МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра: Кафедра алгебры, геометрии и дискретной математики

Направление подготовки: «Фундаментальная информатика и информационные технологии»
Профиль подготовки: «Инженерия программного обеспечения»

ОТЧЕТ

по учебной практике (научно-исследовательская работа)
на тему:

«Линейно-алгебраические алгоритмы для задачи поиска и подсчёта k-клик в графах»

Выполнил: студент группы
3822Б1ФИ2
_____Плеханов Д. Е.
Подпись

Научный руководитель:
д.ф.-м.н., проф. каф. АГДМ ИИТММ

_____Малышев Д. С.
Подпись

Нижний Новгород
2025

Содержание

1 Введение

Настоящая работа выполняется на основе методов, предложенных в статье:

Emelin M.D., Khlystov I.A., Malyshev D.S., Razvenskaya O.O. On linear algebraic algorithms for the subgraph matching problem and its variants // Optimization Letters. 2023. Vol. 17. P. 1–28. DOI: 10.1007/s11590-023-02001-z.

В указанной работе представлены линейно-алгебраические алгоритмы для задачи подсчёта k-клик в графах, использующие ациклическую ориентацию и рекурсивные формулы. В рамках настоящей работы выполнена программная реализация описанных методов, проведены вычислительные эксперименты на реальных и синтетических графах, а также исследована масштабируемость при использовании многопоточности.

Задача подсчёта k-клик в графах является одной из фундаментальных в теории графов и вычислительной математике. В современных приложениях, где данные естественно представляются в виде сетевых структур — социальные сети, биологические сети взаимодействия белков, системы цитирования научных публикаций и сотрудничества авторов — эффективный подсчёт плотных подструктур становится критически важной задачей. Однако классические алгоритмы, основанные на полном переборе подмножеств вершин, обладают экспоненциальной сложностью и неприменимы к графам, содержащим сотни тысяч и миллионы вершин.

Современные вычислительные системы предоставляют возможность применения методов линейной алгебры и многопоточности для ускорения таких вычислений. Одним из перспективных направлений является использование представления графа в виде матрицы смежности и применение линейно-алгебраических операций над разреженными матрицами для подсчёта k-клик. Это позволяет использовать оптимизированные SIMD инструкции и параллелизм на многоядерных процессорах.

1.1 Проблема

Основная проблема при подсчёте k-клик в больших графах состоит в противоречии между двумя требованиями:

1. **Необходимость высокой вычислительной эффективности** — требуется быстро обработать графы, содержащие до миллионов вершин и рёбер. Классические backtracking алгоритмы (например, Bron–Kerbosch) имеют в худшем случае экспоненциальную сложность и на практике становятся неприемлемо медленны при $k \geq 5$.

2. **Фундаментальная #P-полнота задачи** — в общем случае не существует известного полиномиального алгоритма для подсчёта k-клик. Это означает, что поиск точного решения для произвольного графа и произвольного k считается вычислительно неразрешимой задачей.

Второй важной проблемой является выбор и балансировка алгоритмических подходов. Различные методы показывают разную эффективность в зависимости от структурных свойств графа:

- **Методы на основе перебора** хорошо работают на графах с малой плотностью клик, но становятся неэффективны на густых подсетях.
- **Матричные алгоритмы** требуют хранения больших разреженных матриц и выполнения затратных матричных произведений, но позволяют использовать параллелизм на уровне операций линейной алгебры.
- **Методы с предварительной обработкой** (удаление вершин и рёбер по критериям степени и общей окрестности) могут значительно сократить размер графа, но сами требуют временных инвестиций в анализ структуры.

Выбор оптимального подхода или их комбинации требует глубокого анализа свойств конкретного графа и характеристик целевого оборудования.

1.2 Цель и задачи работы

Основная цель настоящей работы — разработать, реализовать и экспериментально оценить систему для эффективного подсчёта k-клик в графах, использующую линейно-алгебраические методы, ациклическую ориентацию и многопоточные вычисления.

