«Сравнительный анализ алгоритмов BFS, DFS,Дейкстры для поиска блоков и шарниров в графе»

Содержание

Введение

Распространённый в различных областях науки метод графического изображения процессов, зависимостей, структур и т.п. с помощью точек и соединяющих их линий позволил создать специфические и удобные для специалистов каждой отрасли графические схемы: электрические схемы, схемы авиалиний и т.д.

Теория графов – это простой и мощный инструмент для решения вопросов связанных с широким кругом проблем.

Эта теория может применяться к любым схематическим представлениям процессов и служит математическим инструментом для их исследования. В некоторых случаях, использование математического аппарата теории графов позволяет сделать некоторые выводы и упрощения, которые не так очевидны в обычных схемах.

Граф – это совокупность узлов и соединяющих их рёбер. Он строится для того, чтобы отобразить отношения на множествах.

Это дает представление о структуре исследуемого объекта, устанавливает отношение между его отдельными узлами, и, если ввести соответствующую характеристику веса для ребер, то можно получить количественную оценку связей.

Операции над графами не ограничиваются анализом технологических схем, а позволяют использовать статистические данные для выявления значимых факторов, которые влияют на процесс, для определения минимального набора критериев оптимизации и для получения другой информации.

Приведем пример, который показывает, как задача нахождения центра графа может возникнуть в приложении теории графов. Предположим, что граф представляет собой сеть дорог: вершины - это населенные пункты, а ребра — дороги между ними. Необходимо оптимально расположить пожарные части, больницы, магазины и т.д. Главный критерий оптимальности – это минимизация расстояний от пунктов обслуживания до наиболее удаленных от них населенных пунктов. Таким образом, исходя из определения центра графа, пункты обслуживания необходимо размещать именно в тех вершинах, которые являются центром.

Развитие теории графов в основном связано с большим количеством различных приложений.

Графы используются при проектировании интегральных схем и схем управления, при исследовании блок-схем программ, в экономике, химии, биологии и т.д.

В виде графов можно, например, интерпретировать схемы дорог и метрополитена, электрических цепей и молекул химических соединений ,а также связи между людьми.

Графы: основные понятия

Обыкновенным графом называется пара G = (V, E), где V – конечное множество, E – множество неупорядоченных пар различных элементов из V. Элементы множества V называются вершинами графа, элементы множества E – его ребрами.

Ребро e соединяющее вершину a с вершиной b и если пара (a,b) считается упорядоченной, то это ребро называется ориентированным, вершина a – его началом, вершина b – его концом. Если эта пара считается неупорядоченной, то ребро называется неориентированным, а его обе вершины – концы.

Граф, состоящий только из ориентированных ребер, называют **ориентированным графом**, если только из неориентированных ребер – **неориентированный**.

Граф называется **связным**, если в нем для любых двух вершин имеется маршрут, соединяющий эти вершины.

Маршрут в графе - это последовательность вершин $x_1, x_2, \ldots, x_k$ такая, что для всякого i = 1, 2, . . . , k-1 вершины x_i и x_{i+1} соединены ребром.

Путь — маршрут, в котором все ребра различны.

Цикл — это замкнутый путь.

Граф называется **ациклическим**, если в нем нет циклов.

Деревом называется связный граф, не имеющий циклов.

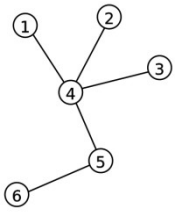


Рис. 1. Дерево

Во всяком дереве, в котором больше одной вершины, имеется не менее двух вершин степени 1. Такие вершины называют листьями.

Если G – **дерево**, то

1) число ребер в нем на 1 меньше числа вершин;

2) в G любая пара вершин соединена единственным путем;

3) при добавлении к G любого нового ребра образуется цикл;

4) при удалении из G любого ребра он превращается в несвязный граф.

Ветвь к вершине x дерева T – это максимальный подграф, содержащий x в качестве висячей вершины. “Вес” w(x) вершины x – наибольший размер ее ветвей (подграфа).

$w(x) = \max\{|T_1|, |T_2|, \ldots, |T_k|\}$, где k = deg(x) – степень вершины x, T_k – подграф вершины x.

Число вершин, смежных с вершиной x, называется степенью вершины x и обозначается

deg(x).

Центроид (или центр масс) дерева T – множество вершин с наименьшим весом:

Centroid(T) = { x ∈ V(T) | w(x) ≤ w(y), $\forall y \in V(T)$}.

Вес любого листа дерева равен размеру дерева.

Шарнир (**точка сочленения**) в графе – вершина, при удалении которой увеличивается число компонент связности.

Блоком графа называется подграф B, удовлетворяющий одному из трех условий:

а) B состоит из одной изолированной вершины графа G (такой блок называется тривиальным);

б) B порождается единственным ребром, которое является перешейком в G;

в) B является максимальным двусвязным подграфом графа G

Метрические характеристики графа:

Расстояние от данной вершины a до наиболее удаленной от нее вершины называется **эксцентриситетом вершины** и обозначается через ecc(a).

ecc(a) = $\max_{x \in VG} d(a, x)$

Величина наименьшего эксцентриситета называется **радиусом** графа.

rad(G) = $\min_{x \in VG} \max_{y \in VG} d(x, y)$

Вершину с наименьшим эксцентриситетом называют **центральной**. Множество всех центральных вершин называется центром графа.

Center(G) = { a∈VG: ecc(a) = rad(G) }

Постановка задачи

1. Изучение алгоритмов обхода произвольного связного графа в ширину и глубину для построения остовного дерева и их реализации на разных структурах хранения графов: матрица смежности и список смежности.
2. Изучение и реализация алгоритмов поиска центра и центроида в дереве.
3. Изучение и реализация алгоритма поиска кратчайшего пути во взвешенном графе: алгоритм Дейкстры.
4. Изучение и реализация алгоритмов поиска шарниров, блоков в графе и построение BC-дерева.
5. Проведение анализа работы алгоритмов при различных способах хранения и насыщенности графов.

1. Структура хранения графа

В компьютерных науках и информационных технологиях граф определяется как нелинейная структура данных.

Линейные структуры данных характеризуются тем, что они связывают элементы отношениями "простого соседства". Линейными структурами данных являются, например, массивы, таблицы, списки, очереди, стеки, строки.

Напротив, нелинейные структуры данных — это те, в которых элементы располагаются на разных уровнях иерархии и разделяются на три вида: исходные, порождённые и подобные.

Граф - нелинейная структура данных.

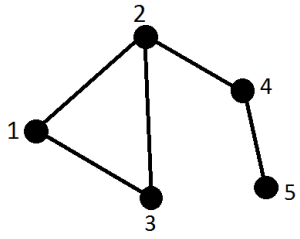
1.1 Матрица смежности

Пусть G – граф с n вершинами, причем VG = {1, 2, …, n}. Квадратная матрица A порядка n, в которой элемент A_{ij}, стоящий на пересечении строки с номером i и столбца с номером j, определяется следующим образом:

$$A = \begin{cases} 1, \text{если } i, j \in EG \\ 0, \text{если } i, j \notin EG \end{cases}$$

Она называется **матрицей смежности** графа.

Для обыкновенного графа она обладает двумя особенностями: из-за отсутствия петель на главной диагонали стоят нули, а так как граф неориентированный, то матрица симметрична относительно главной диагонали. Обратно, каждой квадратной матрице порядка n, составленной из нулей и единиц и обладающей двумя указанными свойствами, соответствует обыкновенный граф с множеством вершин {1, 2, …, n}.



Матрица смежности:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Рис. 2 Граф

Преимущества матрицы смежности в том, что можно получить за один шаг ответ на вопрос: существует ли ребро между двумя вершинами. А также некоторые свойства графов, можно изучить, используя алгебру матриц. (Например, изоморфизм).

Один из главных недостатков использования матрицы смежности - требуется большой объем памяти.

1.2 Список смежности

Пусть G – граф, v_1, v_2, ... , v_n – множество всех его вершин.

Списками смежности графа G называется массив из n списков, в котором, для любого i = 1,2,...,n, i-й список содержит в точности все вершины, смежные с v_i.

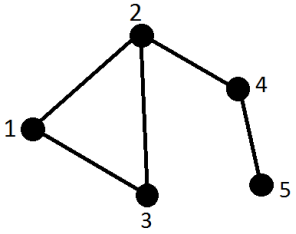


Рис.3 Граф

Список смежности:

1	2	3	4	5
2	1	1	2	4
3	3	2	5	

Списки смежности очень эффективно используют память, но для выяснения вопроса смежности двух вершин, может потребоваться полностью просмотреть список одной из них.

Если граф насыщенный, то количество ребер близко к №, что увеличит время работы с такой структурой хранения.

Сравнение с матрицей смежности

Операция	Список смежности	Матрица смежности
Проверка на наличие ребра (x,y)	O (|E|)	O (1)
Определение степени вершины	O (1)	O (|V|)
Использование памяти для разреженных графов	O(|V| + |E|)	O (|V|2)
Обход графа	O(|V| + |E|)	O (|V|2)

2.Описание алгоритмов

2.1 Алгоритм поиска в ширину (BFS)

Поиск в ширину (англ. Breadth-First search) — метод обхода графа и поиска пути в графе. Идея поиска в ширину заключается в последовательном просмотре отдельных уровней графа: сначала посещается корень – произвольно выбранный узел, затем – все смежные вершины данного узла, после этого посещаются потомки потомков и т.д. Вершины просматриваются в порядке возрастания их расстояния от корня.

На вход алгоритма подается граф G и номер стартовой вершины s.

Для реализации алгоритма потребуются очередь Q и множество посещенных вершин: булевский массив used[].

V(x) – множество всех вершин смежных с вершиной x.

Алгоритм

BFS (graf G, top s)

1. Добавляем вершину s в очередь: s => Q
2. Помечаем вершину s как посещенную: used[s] = true, а для остальных вершин used[] = false.
3. **Пока** Q не пуста:
4. Извлекаем текущую вершину x из очереди Q: x <= Q
5. Просматриваем все смежные вершины $y \in V(x)$ с вершиной x:

Если смежная вершина $y \in V(x)$ не посещенная: used[y] = false
Тогда посещаем вершину y: used[y] = true
Добавляем вершину y в очередь: y => Q

Свойства алгоритма BFS:

1 Алгоритм BFS заканчивает работу после конечного числа шагов.

Число повторений цикла while не превосходит числа вершин.

2. Общее время работы для матрицы смежности оценивается в O ($|V^2|$), а для списка смежности – O (|V| + |E|)
3. Теорема о BFS-дереве. Любое BFS-дерево является деревом кратчайших путей.

2.2 Алгоритм поиска в глубину (DFS)

Поиск в глубину (англ. Depth-First Search) — один из методов обхода графа.

Идея метода - посещается стартовая вершина, далее необходимо продвигаться вдоль ребер графа, до попадания в тупик. Вершина графа является тупиком, если все смежные с ней вершины уже посещены. Если попали в тупик , то необходимо вернуться назад вдоль пройденного пути, до тех пор пока не будет найдена вершина, у которой есть не посещенная вершина, а затем необходимо двигаться в этом новом направлении. Процесс оказывается завершенным при возвращении в начальную вершину, причем все смежные с ней вершины уже должны быть посещены.

На вход алгоритма подается граф G и номер стартовой вершины s.

Для реализации алгоритма потребуются стек S и множество посещенных вершин: булевский массив used[].

V(x) – множество всех вершин смежных с вершиной x.

Алгоритм

DFS (graf G, top s)

1. Стартовая вершина s помечается как посещенная: used[s] = true
2. Добавляем вершину s в стек: s => S
3. **Пока** S не пуст:
   3.1 Смотрим последнюю пришедшую вершину x в стеке S
   3.2 Просматриваем все смежные вершины $y \in V(x)$ с вершиной x:
   - **Если** есть хотя бы одна смежная вершина $y \in V(x)$, которая не посещена: used[y] = false
     Тогда посещаем вершину y: used[y] = true
     Добавляем вершину y в стек : y => S
     Переходим к пункту 3.1
   - **Если** таких вершин нет, то возвращаемся на шаг назад путем извлечения вершины x из стека S: x <= S

Свойства алгоритма DFS:

- Общее время работы для матрицы смежности оценивается в $O(|V^2|)$, а для списка смежности – $O(|V| + |E|)$
- Есть реализация рекурсивного поиска в глубину, но на больших графах такой алгоритм может сильно нагружать стек вызовов. Если есть риск переполнения стека, то используют не рекурсивный вариант поиска.

• *2.3 Алгоритм поиска центра в дереве*

Вершину с наименьшим эксцентриситетом называют **центральной**. По теореме Жордана: центр дерева состоит либо из одной вершины, либо из двух смежных вершин.

Center(G) = { a∈VG: ecc(a) = rad(G) }

Идея алгоритма поиска центра в дереве: удалить из дерева все листья одновременно, тогда эксцентриситет каждой вершины уменьшиться на 1. Продолжим эту операцию до достижения двух или одной вершин, тогда получим вершины(-у) с наименьшим эксцентриситетом, т.е центр.

На вход алгоритма подается дерево T.

Алгоритм поиска центра:

1. Просматриваем последовательно каждую строку, если в строке всего одна 1, то кладем в стек вершину, и отмечаем ее как посещенную.
2. Пока стек не будет пуст, достаем все вершины и удаляем их в дереве
3. Повторяем 1 – 2 до тех пор, пока количество не посещённых вершин не будет равно 1 или 2.
4. Не посещенные вершины и являются центральными.

Свойства для матрицы:

1. Время работы для матрицы смежности оценивается в O ($|V^3|$)

Свойства для списка:

1. Время работы для матрицы смежности оценивается в O (|V*E|)

2.4 Алгоритм поиска центроида в дереве

“Вес” w(x) вершины x – наибольший размер ее ветвей (подграфа).

Центроид (или центр масс) дерева T – множество вершин с наименьшим весом:

Centroid(T) = { x ∈ V(T) | w(x) ≤ w(y), $\forall y \in V(T)$ }.

Вес любого листа дерева равен размеру дерева без этого листа, т.е V – 1.

Теорема. Каждое дерево имеет центроид, состоящий из одной или двух смежных вершин.

Идея алгоритма поиска центра в дереве:

1 Способ – Рассматривая все вершины дерева, для каждой из них необходимо определить «вес», а затем выбрать вершину, у которой наименьший «вес», или две вершины, «вес» которых равен наименьшему. Это и будет центроид дерева.

2 Способ (более эффективный) – Выбираем произвольную вершину x дерева (лучше выбрать вершину, степень которой наибольшая, таким образом, можно сократить количество операций). Определяем «вес» данной вершины. Смотрим смежную вершину с x и определяем ее «вес».

Если «вес» смежной вершины превосходит «вес» вершины x , то необходимо рассмотреть следующую смежную вершину.

Если «вес» смежной вершины меньше, чем «вес» вершины x, то переходим к этой новой вершине и продолжаем от нее.

В случае, если «веса» всех смежных вершин больше «веса» рассматриваемой вершины, то это центроид из одной вершины. Если оказалось, что «вес» смежной вершины равен «весу» рассматриваемой вершины, то тогда это центроид из двух вершин.

Т.е продолжаем двигаться по вершинам, пока их «вес» уменьшается.

На вход алгоритма подается дерево T.

Алгоритм

1. Находим вершину, степень которой наибольшая.

Для списка смежности это сделать проще: выбираем вершину, у которой максимальный размер списка смежных с ней вершин.

Для матрицы смежности: выбираем вершину, у которой наибольшая сумма из 1.

root = max(degree(v) для всех vϵV.

2. Считаем вес корня: N(root) – окрестность корня, соседние вершины

Для всех neighbor ϵ N(root):

Считаем вес подграфа = дереву полученном с помощью алгоритма DFS, без ребра между корнем и соседней с ним вершины.

Вес корня = минимальное DFS-дерево, полученное построением DFS-деревьев из всех соседних вершин, без ребер связывающих их с корнем.

3. Просматриваем соседнюю вершину neighbor с root:

Если neighbor == list:

continue

иначе: считаем вес neighbor: как в пункте 2 для neighbor -> root

Если вес neighbor > веса root:

переходим к следующей соседней вершине

иначе если вес neighbor < веса root:

root = neighbor

вес root = вес neighbor

переходим к началу пункта 3

иначе: центроид из root и neighbor

Если среди всех соседей не оказалось вершин с меньшим весом, тогда центроид состоит из root.

Свойства:

1. Время работы для матрицы смежности оценивается в O ($|V^3|$), а время работы для списка смежности - O (|V*E|).

2. Вершины, которые в деревьях являются центроидами, необязательно совпадают с вершинами, которые в этих деревьях являются центрами.

2.5 Поиск кратчайшего пути. Алгоритм Дейкстры

Алгоритм находит кратчайшее расстояние от одной из вершин графа до всех остальных и работает только для графов без ребер отрицательного веса.

Каждой вершине приписывается **вес**– это вес пути от начальной вершины до данной.. Обходя граф, алгоритм считает для каждой вершины маршрут, и, если он оказывается кратчайшим, выделяет вершину. Весом данной вершины становится вес пути.

Алгоритм Дейкстры

1. Всем вершинам, за исключением первой, присваивается вес равный бесконечности, а первой вершине – 0.
2. Все вершины не выделены.
3. Первая вершина объявляется текущей.
4. Вес всех невыделенных вершин пересчитывается по формуле: вес невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса ребра, соединяющего текущую вершину с невыделенной.
5. Среди невыделенных вершин ищется вершина с минимальным весом. Если таковая не найдена, то есть вес всех вершин равен бесконечности, то маршрут не существует. Следовательно, выход. Иначе, текущей становится найденная вершина. Она же выделяется.
6. Если текущей вершиной оказывается конечная, то путь найден, и его вес есть вес конечной вершины.
7. Переход на шаг 4.

Свойства:

1. Время оценивается в $O(|V|^2+|E|)$.

2.6 Поиск шарниров в графе

Вершина называется шарниром (или точкой сочленения), если при ее удалении число компонент связности увеличивается.

Лемма Пусть T – DFS-дерево графа G с корнем a. Вершина x ≠ a является шарниром графа тогда и только тогда, когда у нее в дереве T имеется такой сын y, что ни один потомок вершины y не соединен ребром ни с одним собственным предком вершины x

Теорема о шарнирах. Корень DFS-дерева является шарниром графа тогда и только тогда, когда его степень в этом дереве больше 1

Алгоритм

1. В процессе поиска в глубину помечаем вершины v натуральными числами n(v) в порядке их обнаружения.
2. При возврате (полагаем, что стек раскручивается после того, как все вершины помечены) вычисляем для каждой извлеченной из стека вершины v число:

   $$low(v) = \min \{ \min_{u \in d(v)} low(u), \min_{w \in A(v) \setminus d(v)} n(w) \}$$

   Здесь d(v) – потомки v и d-дереве (по прямым ребрам из v),
   A(v) – все соседи v (A(v)\d(v) – вершины, доступные из v по обратным ребрам и предок v в d-дереве).
3. Шарнирами являются:
   - Вершина, с которой начали поиск в глубину, если у нее больше 1 потомка в d-дереве.
   - Остальные вершины v, если $\exists u \in d(v): low(u) = n(v)$

Свойства:

1. Время оценивается в O(|V|+|E|).

2.7 Поиск блоков в графе

Блок графа – это его максимальный связный подграф, не имеющий собственных шарниров (т.е. некоторые шарниры графа могут принадлежать блоку, но своих шарниров у блока нет).

Лемма

Пусть x – предок для y в DFS-дереве T. Ребро (x, y) является начальным ребром некоторого блока тогда и только тогда, когда $low(y) = n(x)$)

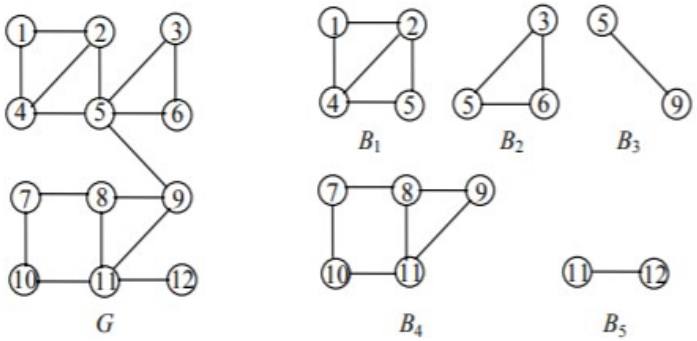
Алгоритм

Описанный выше алгоритм выявления шарниров нетрудно приспособить для отыскания всех блоков графа.

1. В основе алгоритма вычисление функции $low(v)$ из предыдущего раздела. $low(v)$ есть наименьший из глубинных номеров вершин, смежных с потомками вершины v.

Переменная k – счетчик блоков, B(k) – множество вершин блока с номером k.

2. Так же понадобится стек S, в котором будут накапливаться вершины графа, впервые встречающиеся в процессе обхода в глубину (то есть превращающиеся из новых в открытые).
3. Множество вершин нового блока NewBlock строится всякий раз, когда обнаруживается начальное ребро (x, y) некоторого блока (выполняется равенство $low(y) = n(x)$), т.е обнаруживается шарнир.
4. Множество вершин блока NewBlock включает новое множество B(k) и вершины x, y и все вершины, находящиеся в стеке выше вершины y. Эти вершины удаляются из стека (кроме вершины x, которая является начальной вершиной блока и может принадлежать еще и другим блокам).



Свойства:

1. Время оценивается в O (|V|+|E|).

Рис. 4

2.8 Построение BC-дерева

Строение связного графа, состоящего из нескольких блоков, может быть схематически описано с помощью так называемого дерева блоков и шарниров, кратко именуемого BC-деревом.
В этом дереве имеются две категории вершин - одни поставлены в соответствие блокам графа, другие - его шарнирам.

Каждая висячая вершина дерева bc(G) является блоком (т.е. компонентой двусвязности) связного графа G.
Каждый связный граф хотя бы с двумя компонентами двусвязности содержит не менее двух висячих блоков.

На рис.5 изображено BC-дерево для графа с рис.4.

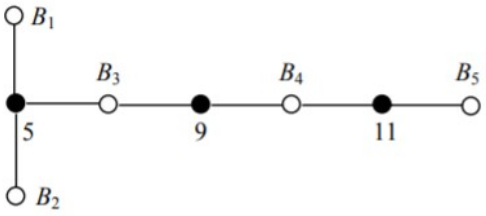


Рис. 5

Блоки изображены белыми, а шарниры черными кружками.

Алгоритм

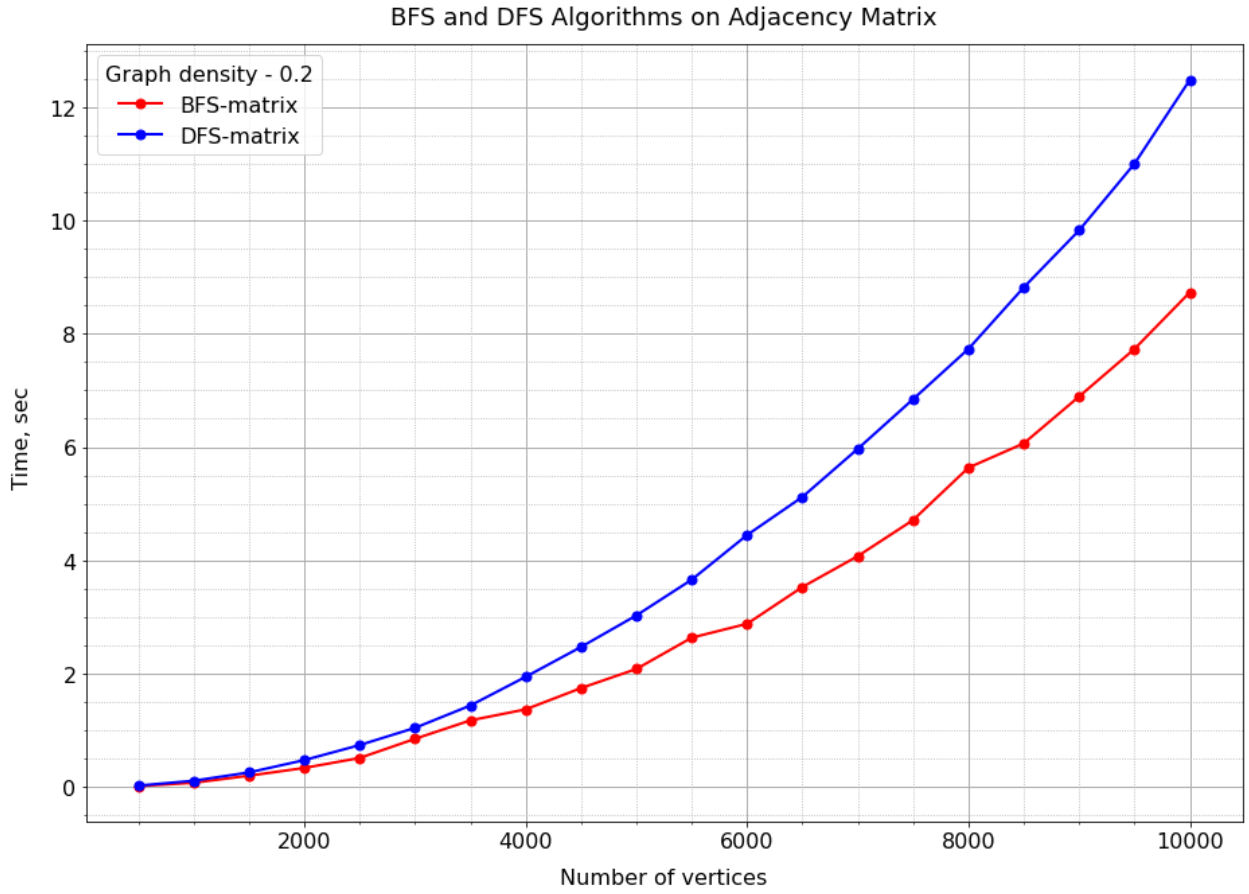
Определяем блоки для графа G и шарниры по алгоритмам выше. Заменяем блок вершиной. Вершина-блок в дереве соединяется ребром с вершиной-шарниром, если в графе соответствующий шарнир принадлежит соответствующему блоку.

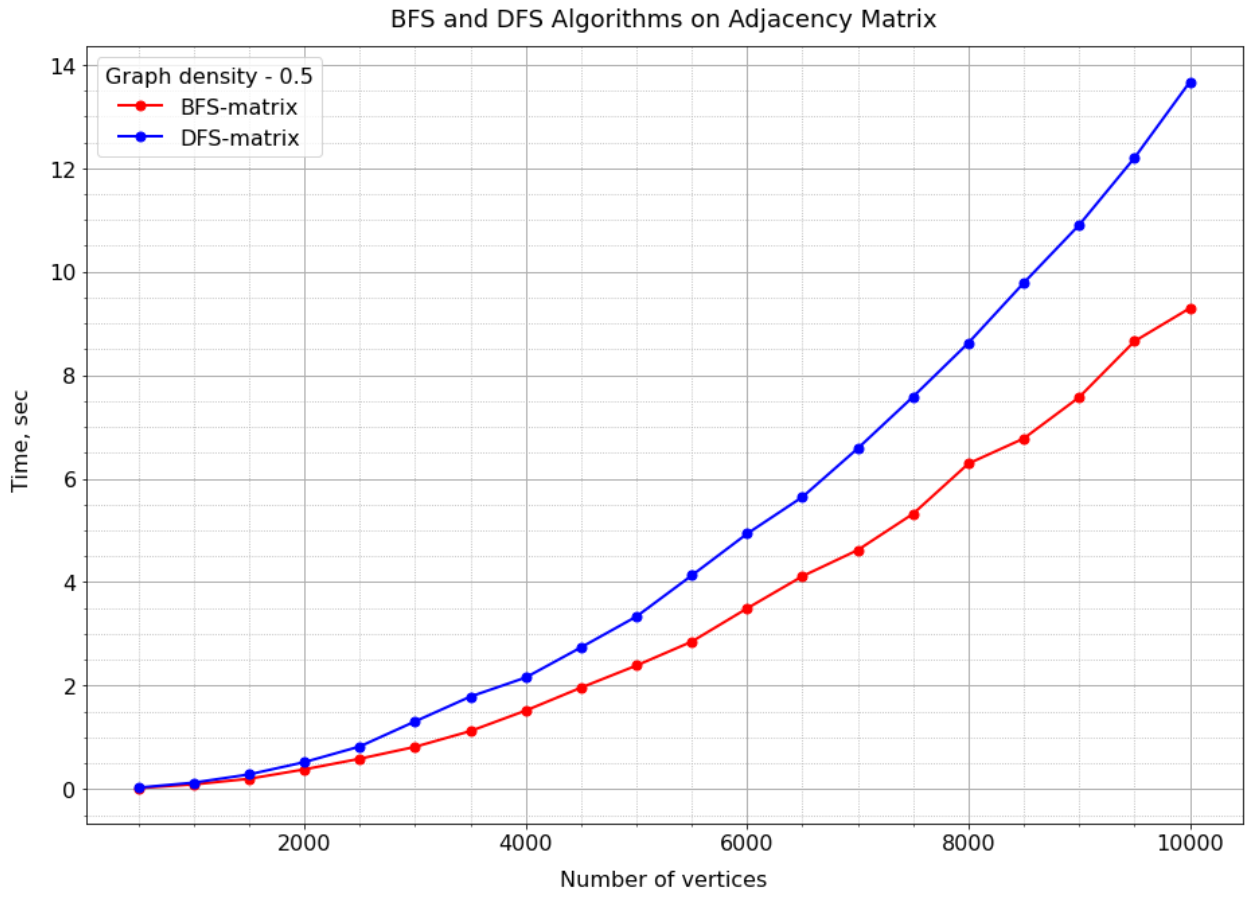
Свойства:

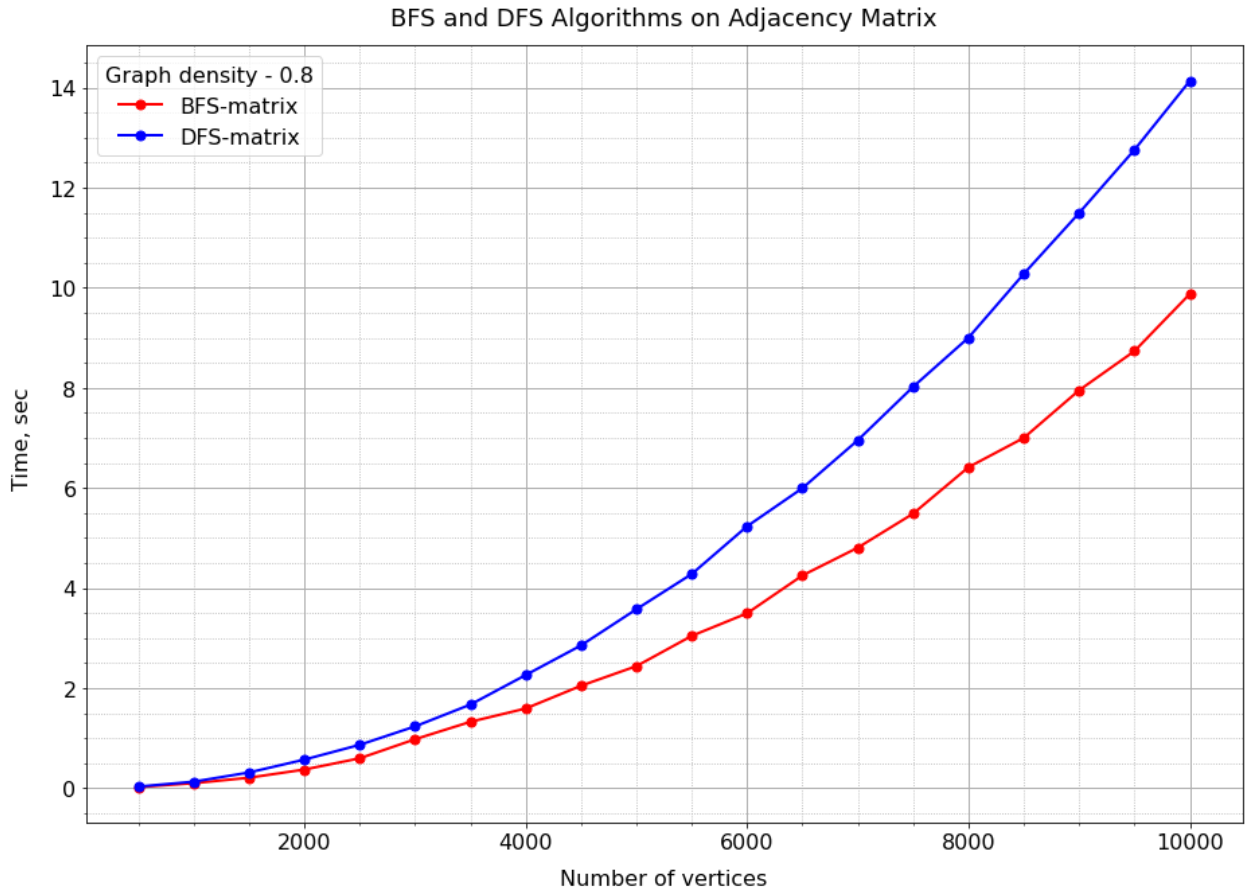
1. Время оценивается в O (|V|+|E|), время оценки поиска блоков в графе..

