ребра из вершины n в вершину m. Второй параметр во всех трех реализациях — индекс стартовой вершины.

Возвращаемым значением в реализации с использованием PyGraphBlas является вектор, координаты которого — уровни, на которых были достигнуты вершины. В реализации с SciPy и IGraph результат — коллекция, элементы которой — номера соответствующих вершин в порядке обхода в

ширину. Однако, в первом случае данная коллекция представляет из себя ndarray, а во втором — список.

Перед началом работы, каждая из реализаций преобразует входные данные в нужный формат. Для PyGraphBlas и SciPy это разреженные матрицы, для IGraph — матрица смежности. Далее в реализациях под SciPy и IGraph, используются библиотечные методы (breadth_first_order и bfs соответсвенно). В случае с PyGraphBlas на каждом шаге постепенно формируется результирующий вектор. Делается это следующим образом: на каждом уровне обхода, путем умножения результирующего вектора на матрицу графа, получается вектор вершин, достигнутых на данном уровне и в результирующем векторе значение уровня сопоставляется элементам, соответсвующим этим вершинам.

2.2 Подсчет треугольников

Единственный параметр подсчета треугольников совпадает с первым параметром поиска в ширину. Однако, здесь есть ограничение — граф должен быть невзвешенным. Это значит, что веса всех ребер должны быть равны 1.

Возвращаемым значением во всех трех реализациях является единственное число — количество треугольников в графе.

Перед началом работы производятся те же преобразования данных, что и в поиск в ширину. Далее в реализации с IGraph вызывается стандартный метод cliques. В реализациях под PyGraphBlas и SciPy сначала строится нижняя треугольная матрица T для матрицы смежности, затем суммируются все элементы этой матрицы, которые равны единице и в T, и в T^2.

2.3 Поиск кратчайших путей из одной вершины

Входные параметры алгоритма поиска кратчайших путей из одной вершины такие же, как и у поиска в ширину.

Тип результата также совпадает со случаем поиска в ширину, однако каждый элемент полученных колекций содержит длину кратчайшего пути

до соответветствующей вершины из начальной.

Перед началом работы, происходят те же преобразования, как и в предыдущих алгоритмах. Далее в реализации с использованием PyGraphBlas, на каждом шаге алгоритма формируется результирующий вектор. Делается это путем применения операции MinPlus к текущему состоянию результирующего вектора и матрице графа. Результат работы алгоритма — вектор, координаты которого — длины кратчайших путей из стартовой вершины до соответствующей вершины. Вершине с номером n, соответствует значение n-ой координаты. На самом деле, вышеописанный алгоритм — это алгоритм Беллмана-Форда. В библиотеке SciPy есть стандартный метод bellman_ford, который используется в реализации для данного модуля и работает подобным образом. В свою очередь, библиотека IGraph содержит встроенный метод shortest_paths, который, на самом деле, в зависимости от входных данных, использует алгоритм Беллмана-Форда, Дейкстры или Джонсона. Данный метод используется в реализации под IGraph.

Глава 3. Постановка экспериментов и измерение производительности

3.1 Эксперименты

Измерения производились на компьтере с процессором INTEL Core i5 8th Gen (2.30GHz), 16 гб оперативной памяти и операционной системой Ubuntu 20.04. Для эксперимента использовался датасет p2p-Gnutella06 с сайта [4].

Первый эксперимент был поставлен следующим образом: в каждой реализации алгоритмы обхода в ширину и поиска кратчайших путей из одной вершины запускались для 100 случайных вершин на 2 подграфах — 100 вершин, 1000 вершин и на всем графе — 8717 вершин. Вес каждого ребра генерировался случайно отдельно для каждого эксперимента. Каждый опыт проводился 3 раза. При каждом запуске производился замер, а потом из всех запусков вычислялось среднее, максимальное и минимальное затраченное время. Алгоритм подсчета треугольников запускался на