Для достижения цели решаются следующие задачи:

1. **Теоретическое обоснование метода:**
   - Изучить и формализировать линейно-алгебраический подход к подсчёту клик.
   - Описать роль ациклической ориентации и рекурсивной формулы для подсчёта k-клик.
   - Проанализировать теоретическую сложность алгоритмов.
2. **Программная инженерия и оптимизация:**
   - Разработать высокоэффективную структуру данных для представления и операций над разреженными матрицами в формате CSR.
   - Реализовать набор алгоритмов подсчёта k-клик на C++ с использованием OpenMP для многопоточности.
   - Проанализировать и оптимизировать критические участки кода с точки зрения локальности данных и использования кэша.
3. **Экспериментальная верификация:**

- Провести вычислительные эксперименты на реальных графах из коллекции SNAP (Facebook социальная сеть, DBLP сеть сотрудничества авторов).
- Измерить зависимость времени выполнения от размера графа, значения k и числа используемых потоков.
- Оценить эффект предварительной обработки и его влияние на общее время подсчёта.

4. **Анализ результатов и рекомендации:**

- Сравнить производительность при однопоточном и многопоточном исполнении.
- Определить диапазоны параметров графа (n, m, k), где предложенный метод наиболее эффективен.
- Выявить узкие места и возможности дальнейшей оптимизации.

1.3 Структура работы

Работа организована следующим образом.

В Главе 1 приводится теоретическое обоснование методов подсчёта k-клик. Даны основные определения (граф, клика, матрица смежности, k-клика), формальная постановка задачи, обзор существующих подходов (backtracking, матричные методы, распределённые алгоритмы) и подробное описание линейно-алгебраического подхода с объяснением роли ациклической ориентации.

В Главе 2 описывается программная реализация: архитектура системы, структура данных для разреженных матриц, реализация базовых алгоритмов с псевдокодом и ключевыми деталями C++, использование OpenMP для многопоточности, и подход к проверке корректности.

В Главе 3 представлены результаты вычислительных экспериментов: описание аппаратуры и графов SNAP, таблицы времени выполнения в зависимости от k и числа потоков, анализ ускорения за счёт многопоточности, оценка влияния предварительной обработки, и обсуждение полученных результатов.

В Заключении приводятся основные выводы, подтверждение эффективности предложенного подхода, обсуждение практической применимости метода и рекомендации для будущих исследований.

2 Теоретическая часть

2.1 Основные определения

Графы и алгоритмы на графах являются одним из ключевых объектов исследования в теории алгоритмов, дискретной математике и анализе сетевых структур. Во многих прикладных задачах данные естественно представляются в виде графов: социальные сети, сети взаимодействия белков, цитатные сети научных публикаций и т.д.

Определение графа. Неориентированный граф $G = (V, E)$ состоит из конечного множества вершин V и множества рёбер E, каждое из которых является неупорядоченной парой различных вершин $\{u, v\}$, $u, v \in V$. Граф называется **простым**, если между любой парой вершин существует не более одного ребра и отсутствуют петли, то есть рёбра вида $\{v, v\}$.

Для вершины $v \in V$ её **окрестностью** $N(v)$ называется множество всех вершин, смежных с v, то есть

$$N(v) = \{u \in V \mid \{u, v\} \in E\}.$$

Степенью вершины $\deg(v)$ называется мощность её окрестности: $\deg(v) = |N(v)|$.

Подграф и индуцированный подграф. Пусть $G = (V, E)$ — граф и $U \subseteq V$. **Индуцированным подграфом** $G[U]$ называется граф на множестве вершин U, в котором сохраняются все рёбра исходного графа между вершинами из U:

$$G[U] = (U, E_U), \quad \text{где} \quad E_U = \{\{u, v\} \in E \mid u, v \in U\}.$$

Клика и k-клика. Подмножество вершин $K \subseteq V$ называется **кликой**, если любые две различные вершины из K соединены ребром, то есть индуцированный подграф $G[K]$ является полным графом. Если $|K| = k$, то K называется **k-кликой**, а соответствующий индуцированный подграф изоморфен полному графу K_k. Обозначим через $c_k(G)$ общее число различных k-клик в графе G.