3 Анализ алгоритмов

3.1 BFS- и DFS-деревья на матрице смежности

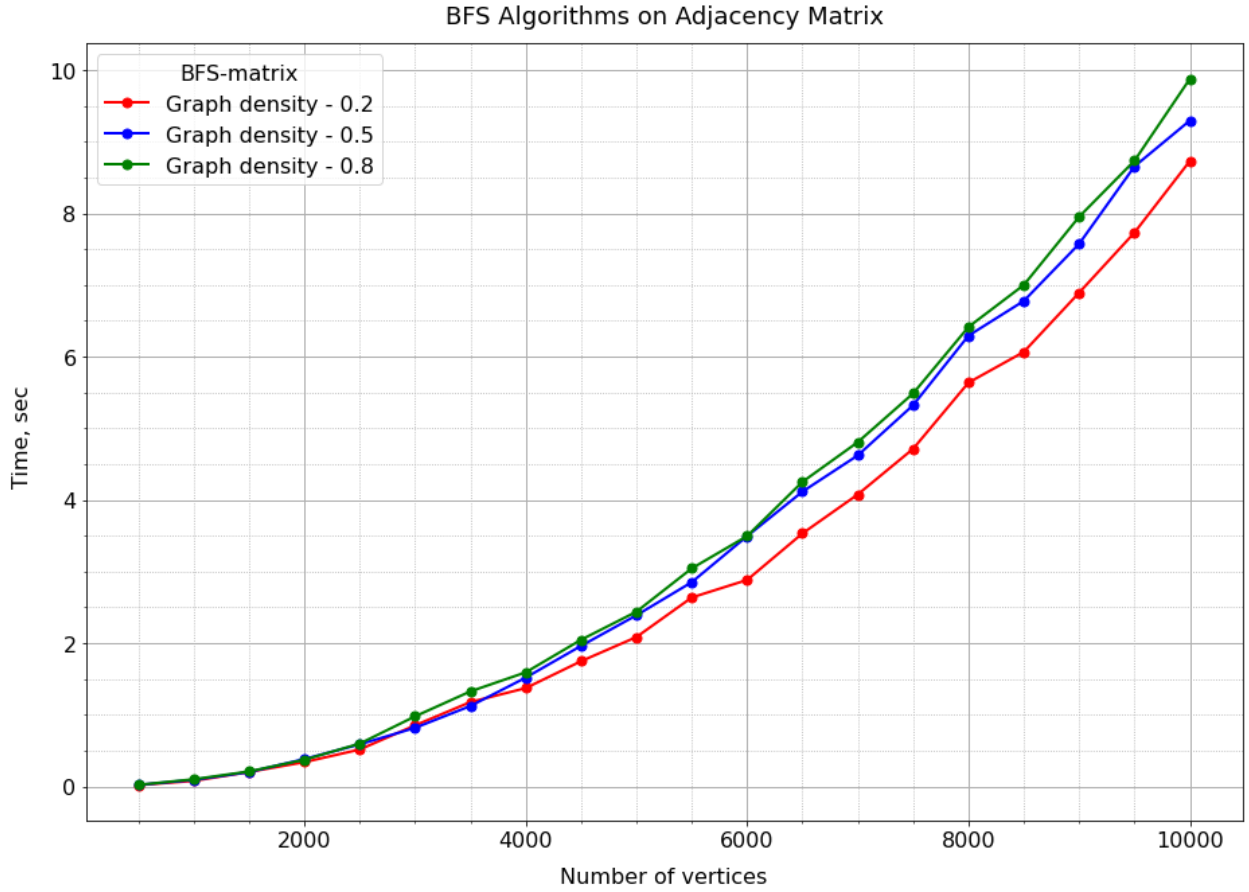




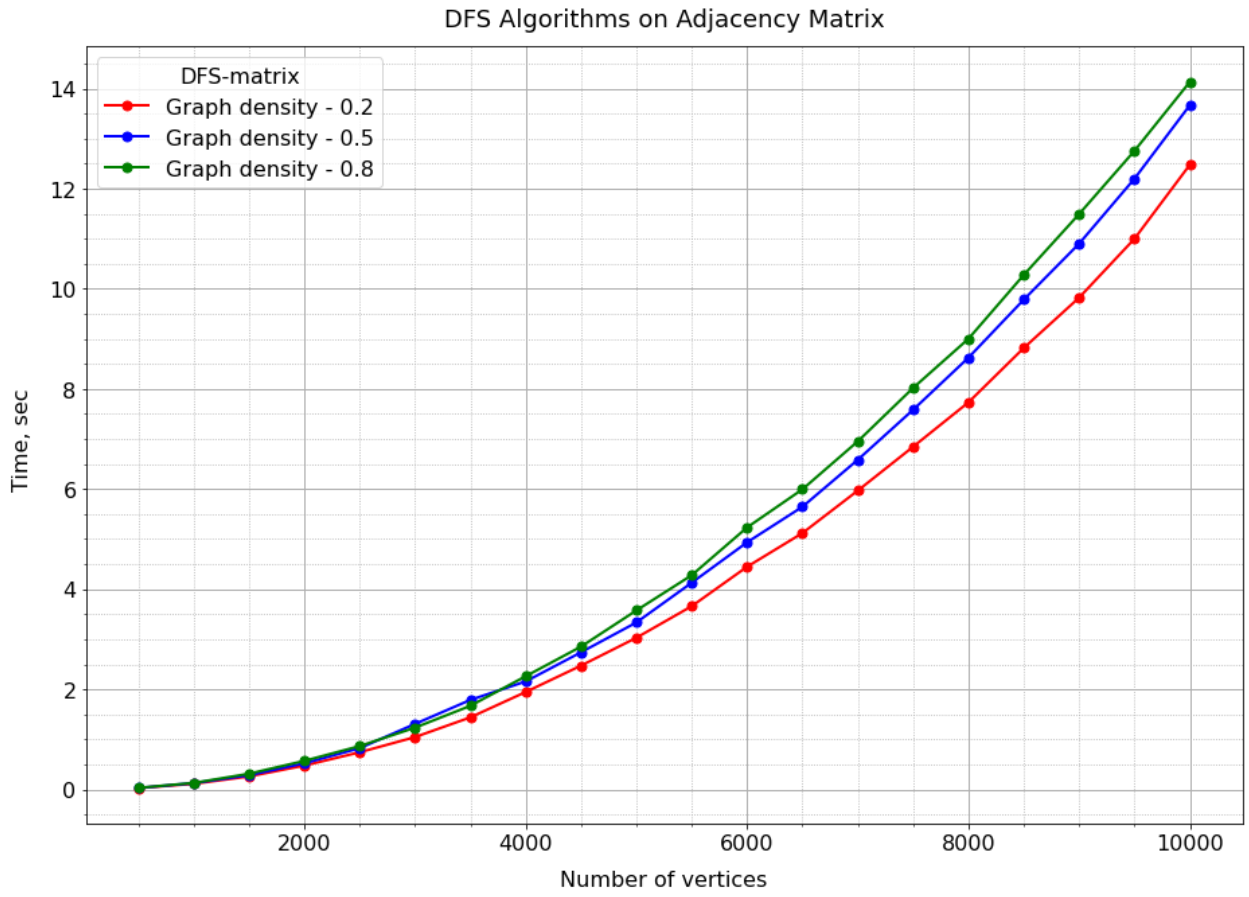


На данной выборке графов преимущества у BFS-алгоритма, по отношению к DFS-

алгоритму, это связанно с тем, что в DFS-алгоритме, есть необходимость возврата. Но разница несильно существенна.

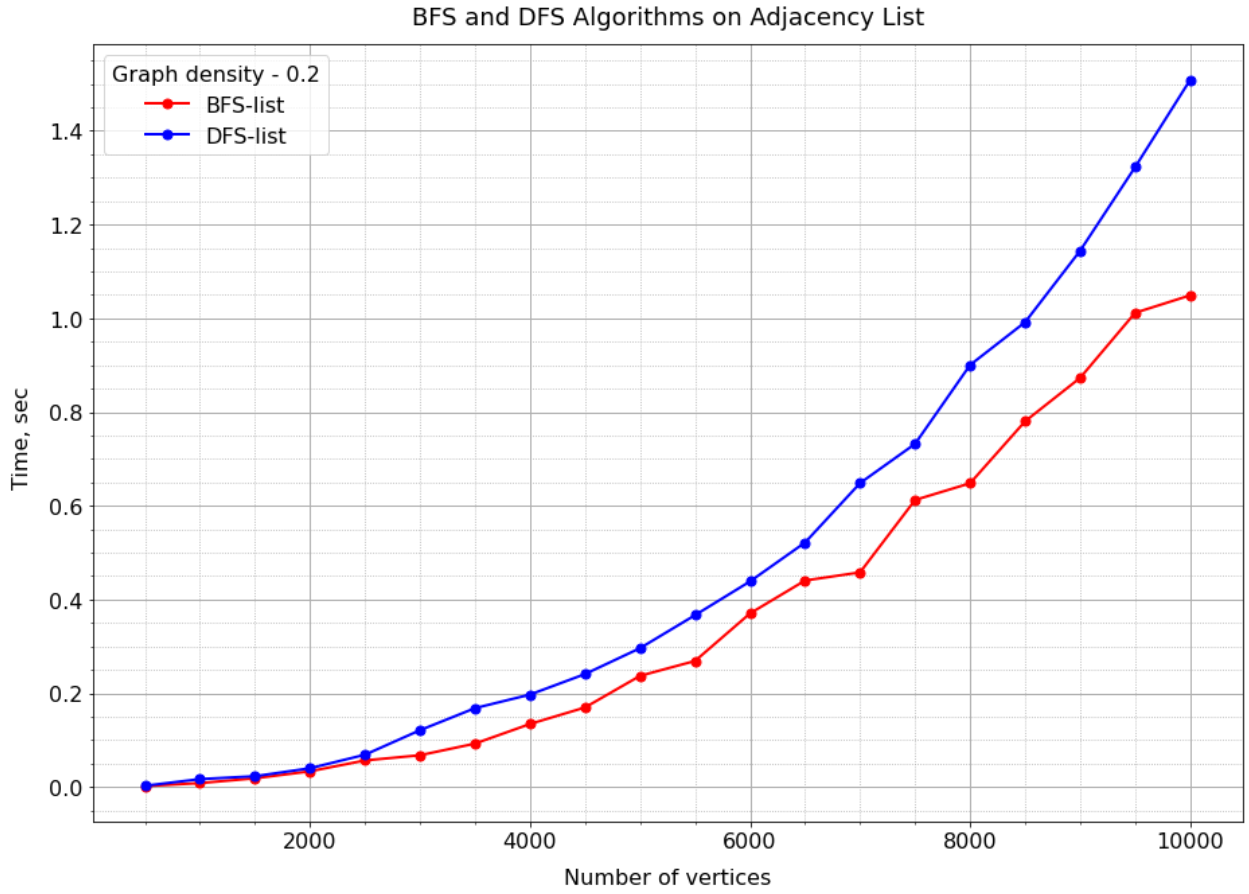


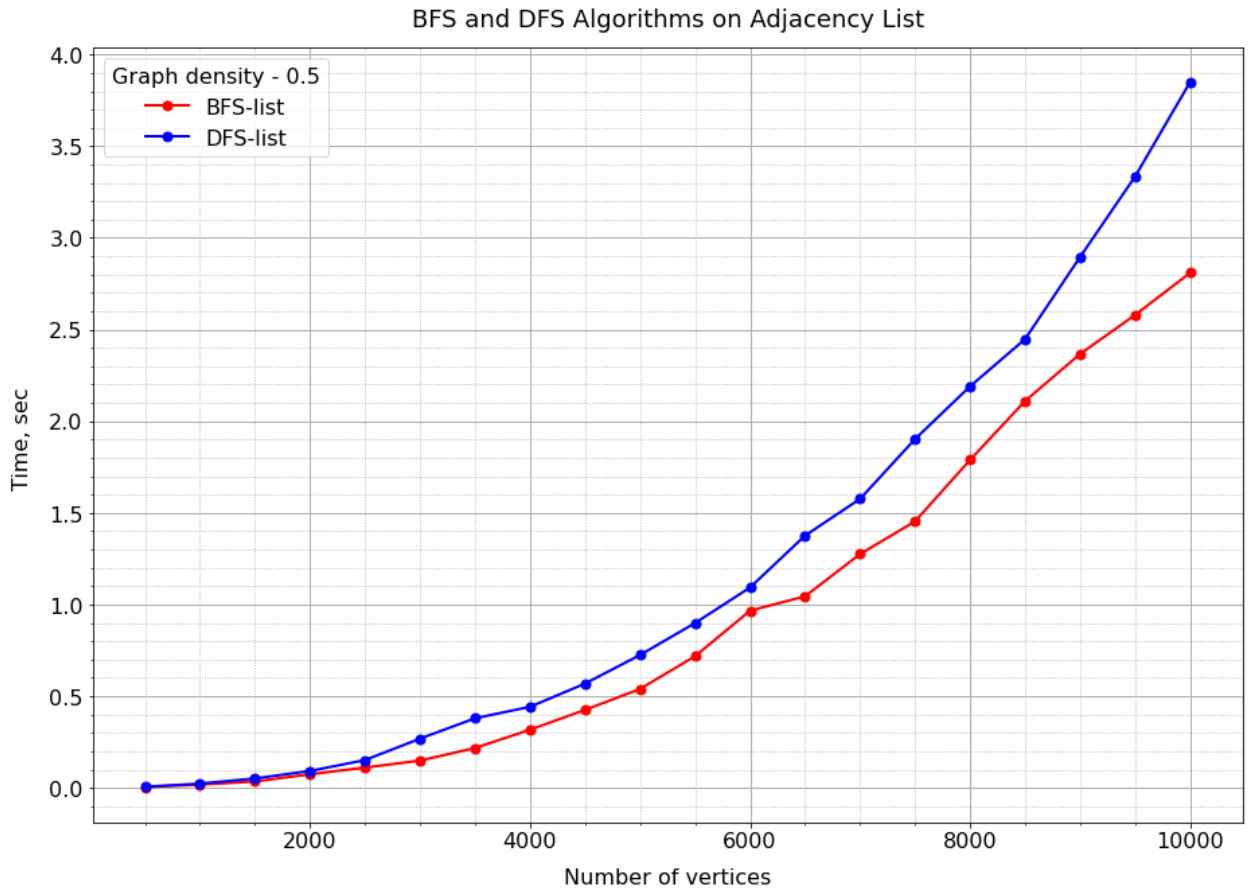
На графах с малым количеством вершин разницы во времени построения каркаса BFS-алгоритмом нет. На разреженных графах алгоритм справляется быстрее.

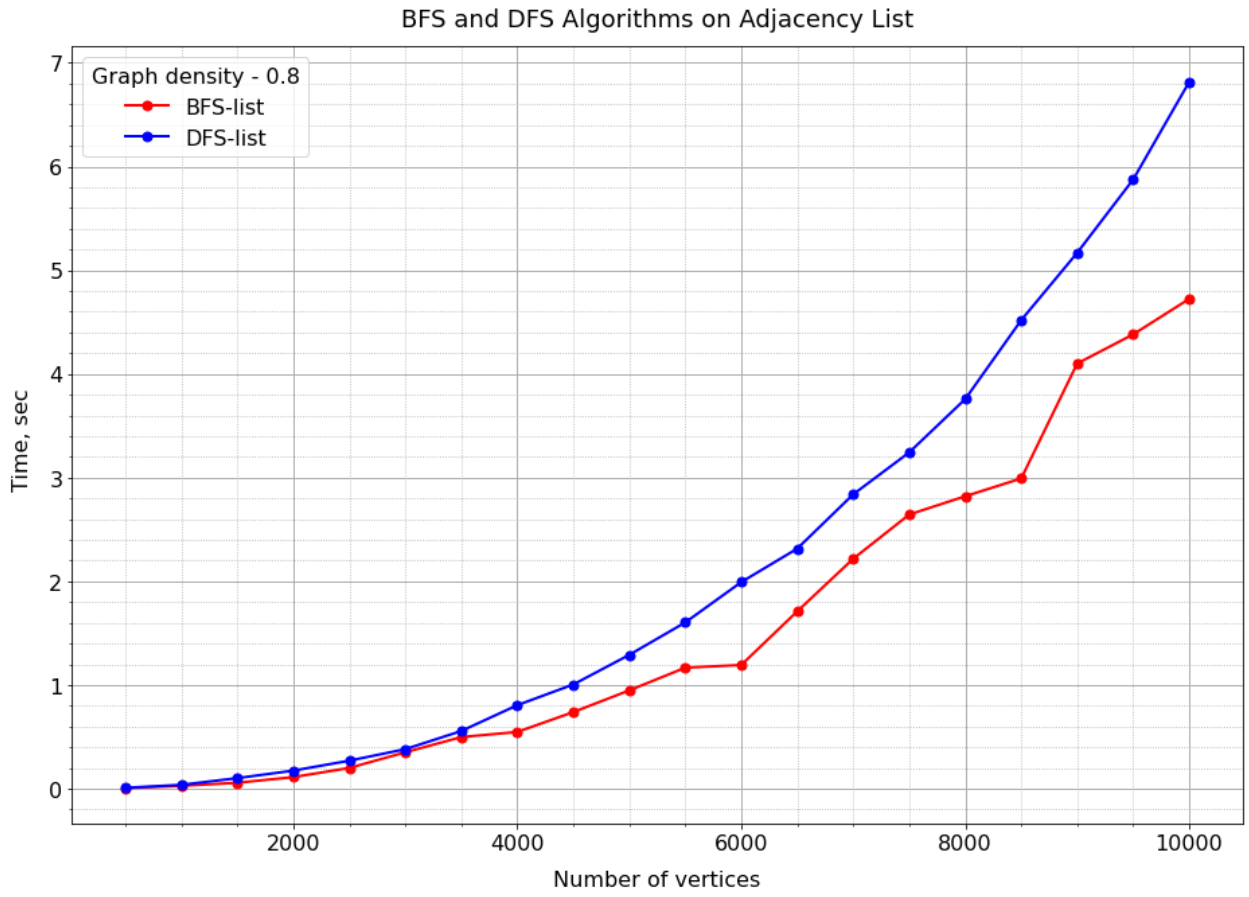


Для DFS- алгоритма ситуация аналогичная. На более насыщенных графах алгоритм занимает чуть больше времени.

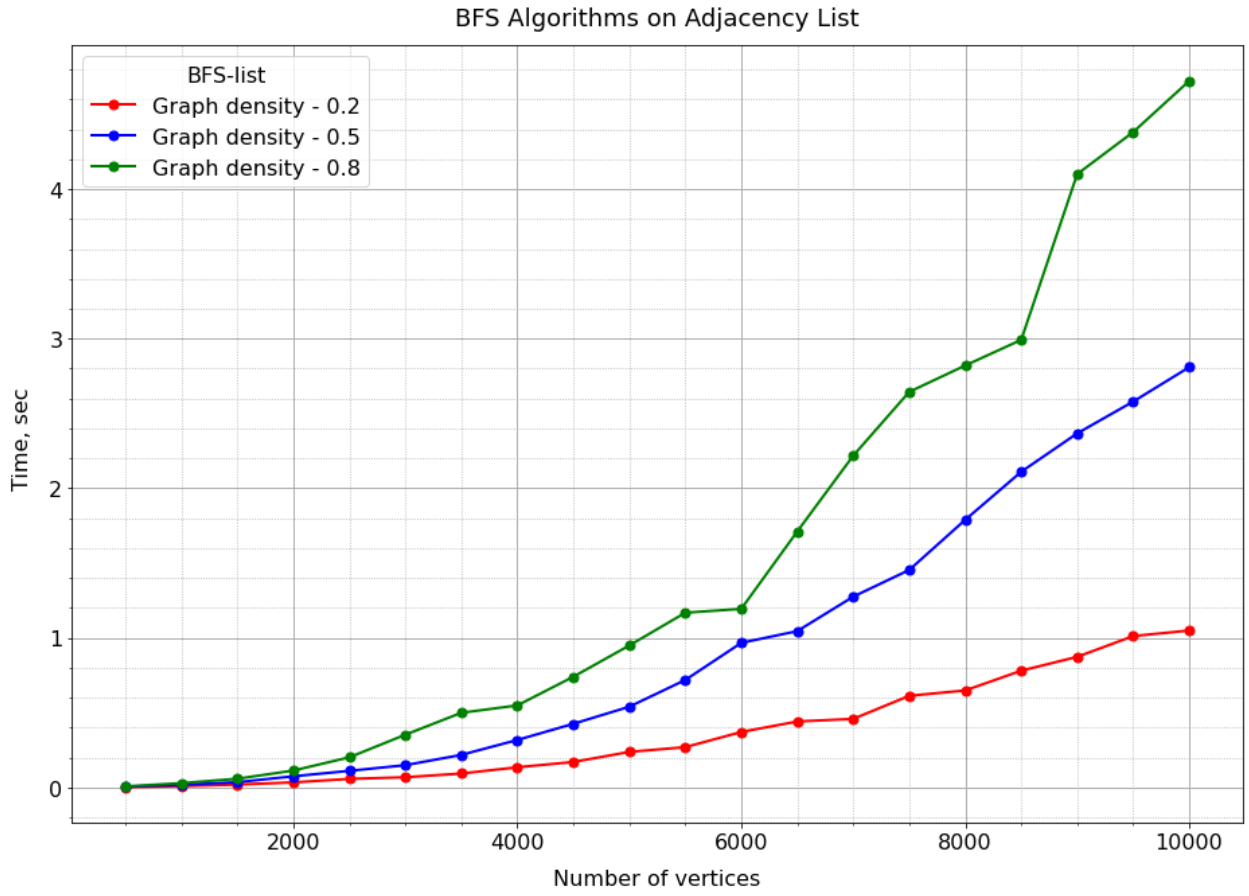
3.2 BFS- и DFS-деревья на списке смежности

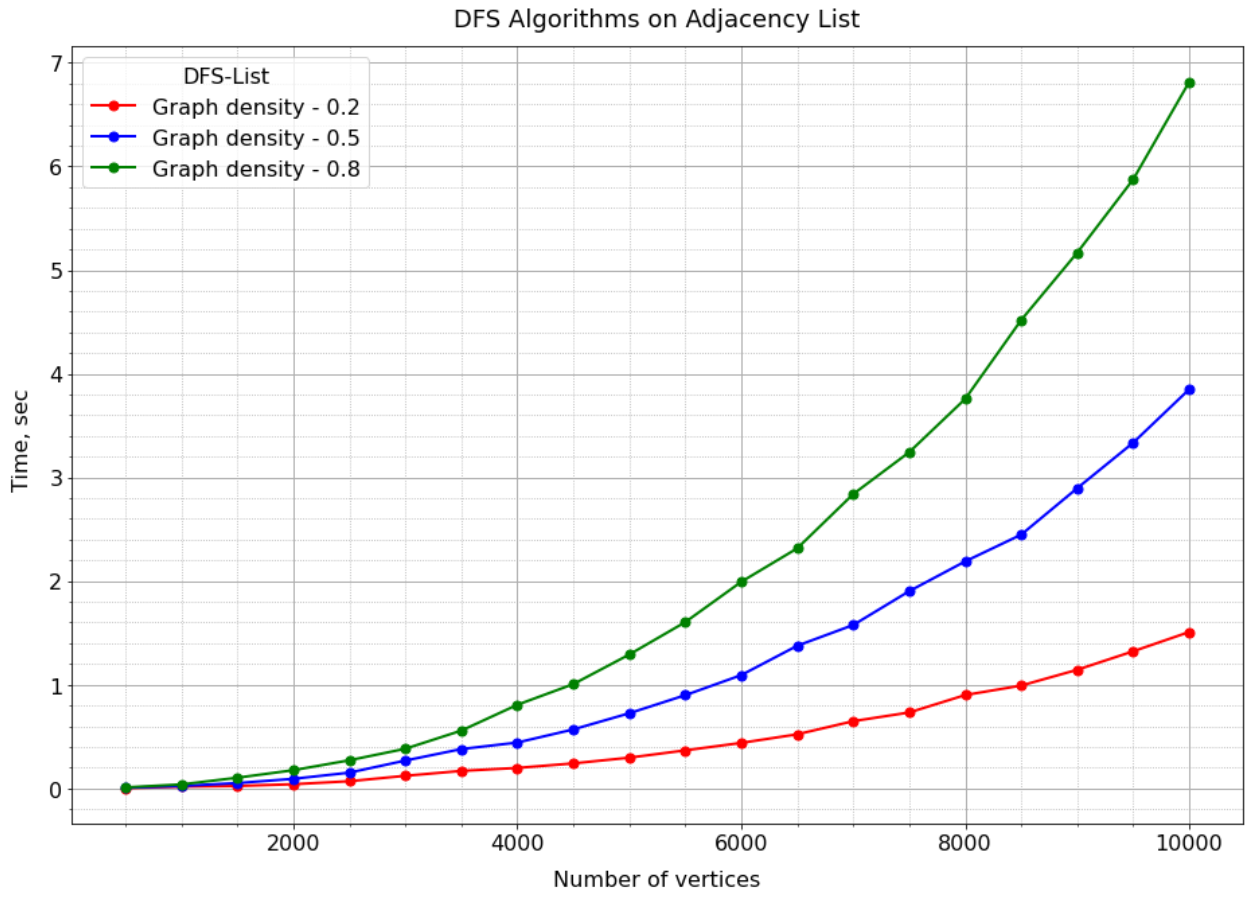






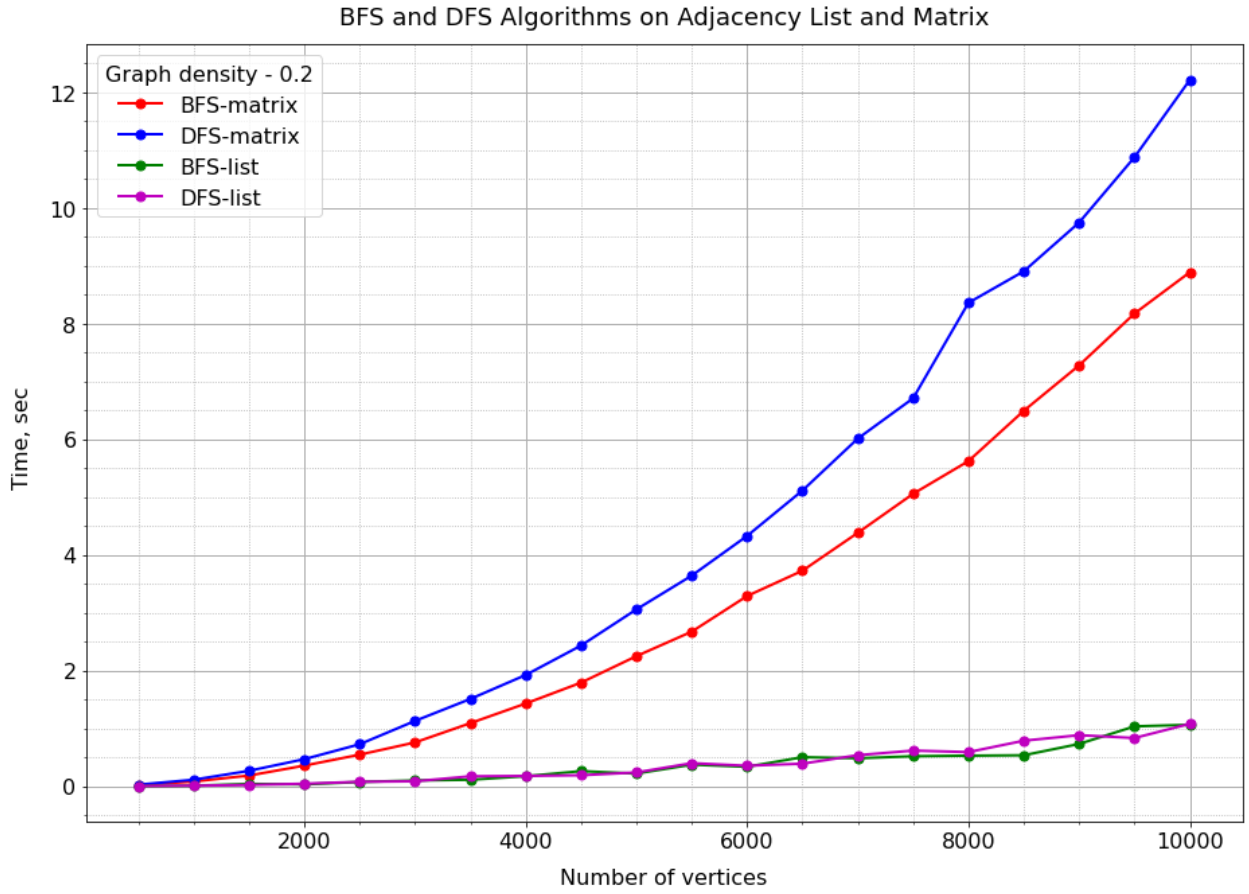
На другой структуре данных: списке смежности все так же преимущества у BFS-алгоритма, рекурсия позволила бы сэкономить время DFS-алгоритму, но есть риск переполнения стека вызовов.