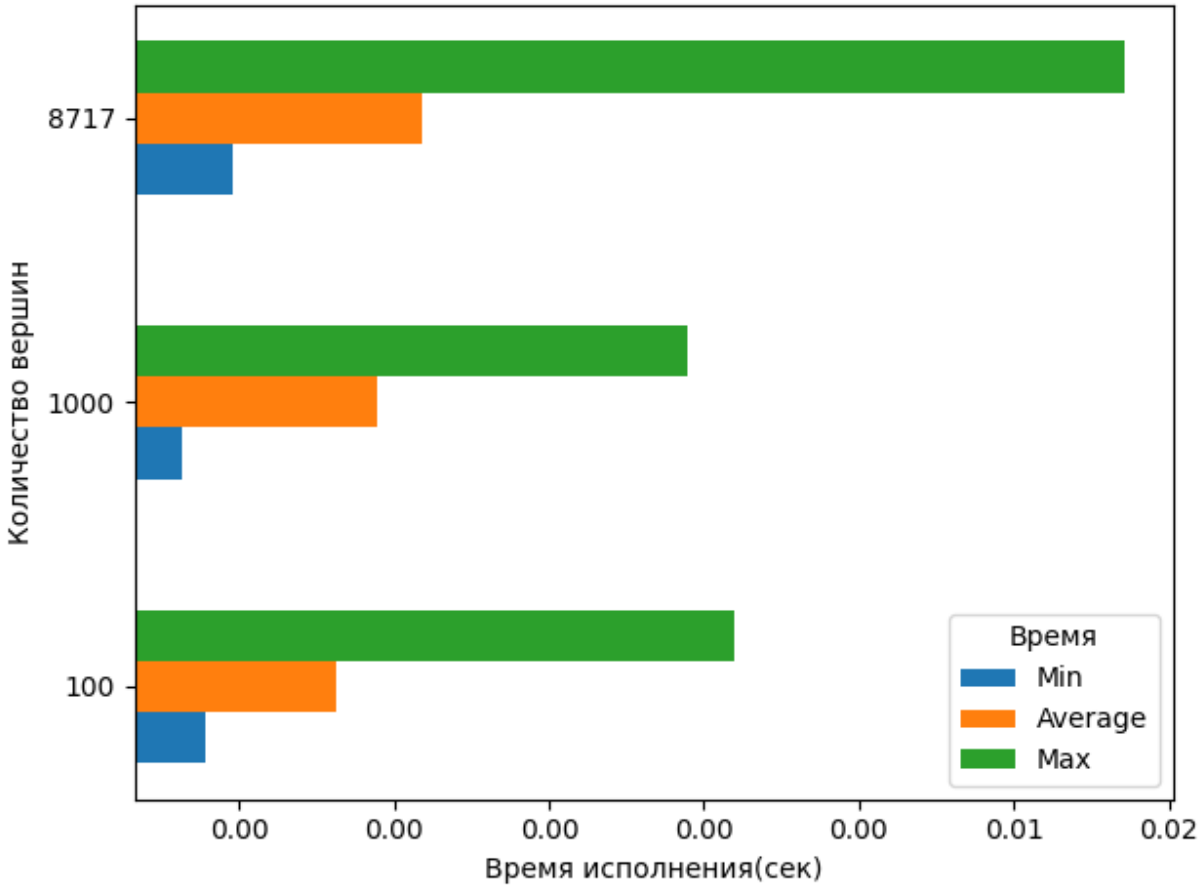

Рис. 1: PyGraphBlas: BFS.

каждом из вышеописанных подграфов по 10 раз, однако тут генерируемый вес каждого ребра может быть равен 1 (есть ребро) или 0 (нет ребра). В данном алгоритме проводился только замер затраченного времени для каждого из тестовых подграфов. Стоит отметить, что замер производился без учета начального преобразования входных данных к нужному формату. Результаты проведения опытов представлены на диаграмах 1, 2, 3, 4, 5, 6, 7, 8, 9

Второй эксперимент проводился на значительно большем графе. Он был взят из датасета Youtube.com с сайта [4]. Данный граф имеет 1134890 вершин, что существенно превосходит граф из предыдущего опыта. Эксперимент проводился так: алгоритмы обхода в ширину и поиска кратчайших путей запускались из 10 случайных вершин и вычислялось максимальное, минимальное и среднее время работы из этих замеров. Алгоритм подсчета треугольников запускался на данном графе трижды. Поскольку в отличие

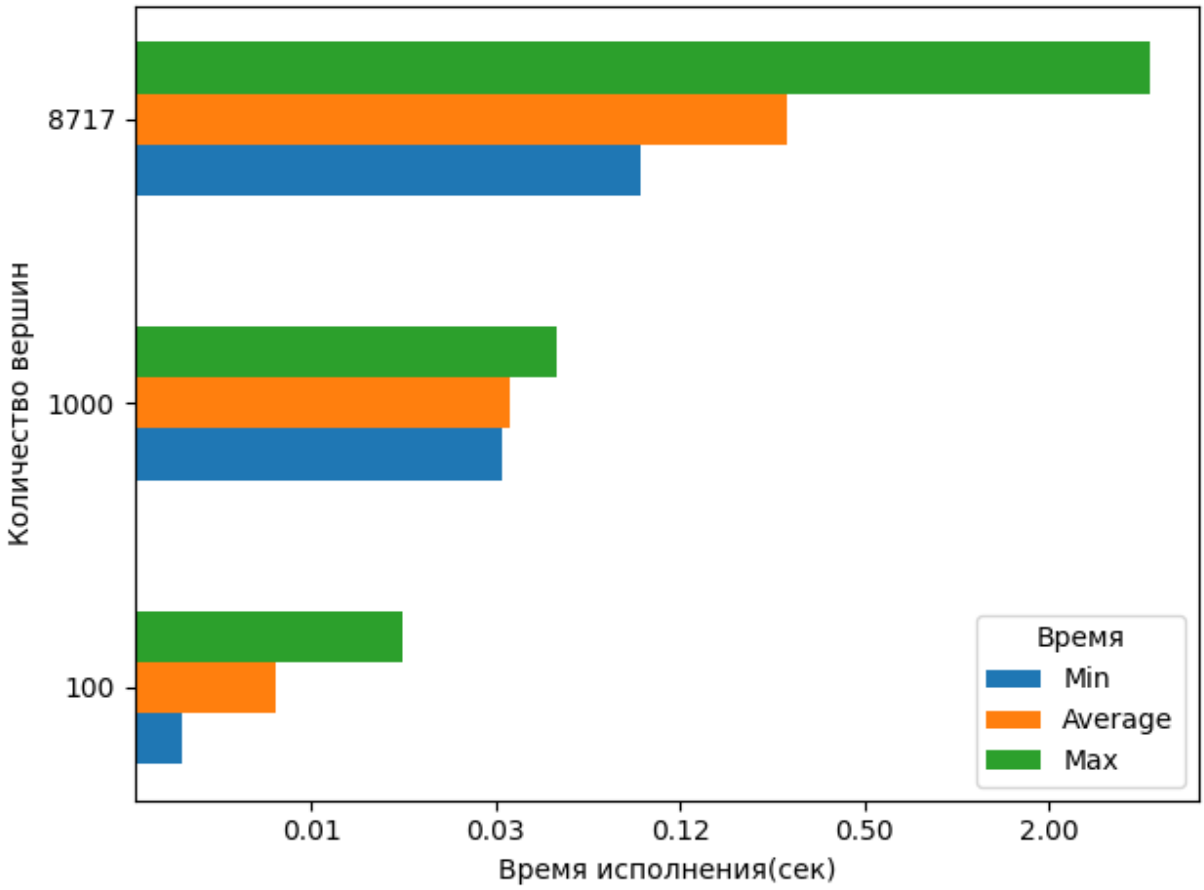

Рис. 2: PyGraphBlas: SSSP.