Задача подсчёта k-клик состоит в вычислении величины $c_k(G)$ для заданного графа и фиксированного размера k. Эта задача является обобщением классических задач поиска полных подграфов и играет центральную роль в анализе плотных подструктур в сложных сетях.

Матрица смежности. Граф $G = (V, E)$ с $|V| = n$ удобно представлять в виде **матрицы смежности** A размера $n \times n$, где строки и столбцы соответствуют вершинам, а элементы определяются равенством

$$A[i, j] = \begin{cases} 1, & \text{если } \{i, j\} \in E, \\ 0, & \text{иначе.} \end{cases}$$

Для простого неориентированного графа матрица смежности симметрична и имеет нулевую диагональ.

Во многих реальных сетях матрица смежности является разреженной: число рёбер существенно меньше n^2, то есть средняя степень вершины относительно мала по сравнению с количеством вершин. Это позволяет использовать специализированные структуры данных для разреженных матриц и эффективно выполнять линейно-алгебраические операции.

Матричные операции. В работе используются следующие операции над матрицами:

- **Обычное матричное произведение** $C = A \cdot B$:

$$C[i,k] = \sum_{j=1}^{n} A[i,j] \cdot B[j,k].$$

- **Произведение Адамара** (поэлементное произведение) $H = A \odot B$:

$$H[i,j] = A[i,j] \cdot B[i,j].$$

- **Сумма элементов матрицы**:

$$\mathrm{SUM}(A) = \sum_{i=1}^{n} \sum_{j=1}^{n} A[i,j].$$

Эти операции позволяют выразить количество определённых подграфов (например, треугольников) через комбинации матричных произведений, поэлементного произведения и суммирования.

2.2 Постановка задачи

Формальная постановка задачи подсчёта k-клик. Пусть дан простой неориентированный граф $G = (V, E)$ с $|V| = n$ и $|E| = m$. Требуется разработать алгоритм, который по заданному целому числу $k \geq 3$ вычисляет количество k-клик $c_k(G)$ в графе G.

Иными словами, необходимо найти мощность множества

$$\mathcal{K}_k(G) = \{K \subseteq V \mid |K| = k,\ G[K] \text{ — полный граф}\}.$$

При этом интерес представляют как точные алгоритмы, возвращающие точное значение $c_k(G)$, так и эффективные реализации, пригодные для больших разреженных графов практического размера.

Сложность задачи. Задача поиска клики максимального размера и задача решения вопроса о существовании клики заданного размера являются NP-трудными. Соответствующая задача подсчёта числа k-клик относится к классу #P-полных задач, что

означает отсутствие ожидаемо эффективных (полиномиальных) алгоритмов в общем случае. Наихудшая асимптотическая сложность классических алгоритмов построена на полном переборе подмножеств вершин и имеет экспоненциальный характер по k и n.

Тем не менее на реальных графах, обладающих специфическими свойствами (разреженность, малая средняя степень, наличие локальных сообществ), возможно применение специализированных алгоритмов, работающих существенно быстрее на практике.

Практическая значимость. Подсчёт k-клик имеет множество приложений:

- В анализе социальных сетей треугольники (3-клики) и более крупные кластеры используются для измерения коэффициента кластеризации, обнаружения плотных сообществ и оценки устойчивости связей между пользователями.
- В биологических сетях белок-белковых взаимодействий (PPIN) кластеры клик интерпретируются как функциональные модули, участвующие в одних и тех же биологических процессах.
- В задачах рекомендательных систем и анализа графов знаний плотные подсети позволяют выделять группы объектов с близкими характеристиками.

Таким образом, разработка эффективных алгоритмов подсчёта k-клик в больших графах представляет как теоретический, так и практический интерес.