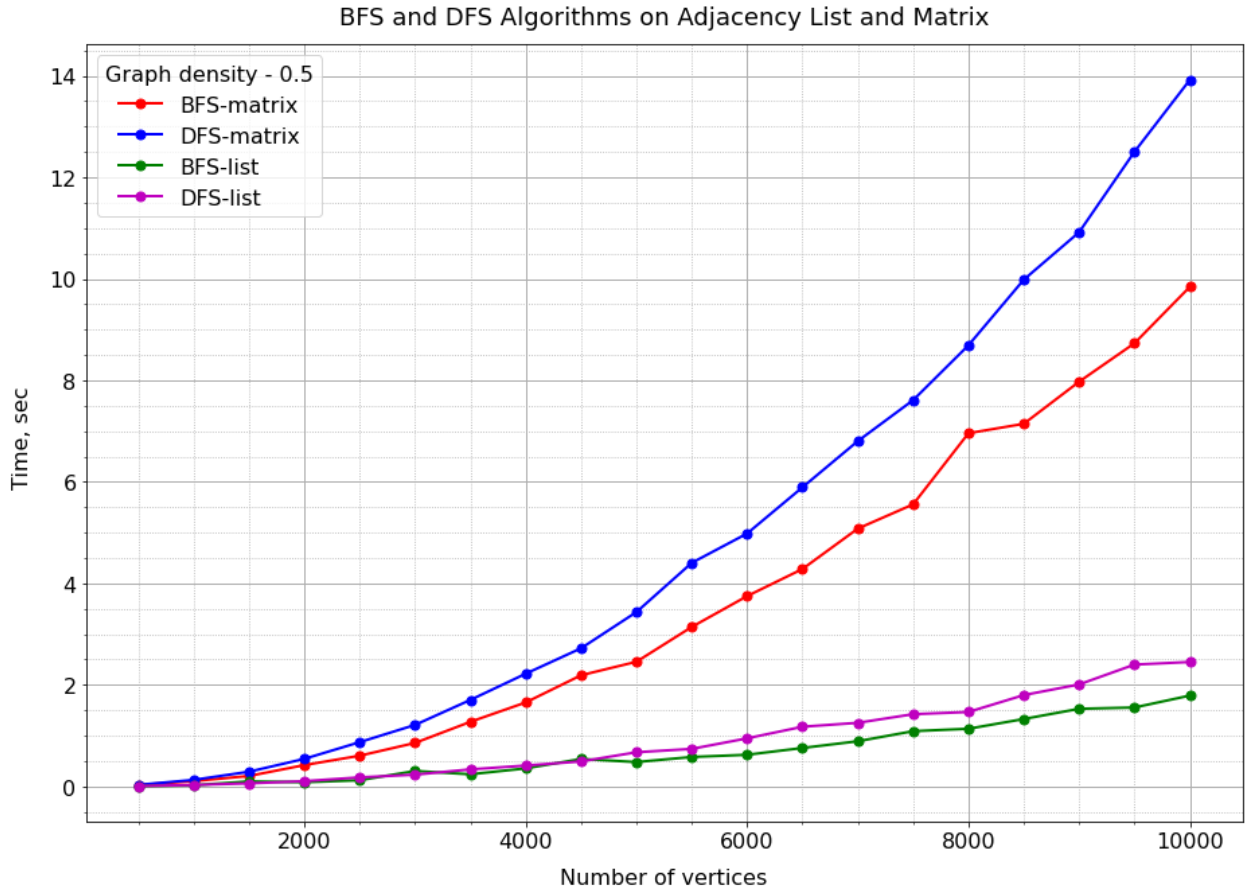


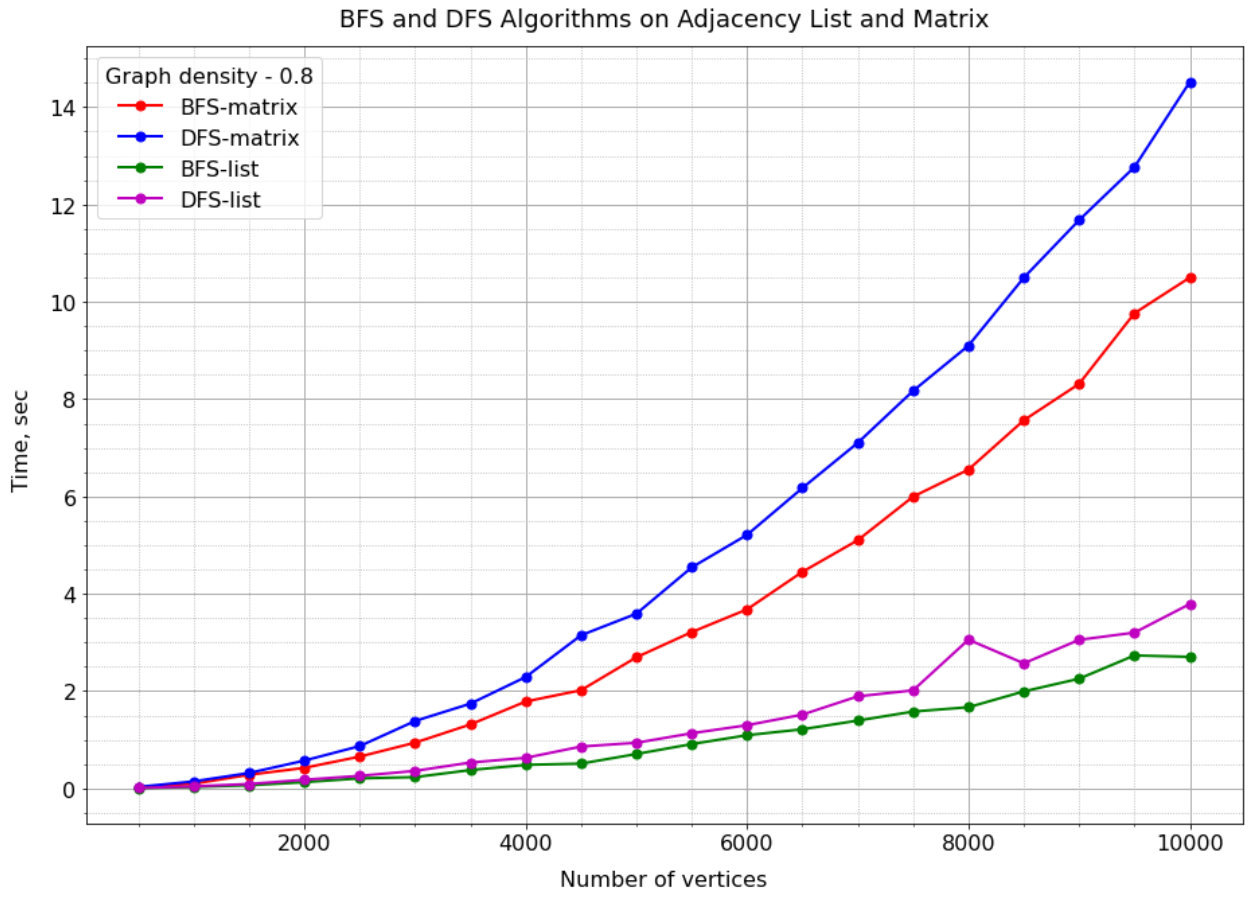


В случае списка смежности зависимость времени работы алгоритмов от насыщенности графа нагляднее. Хуже всего время у графов с наибольшим количеством ребер, это связано с тем, что при увеличении числа ребер список расширяется, что замедляет работу алгоритма.

3.3 BFS- и DFS-деревья на матрице и на списке смежности

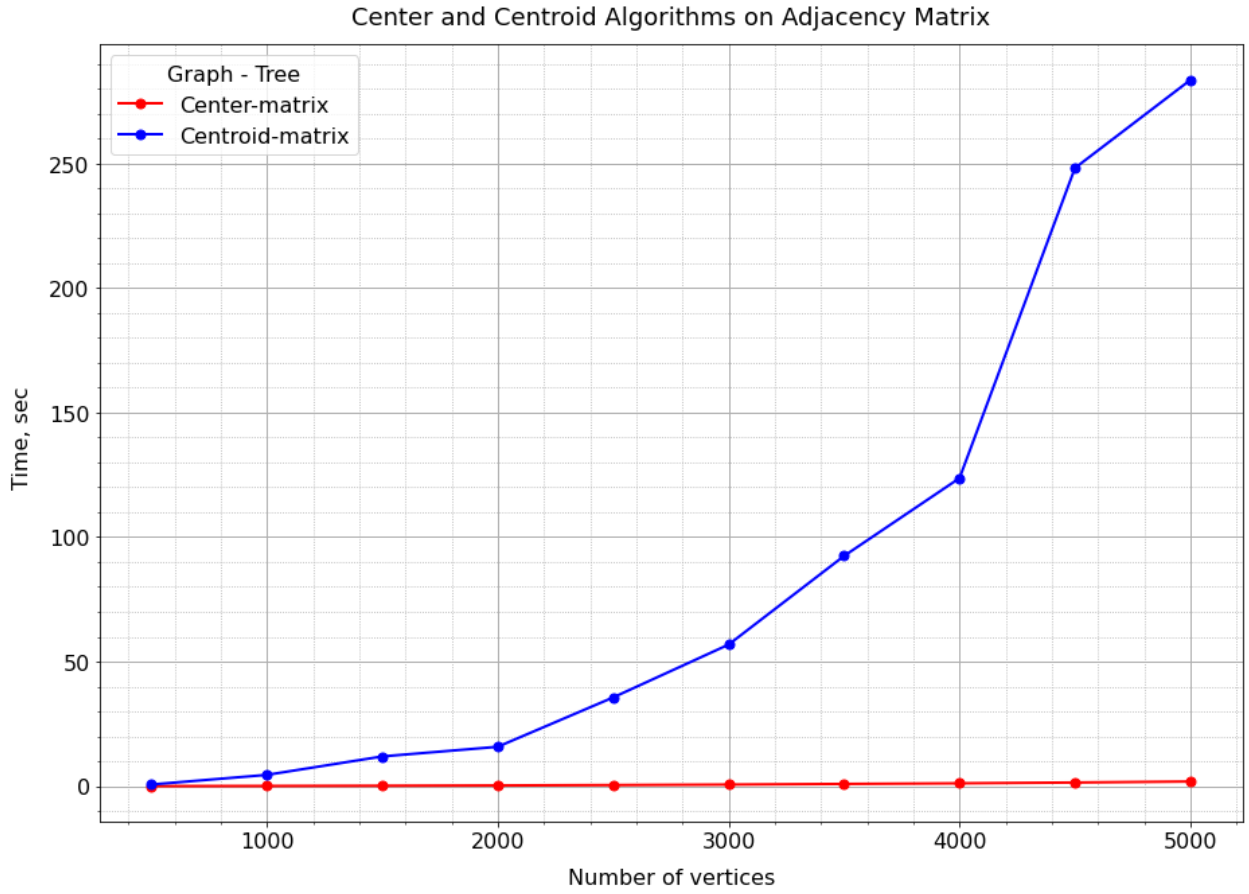






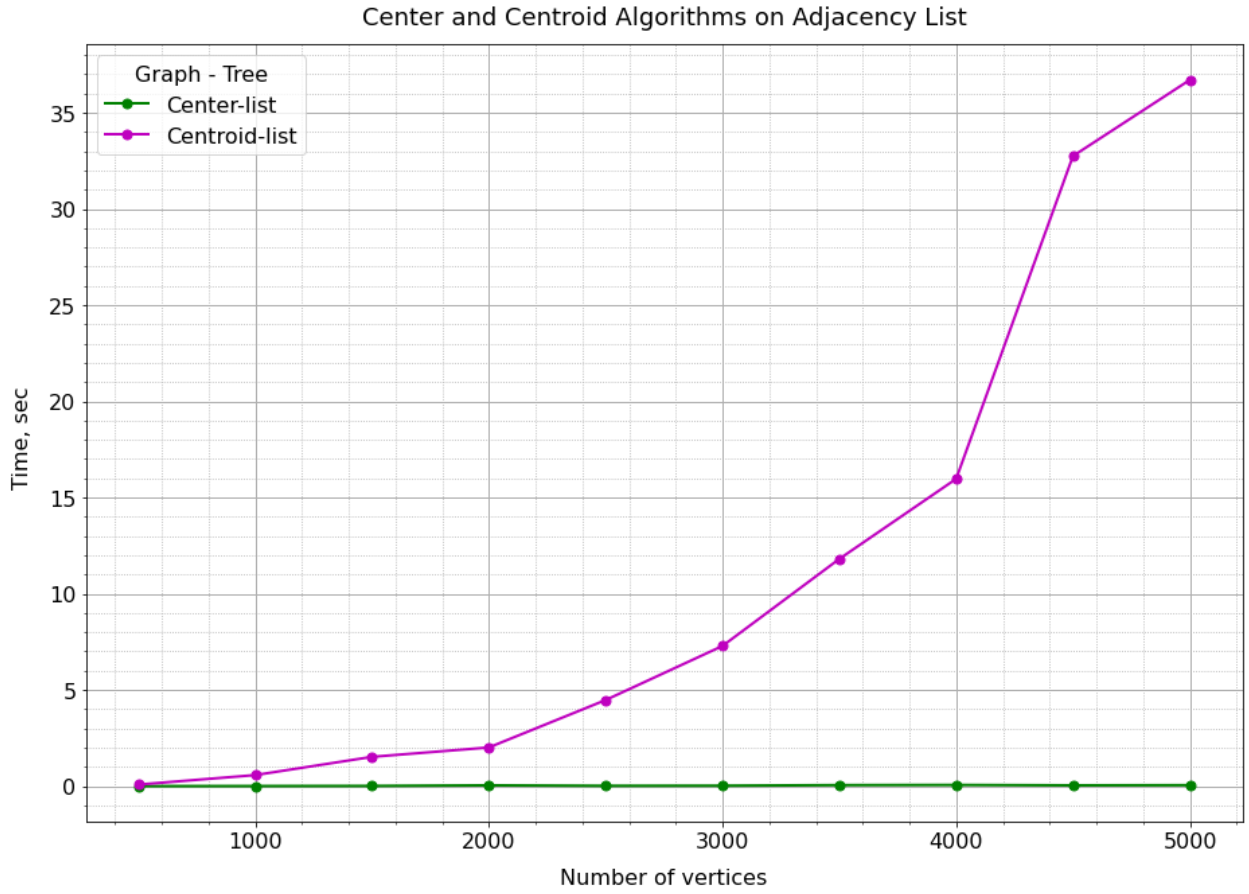
Построение BFS- и DFS-деревьев значительное быстрее происходит при использовании списка смежности, что связанно с более быстрым доступам к соседним вершинам.

3.4 Поиск центра и центроида на матрице смежности



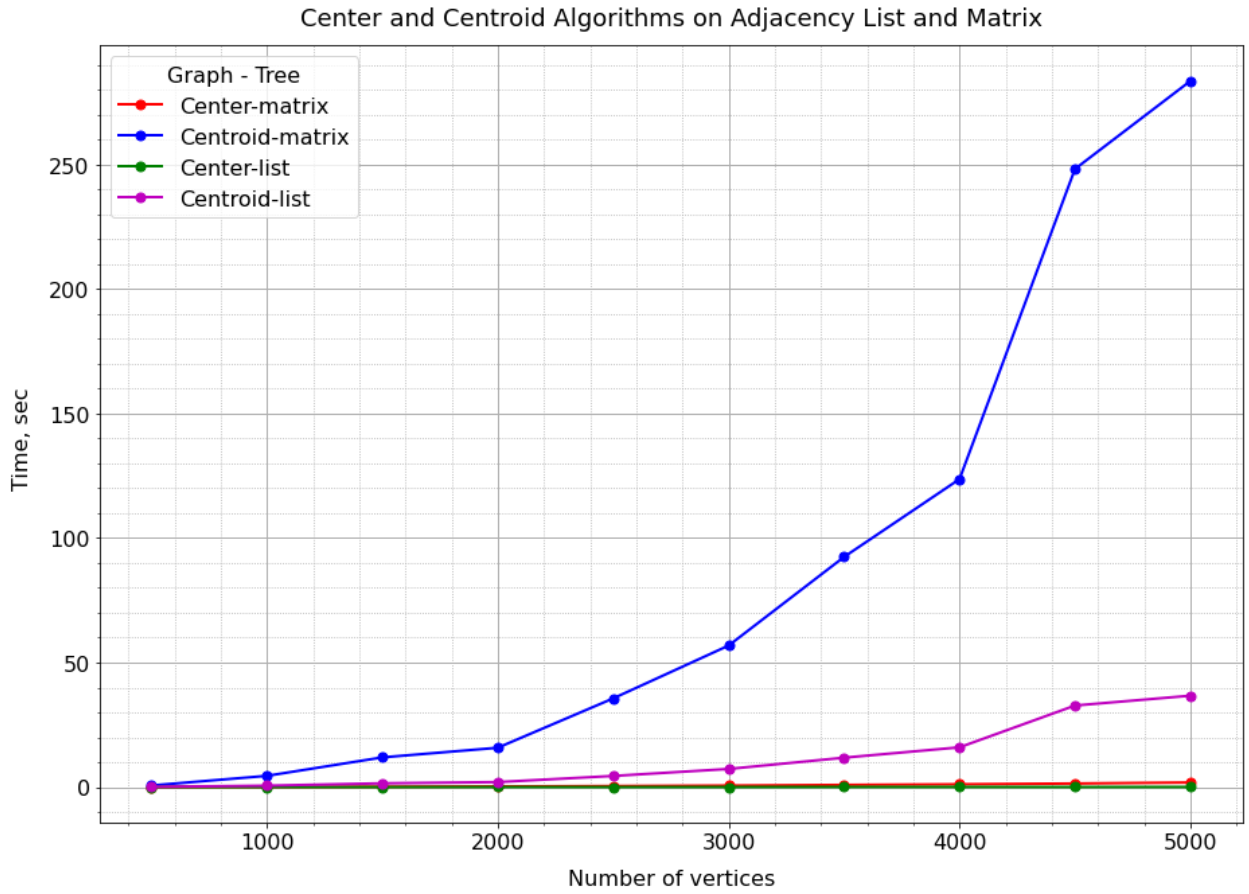
Алгоритм поиска центра в дереве работает быстрее, так как не нужно вычислять веса вершин.

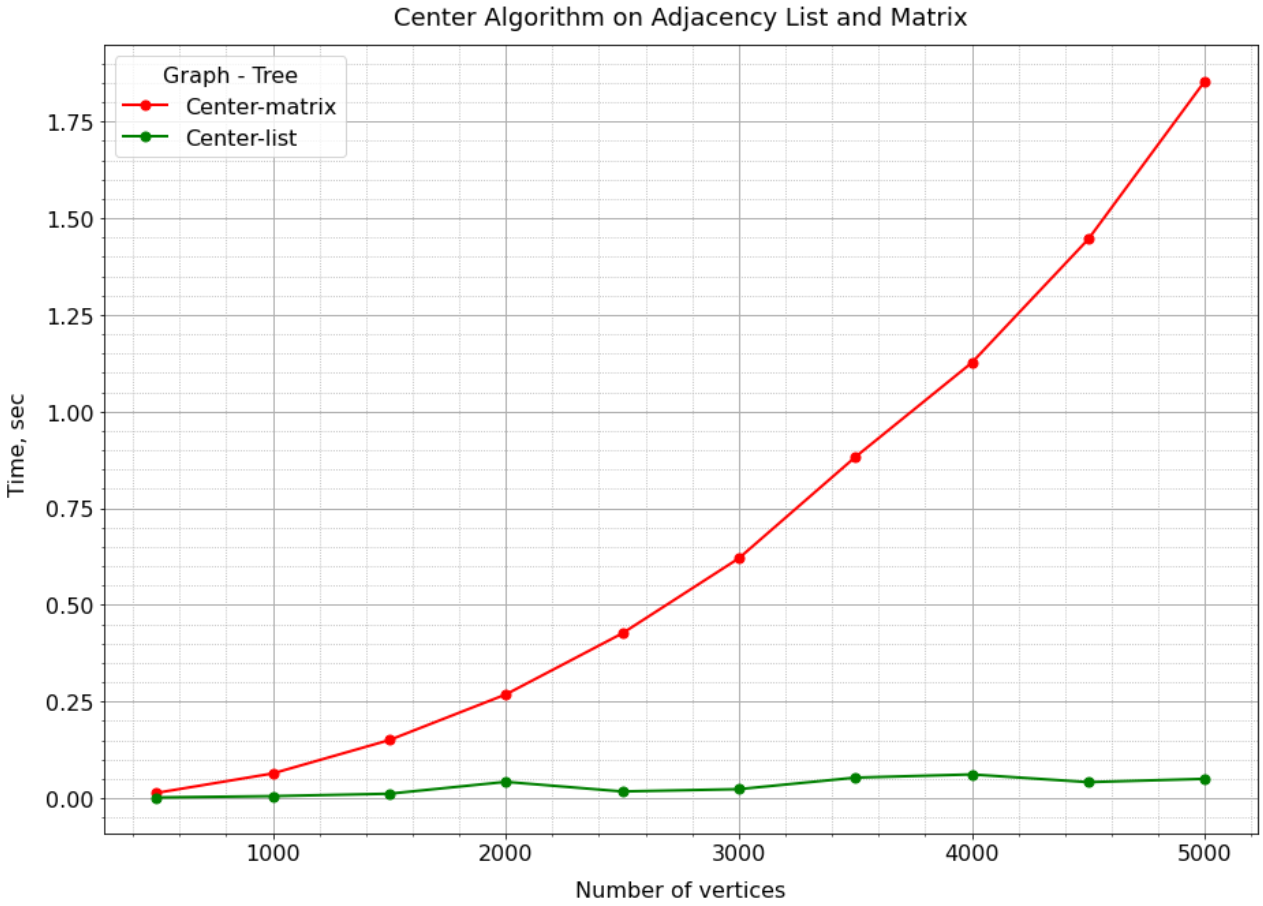
3.5 Поиск центра и центроида на списке смежности



На данной структуре хранения все также быстрее работает алгоритм поиска центра в дереве.

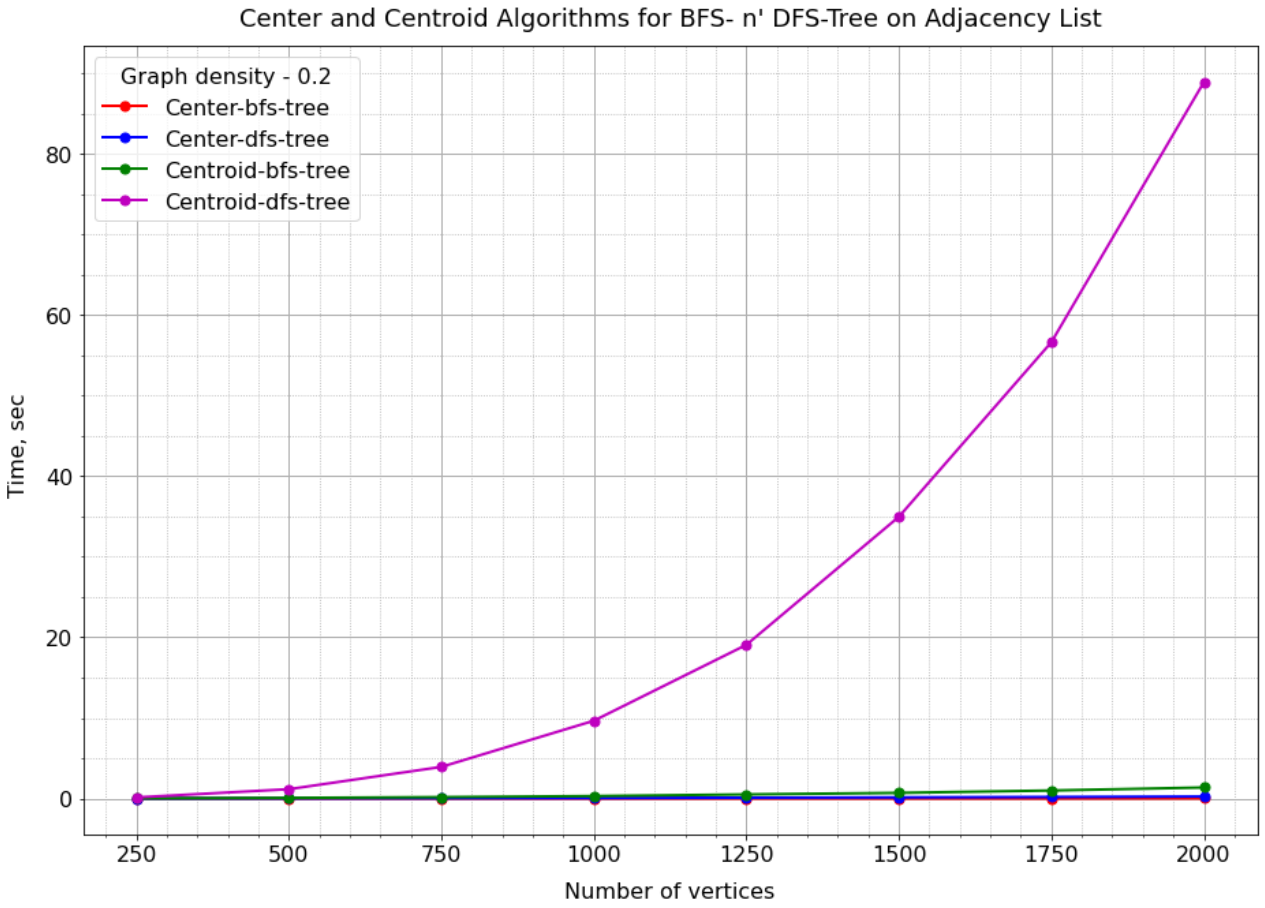
3.6 Поиск центра и центроида на списке и матрице смежности

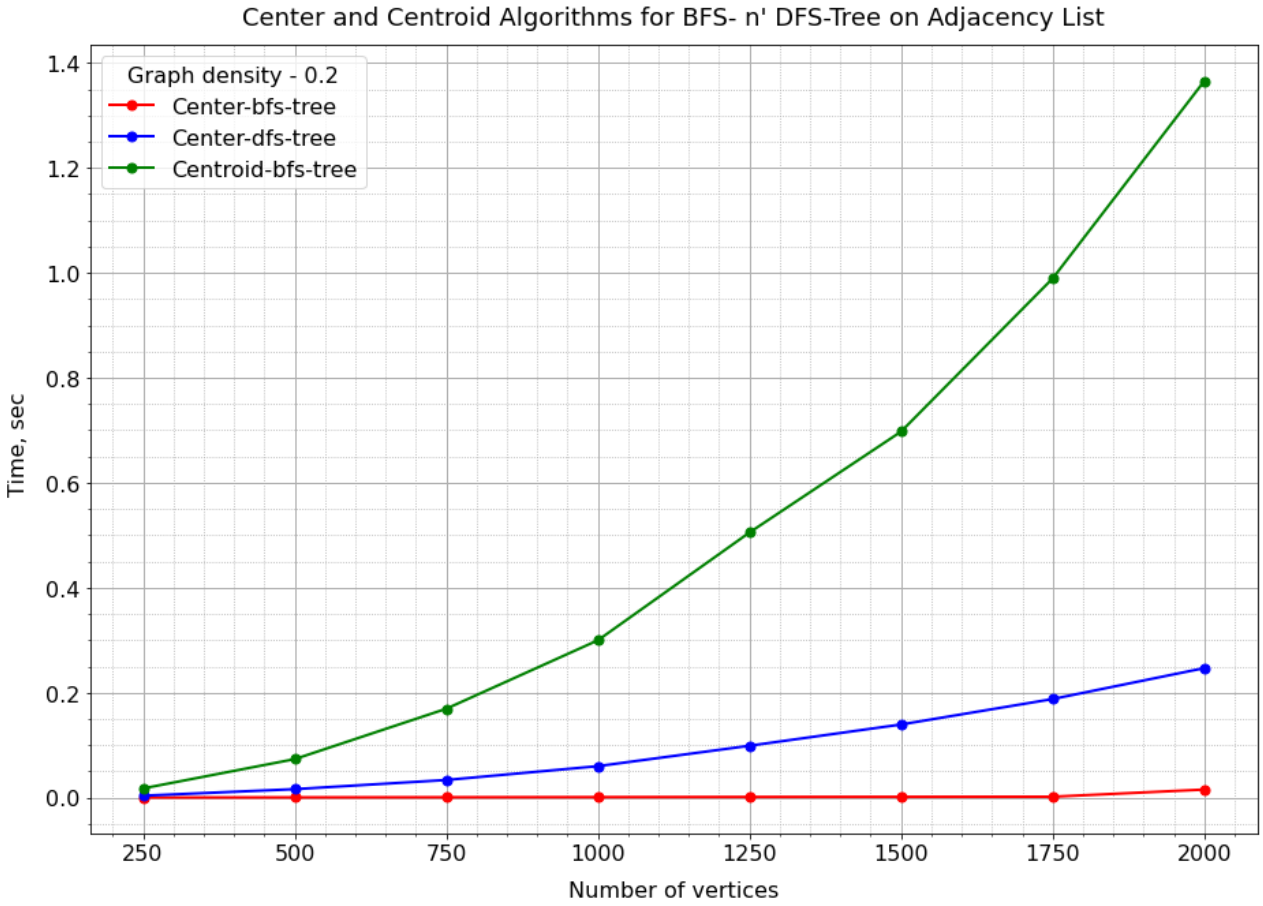


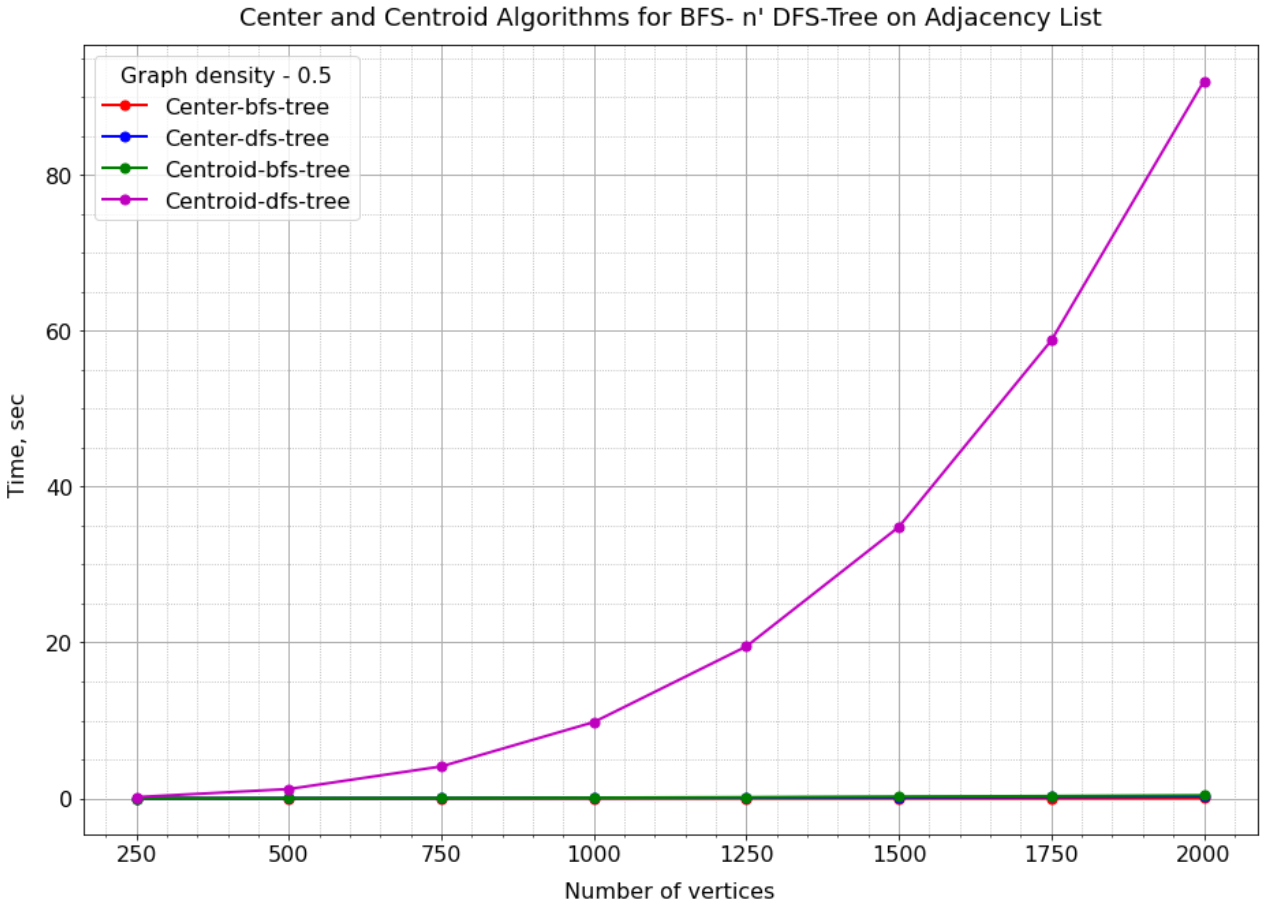


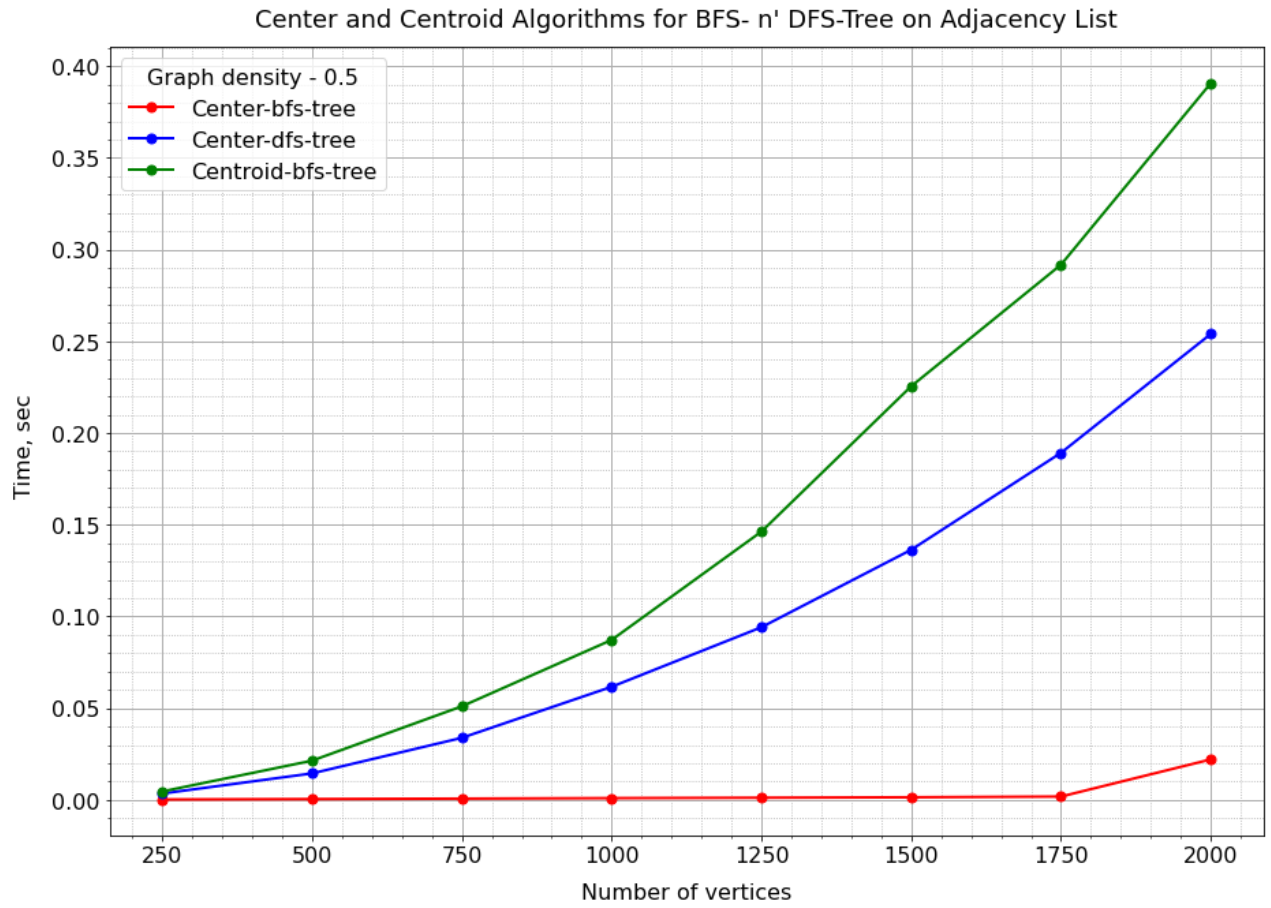
Заметное отставание у алгоритма поиска центроида в матрице смежности. Преимущества все так же у алгоритмов на списке смежности.