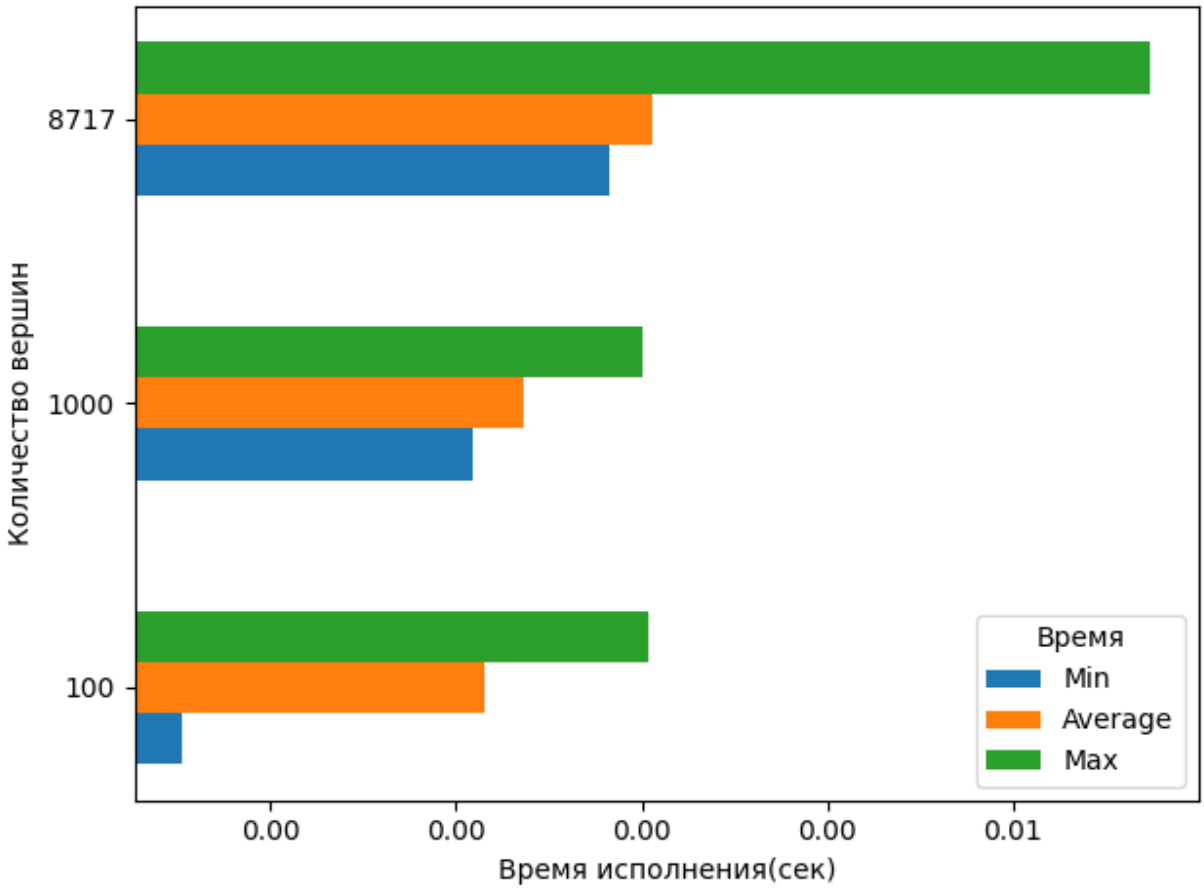

Рис. 3: PyGraphBlas: Triangles count.

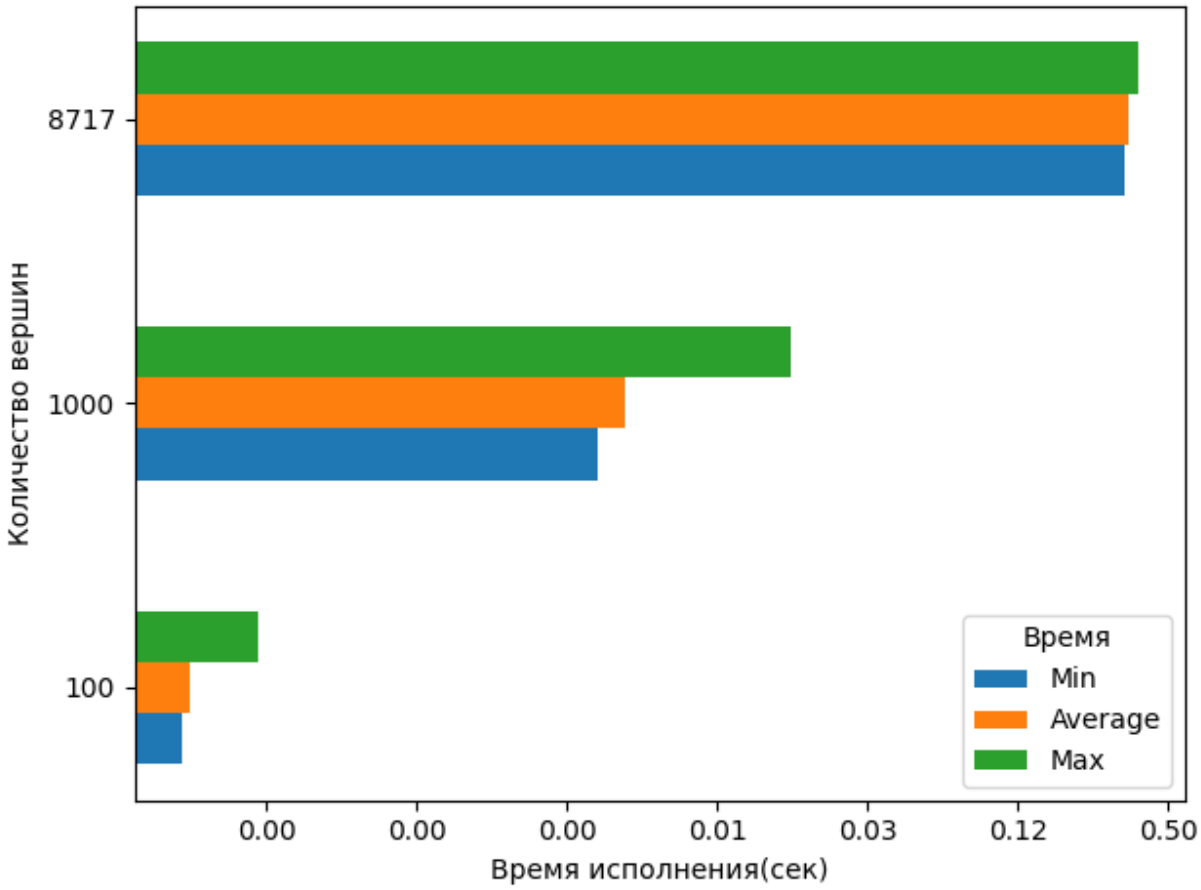

Рис. 4: SciPy: BFS.