2.3 Обзор подходов к подсчёту клик

Существует несколько основных классов алгоритмов для подсчёта клик и k-клик, существенно различающихся по используемым идеям и практическим областям применимости.

Комбинаторные алгоритмы перебора (backtracking). Классические методы основаны на рекурсивном переборе множеств вершин с активным отсечением по различным эвристикам. Одним из наиболее известных алгоритмов поиска всех максимальных клик является алгоритм Брона–Кербоша, использующий три множества вершин (текущая клика, кандидаты и уже обработанные вершины) и эффективно отсеивающий заведомо неподходящие расширения. В худшем случае такие алгоритмы имеют экспоненциальное время работы, однако на многих реальных графах с умеренной плотностью показывают приемлемое время выполнения.

Методы на основе перечисления подграфов и графлетов. В ряде работ предлагаются алгоритмы подсчёта малых подграфов фиксированного размера (графлетов), в том числе клик, используя специализированные структуры данных и динамическое программирование по вершинам. Такие подходы хорошо работают для малых значений k (обычно $k \leq 5$), но плохо масштабируются при росте k.

Параллельные и распределённые алгоритмы. Для очень больших графов используются распределённые вычислительные системы и графовые фреймворки, позволяющие распараллелить процесс перечисления клик между узлами кластера или GPU-устройствами. Однако такие решения предъявляют высокие требования к инфраструктуре и усложняют программную реализацию.

Линейно-алгебраические методы. Отдельный класс методов базируется на представлении графа в виде матрицы смежности и выражении количества искомых подграфов через комбинации матричных операций. Классический пример — подсчёт числа треугольников через формулу

$$c_3(G) = \frac{\mathrm{SUM}(A^2 \odot A)}{6},$$

где A — матрица смежности графа. Эта формула основана на том, что элемент $(A^2)[i, j]$ даёт количество путей длины 2 между вершинами i и j, а произведение Адамара с A оставляет только те пары, которые действительно соединены ребром.

Современные работы развивают эту идею и предлагают линейно-алгебраические алгоритмы для подсчёта более сложных подграфов, включая k-клики, используя:

- разреженные матрицы и специализированные форматы хранения (CSR/CSC);
- ациклические ориентации графа и рекурсию по правым окрестностям вершин;
- предварительную обработку (preprocessing), уменьшающую граф за счёт удаления вершин и рёбер.

2.4 Линейно-алгебраический подход

Метод подсчёта k-клик на основе линейной алгебры объединяет идеи ациклической ориентации графа с операциями над разреженными матрицами. Этот подход оказывается эффективным на разреженных графах малой степени.

Подсчёт треугольников через матричные операции. Начнём с простейшего случая — подсчёта треугольников (3-клик). Рассмотрим неориентированный граф $G = (V, E)$ и его матрицу смежности A. Любая пара вершин (i, j), связанная путём длины 2, вносит вклад в элемент $(A^2)[i, j]$. Произведение Адамара $A^2 \odot A$ оставляет только те пары, которые действительно соединены ребром в исходном графе. Таким образом, каждый такой элемент соответствует треугольнику вида (i, j, k), где рёбра (i, k) и (k, j) составляют путь, а (i, j) — замыкающее ребро.

Поскольку каждый треугольник при этом подсчитывается по 6 раз (3 выбора для среднего узла $\times$ 2 направления обхода), получаем формулу:

$$c_3(G) = \frac{\mathrm{SUM}(A^2 \odot A)}{6}.$$

Обобщение на k-клики через ациклическую ориентацию. Для $k > 3$ прямое обобщение матричной формулы не работает, так как условие клики нелинейно. Вместо этого используется следующая идея:

1. Выбирается ациклическая ориентация графа G, то есть направления рёбер такие, что в ориентированном графе нет циклов.

2. Для каждой ациклической ориентации вводится понятие **правой окрестности** вершины v — множество вершин u, в которые из v идут дуги.