3.7 Поиск центра и центроида в BFS- и DFS-деревьях на списке и матрице смежности

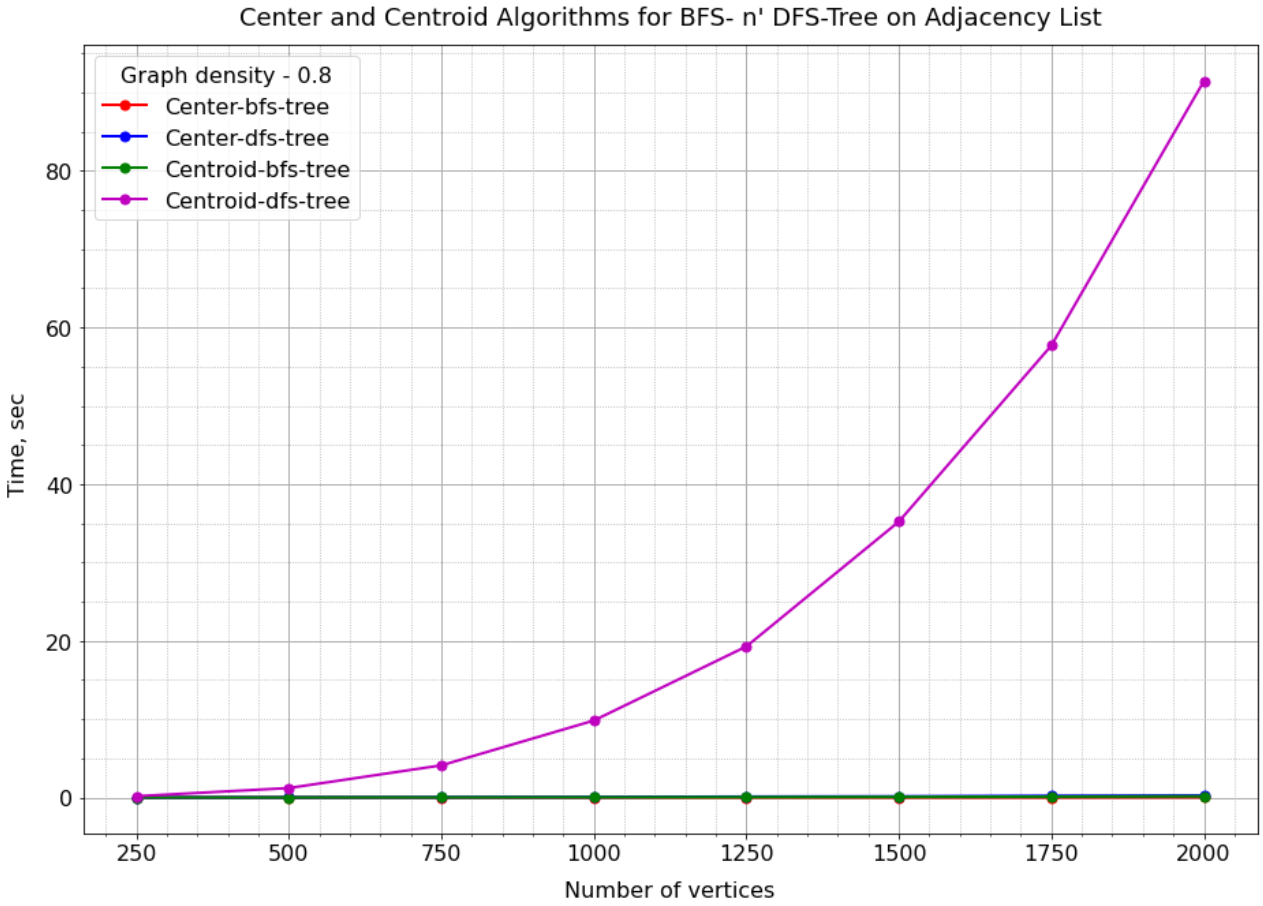


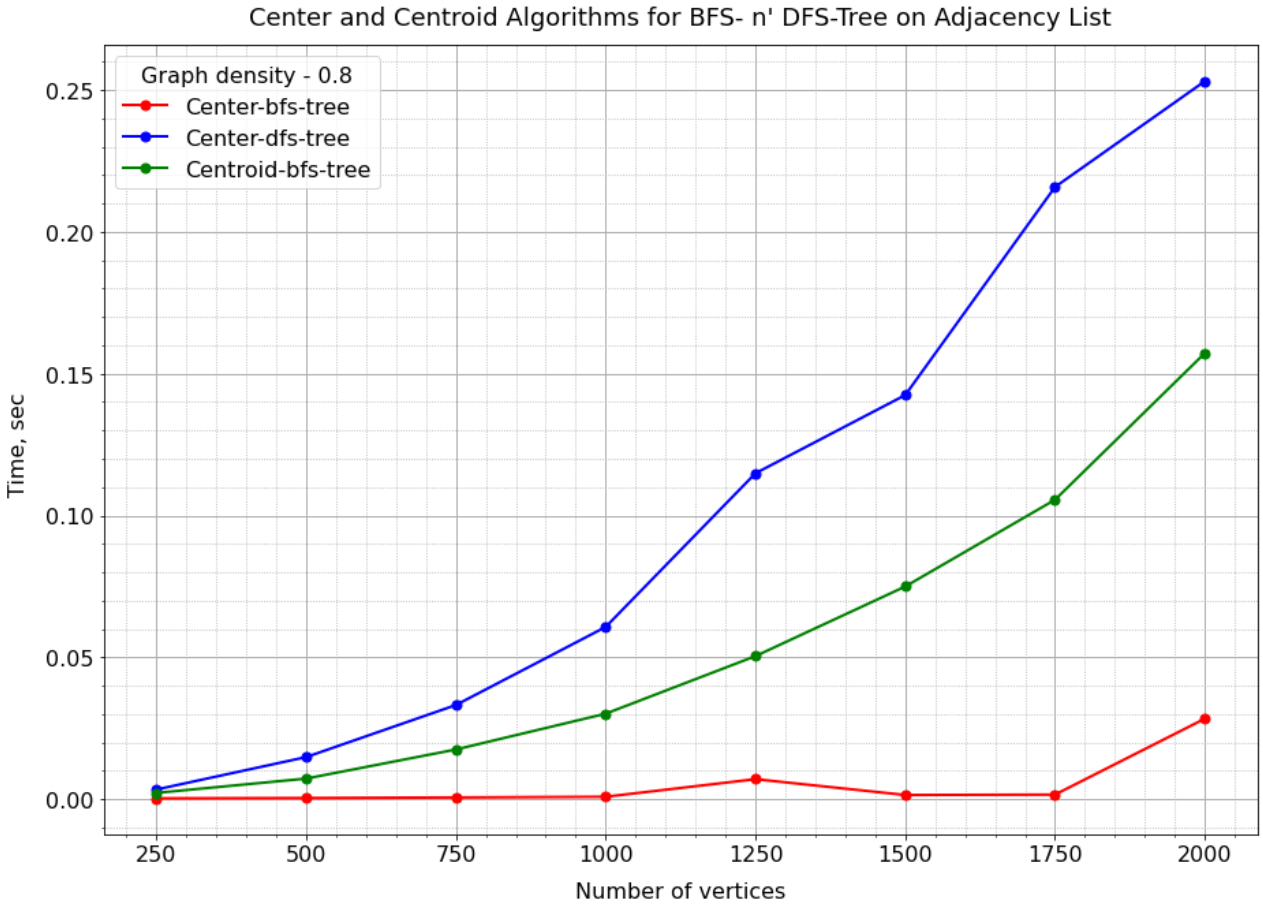

Center and Centroid Algorithms for BFS- n' DFS-Tree on Adjacency List
Graph density - 0.2
Center-bfs-tree
Center-dfs-tree
Centroid-bfs-tree
Time, sec
Number of vertices
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
250
500
750
1000
1250
1500
1750
2000


Center and Centroid Algorithms for BFS- n' DFS-Tree on Adjacency List
Graph density - 0.5
Center-bfs-tree
Center-dfs-tree
Centroid-bfs-tree
Centroid-dfs-tree
Time, sec
Number of vertices
0
20
40
60
80
250
500
750
1000
1250
1500
1750
2000

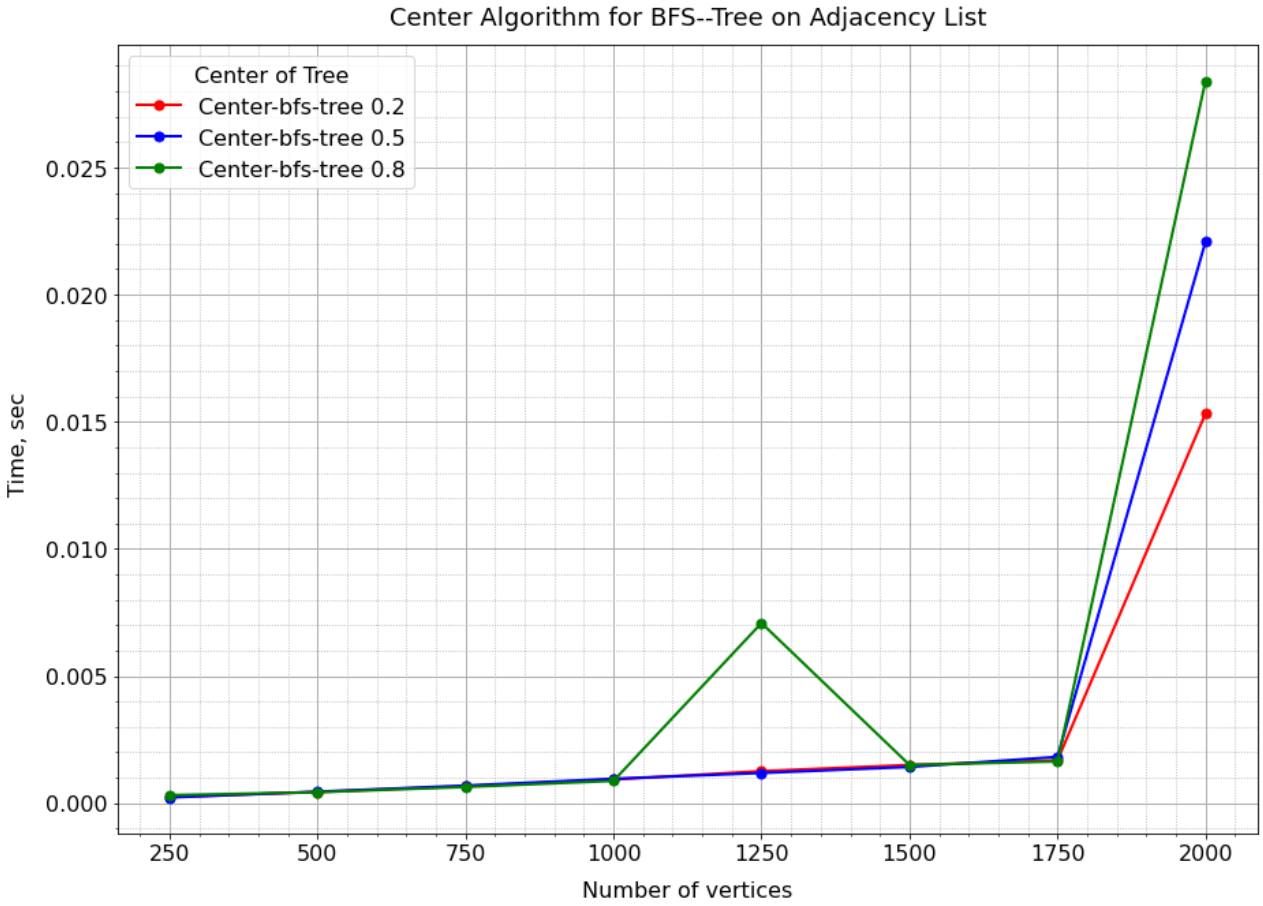


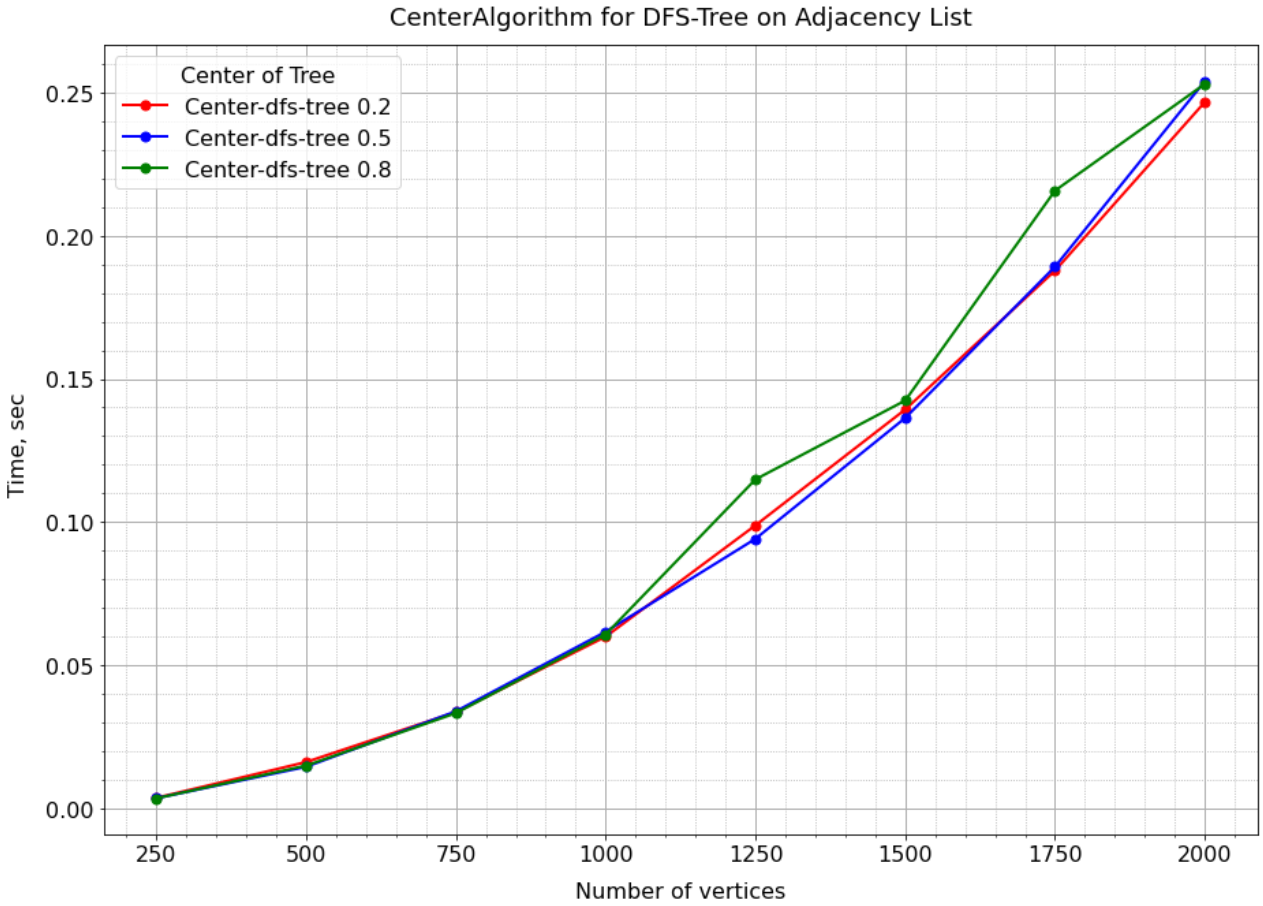
Алгоритм поиска центроида в DFS-дереве работает дольше всего. Это связанно с тем, что у DFS-дерева большая глубина и приходится много раз вычислять веса вершин. Быстрее всего работает поиск центра в BFS-дереве, как раз из-за того что оно имеет меньшую глубину.



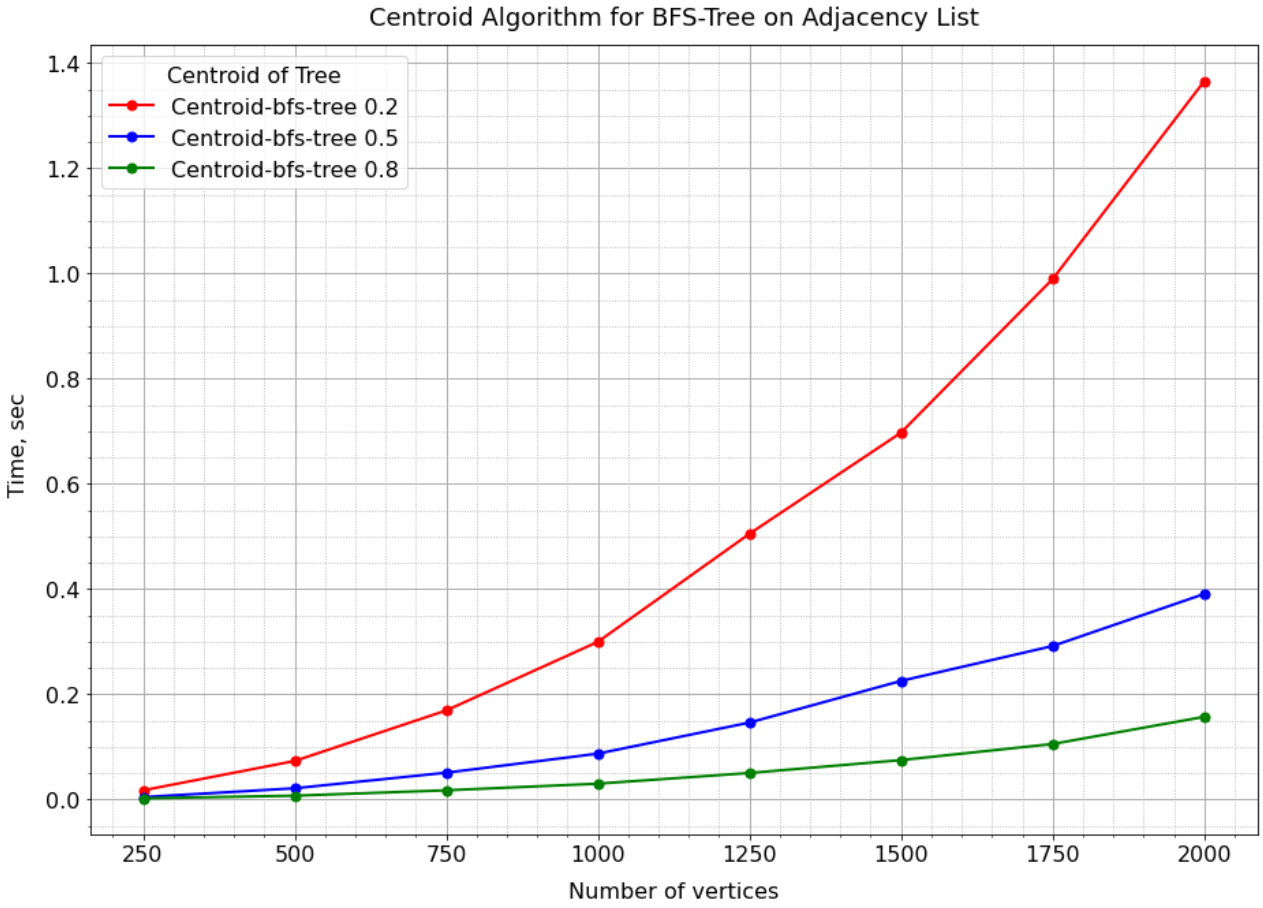


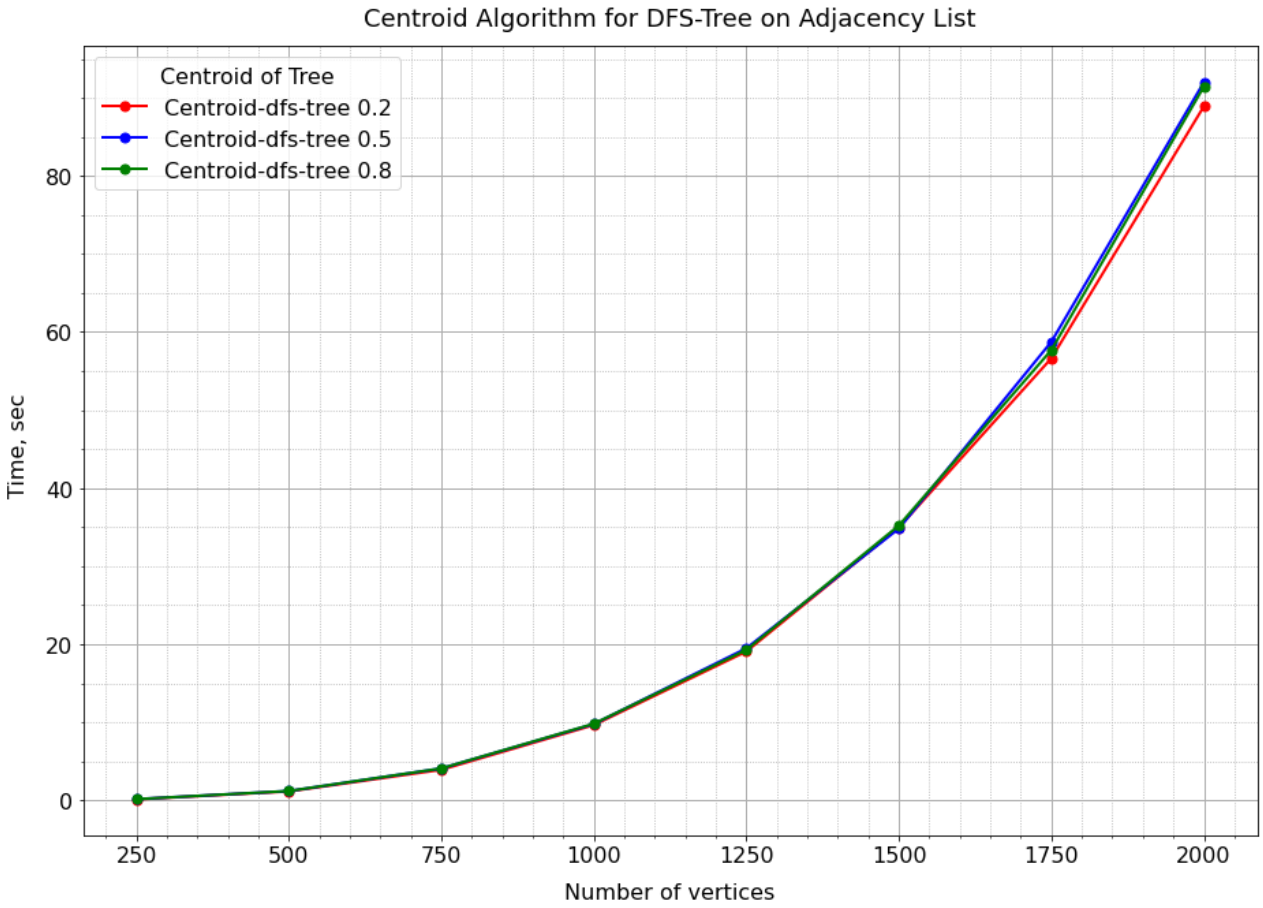
В случае насыщенного графа алгоритм поиска центра в DFS-дереве начал работать в разы хуже, что так же связанно с большой глубиной построенного каркаса графа.





В случае поиска центра в деревьях большой разницы во времени работы алгоритма в зависимости от построения каркасов BFS- и DFS- деревьев разной насыщенности графа нет.

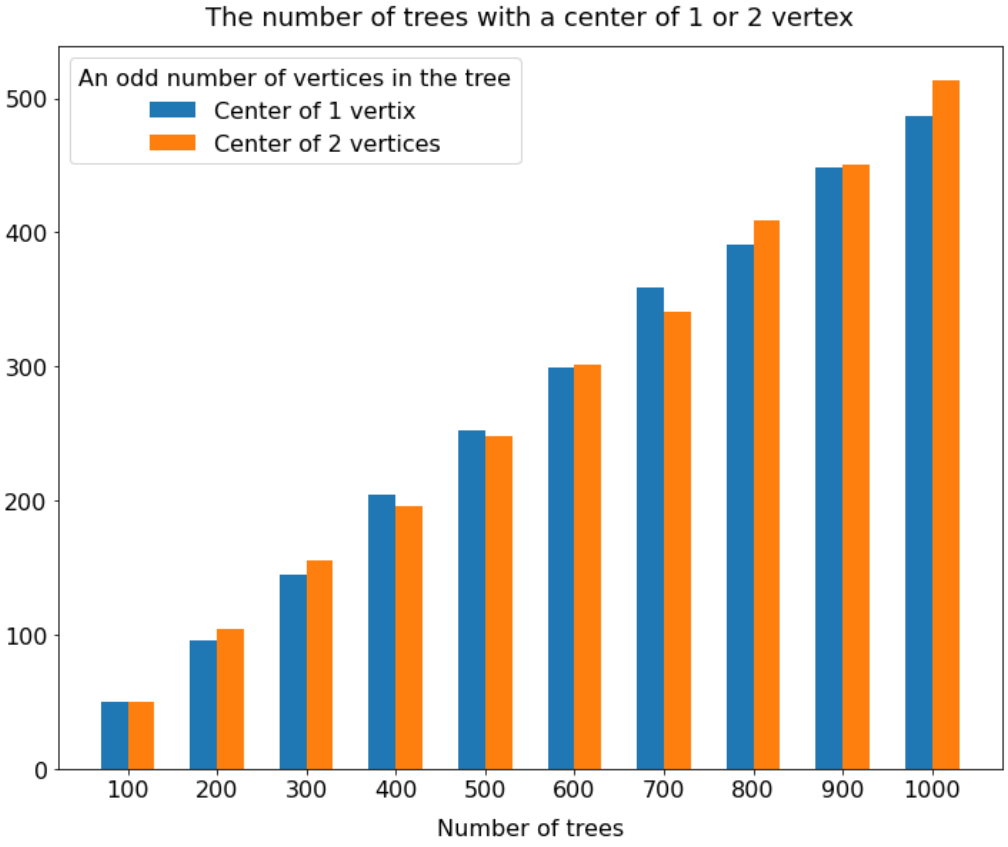


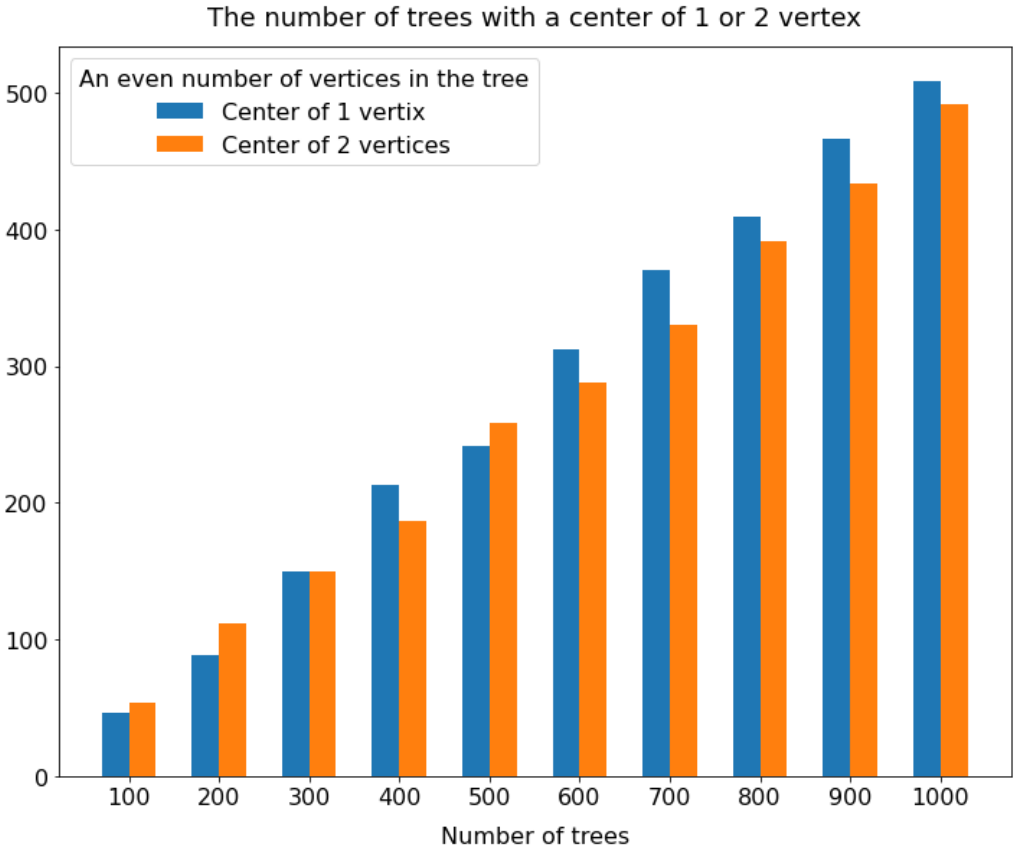


При поиске центроида в деревьях построенных, как каркасы BFS- и DFS-алгоритмов с различной насыщенностью графов есть разница. Поиск центроида в DFS-деревьях оказался

одинаковым, в отличие от поиска центроида в BFS-деревьях.