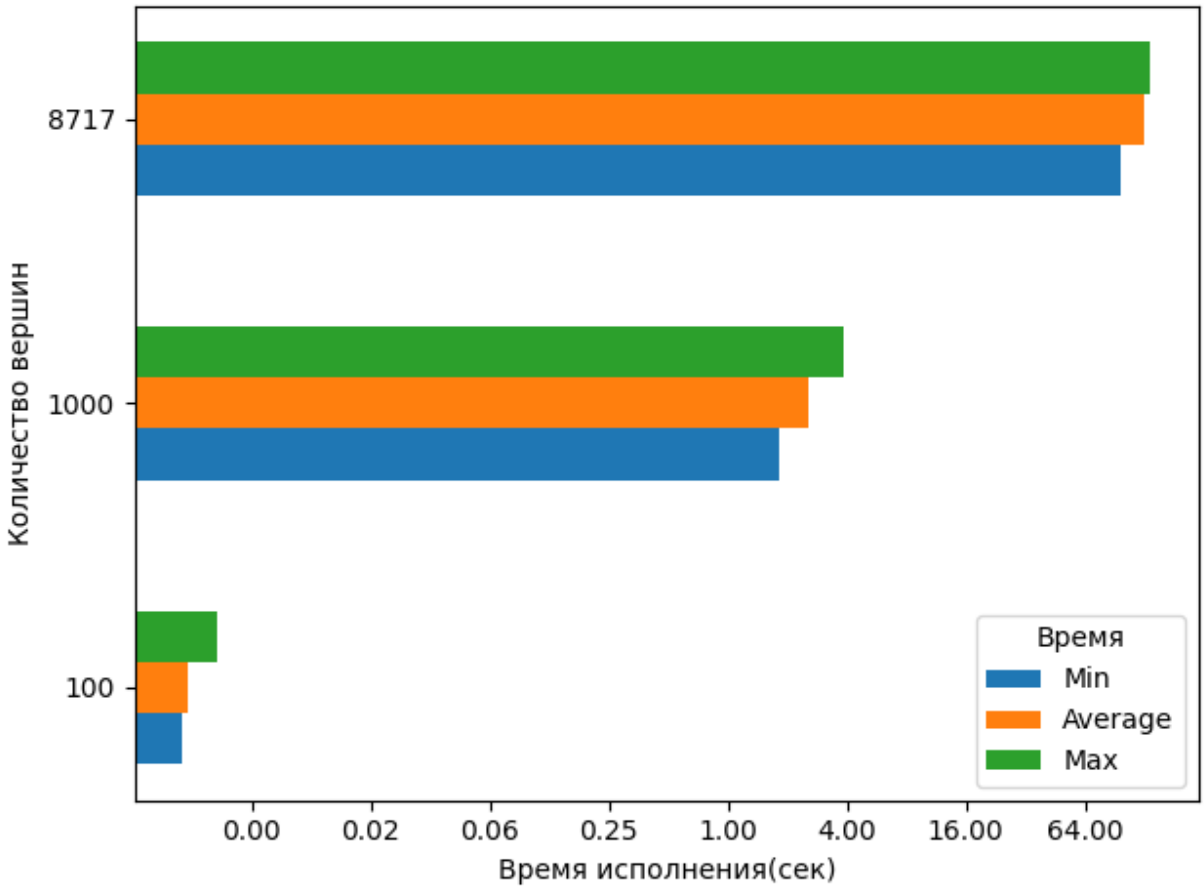

Рис. 5: SciPy: SSSP.