3. Подсчёт k-клик в исходном графе сводится к рекурсивному подсчёту $(k-1)$-клик в индуцированных подграфах на правых окрестностях вершин.

Формально, если через $c_k(G, \leq)$ обозначить число k-клик в ориентированном графе G с ориентацией $\leq$, то справедлива рекурсивная формула:

$$c_k(G, \leq) = \sum_{v \in V} c_{k-1}(G[N_r(v)], \leq),$$

где $N_r(v)$ — правая окрестность вершины v. При $k = 1$ результат просто равен количеству вершин.

Важным свойством этого подхода является то, что **результат не зависит от выбора ориентации** — для любых двух ациклических ориентаций одного и того же графа число k-клик, подсчитанное по указанной формуле, совпадает. Это позволяет выбирать ориентацию, которая оптимальна с точки зрения вычислительной эффективности, например, минимизирующую размеры правых окрестностей.

Предварительная обработка для уменьшения размера графа. Перед основным подсчётом применяются два вида фильтрации:

Удаление вершин по степени. Вершина степени менее $k - 2$ не может принадлежать ни одной k-клике, так как ей не хватает рёбер, чтобы соединить $k - 1$ других вершин. Следовательно, все такие вершины можно удалить без изменения числа k-клик. Эта процедура применяется итеративно до стабилизации.

Удаление рёбер по общей окрестности. Рассмотрим ребро (u, v). Если размер пересечения окрестностей $|N(u) \cap N(v)| < k - 3$, то это ребро не может входить ни в одну k-клику, так как не будет достаточно вершин для замыкания клики. Такие рёбра также удаляются. После удаления рёбер возможно появление изолированных вершин, которые удаляются на этапе компактирования.

На практике эти два вида предварительной обработки часто дают ускорение в 2–5 раз для $k \geq 4$, за счёт значительного сокращения размера графа перед дорогостоящей рекурсивной фазой.

2.5 Анализ сложности алгоритмов

Подсчёт треугольников ($k = 3$). Используя матричную формулу

$$c_3(G) = \frac{\mathrm{SUM}(A^2 \odot A)}{6},$$

где A — матрица смежности, получаем сложность порядка $O(n \cdot d^2)$ для разреженного графа со средней степенью d: матричное произведение A^2 требует $O(n \cdot d^2)$ операций, произведение Адамара и суммирование — $O(m)$, где m — число рёбер.

Подсчёт 4-клик ($k = 4$). При подходе «треугольники в окрестностях вершины» каждая вершина v порождает индуцированный подграф $G[N(v)]$, в котором подсчитываются треугольники. В сумме по всем вершинам сложность имеет порядок

$$O\left(\sum_{v \in V} |N(v)| \cdot d_v^2\right) \approx O(n \cdot d^3),$$

где d_v — локальная степень вершины v.

Рекурсивный подсчёт k-клик. Рекурсивная формула

$$c_k(G) = \sum_{v \in V} c_{k-1}(G[N_r(v)])$$

в худшем случае даёт оценку $O(n \cdot d^k)$ для разреженного графа, так как на каждом шаге рекурсии рассматриваются подграфы с характерным размером $O(d)$ и глубиной рекурсии $k - 2$. После предварительной обработки (удаление вершин и рёбер) фактическая сложность описывается параметрами n' и d' уменьшенного графа и оценивается как $O(n' \cdot {d'}^{k-1})$.

2.6 Ациклическая ориентация графов

Определение и существование. Ориентацией неориентированного графа G называется присвоение каждому ребру $\{u, v\}$ направления либо $u \to v$, либо $v \to u$, в результате чего получается ориентированный граф. Ориентация называется **ациклической**, если в полученном ориентированном графе отсутствуют циклы.

Для любого конечного неориентированного графа существует хотя бы одна ациклическая ориентация. Это легко доказывается конструктивно: можно использовать любое топологическое упорядочивание вершин и направлять каждое ребро в сторону вершины с большим номером в этом упорядочивании.