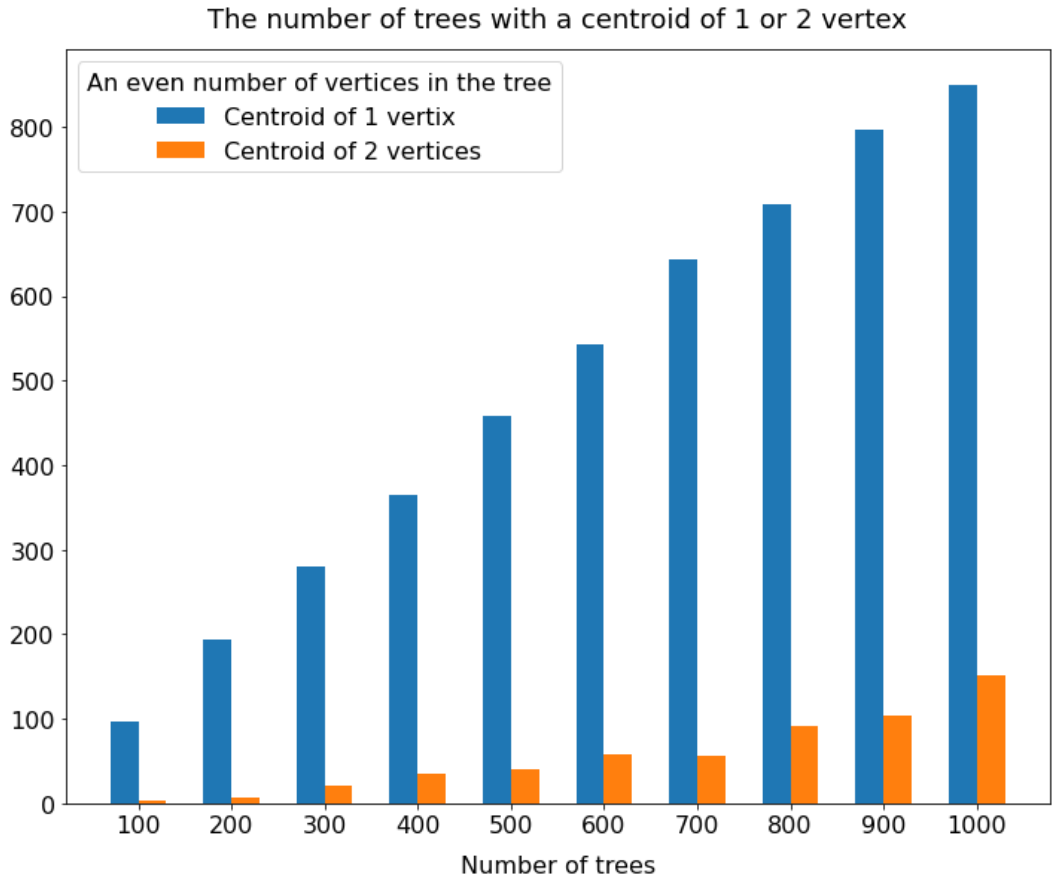
3.8 Гистограммы центра и центроида

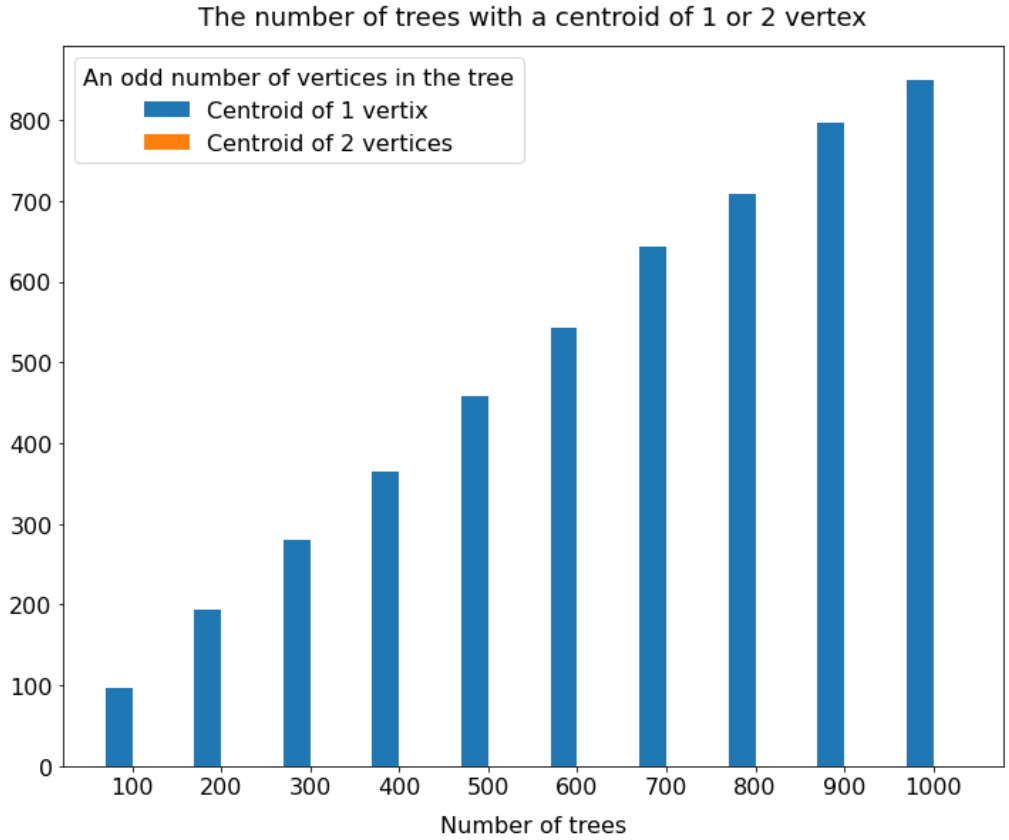




Независимо от количества вершин в дереве, вероятность появления графа с центром из

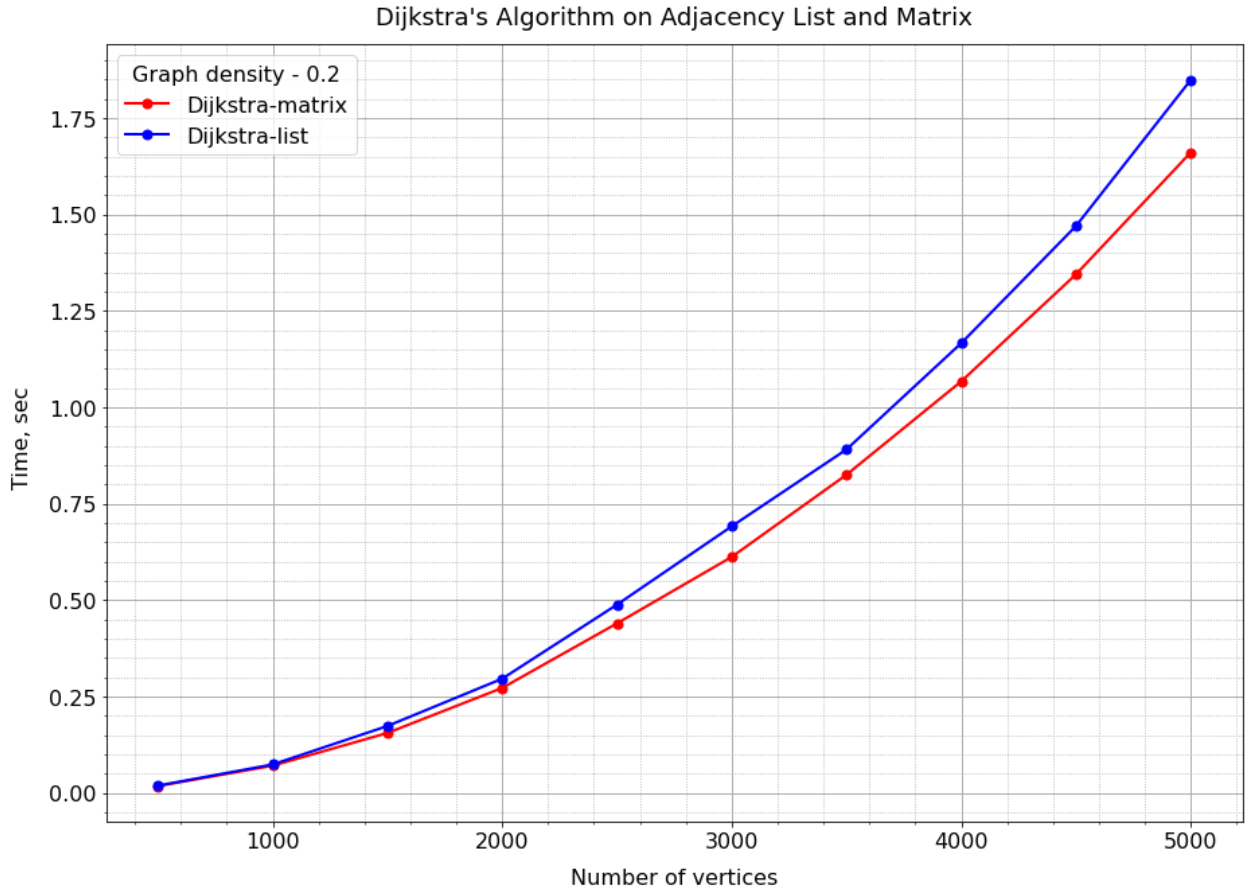
одной или из двух вершин одинакова.

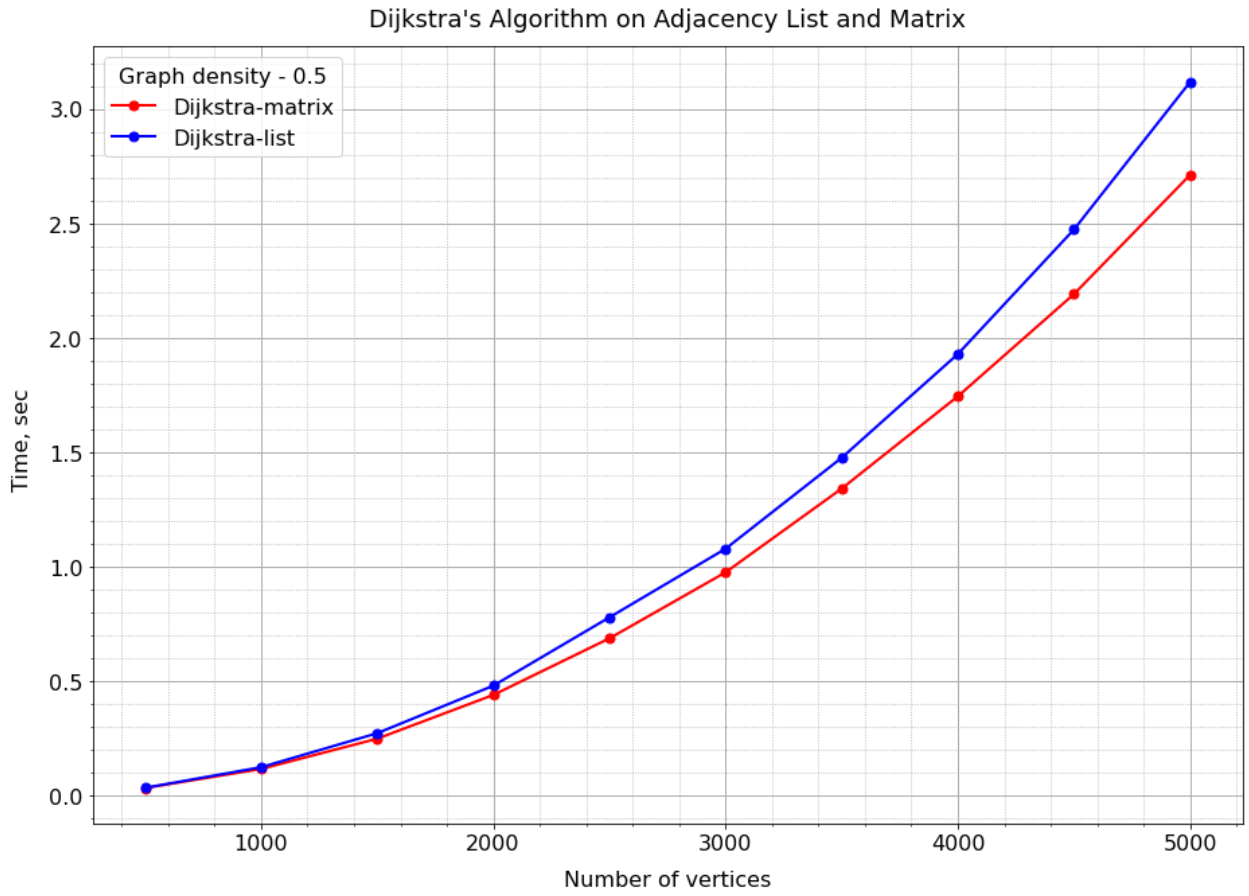


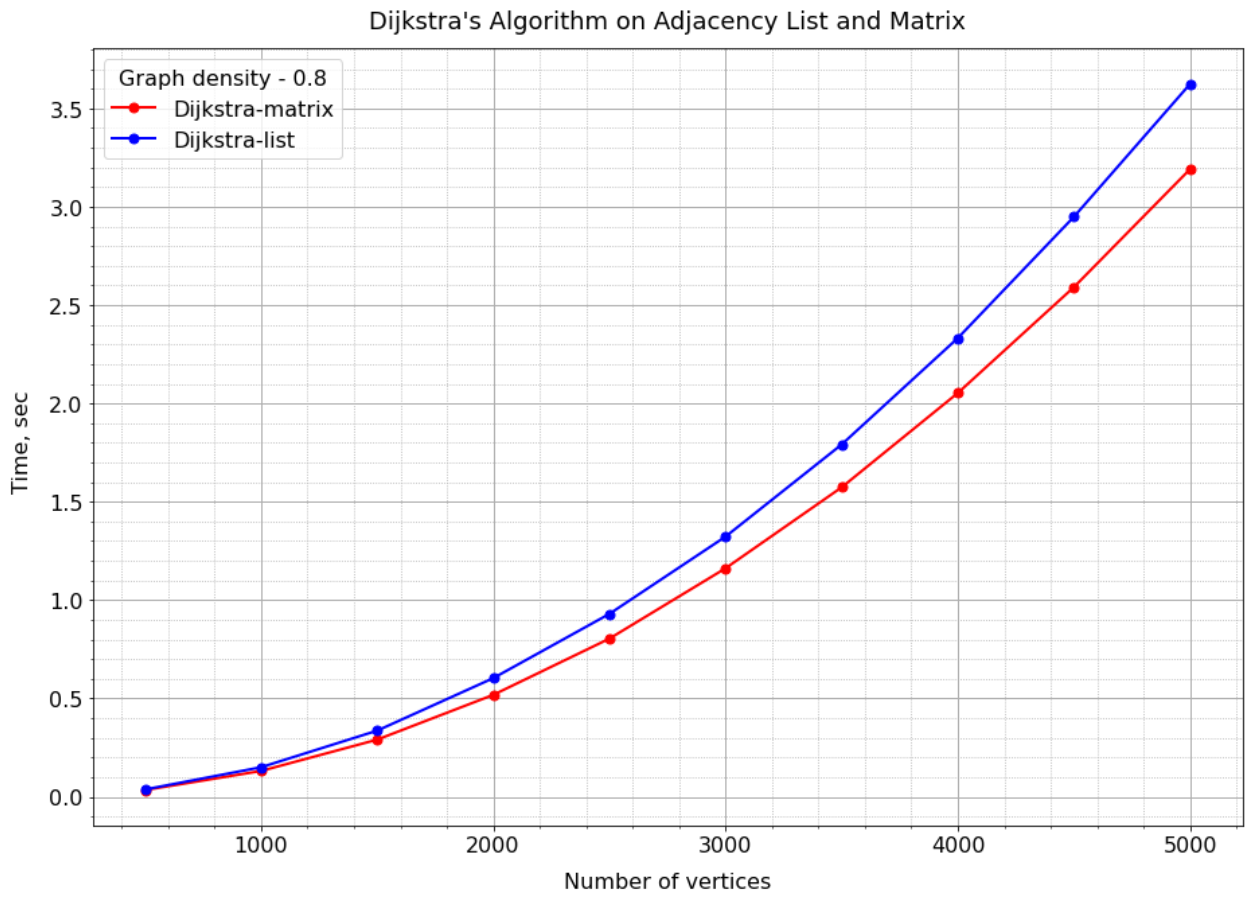


При четном числе вершин в дереве крайне редко, но встречается центроид из двух вершин. В случае нечетного числа вершин центроид состоит только из одной вершины.

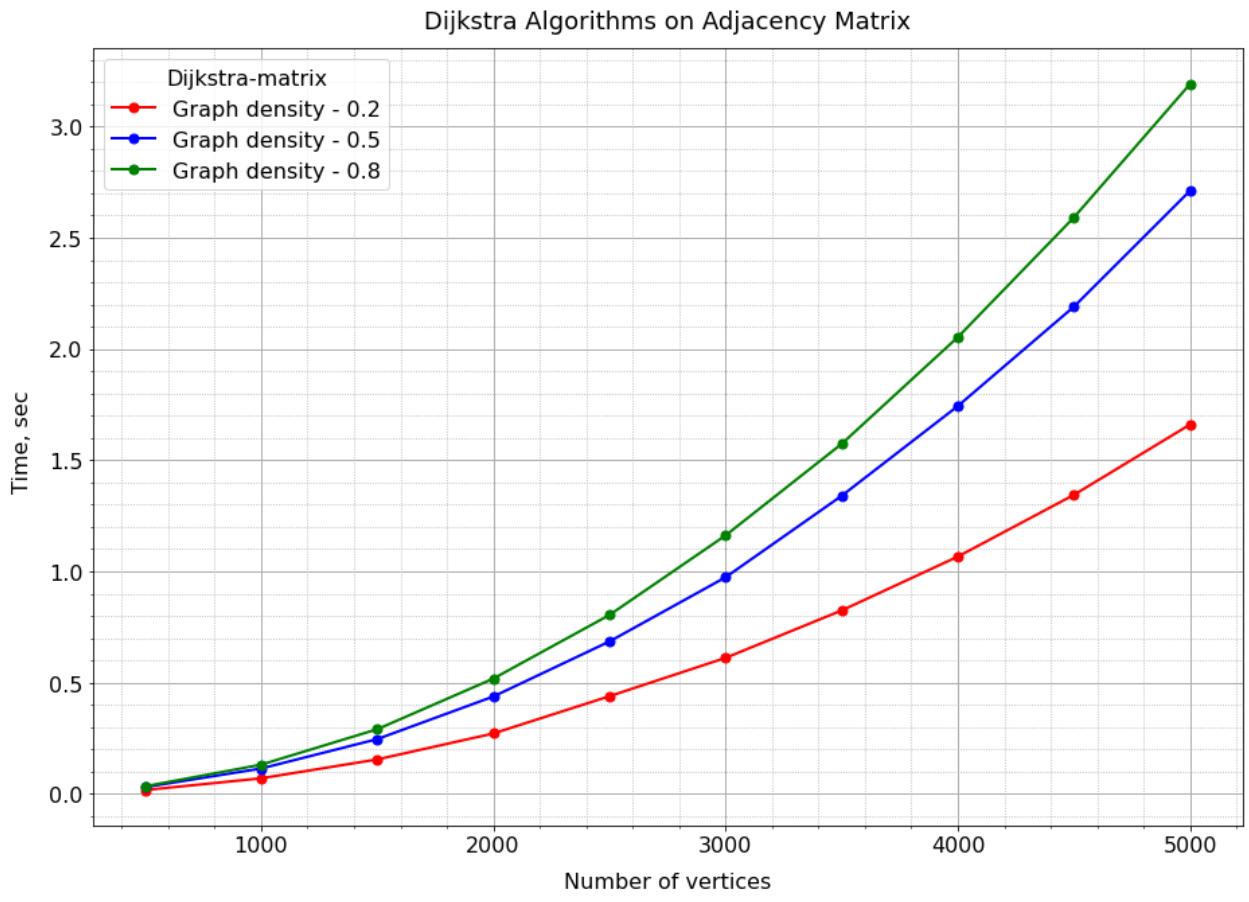
3.9 Алгоритм Дейкстры на матрице и списке смежности

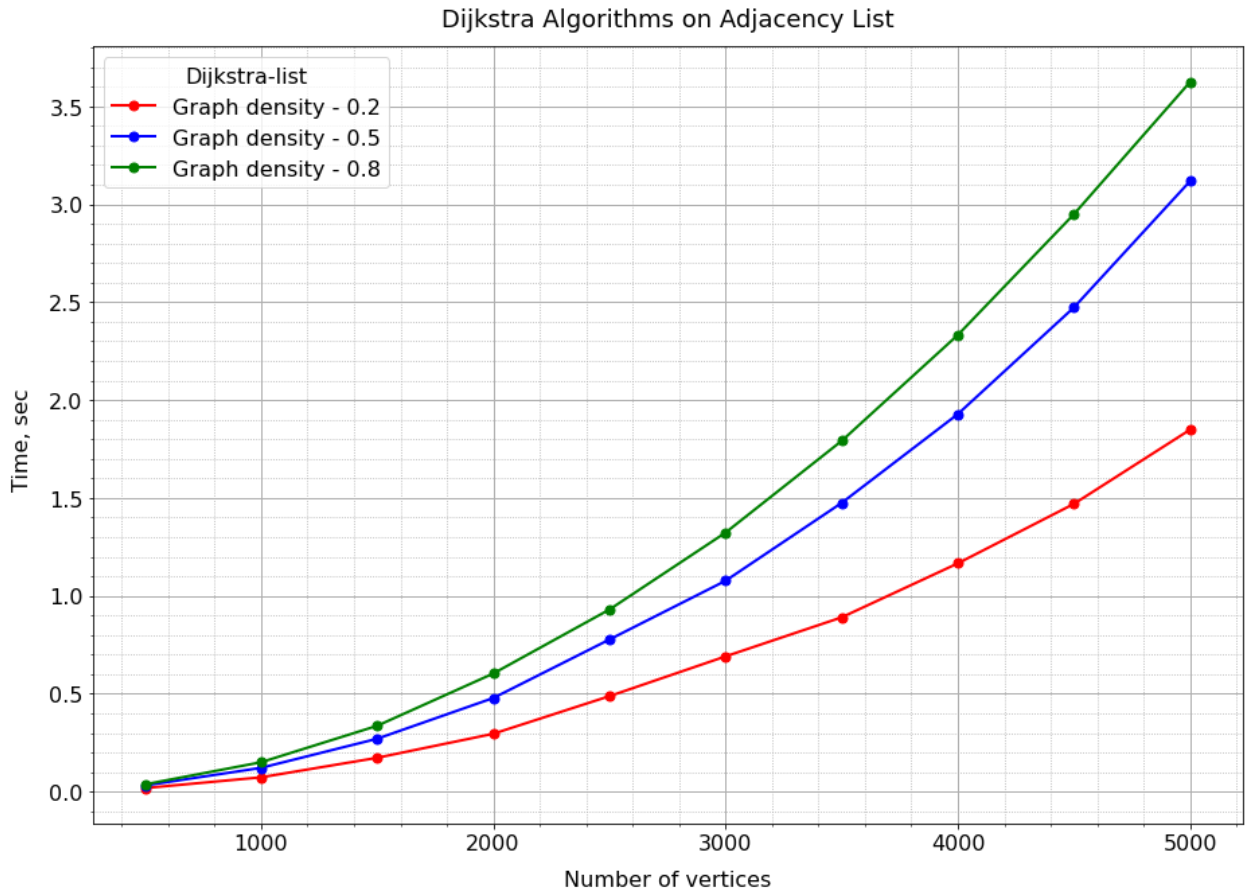






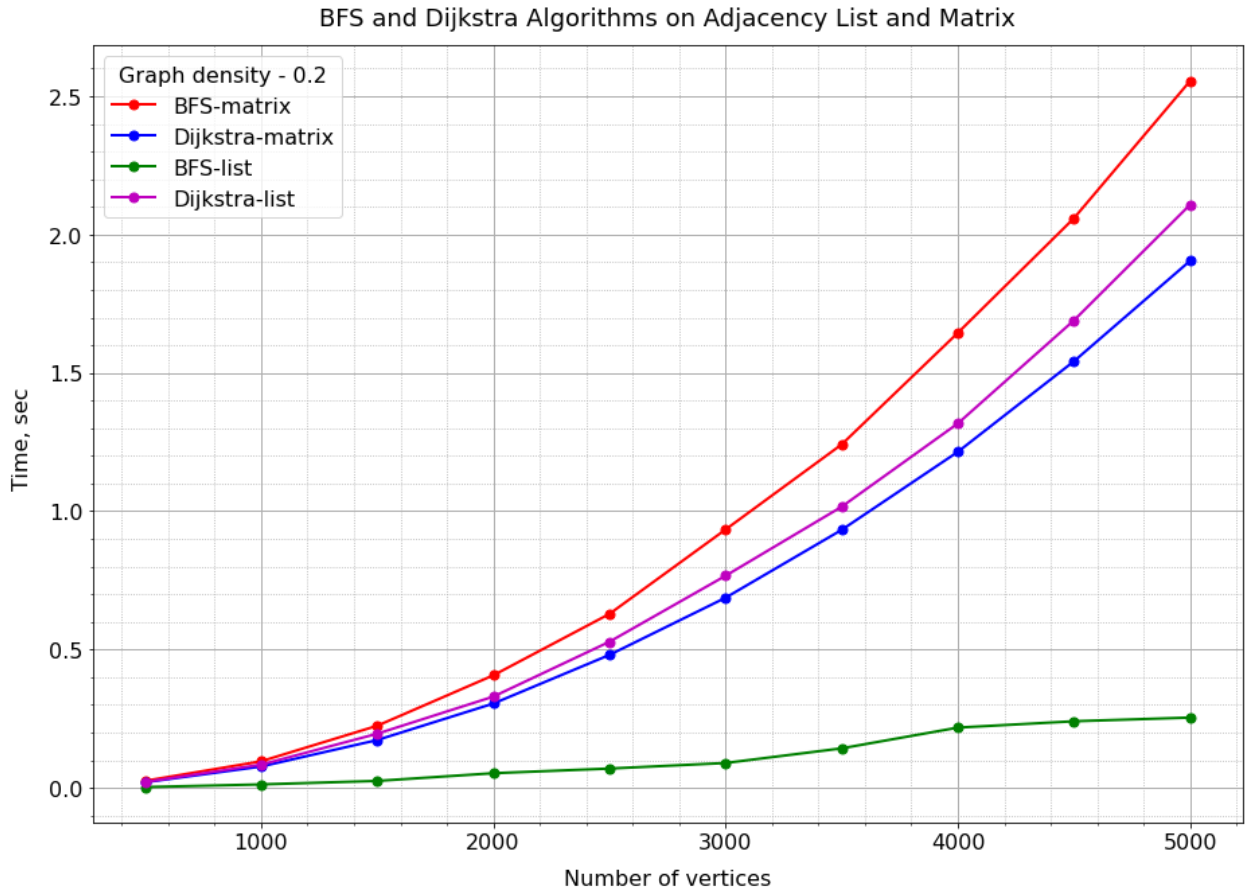
Алгоритм поиска кратчайших путей быстрее работает на матрице смежности. Разницы существенной во времени нет.



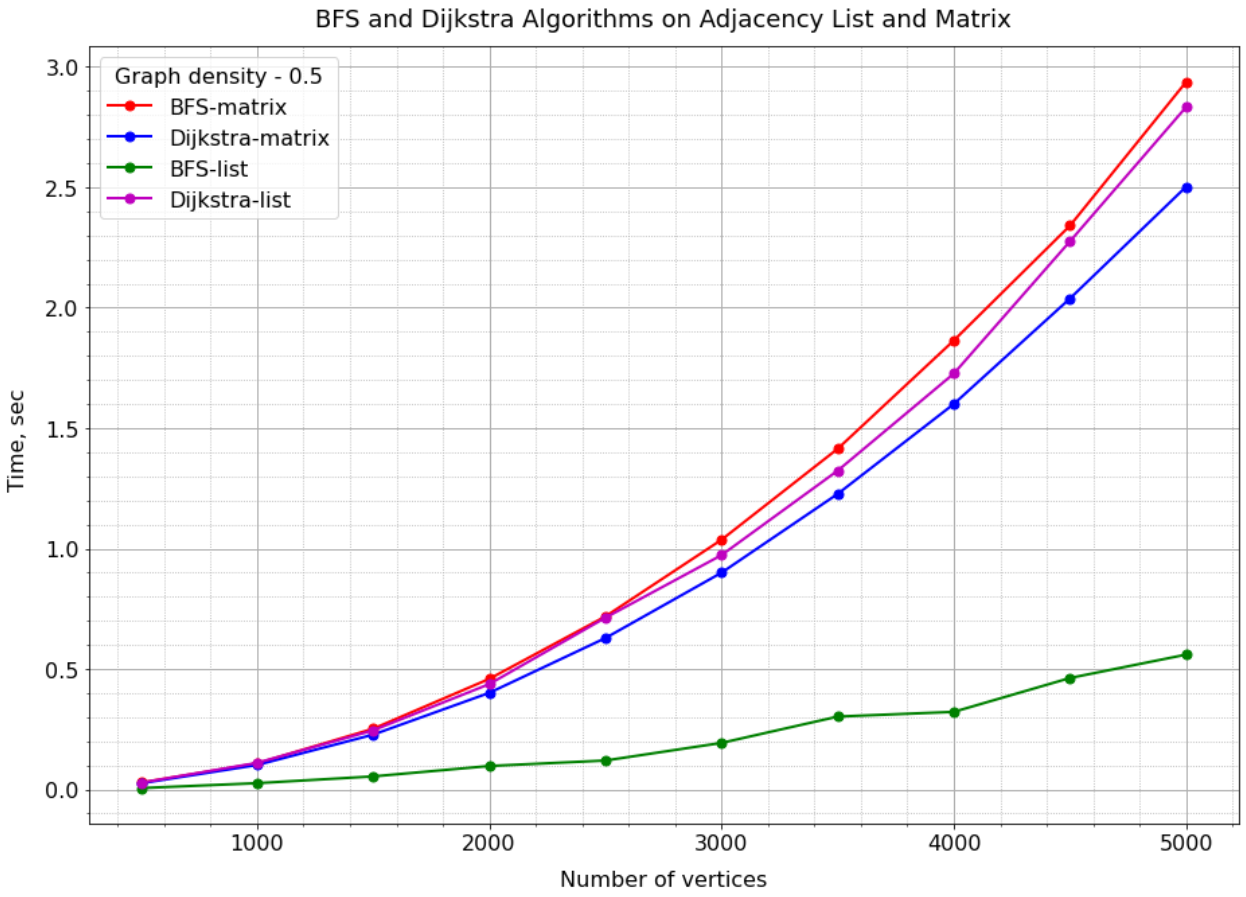


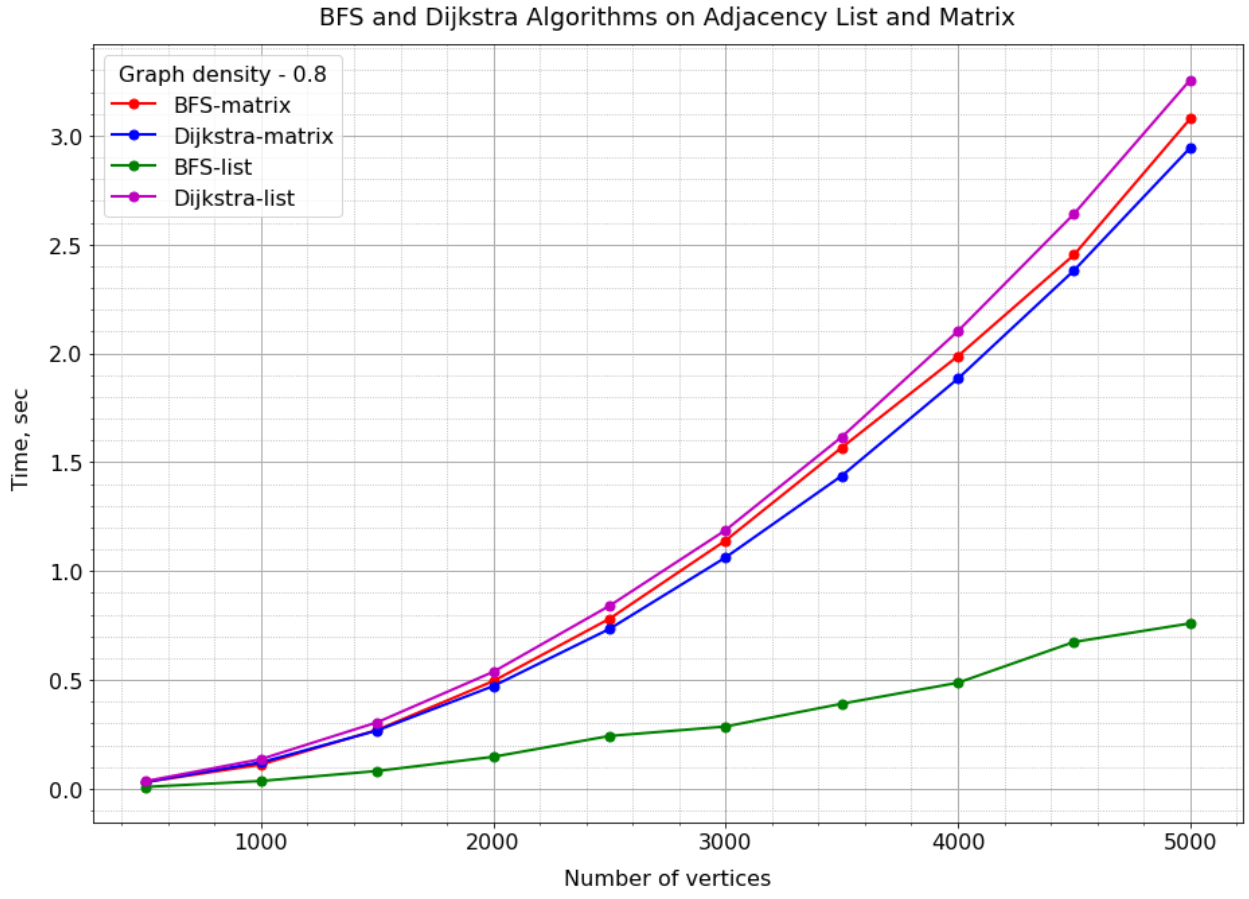
При увеличении ребер в графе алгоритм работает дольше.