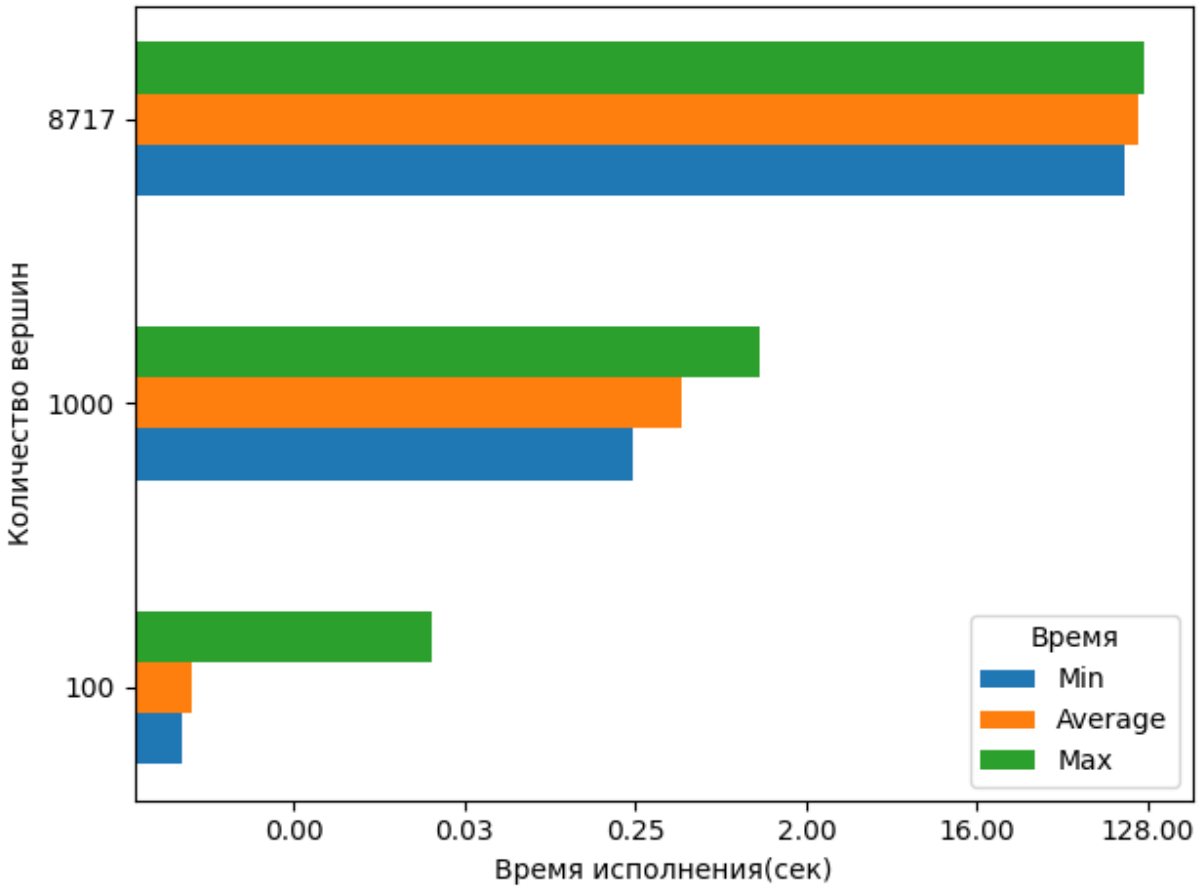

Рис. 6: SciPy: Triangles count.

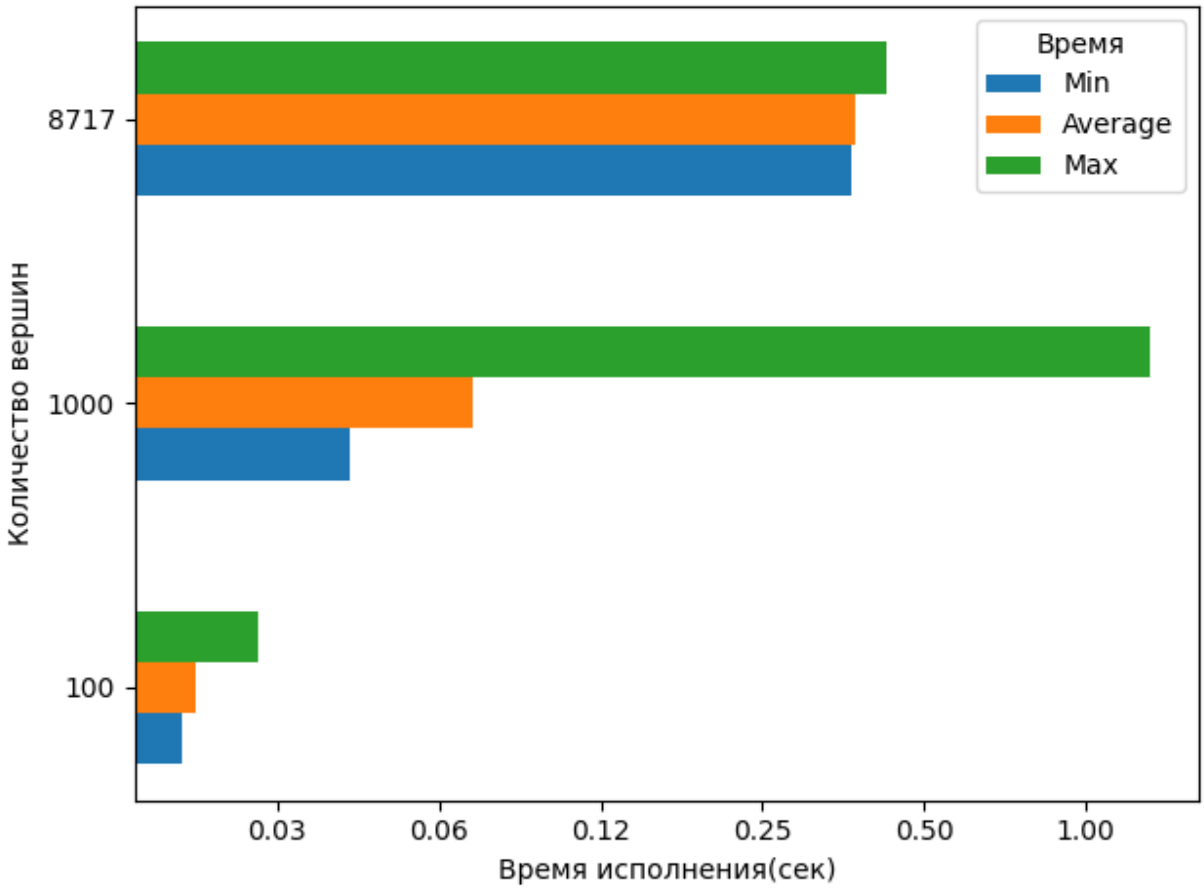

Рис. 7: IGraph: BFS.