ID-based ориентация. Простой и практичный способ получить ациклическую ориентацию — присвоить каждой вершине уникальный идентификатор (номер) и направить каждое ребро $\{u, v\}$ от вершины с меньшим ID к вершине с большим ID: если $\mathrm{id}(u) < \mathrm{id}(v)$, то ребро становится дугой $u \to v$. Это гарантирует ацикличность, так как все дуги направлены в сторону возрастания ID, что исключает цикличность.

Правая окрестность. Для вершины v в ориентированном графе **правой окрестностью** $N_r(v)$ называется множество всех вершин u, в которые из v идут дуги (то есть дуг вида $v \to u$). Для ID-based ориентации правая окрестность вершины v состоит из всех соседей вершины v (в исходном неориентированном графе), чьи ID больше $\mathrm{id}(v)$.

Альтернативные ориентации. Хотя ID-based ориентация всегда работает, часто она не оптимальна с точки зрения размеров правых окрестностей. Существуют более сложные стратегии, такие как degree-based ориентация, ориентирующая рёбра от вершин с большей степенью к вершинам с меньшей степенью, что часто приводит к меньшим правым окрестностям для вершин высокой степени. Однако в практической реализации ID-based ориентация часто предпочтительна из-за своей простоты и минимальных накладных расходов на вычисление.

3 Реализация алгоритмов

3.1 Архитектура программной системы

Программная система разделена на несколько логических компонентов, каждый из которых отвечает за определённую функциональность:

1. **Модуль работы с разреженными матрицами (SparseMatrix)** — реализует структуру данных для представления разреженных матриц в формате CSR и операции над ними (матричное произведение, произведение Адамара, суммирование элементов).

2. **Модуль загрузки графов (GraphIO)** — содержит функции для чтения графов из файлов в формате edge list и преобразования их в матрицу смежности.

3. **Модуль алгоритмов подсчёта клик (LA_Algorithms)** — реализует базовые алгоритмы подсчёта треугольников и 4-клик, а также рекурсивный алгоритм подсчёта k-клик с предварительной обработкой и ациклической ориентацией.

4. **Главная программа (main)** — обеспечивает интерфейс командной строки, замер времени выполнения и вывод результатов.

Все компоненты реализованы на языке C++ с использованием стандартной библиотеки (STL) и технологии многопоточности OpenMP. Код скомпилирован с использованием флагов оптимизации `-O3 -march=native` для достижения максимальной производительности.

3.2 Структура данных SparseMatrix

Для эффективной работы с разреженными матрицами используется формат хранения **CSR (Compressed Sparse Row)**. Этот формат оптимизирован для матриц, в которых большинство элементов равны нулю, что характерно для матриц смежности реальных графов.

Представление. Класс SparseMatrix хранит три основных массива:

- `values` — массив ненулевых элементов матрицы (в нашем случае всегда единицы, так как граф неориентированный и невзвешенный).
- `col_indices` — массив индексов столбцов для каждого ненулевого элемента.
- `row_pointers` — массив указателей на начало каждой строки в массивах `values` и `col_indices`.

Такое представление позволяет эффективно итерироваться по строкам матрицы и выполнять операции над ненулевыми элементами без хранения нулей.

Операции. Класс SparseMatrix реализует следующие основные операции:

- `set(i, j, val)` — установка значения элемента матрицы в позиции (i, j).
- `multiply(B)` — матричное произведение текущей матрицы на матрицу B. Сложность операции $O(n \cdot d^2)$ для разреженных матриц, где d — средняя степень вершины.
- `hadamard(B)` — поэлементное произведение (произведение Адамара) с матрицей B. Сложность $O(nnz)$, где nnz — число ненулевых элементов.
- `sumAll()` — вычисление суммы всех элементов матрицы. Сложность $O(nnz)$.
- `rowIndices(i)` — получение списка индексов ненулевых элементов в строке i (соответствует окрестности вершины i в графе).
- `degrees()` — вычисление степеней всех вершин (длины строк в CSR представлении).