3.10 Алгоритмы поиска кратчайших путей: BFS и Дейкстры на матрице и списке смежности



В случае если веса ребер одинаковые можно использовать для поиска кратчайших путей BFS-алгоритм. Лучше всего кратчайшие пути от вершины до всех остальных находит BFS-алгоритм на списке смежности. На разреженных графах алгоритм Дейкстры справился получше BFS-алгоритма на матрице смежности.



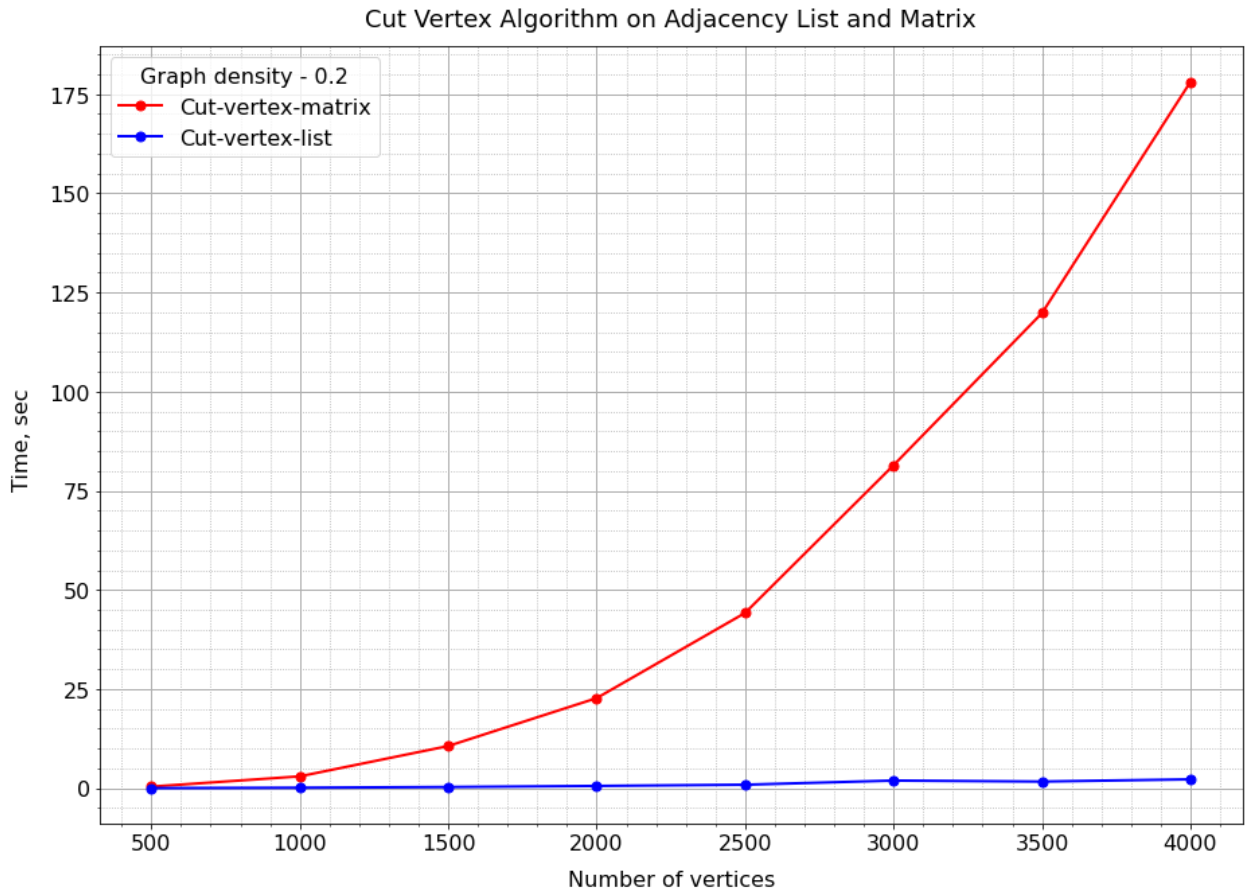


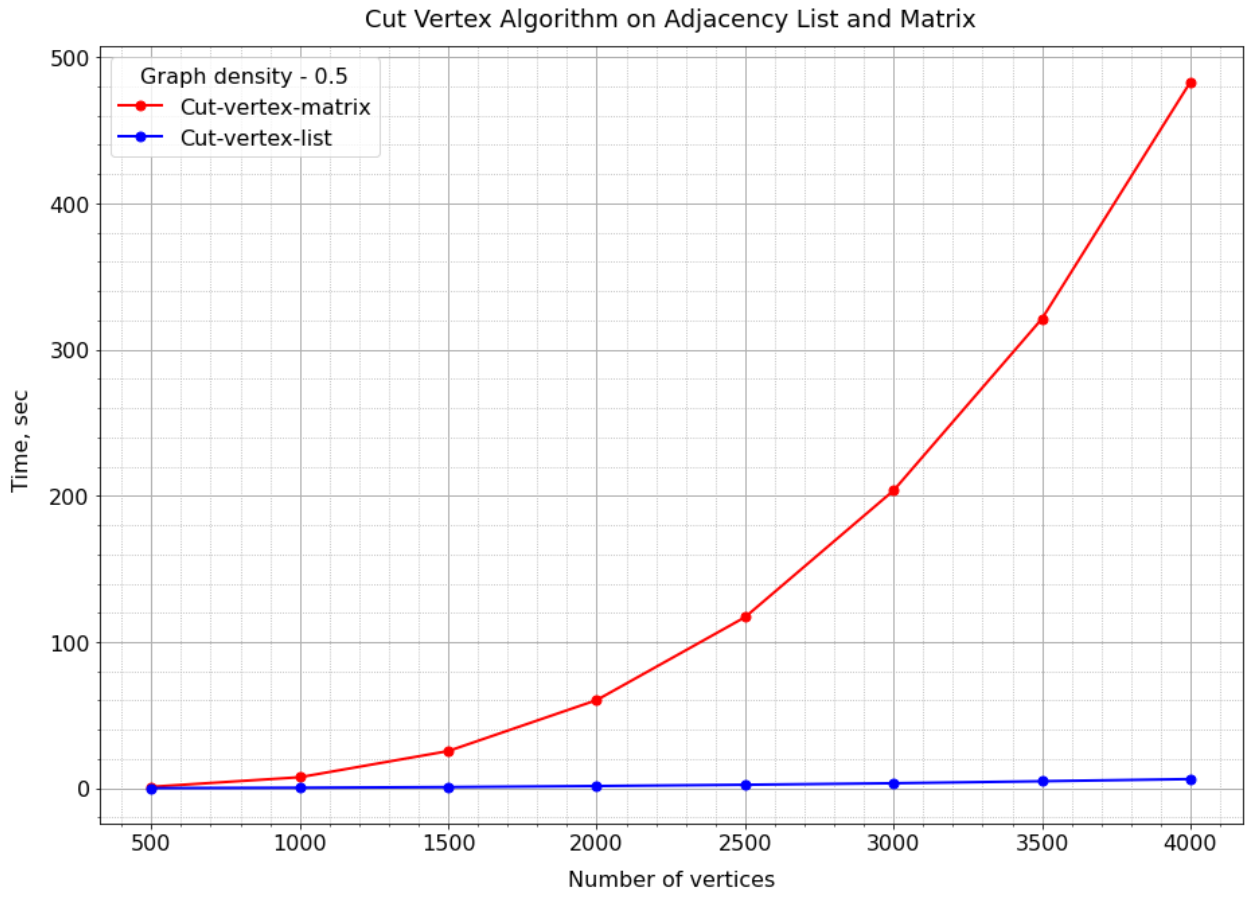
При увеличении ребер в графе BFS-алгоритм на матрице смежности начинает немного

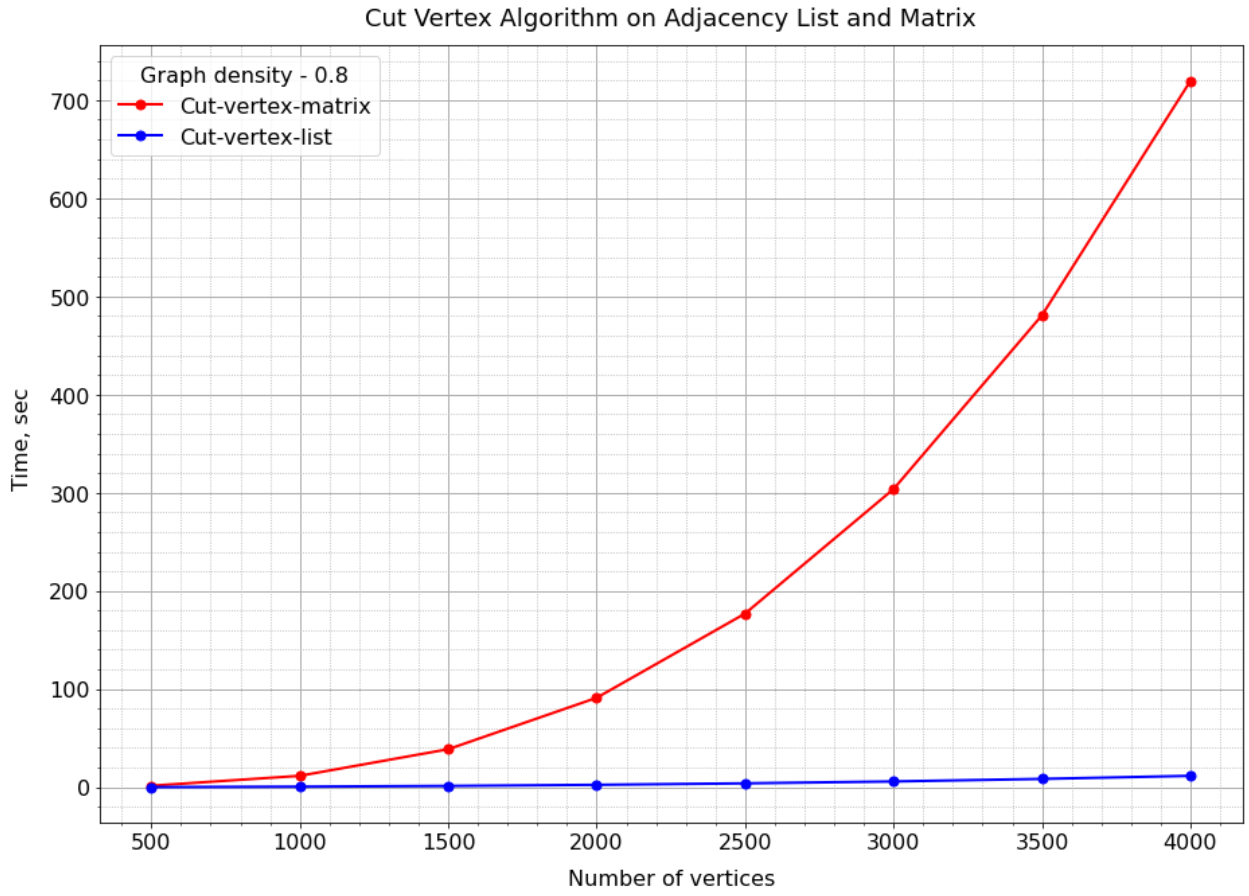
уступать алгоритму Дейкстры. В случае одинаковых весов у ребер, будет оптимальнее использовать BFS-алгоритм.

Лучше всего кратчайшие пути от вершины до всех остальных в случае одинаковых весов находит BFS- алгоритм на списке смежности.

3.11 Алгоритм поиска шарниров на матрице и списке смежности

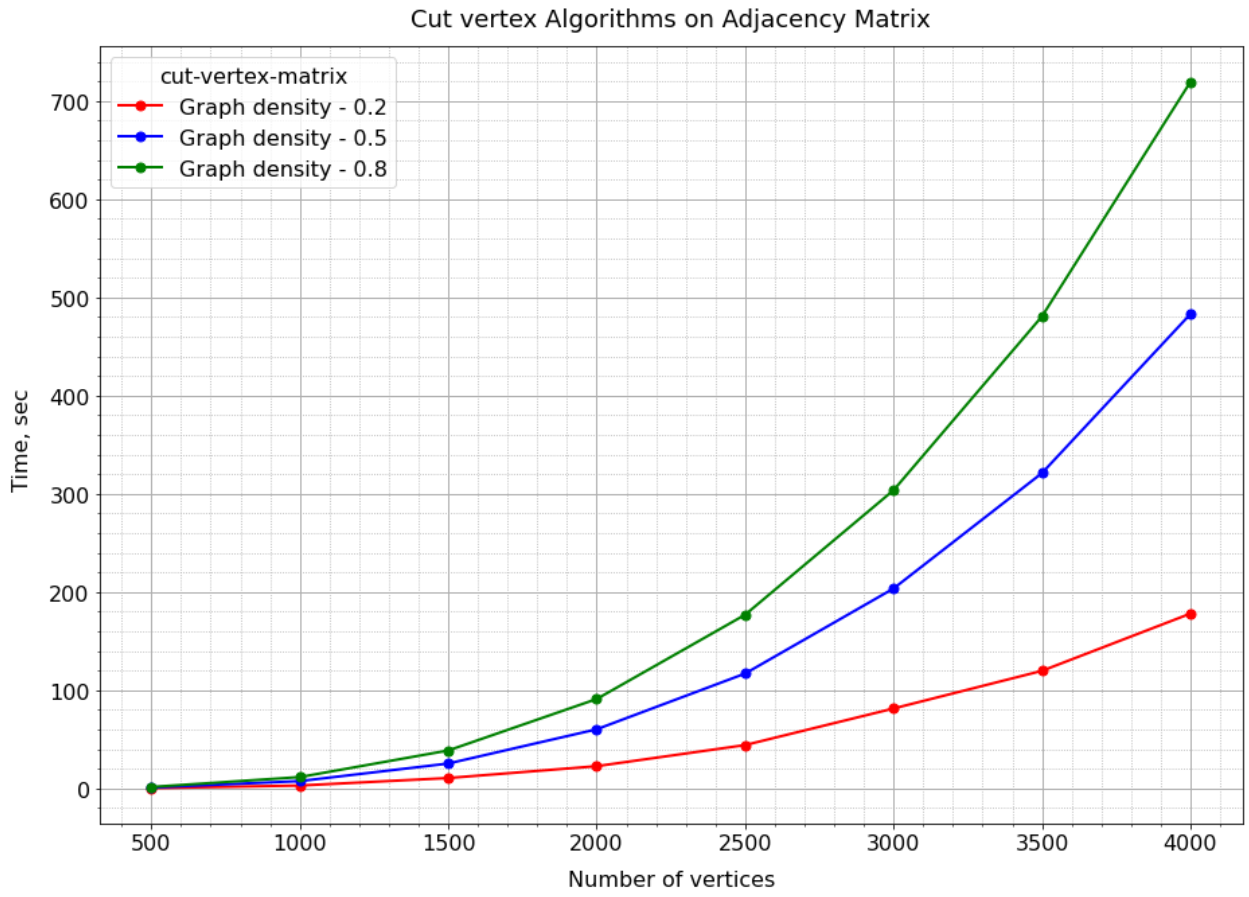


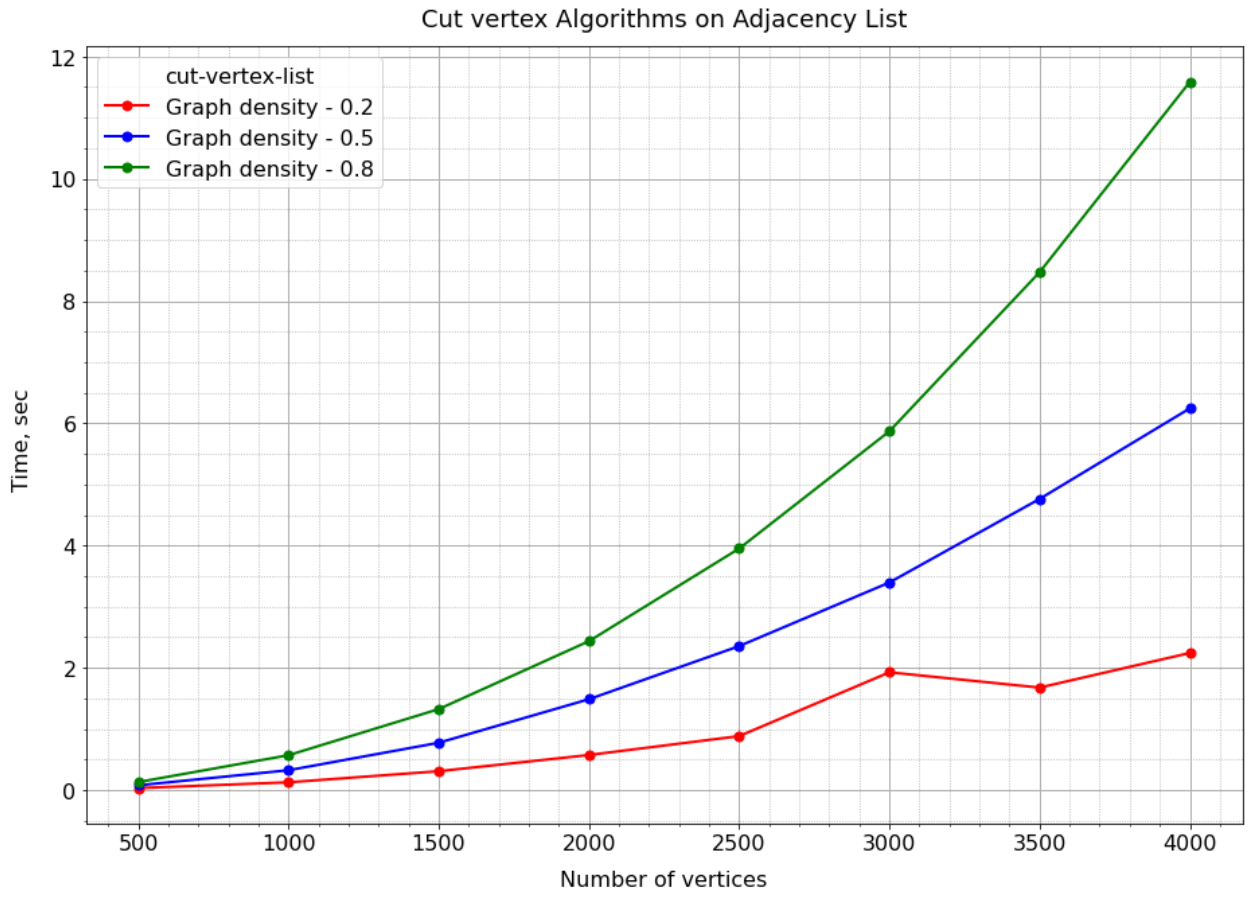




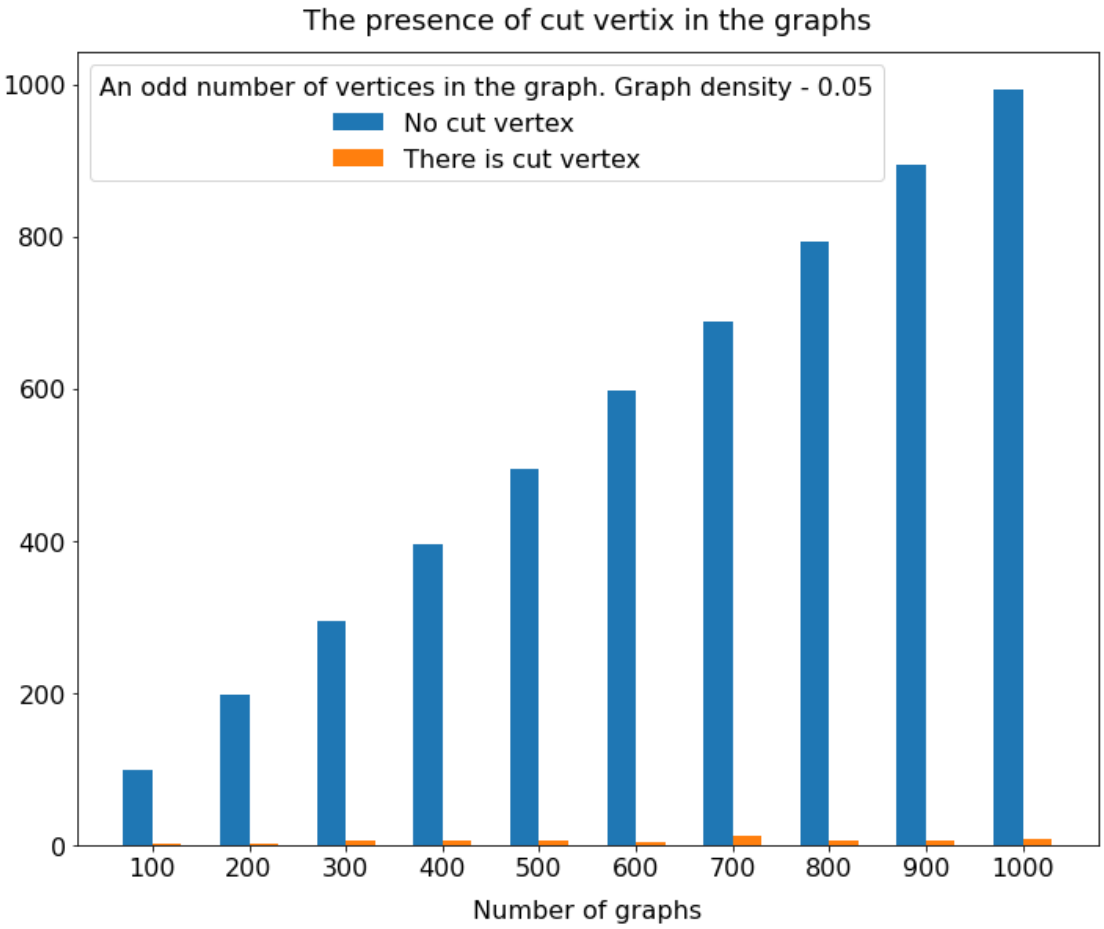
Алгоритм поиска шарниров в графе оптимальнее работает на списке смежности.

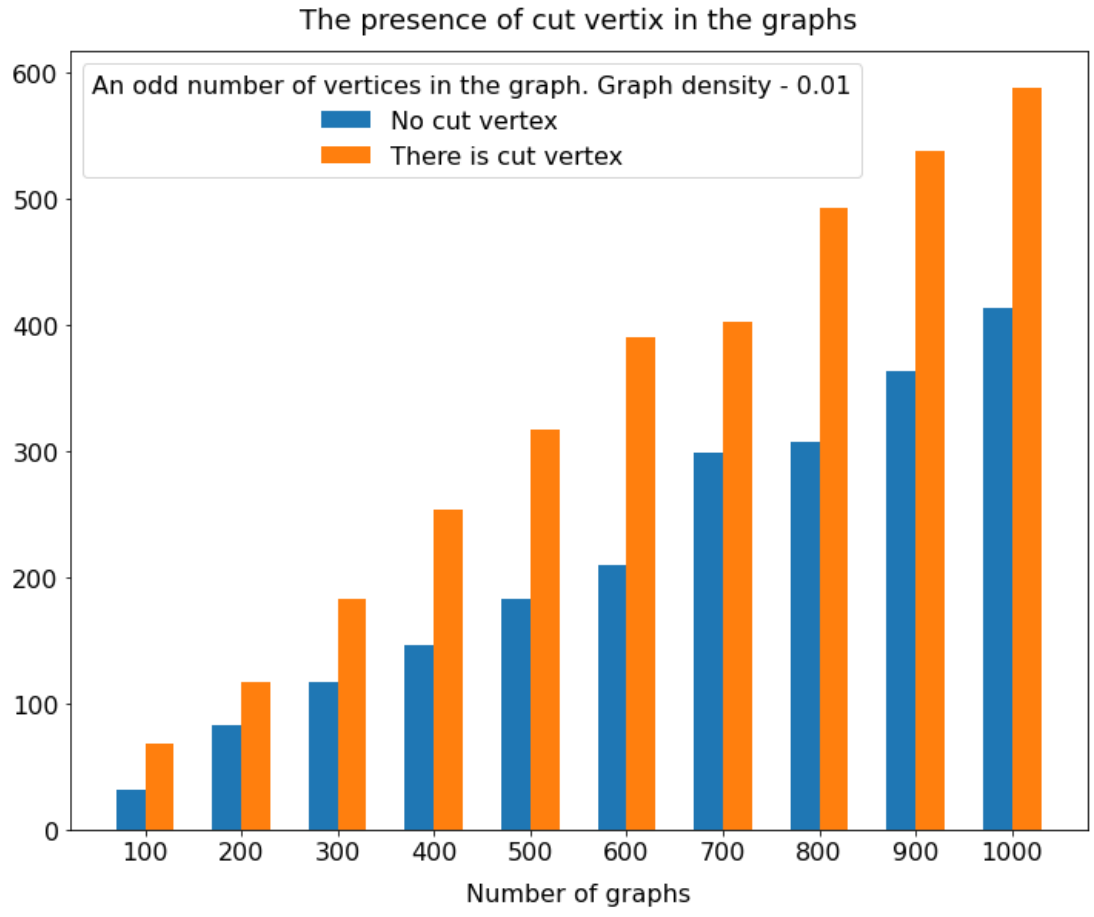
В матрице смежности больше времени уходит на поиск соседней вершины, в отличие от списка смежности.

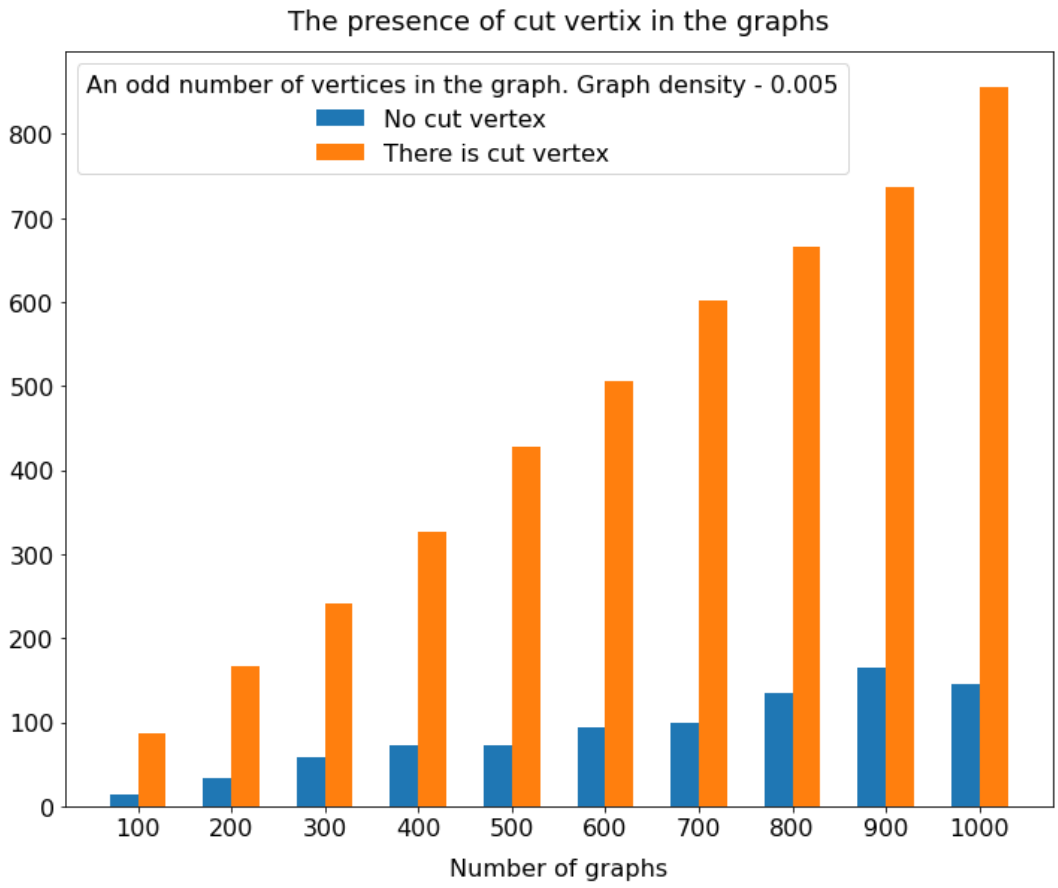




При увеличении числа ребер в графе алгоритм поиска шарниров начинает работать медленнее.