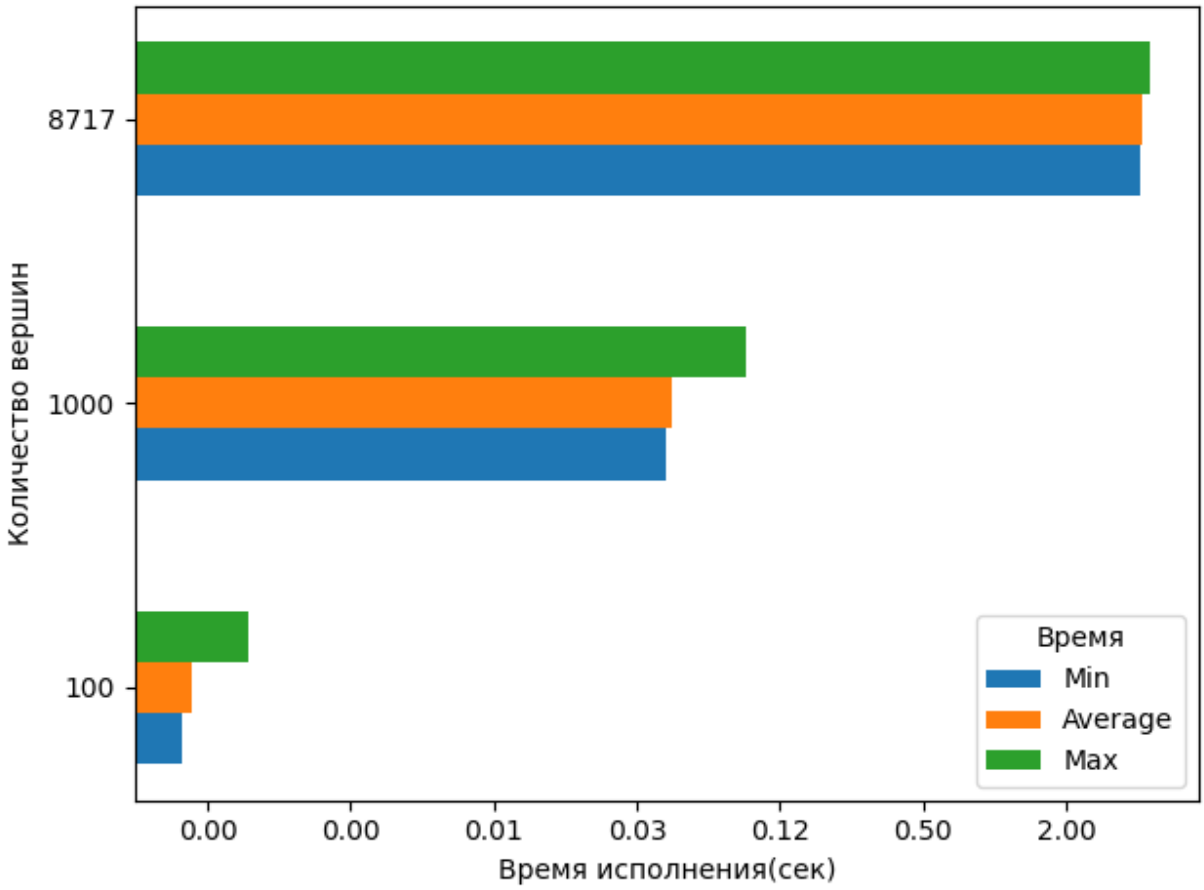

Рис. 8: IGraph: SSSP.

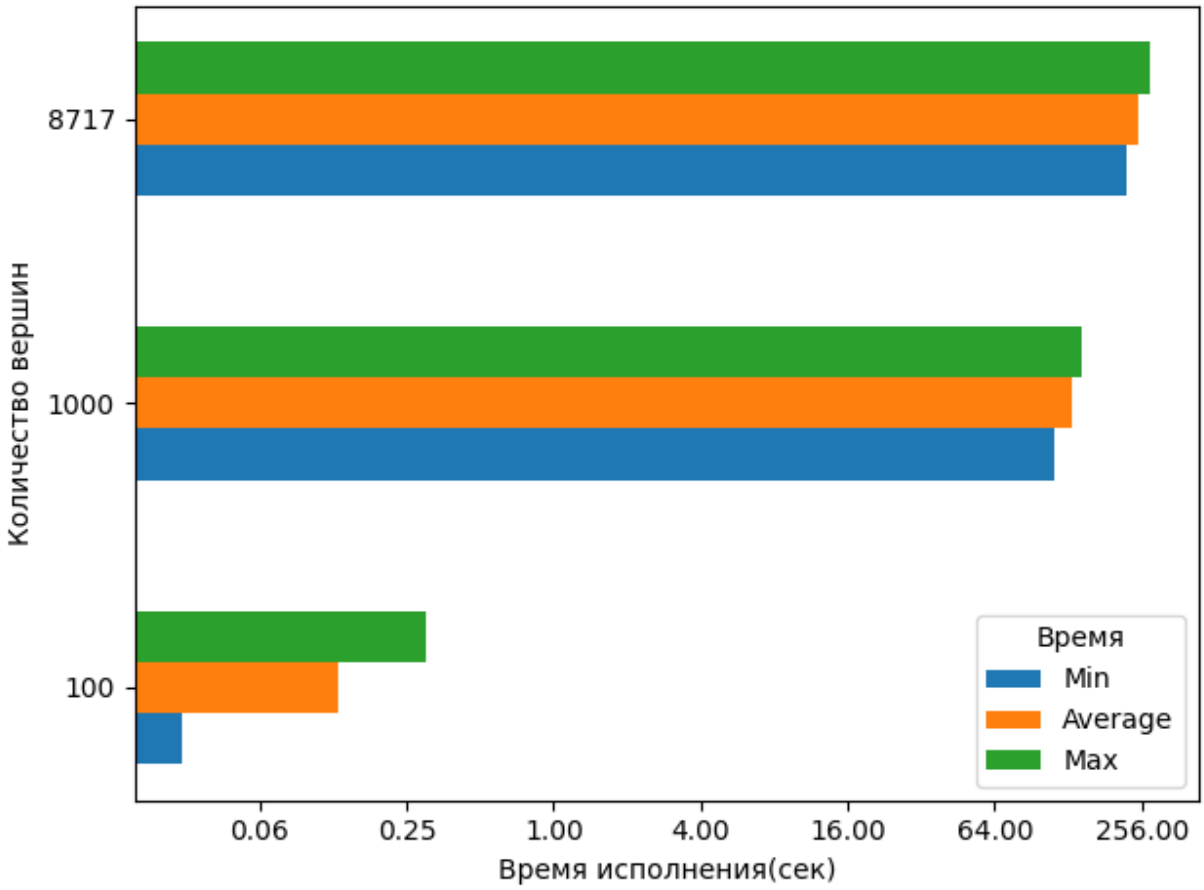

Рис. 9: IGraph: Triangles count.