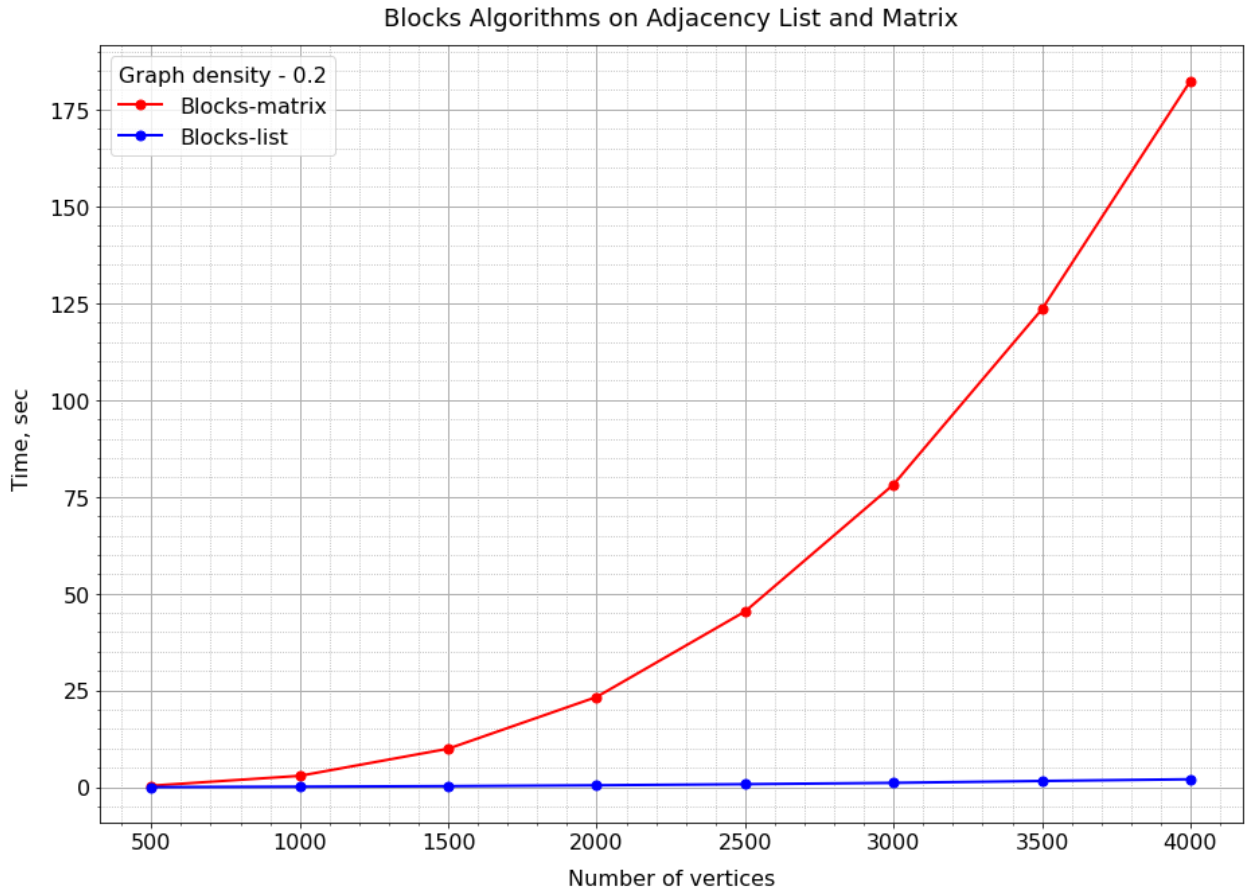

The presence of cut vertix in the graphs
An odd number of vertices in the graph. Graph density - 0.05
No cut vertex
There is cut vertex
1000
800
600
400
200
0
100
200
300
400
500
600
700
800
900
1000
Number of graphs

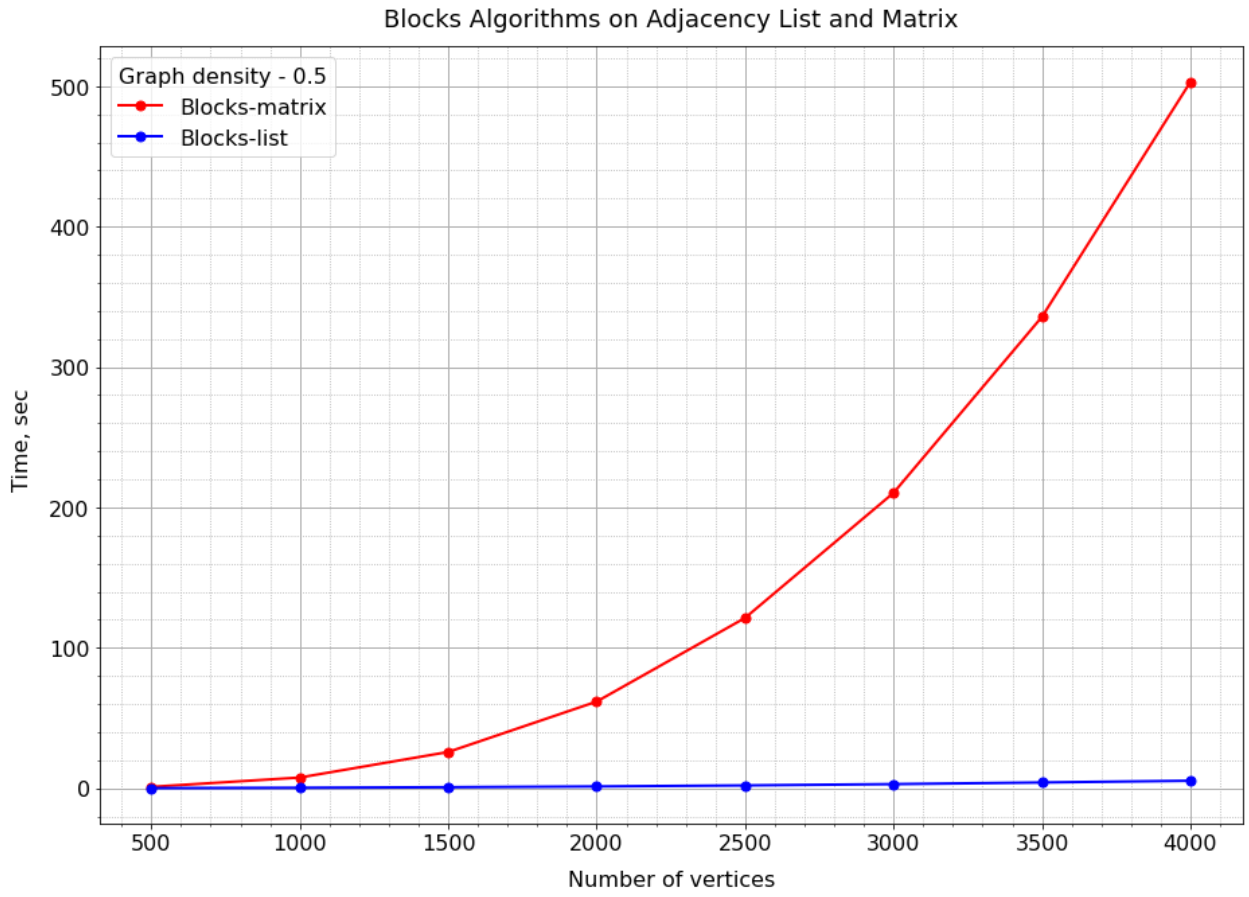

The presence of cut vertix in the graphs
An odd number of vertices in the graph. Graph density - 0.01
No cut vertex
There is cut vertex
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
900
1000
Number of graphs

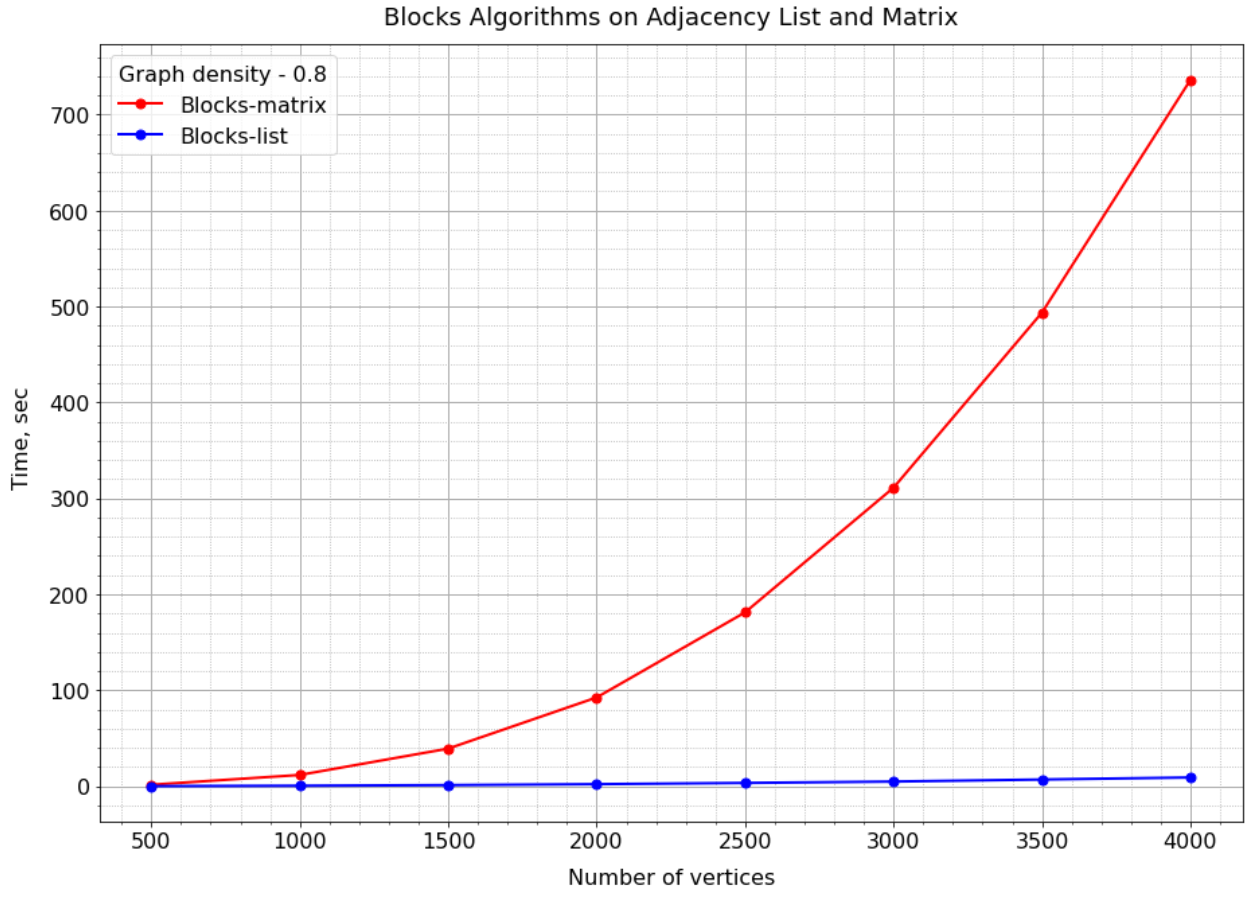


При уменьшении числа ребер в графе и отсутствии ребра между первой вершиной и последней позволяют получить большее количество шарниров, так как становится все меньше циклов.

3.12 Алгоритм поиска блоков на матрице и списке смежности

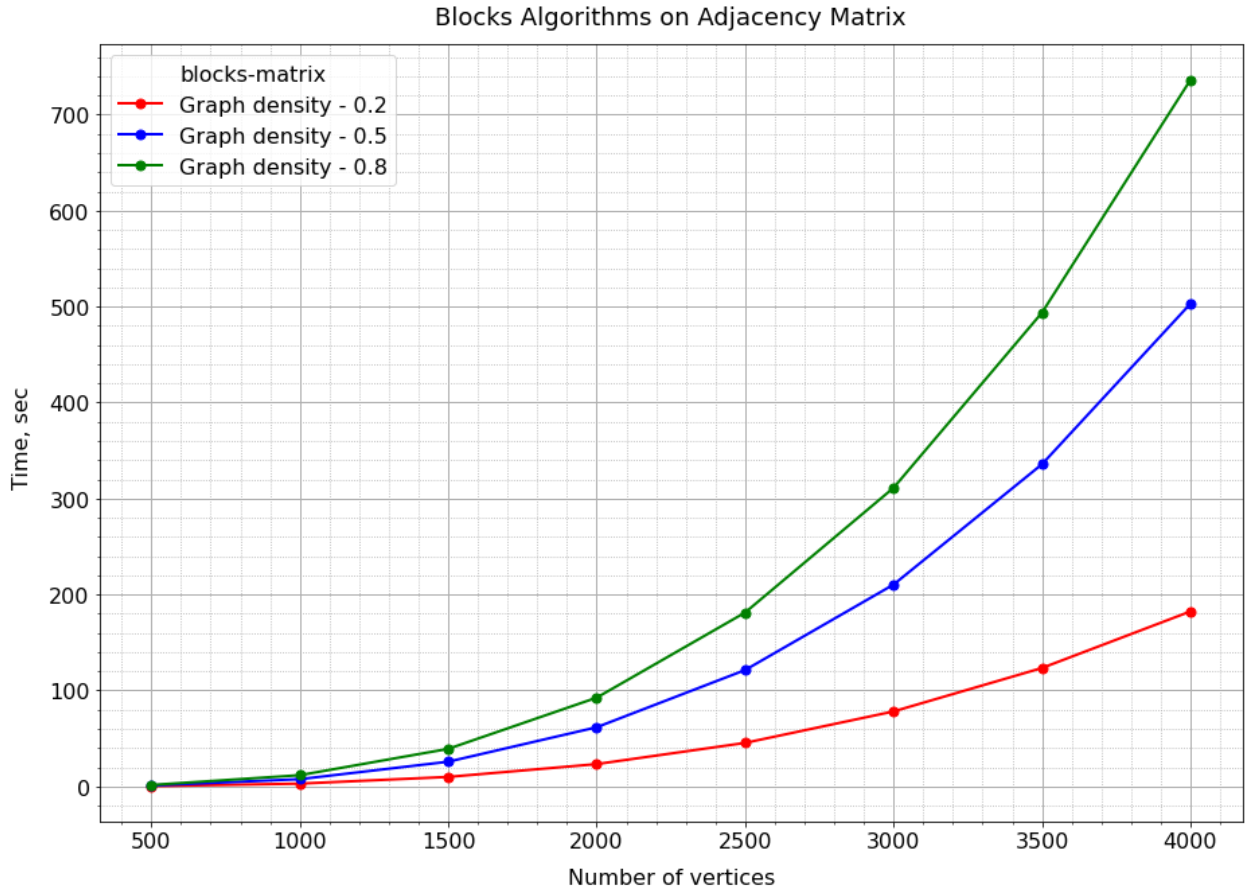


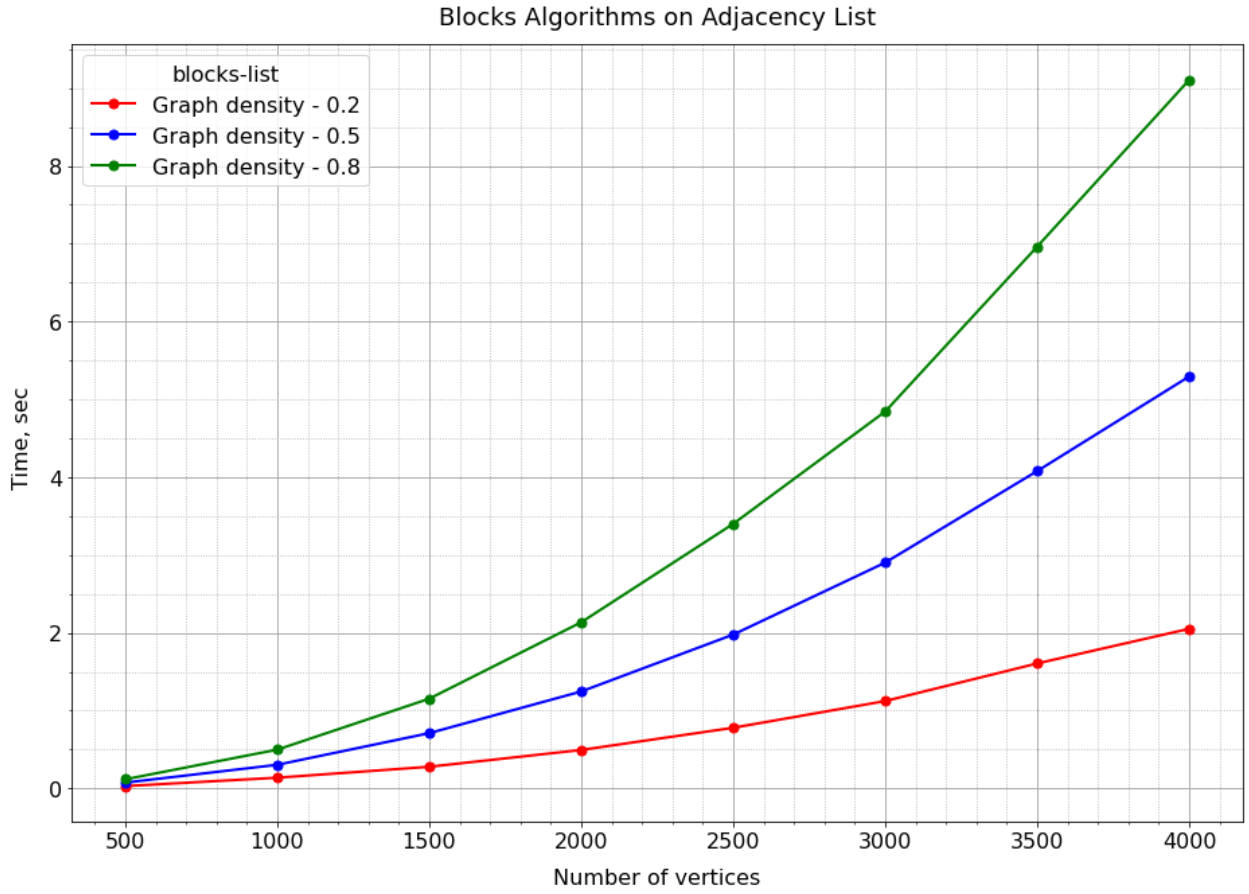




Алгоритмы поиска блоков в графе работают быстрее на списках смежности. Преимущества из-за поиска

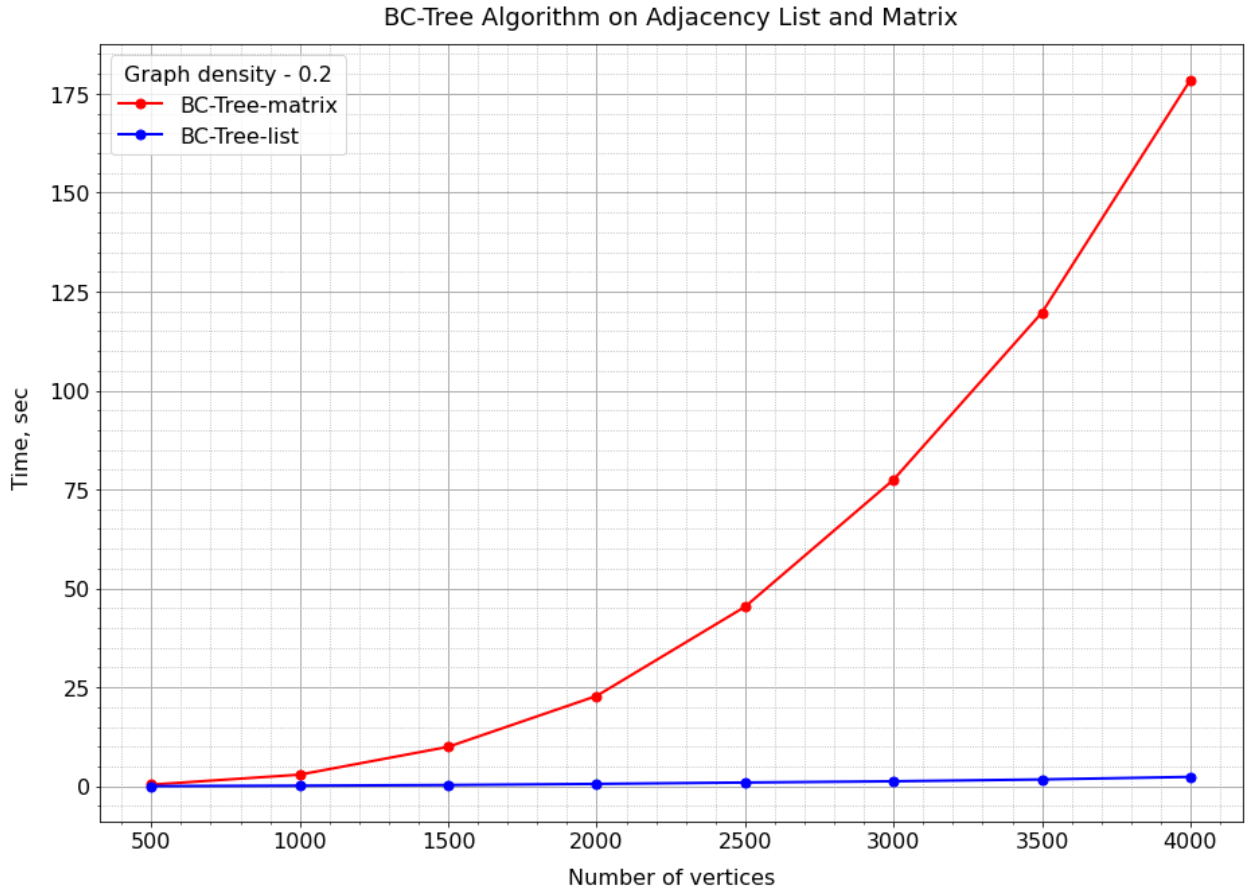
шарниров в графе.

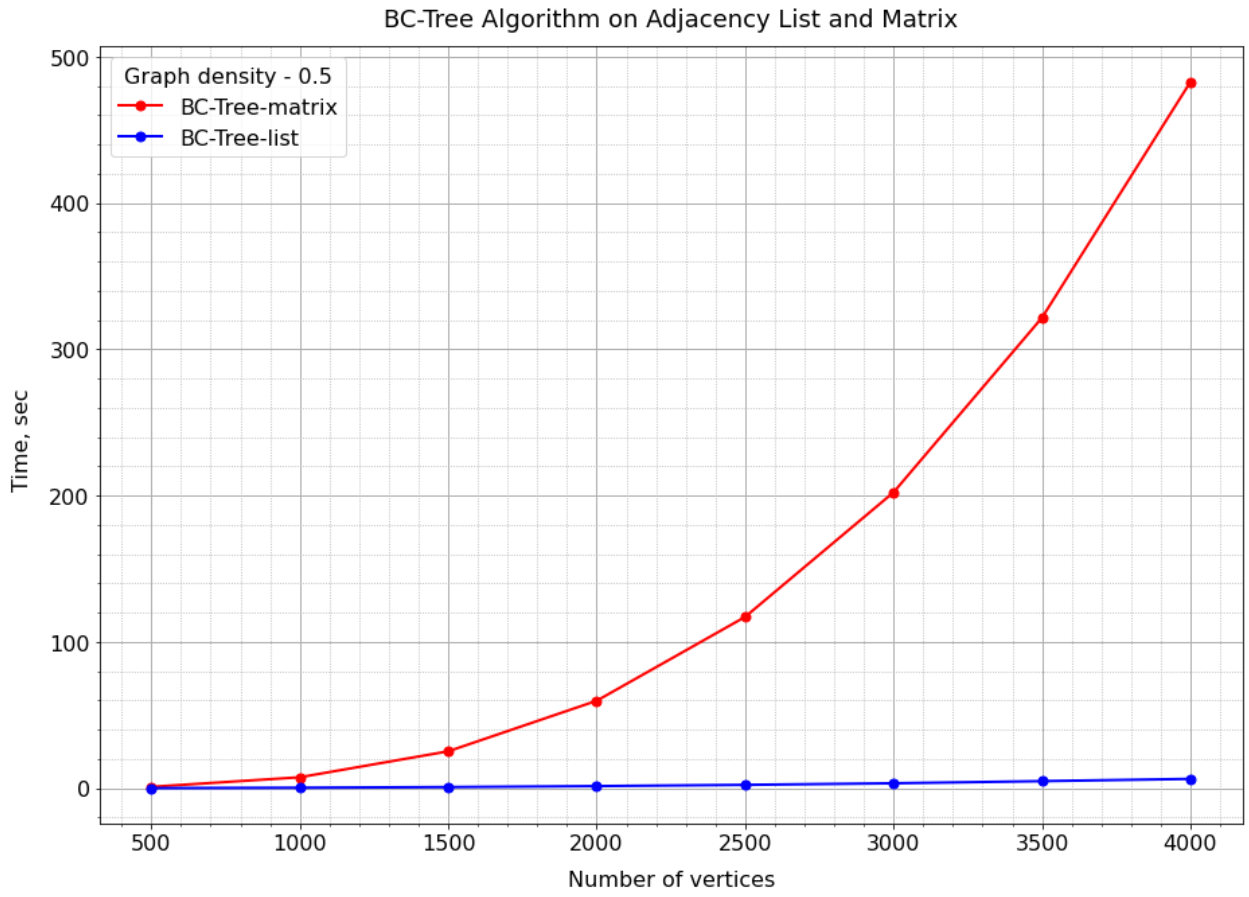


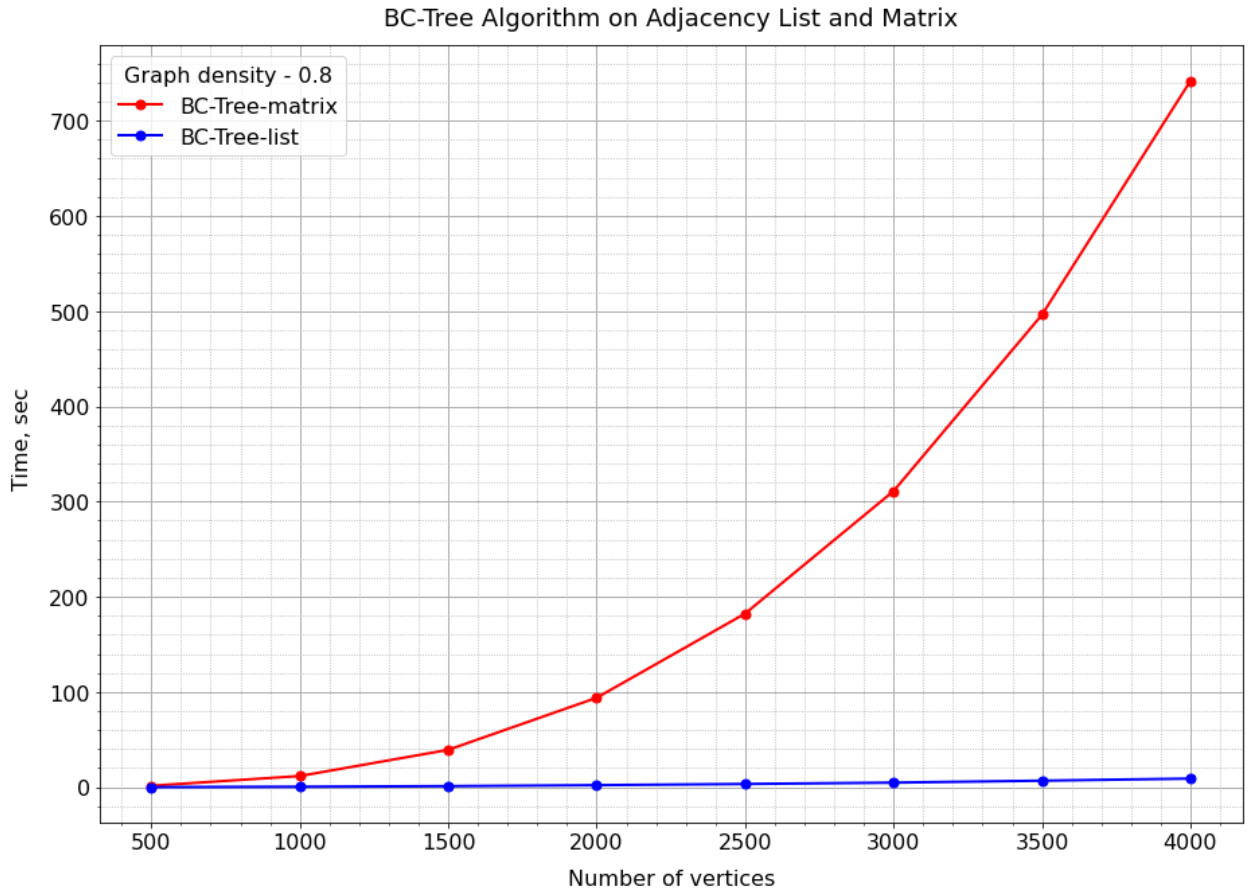


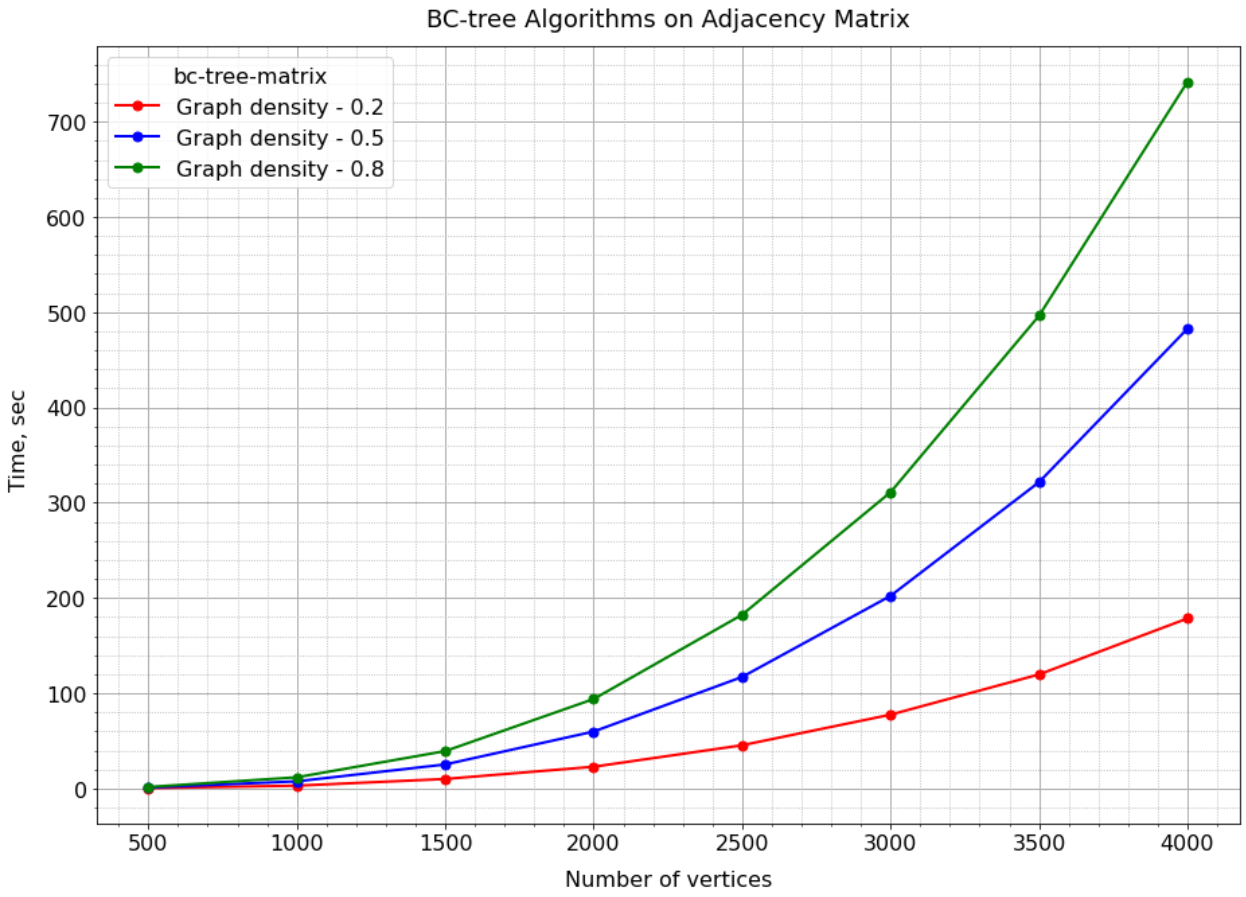
При увеличении числа ребер в графе алгоритм стал работать медленнее на обоих структурах данных, что вполне очевидно, такая же ситуация была и с DFS-алгоритмом, а в основе этого алгоритма и лежит обход графа в глубину и поиск шарниров.

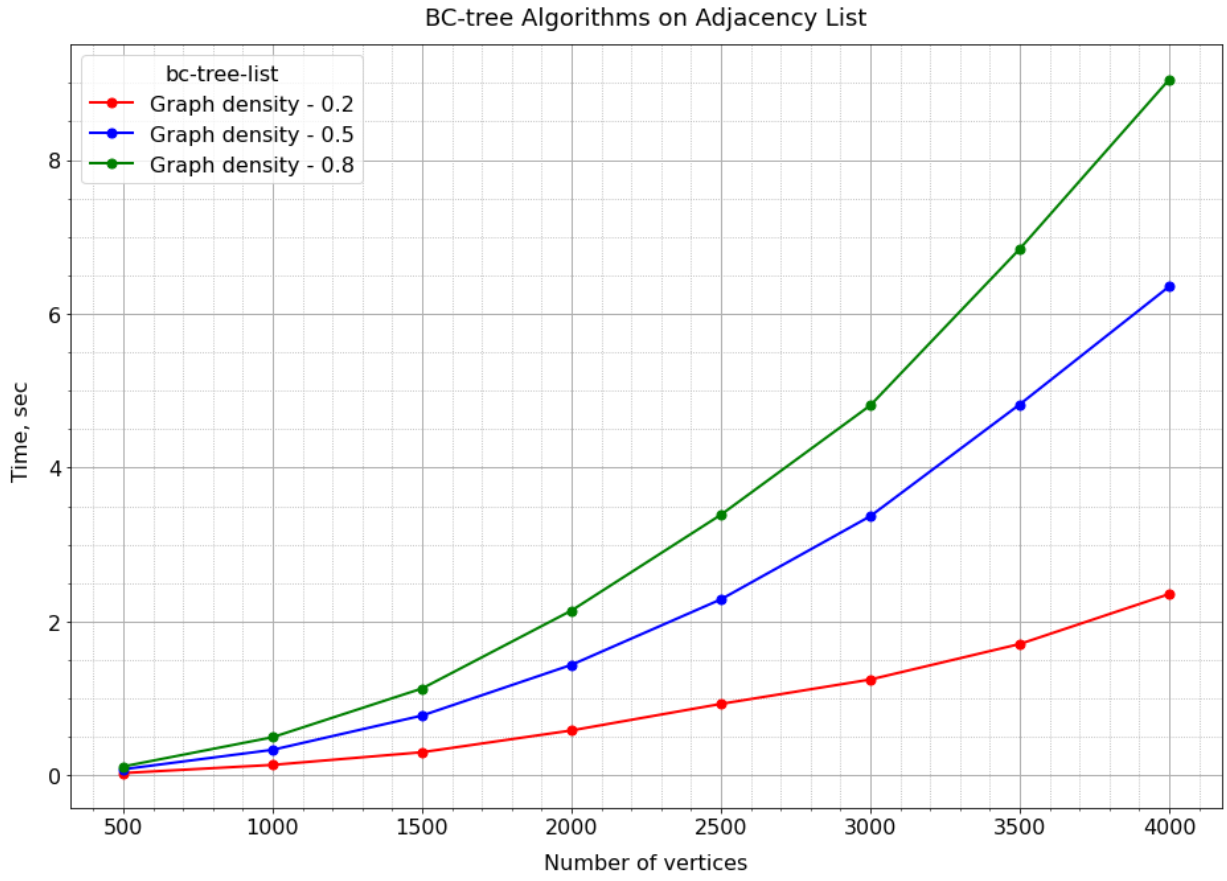
3.13 Алгоритм построения BC-tree на матрице и списке смежности




BC-Tree Algorithm on Adjacency List and Matrix
Graph density - 0.5
BC-Tree-matrix
BC-Tree-list
Time, sec
Number of vertices
0
100
200
300
400
500
1000
1500
2000
2500
3000
3500
4000


BC-Tree Algorithm on Adjacency List and Matrix
Graph density - 0.8
BC-Tree-matrix
BC-Tree-list
Time, sec
Number of vertices
0
100
200
300
400
500
600
700
1000
1500
2000
2500
3000
3500
4000





Аналогичная ситуация с алгоритмом поиска шарниров в графе, так как этот алгоритм лежит в основе.