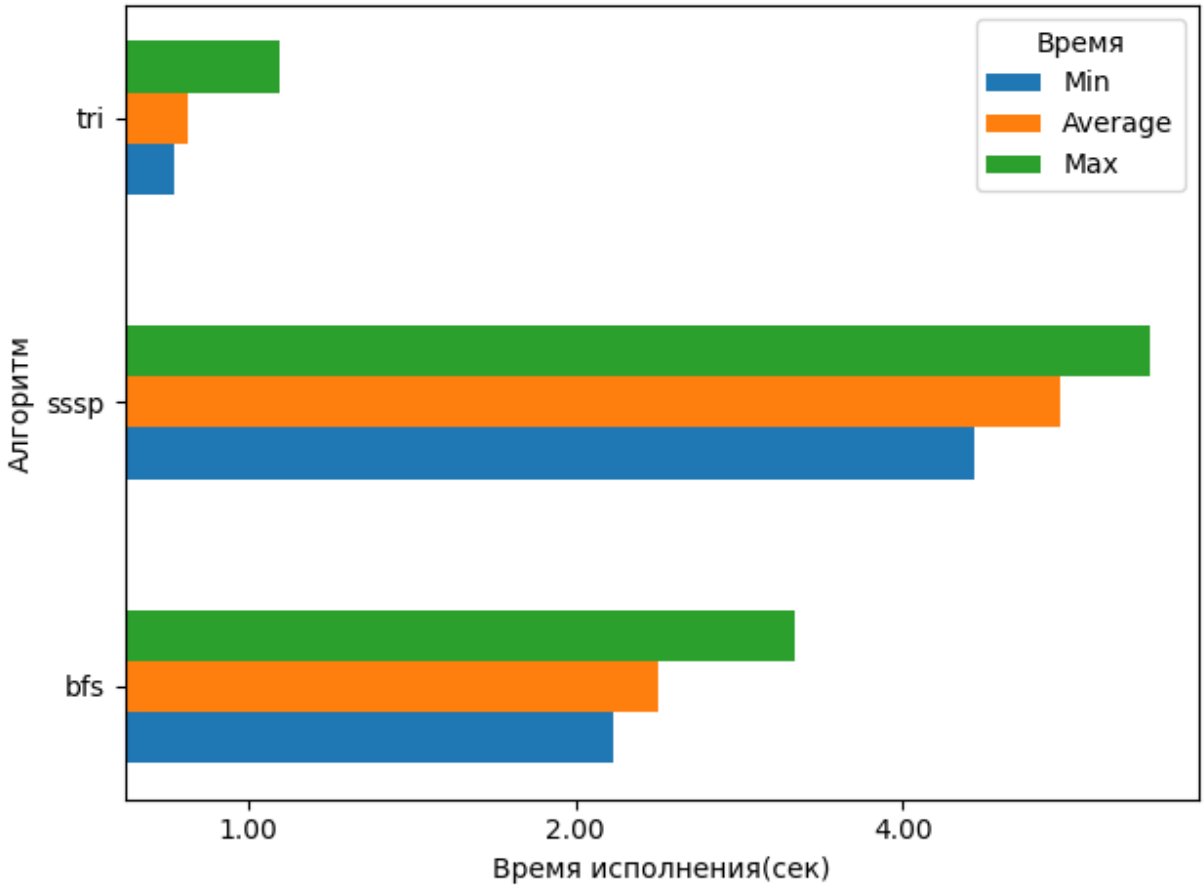

Рис. 10: PyGraphBlas: Youtube dataset.

от предыдущего, данный граф изначально является ненаправленным, то здесь при подсчете треугольников достаточно все ребра, не равные нулю, считать равными единице. Затем также вычислялось минимальное, максимальное и среднее время работы алгоритма из этих замеров. Результаты представлены на диаграмах 10, 11, 12.

3.2 Анализ

Почти на всех входных данных библиотека IGraph показывает значительно меньшую производительность в сравнении с двумя другими. Это обусловлено тем, что данная библиотека не направлена на значительные оптимизации при работе с графами и использует стандартные средства. Что же касается PyGraphBlas и SciPy, это скорее библиотеки для инженерно-математических задач, в связи с чем большое внимание тут уделено оптимизации работы алгоритмов.

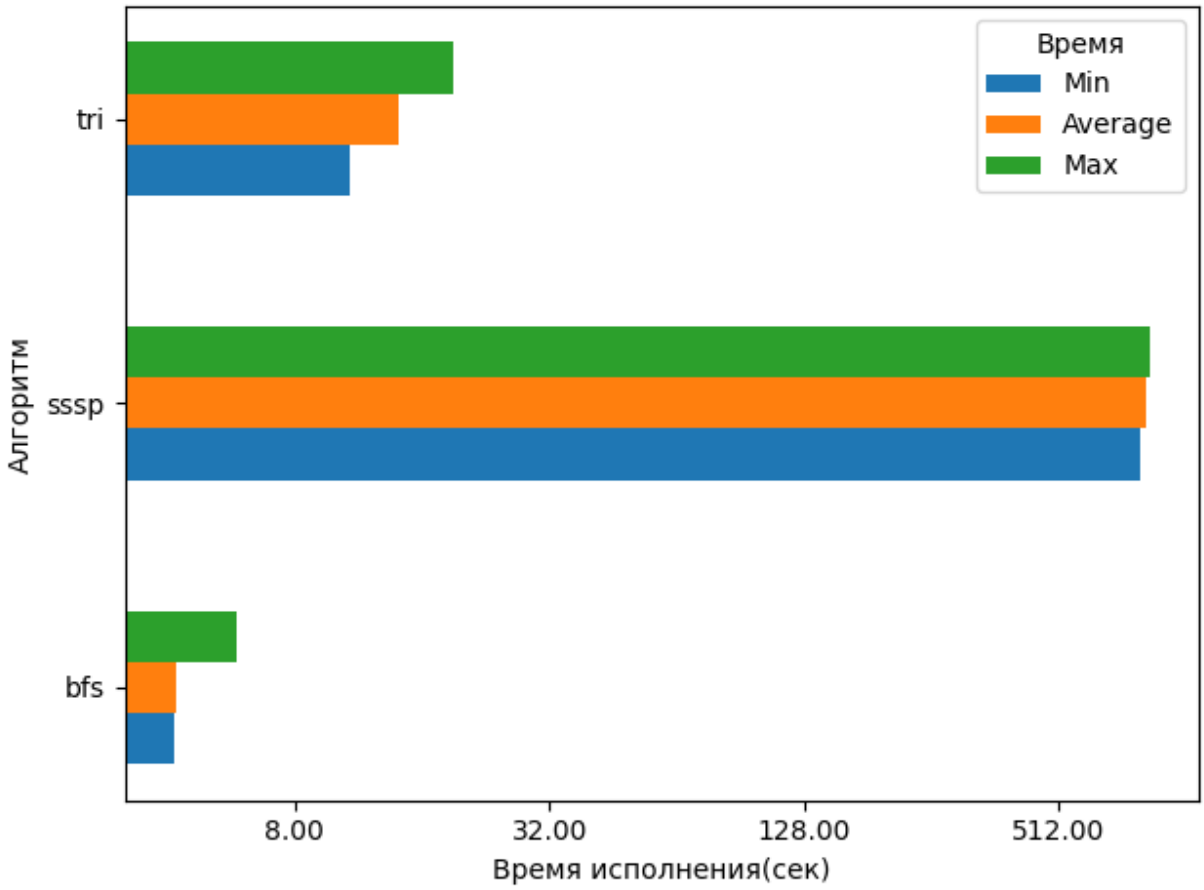

Рис. 11: SciPy: Youtube dataset.