Заключение

Алгоритмы поиска в ширину и глубину для построения деревьев из произвольных графов работают почти одинаково на обоих структурах хранения. На данной выборке графов эффективнее оказался алгоритм поиска в ширину.

В зависимости от количества ребер работа алгоритмов DFS и BFS на матрице смежности заметно увеличена в случае более насыщенного графа. Список смежности работает медленнее в зависимости от количества ребер, т.к при увеличении числа ребер список расширяется. Алгоритм DFS уступает BFS алгоритмы, так как есть необходимость возврата. Для работы с алгоритмами BFS и DFS оптимальнее использовать список смежности.

Алгоритм поиска кратчайших путей в разы эффективнее выполняется. на матрице смежности. При увеличении числа ребер в графе алгоритм начинает замедляться как на списке смежности, так и на матрице смежности. Стоит попробовать использовать приоритетную очередь в алгоритме, чтобы сократить время выполнения. В случае графа с ребрами одинаковых весов лучше всего использовать BFS-алгоритм для списка смежности.

Алгоритм поиска шарниров и блоков в графе показал вполне ожидаемые результаты схожие с результатами алгоритма построения DFS-дерева, так как в основе алгоритма поиска лежит обход графа в глубину. Построение ВС-дерева напрямую зависит от работы алгоритмов поиска шарниров и блоков в графе. Алгоритм поиска центра быстрее всего работает на списке смежности. При поиске центроида в дереве следует отдать предпочтение списку смежности.

Алгоритм поиска центра работает быстрее в отличие от поиска центроида в дереве, так как не нужно вычислять веса вершин. При четном числе вершин в дереве крайне редко, но встречается центроид из двух вершин. В случае нечетного числа вершин центроид состоит только из одной вершины.

Независимо от количества вершин в дереве, вероятность появления графа с центром из одной или из двух вершин одинакова.

Список используемых источников

[1] Алексеев, В. Е. Графы. Модели вычислений. Структуры данных: Учебник / В. Е. Алексеев, В. А. Таланов. — Нижний Новгород: изд-во Нижегородского ННГУ. — 2005. — 250 с. 4

[2] Алексеев, В. Е, Захарова Д. В. О симметрических пространствах графов, Дискретн. анализ и исслед. опер., сер. 1, 14:1/ В. Е. Алексеев, Д. В Захарова. 2007, 21–26 с; J. Appl. Industr. Math.,2:2 2008, 151–154

[3] Алябышева, Ю. А. Алгоритм Дейкстры поиска кратчайшего пути в графе с использованием строковых матриц смежности / Ю. А. Алябышева // Сборник трудов Всероссийской конференции по математике с международным участием "МАК-2018" / АлтГУ [и др.]. — Барнаул : Изд-во АлтГУ. — 2018. — С. 223–227.

[4] Апанович, З. В. Визуализация больших графов и матрицы смежности / З. В. Апанович // Научный сервис в сети Интернет: труды XX Всероссийской научной конференции (Новороссийск, 17-22 сентября 2018 г.). — М.: ИПМ им. М.В. Келдыша. — 2018. — С. 28-41. — URL: http://keldysh.ru/abrau/2018/theses/22.pdfdoi:10.20948/abrau-2018-22, свободный.

[5] Белов, Ю. А. Уточнение свойств центроида дерева / Ю. А. Белов, С. И. Вовчок // Моделирование и анализ информационных систем. — 2017. — № 24 (4). — С. 410-414.

[6] Изотова, Т. Ю. Обзор алгоритмов поиска кратчайшего пути в графе / Т.Ю. Изотова // Новые информационные технологии в автоматизированных системах. — 2016. — №19. — С. 341-344.

[7] Зыков, А. А. Основы теории графов / А. А. Зыков. — М.: Наука. — 1987. — 384 с.

[8] Кормен, Т. Х. Алгоритмы: построение и анализ / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — М.: Изд-во «И. Д. Вильямс». — 2013. — 1328 с.

[9] Лебедев С. С., Новиков Ф. А. Необходимое и достаточное условие применимости алгоритма Дейкстры /Лебедев С. С., Новиков Ф. А .-- Компьютерные инструменты в образовании. -- 2017.-- № 4.-- С. 5–13.

[10] Попов, Д. А. Поиск кратчайших маршрутов на графах дорожной сети. Определение методов ускорения поиска / Д. А. Попов // Научный альманах. — 2016. — №1-1(15). — С. 477-480.

[11] Banerjee N., S. Chakraborty, and V. Raman. Improved space efficient algorithms for BFS, DFS and applications.-- In 22nd COCOON. -- volume 9797.— Springer. -- LNCS.--2016.-- pages 119–130.

[12] Chakraborty S, Raman V, Satti SR. Biconnectivity, st-numbering and other applications of DFS using O(*n*) bits..--*J. Comput. Syst. Sci.*2017.—pages 90:63–79. doi: 10.1016/j.jcss.2017.06.006.

[12] Discrete Applied Mathematics Volume 156, Issue 1. — 2008. — 55-75. URL: https://www.sciencedirect.com/journal/discrete-applied-mathematics/vol/156/issue/1, свободный.

[13] Farina, G. A linear time algorithm to compute the impact of all the articulation points./ Gabriele Farina -- Polytechnic University of Milan—2015.— arXiv:1504.00341v3 [cs.DS] 10 May 2015

[14] Karpov, D.V. Blocks in k-connected graphs./ Karpov D.V.J Math Sci 126. 2005.1167–1181.- URL: https://doi.org/10.1007/s10958-005-0084-4, свободный.

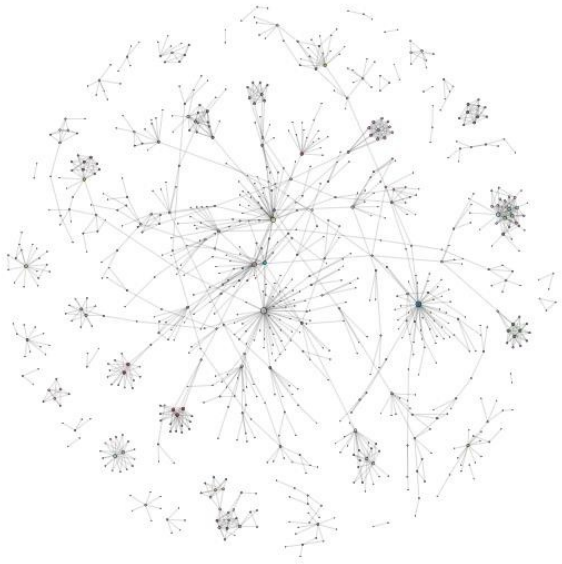
[15] Schmidt J. M. A simple test on 2-vertex- and 2-edge-connectivity./ J. M Schmidt. -- Information Processing Letters Volume 113,-- Issue 7.--15 April 2013.--Pages 241-244. URL.: https://doi.org/10.1016/j.ipl.2013.01.016

Приложение 1

Работа алгоритмов на глобальных данных

Исходные данные:

Dataset statistics	
Nodes	875713
Edges	5105039
Nodes in largest WCC	855802 (0.977)
Edges in largest WCC	5066842 (0.993)
Nodes in largest SCC	434818 (0.497)
Edges in largest SCC	3419124 (0.670)
Average clustering coefficient	0.5143
Number of triangles	13391903
Fraction of closed triangles	0.01911
Diameter (longest shortest path)	21
90-percentile effective diameter	8.1



Данные взяты с сайта: https://snap.stanford.edu/data/web-Google.html

Исходный граф: ориентированный, несвязный, количество ребер: 5105039, количество вершин: 875713

Связный, ориентированный: количество ребер: 5066842, количество вершин: 855802

Связный, неориентированный граф: количество ребер: 4322051, количество вершин: 875713

Время работы алгоритмов:

1. BFS : 12.307570695877075 (Edges 855801 , nodes 855802)

2.DFS: 16.13267493247986 (Edges 855801, nodes 855802)

3.Center: 6.82934045791626 ([210104, 556134])

4.Centroid: 776. 46574042391829 (([280367, 436711])

5. Blocks, cut vertex: 17.332032442092896 (Block 9638,Cut 9637)

6. BC-tree: 521.5348505973816 (Edges 28284 nodes 19275)

Приложение 2

Код программы

```
"""module for algorithm of bfs"""             ### НОВАЯ ВЕРСИЯ
                                               #улучшено время
import bisect
from collections import deque

def bfs_for_matrix(matrix, root):
    tree_matrix = [[0] * len(matrix) for i in range(len(matrix))] #возможно
 для
                                                                  # экономи
и памяти использовать диагональную матрицу (если будет время)
    queue = deque([root])      #очередь
    visited = [False for i in range(len(matrix))]             #посещенные в
ершины (флаги)
    visited[root] = True
    while queue:
        vertex = queue.popleft()
        for i in range(len(matrix)):        # проходим по соседям (все инде
ксы)
            if matrix[vertex][i] == 1  and not visited[i]:
                visited[i] = True
                queue.append(i)
                tree_matrix[vertex][i] = tree_matrix[i][vertex] = 1 # запол
няем дерево
    return tree_matrix


def bfs_for_list(list_, root):
    tree_list = [[] for i in range(len(list_))]

    queue = deque([root])      #очередь
    visited = [False for i in range(len(list_))]             #посещенные ве
ршины (флаги)
    visited[root] = True

    while queue:
        vertex = queue.popleft()
        for neighbour in list_[vertex]:     # проходим по соседям
            if not visited[neighbour]:
                visited[neighbour] = True
                queue.append(neighbour)
                bisect.insort(tree_list[vertex],neighbour)     #заполняем д
ерево с сортировкой
                bisect.insort(tree_list[neighbour],vertex)

    return tree_list
```

```
"""module for algorithm dfs"""   ### НОВАЯ ВЕРСИЯ

import bisect
from collections import deque

def dfs_for_matrix(matrix, root):
    tree_matrix = [[0] * len(matrix) for i in range(len(matrix))]  #можно и
спользовать диагональную матрицу для экономия памяти

    stack = deque([root])      #стек / чтобы избежать рекурсии
    visited =  [False for i in range(len(matrix))]            #посещенные в
ершины (флаги)
    visited[root] = True
    while stack:
        flag_neighbour = False
        vertex = stack[-1]
        for i in range(len(matrix)):
            if matrix[vertex][i] != 0 and not visited[i]:  #просмотр соседе
й
                flag_neighbour  = True
                visited[i] = True
                stack.append(i)
                tree_matrix[vertex][i] = tree_matrix[i][vertex] = 1    # за
полняем дерево
                break
        if not flag_neighbour and stack:
            stack.pop()
    return tree_matrix


def dfs_for_list(list_, root):
    tree_list = [[] for i in range(len(list_))]

    stack = deque([root])      #стек / чтобы избежать рекурсии
    visited = [False for i in range(len(list_))]            #посещенные вер
шины (флаги)
    visited[root] = True

    while stack:
        flag_neighbour = False
        vertex = stack[-1]
        for neighbour in list_[vertex]:
          if not visited[neighbour]:
                flag_neighbour = True
                visited[neighbour] = True
                stack.append(neighbour)
                bisect.insort(tree_list[vertex], neighbour)     #заполняем
дерево
                bisect.insort(tree_list[neighbour],vertex)
                break
        if not flag_neighbour:
```

```
            stack.pop()
    return tree_list


"""module for center search algorithm in tree"""  ### НОВАЯ ВЕРСИЯ

import tree

def center_for_matrix(matrix):
    copy_matrix = [elem[:] for elem in matrix]
    not_visited = [1 for i in range(len(matrix))]
    count_leafs = 0   # количество вершин, которые можно посещать до 1 или
2
    while count_leafs != 1 and count_leafs != 2:

      leafs = []
      for i in [k for k, j in enumerate(not_visited) if j == 1]:     # поис
к листов
        if sum(copy_matrix[i]) == 1:
          leafs.append(i)
          not_visited[i] = 0
      for top in leafs:                        # удаление листа
        index = copy_matrix[top].index(1)
        copy_matrix[top][index] = copy_matrix[index][top] = 0
      count_leafs = sum(not_visited)

    center = [i+1 for i, j in enumerate(not_visited) if j == 1]    # вершин
ы которые являются центром
    return center


def center_for_list(list_):
    list_copy = [elem[:] for elem in list_]
    not_visited = [1 for i in range(len(list_copy))]
    count_leafs = 0   # количество вершин, которые можно посещать до 1 или
2

    while count_leafs != 1 and count_leafs != 2:
      leafs = []
      for i in [k for k, j in enumerate(not_visited) if j == 1]:
         if len(list_copy[i]) == 1:
           leafs.append(i)
           not_visited[i] = 0
      for top in leafs:
        neighbour = list_copy[top]
        list_copy[neighbour[0]].remove(top)
      count_leafs = sum(not_visited)

    center = [i+1 for i, j in enumerate(not_visited) if j == 1]
    return center
```

```
"""module for centroid search algorithm in tree"""   ###НОВАЯ ВЕРСИЯ

import tree

def size_of_subgraph_for_matrix(matrix, root, neighbour):     # размер подг
рафа
    stack = [neighbour]
    path = []
    while stack:
        vertex = stack.pop()
        if vertex in path:
            continue
        path.append(vertex)
        for i in range(len(matrix)):
            if matrix[vertex][i] != 0:  #просмотр соседей
              if i == root:
                continue
              stack.append(i)
    return len(path)

def weight_of_vertex_for_matrix(matrix, vertex):            # вес вершины
  weight = 0
  for neighbour in range(len(matrix)):
    if matrix[vertex][neighbour] == 1:
      weight_neighbour = size_of_subgraph_for_matrix(matrix,vertex,neighbou
r)

      if weight < weight_neighbour:
        weight = weight_neighbour
  return weight

def size_of_subgraph_for_list(list_, root, neighbour):     # размер подграф
а
    stack = [neighbour]
    path = []
    while stack:
        vertex = stack.pop()
        if vertex in path:
            continue
        path.append(vertex)
        for neighbour in list_[vertex]:
          if neighbour == root:
            continue
          stack.append(neighbour)
    return len(path)

def weight_of_vertex_for_list(list_, vertex):            # вес вершины
  weight = 0
  for neighbour in list_[vertex]:
```

```
    weight_neighbour = size_of_subgraph_for_list(list_,vertex,neighbour)

    if weight < weight_neighbour:
      weight = weight_neighbour
  return weight

def centroid_for_matrix(matrix):
    centroid = []

    degree_vertex = sum(matrix[0])        # Выбор вершины с наибольшей степ
енью
    root = 0
    for i in range(len(matrix)):
      if degree_vertex < sum(matrix[i]):
        degree_vertex = sum(matrix[i])
        root = i

    weight_root = weight_of_vertex_for_matrix(matrix,root)

    flag = True
    while flag:

      for neighbour in range(len(matrix)):
        if matrix[root][neighbour] == 1:
          if sum(matrix[neighbour]) == 1:   # лист
            flag = False
            continue
          else:
            weight_neighbour = weight_of_vertex_for_matrix(matrix,neighbour
)

          if weight_neighbour > weight_root:
            flag = False
            continue
          elif weight_neighbour < weight_root:
            root = neighbour
            weight_root = weight_neighbour
            flag = True
            break
          else:
            centroid.append(root)
            centroid.append(neighbour)
            flag = False
            break

    if not centroid:
      centroid.append(root)

    return centroid
```

```
def centroid_for_list(list_):
    centroid = []

    degree_vertex = len(list_[0])        # Выбор вершины с наибольшей степе
нью
    root = 0
    for i in range(len(list_)):
      if degree_vertex < len(list_[i]):
        degree_vertex = len(list_[i])
        root = i

    weight_root = weight_of_vertex_for_list(list_,root)


    flag = True
    while flag:

      for neighbour in list_[root]:
        if len(list_[neighbour]) == 1:   # лист
          flag = False
          continue
        else:
          weight_neighbour = weight_of_vertex_for_list(list_,neighbour)

        if weight_neighbour > weight_root:
          flag = False
          continue
        elif weight_neighbour < weight_root:
          root = neighbour
          weight_root = weight_neighbour
          flag = True
          break
        else:
          centroid.append(root)
          centroid.append(neighbour)
          flag = False
          break

    if not centroid:
      centroid.append(root)

    return centroid

"""module for finding the shortest paths in a graph (Dijkstra's algorithm)"
""   ### НОВАЯ ВЕРСИЯ
import random
import time
#from weighted_graph import GraphList, GraphMatrix
```

```
def dijkstra_for_matrix(matrix, root):

    pred = [-1 for i in range(len(matrix))]
    dist = [float('inf') for i in range(len(matrix))]
    nodes = [i for i in range(len(matrix))]
    nodes.remove(root)
    dist[root] = 0

    for item in nodes:
      if matrix[root][item]!= 0:
        dist[item] = matrix[root][item]
        pred[item] = root

    print('matr = ', end_)
    while nodes:

      min_dist = float('inf')
      vertex = -1
      for item in nodes:
        min_ = dist[item]
        if min_dist > min_:
          min_dist = min_
          vertex = item

      nodes.remove(vertex)

      for item in nodes:
        if matrix[vertex][item] != 0:
          new_dist = dist[vertex] + matrix[vertex][item]
          if new_dist < dist[item]:
            pred[item] = vertex
            dist[item] = new_dist
    return dist, pred

# ##_____________________________________LIST_____________________________
_____________________## #


def dijkstra_for_list(list_, root):

    pred = [-1 for i in range(len(list_))]
    dist = [float('inf') for i in range(len(list_))]
    nodes = [i for i in range(len(list_))]
    nodes.remove(root)
    dist[root] = 0
    for neighbour, cost in list_[root].items():
      dist[neighbour] = cost
      pred[neighbour] = root

    while nodes:
```

```
      min_dist = float('inf')
      vertex = -1
      for item in nodes:
        min_ = dist[item]
        if min_dist > min_:
          min_dist = min_
          vertex = item
      nodes.remove(vertex)
      for item in nodes:
        if item in list_[vertex]:
         new_dist = dist[vertex] + list_[vertex][item]
         if new_dist < dist[item]:
           dist[item] = new_dist
           pred[item] = vertex
    return dist, pred

"module  articulation points of matrix and of list"   ### НОВАЯ ВЕРСИЯ

from collections import deque


def cut_vertex_for_matrix(matrix, root):
    copy_matrix = [elem[:] for elem in matrix]
    cut_vertex = []
    c = 1
    sons_of_root = set()
    dnum = [0 if i != root else c for i in range(len(matrix))]
    low = [0 if i != root else c for i in range(len(matrix))]
    stack = deque([root])      #стек / чтобы избежать рекурсии

    while stack:
        vertex = stack[-1]
        if sum(copy_matrix[vertex]) != 0:
          neighbour = copy_matrix[vertex].index(1)
          copy_matrix[vertex][neighbour] =  0
          if dnum[neighbour] == 0:
             c = c + 1
             stack.append(neighbour)
             dnum[neighbour] = c
             low[neighbour] = c
             if vertex == root:
               sons_of_root.add(neighbour)
          else:
            low[vertex] = min(low[vertex],dnum[neighbour])

        else:
          top = stack.pop()
          if top != root:
            vertex = stack[-1]
            if vertex != root:
```

```
                low[vertex]=min(low[vertex],low[top])
                if low[top] == dnum[vertex]:
                  cut_vertex.append(vertex)
    if len(sons_of_root) > 1:
      cut_vertex.append(root)

    return cut_vertex


def cut_vertex_for_list(list_, root):
    copy_list = [elem[:] for elem in list_]
    cut_vertex = []
    c = 1
    sons_of_root = set()
    dnum = [0 if i != root else c for i in range(len(list_))]
    low = [0 if i != root else c for i in range(len(list_))]
    stack = deque([root])      #стек / чтобы избежать рекурсии

    while stack:
        vertex = stack[-1]
        if copy_list[vertex]:
          neighbour = copy_list[vertex][0]
          copy_list[vertex].remove(neighbour)
          if dnum[neighbour] == 0:
             c = c + 1
             stack.append(neighbour)
             dnum[neighbour] = c
             low[neighbour] = c
             if vertex == root:
               sons_of_root.add(neighbour)
          else:
            low[vertex] = min(low[vertex],dnum[neighbour])

        else:
          top = stack.pop()
          if top != root:
            vertex = stack[-1]
            if vertex != root:
              low[vertex]=min(low[vertex],low[top])
              if low[top] == dnum[vertex]:
                cut_vertex.append(vertex)
    if len(sons_of_root) > 1:
      cut_vertex.append(root)

    return cut_vertex

"module  blocks of matrix and of list"   ### НОВАЯ ВЕРСИЯ

from collections import deque
```

```
def new_block(stack, cut, son):
  block = []
  curr_top = -1
  block.append(cut)
  while curr_top != son:
     curr_top = stack.pop()
     block.append(curr_top)

  return block


def blocks_for_matrix(matrix, root=0):
    copy_matrix = [elem[:] for elem in matrix]
    cut_vertex = []
    c = 1
    sons_of_root = []
    dnum = [0 if i != root else c for i in range(len(matrix))]
    low = [0 if i != root else c for i in range(len(matrix))]
    stack = deque([root])      #стек / чтобы избежать рекурсии
    blocks = []  # блоки
    stack_for_block = deque([root])

    while stack:
        vertex = stack[-1]
        if sum(copy_matrix[vertex]) != 0:
          neighbour = copy_matrix[vertex].index(1)
          copy_matrix[vertex][neighbour] =  0
          if dnum[neighbour] == 0:
             stack_for_block.append(neighbour)
             c = c + 1
             stack.append(neighbour)
             dnum[neighbour] = c
             low[neighbour] = c
             if vertex == root:
               sons_of_root.append(neighbour)
          else:
            low[vertex] = min(low[vertex],dnum[neighbour])

        else:
          top = stack.pop()
          if top != root:
            vertex = stack[-1]
            if vertex != root:
              low[vertex]=min(low[vertex],low[top])
              if low[top] == dnum[vertex]:
                cut_vertex.append(vertex)
                block = new_block(stack_for_block, vertex, top)
                blocks.append(block)
    if len(sons_of_root) > 1:
      cut_vertex.append(root)
      for son in sons_of_root[1:]:
```

```
            block = new_block(stack_for_block, root, son)
            blocks.append(block)
        if stack_for_block:
          block = []
          while stack_for_block:
            block.append(stack_for_block.pop())
          blocks.append(block)

        return blocks, cut_vertex


def blocks_for_list(list_, root=0):
    copy_list = [elem[:] for elem in list_]
    cut_vertex = []
    c = 1
    sons_of_root = []
    dnum = [0 if i != root else c for i in range(len(list_))]
    low = [0 if i != root else c for i in range(len(list_))]
    stack = deque([root])      #стек / чтобы избежать рекурсии
    blocks = []  # блоки
    stack_for_block = deque([root])

    while stack:
        vertex = stack[-1]
        if copy_list[vertex]:
          neighbour = copy_list[vertex][0]
          copy_list[vertex].remove(neighbour)
          if dnum[neighbour] == 0:
             stack_for_block.append(neighbour)
             c = c + 1
             stack.append(neighbour)
             dnum[neighbour] = c
             low[neighbour] = c
             if vertex == root:
               sons_of_root.append(neighbour)
          else:
            low[vertex] = min(low[vertex],dnum[neighbour])

        else:
          top = stack.pop()
          if top != root:
            vertex = stack[-1]
            if vertex != root:
              low[vertex]=min(low[vertex],low[top])
              if low[top] == dnum[vertex]:
                cut_vertex.append(vertex)
                block = new_block(stack_for_block, vertex, top)
                blocks.append(block)
    if len(sons_of_root) > 1:
      cut_vertex.append(root)
      for son in sons_of_root[1:]:
```

```
            block = new_block(stack_for_block, root, son)
            blocks.append(block)
        if stack_for_block:
          block = []
          while stack_for_block:
            block.append(stack_for_block.pop())
          blocks.append(block)

        return blocks, cut_vertex

"""class for bc-tree"""
import random

class BCTreeList:

    tree_list_ = []   # tree
    items = None        # blocks and cut vertex

    def __init__(self, blocks, cut_vertex):
      if cut_vertex:
        self.items = blocks + [[i] for i in cut_vertex]
        self.tree_list_ = [[] for i in range(len(blocks)+len(cut_vertex))]
        for i in range(len(blocks)):
          for j in range(len(cut_vertex)):
            if cut_vertex[j] in blocks[i]:
              self.tree_list_[i].append(j+len(blocks))
              self.tree_list_[j+len(blocks)].append(i)
      else:
        self.tree_list_ = [[]]
        self.items = blocks

    def __str__(self):
        string_list = ''
        i = 0
        for row in self.tree_list_:
            string_list += (str(i) + ' - ' + '[' + ' '.join([str(elem) for
elem in self.items[i]]) +']' + ': ' + ' '.join([str(elem) for elem in row])
 + '\n')
            i += 1
        return string_list

class BCTreeMatrix:

    items = None
    tree_matrix = []

    def __init__(self, blocks, cut_vertex):
      if cut_vertex:
        self.items = blocks + [[i] for i in cut_vertex]
```

```
        self.tree_matrix = [[0] * (len(blocks)+len(cut_vertex)) for i in ra
nge(len(blocks)+len(cut_vertex))]
        for i in range(len(blocks)):
          for j in range(len(cut_vertex)):
            if cut_vertex[j] in blocks[i]:
              self.tree_matrix[i][j+len(blocks)] = 1
              self.tree_matrix[j+len(blocks)][i] = 1
      else:
        self.tree_matrix = [[0]]
        self.items = blocks

    def __str__(self):
        string_matrix = 'Tree: ' + '\n'
        for row in self.tree_matrix:
            string_matrix += (' '.join([str(elem) for elem in row]) + '\n')
        string_matrix += '\n'
        for i in range(len(self.tree_matrix)):
          string_matrix += (str(i) + ' - ' +'[' + ' '.join([str(elem) for e
lem in self.items[i]])+']' + '\n')

        return string_matrix


"module bc-
tree for matrix and of list"   ### НОВАЯ ВЕРСИЯ   #  len(blocks)   +   len(
cut_vertex)

def BCTree_for_matrix(matrix, root=0):
    blocks, cut_vertex = blocks_for_matrix(matrix,root)
    bctree = BCTreeMatrix(blocks, cut_vertex)
    return bctree


def BCTree_for_list(list_, root=0):
    blocks, cut_vertex = blocks_for_list(list_,root)
    bctree = BCTreeList(blocks, cut_vertex)
    return bctree

"""class Tree: adjacency matrix and list"""
import random
#from graph import AdjacencyList, AdjacencyMatrix
class Tree:
    list_tree = AdjacencyList()
    matrix_tree = AdjacencyMatrix()

    def __init__(self, size=2):
        pru_cod = [random.randint(1, size) for i in range(size - 2)]

        self.matrix_tree = AdjacencyMatrix(size, create=False)
        for i in range(len(pru_cod)):
            flag = 0
```

```
            visits = [False for i in range(size)]
            for j in range(len(pru_cod)):
                visits[pru_cod[j] - 1] = True
            for j in range(len(visits)):
                if not visits[j]:
                    flag += 1
                    if flag == 1:
                        self.matrix_tree.matrix[pru_cod[i] - 1][j] = 1
                        self.matrix_tree.matrix[j][pru_cod[i] - 1] = 1
                        pru_cod[i] = j + 1

        flag = 0
        visits = [False for i in range(size)]
        for j in range(len(pru_cod)):
            visits[pru_cod[j] - 1] = True
        for i in range(size):
            if not visits[i]:
                if flag == 0:
                    index = i
                    flag += 1
                else:
                    self.matrix_tree.matrix[i][index] = 1
                    self.matrix_tree.matrix[index][i] = 1

        self.list_tree = AdjacencyList(size, create=False)
        self.list_tree.conversion_from_matrix_to_list(self.matrix_tree.matr
ix)

"""class for representing a graph as an adjacency list"""
import random

#from bfs import bfs_for_list, bfs_for_matrix
#from dfs import dfs_for_list, dfs_for_matrix

class AdjacencyList:
    list_ = []

    def __init__(self, size=1, probability=0.5, create=True):
        self.list_ = [[] for i in range(size)]
        if create:
            self.list_[0].append(1)
            self.list_[size - 1].append(size - 2)
            for i in range(1, size - 1):
                self.list_[i].append(i - 1)
                self.list_[i].append(i + 1)
            count = 0
            for i in range(size-2):
                for j in range(i+2,size):
                    random_probability = random.random()
```

```
                    if random_probability <= probability:

                        self.list_[i].append(j)
                        self.list_[j].append(i)
            for k in range(size):
                self.list_[k].sort()

    def __str__(self):
        string_list = ''
        i = 0
        for row in self.list_:
            string_list += (str(i) + ' - ' + ' '.join([str(elem) for elem i
n row]) + '\n')
            i += 1
        return string_list

    def conversion_from_matrix_to_list(self, matrix):
        self.list_ = [[] for i in range(len(matrix))]
        for i in range(len(matrix)):
            for j in range(len(matrix)):
                if matrix[i][j] == 1:
                    self.list_[i].append(j)


"""class for representing a graph as an adjacency matrix"""
class AdjacencyMatrix:

    matrix = []

    def __init__(self, size=1, probability=0.5, create=True):
        self.matrix = [[0] * size for i in range(size)]
        if create:
            for i in range(1, size):
                self.matrix[i - 1][i] = 1
                self.matrix[i][i - 1] = 1
            for i in range(size):
                for j in range(2 + i, len(self.matrix[i])):
                    random_probability = random.random()
                    if random_probability <= probability:
                        self.matrix[i][j] = 1
                        self.matrix[j][i] = 1

    def __str__(self):
        string_matrix = ''
        for row in self.matrix:
            string_matrix += (' '.join([str(elem) for elem in row]) + '\n')
        return string_matrix

    def conversion_from_list_to_matrix(self, list_):
        self.matrix = [[0] * len(list_) for i in range(len(list_))]
        for i in range(len(list_)):
```

```
            for j in range(len(list_[i])):
                self.matrix[i][list_[i][j]] = 1

"""class for weight a graph as an adjacency list"""
import random
class GraphList:
    list_ = []

    def __init__(self, size=1, probability=0.5, max_weight=100, create=True
):

        if create:
            self.list_ = [{} for i in range(size)]

            rand = int(random.random() * max_weight) + 1
            self.list_[0][1] = rand
            self.list_[1][0] = rand

            rand1 = int(random.random() * max_weight) + 1
            self.list_[size - 1][size - 2] = rand1
            self.list_[size - 2][size - 1] = rand1

            for i in range(1, size - 1):
                rand = int(random.random() * max_weight) + 1
                self.list_[i][i - 1] = rand
                self.list_[i - 1][i] = rand
                rand1 = int(random.random() * max_weight) + 1
                self.list_[i][i + 1] = rand1
                self.list_[i + 1][i] = rand1

            for i in range(size):
                for j in range(size - len(self.list_[i]) - 1):
                    if random.random() < probability:
                        set_list_ = set(range(size))
                        set_list_ -= {i} | set(self.list_[i])
                        elem = set_list_.pop()
                        rand = int(random.random() * max_weight) + 1
                        self.list_[i][elem] = rand
                        self.list_[elem][i] = rand


    def __str__(self):
        string_list = ''
        for i in range(len(self.list_)):
            string_list += (str(i) + ' - ' +
                            '[' + ' '.join(['{'
                                            + str(elem) + ': ' + str(self.l
ist_[i][elem]) +
                                            '}'
                                            for elem in self.list_[i]]) +
```

```
                                ']' + '\n')
        return string_list

    def conversion_from_matrix_to_list(self, matrix):
        self.list_ = [{} for i in range(len(matrix))]
        for i in range(len(matrix)):
            for j in range(len(matrix)):
                if matrix[i][j] != 0 and matrix[i][j] != float('inf'):
                    self.list_[i][j] = matrix[i][j]


"""class for weight a graph as an adjacency matrix"""
class GraphMatrix:
    #inf = 1000000000   # replace float('inf) to inf
    matrix = []

    def __init__(self, size=1, probability=0.5, max_weight=100):
        self.matrix = [[0] * size for i in range(size)]
        for i in range(1, size):
            rand = int(random.random() * max_weight) + 1
            self.matrix[i - 1][i] = rand
            self.matrix[i][i - 1] = rand

        for i in range(size):
            for j in range(2 + i, len(self.matrix[i])):
                if random.random() < probability:
                    rand = int(random.random() * max_weight) + 1
                    self.matrix[i][j] = rand
                    self.matrix[j][i] = rand
                else:
                    self.matrix[i][j] = 0
                    self.matrix[j][i] = 0

    def __str__(self):
        string_matrix = ''
        for row in self.matrix:
            string_matrix += (' '.join([str(elem) for elem in row]) + '\n')
        return string_matrix
```