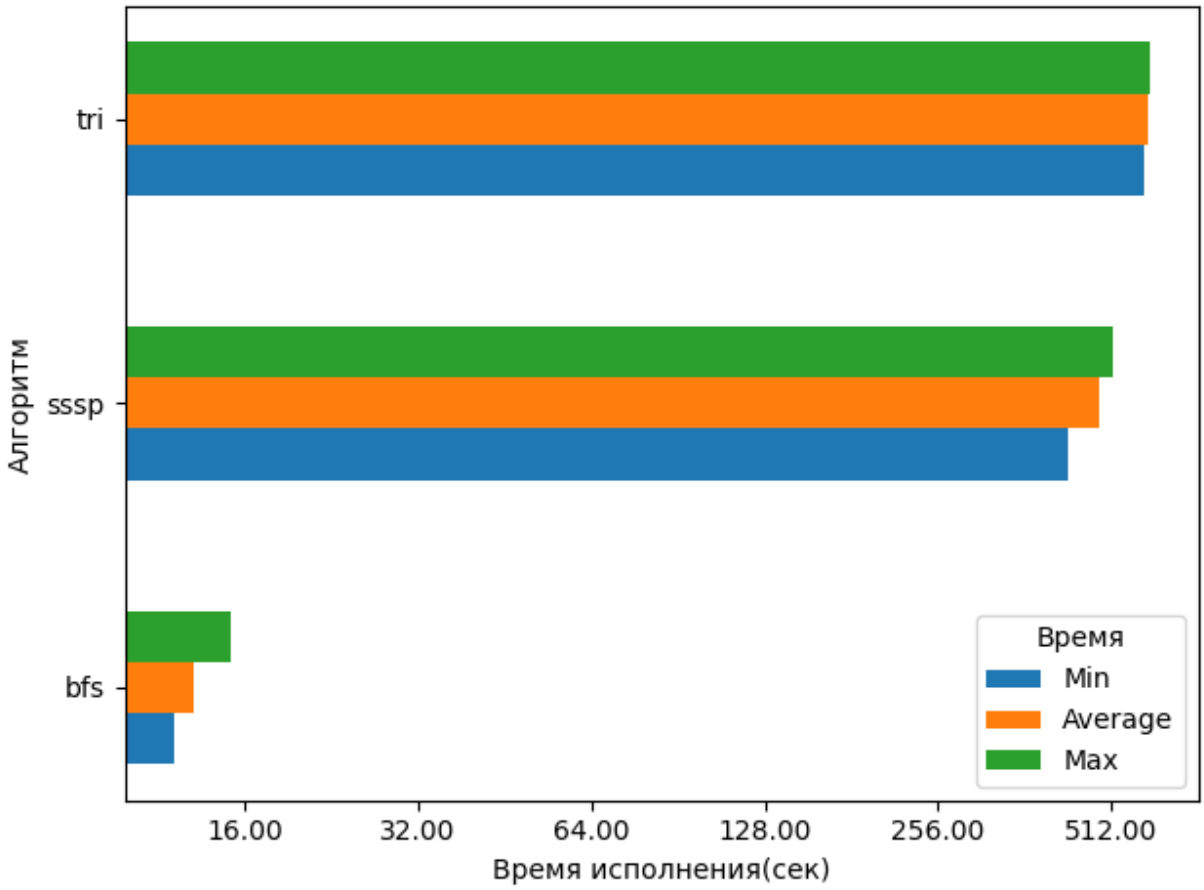

Рис. 12: IGraph: Youtube dataset.

Реализация под PyGraphBlas уступает реализации под SciPy только на малых данных и только на алгоритмах BFS и SSSP. Однако, в остальных случаях PyGraphBlas показывает лучший результат, и, с увеличением входных данных, разница становится все более заметной. Это обуславливается тем, что обе реализации используют матричные операции, на которые в PyGraphBlas сделан основной упор.

Также стоит отметить, что с ростом входных данных наименьший рост времени работы алгоритма наблюдается в реализации с использованием библиотеки PyGraphBlas. Особенно это заметно в случае с подсчетом треугольников. Здесь при увеличении количества вершин почти в сто раз, наблюдается увеличение времени всего в два раза, в то время, как реализации с SciPy и IGraph работают более двух минут. Это также связано с высокой оптимизацией матричных операций в PyGraphBlas.

Глава 4. Заключение

Наименее эффективной реализацией стала реализация с использованием IGraph. Это ожидаемо, потому что данная библиотека предназначена для других целей. В ней содержится множество методов для анализа социальных сетей, но платой за универсальность в этом случае, является скорость.

На малых данных самой эффективной оказалась библиотека SciPy. Однако скорость работы алгоритма сильно растет при увеличении входных данных, а именно размеров графа, который нужно обработать.

Что же касается работы с графом, у которого больше 10000 вершин, тут существенное преимущество за PyGraphBlas. Благодаря матричным примитивам, которые высоко оптимизированы в данном модуле, скорость работы при существенном увеличении графа, увеличивается несущественно и даже на графе с 1000 вершин, реализации с использованием PyGraphBlas уже намного быстрее, чем в случае со SciPy или IGraph.

Список литературы

[1] PyGraphBlas documentation. URL: https://graphegon.github.io/pygraphblas/pygr

[2] SciPy documentation. URL: https://www.scipy.org/docs.html

[3] IGraph documentation. URL: https://igraph.org/python/doc/tutorial/

[4] Dataset source. URL: https://snap.stanford.edu/data/