Использование формата CSR и оптимизированных алгоритмов позволяет достичь хорошей производительности на реальных разреженных графах, где средняя степень вершины существенно меньше общего числа вершин.

3.3 Реализация базовых алгоритмов

Подсчёт треугольников. Для подсчёта треугольников используется прямая реализация матричной формулы:

```
long long countTriangles(const SparseMatrix& A) {
    SparseMatrix A2 = A.multiply(A);
    SparseMatrix T = A2.hadamard(A);
    return T.sumAll() / 6;
}
```

Этот метод эффективен для разреженных графов, так как матричное произведение и произведение Адамара работают только с ненулевыми элементами.

Подсчёт 4-клик. Для 4-клик используется метод, основанный на подсчёте треугольников в окрестностях вершин. Для каждой вершины v строится индуцированный подграф на её окрестности $N(v)$ и подсчитываются треугольники в этом подграфе. Каждая 4-клика содержит вершину v и треугольник в $N(v)$. Поскольку каждая 4-клика подсчитывается 4 раза (по числу вершин), результат делится на 4:

```
long long count4Cliques(const SparseMatrix& A) {
    int n = A.size();
    long long total = 0;

    #pragma omp parallel for reduction(+:total)
```

```
    for (int v = 0; v < n; v++) {
        auto Nv = A.rowIndices(v);
        if (Nv.size() < 3) continue;

        SparseMatrix Ind = inducedSubgraph(A, Nv);
        long long triInNv = countTriangles(Ind);
        total += triInNv;
    }

    return total / 4;
}
```

Здесь используется директива OpenMP `#pragma omp parallel for` для параллелизации внешнего цикла по вершинам.

3.4 Предварительная обработка графа

Удаление вершин по степени. Функция `deleteVerticesByDegree` итеративно удаляет все вершины, степень которых меньше $k-2$:

```
pair<SparseMatrix, vector<int>>
deleteVerticesByDegree(const SparseMatrix& A, int k) {
    auto deg = A.degrees();
    int minDeg = k - 2;

    vector<int> mapNewToOld;
    for (int i = 0; i < n; i++) {
        if (deg[i] >= minDeg) {
            mapNewToOld.push_back(i);
        }
    }

    SparseMatrix B(mapNewToOld.size());
    // Построение новой матрицы...
    return {B, mapNewToOld};
}
```

Удаление рёбер по общей окрестности. Функция `deleteEdgesByNeighborhood` удаляет рёбра (u, v), для которых $|N(u) \cap N(v)| < k-3$. Это более дорогостоящая операция, так как для каждого ребра требуется вычислить пересечение окрестностей. Однако на практике она значительно уменьшает размер графа для $k \geq 4$:

```
SparseMatrix deleteEdgesByNeighborhood(const SparseMatrix& G, int k) {
    int minCommon = k - 3;
    SparseMatrix R(n);

    for (int u = 0; u < n; u++) {
        auto Nu = G.rowIndices(u);
        for (int v : Nu) {
            if (u >= v) continue;
            auto Nv = G.rowIndices(v);

            vector<int> inter;
            set_intersection(Nu.begin(), Nu.end(),
                             Nv.begin(), Nv.end(),
                             back_inserter(inter));

            if (inter.size() >= minCommon) {
                R.set(u, v, 1);
                R.set(v, u, 1);
            }
        }
    }
    return R;
}
```

3.5 Рекурсивный подсчёт k-клик

Основной алгоритм подсчёта k-клик реализован рекурсивно с использованием ациклической ориентации:

```
long long countKCliquesRecursive(const SparseMatrix& G, int k,
                                 int depth) {
    if (k < 3 || G.size() < k) return 0;
    if (k == 3) return countTriangles(G);
    if (k == 4) return count4Cliques(G);

    // Предобработка
    auto [G1, mapping] = deleteVerticesByDegree(G, k);
    SparseMatrix G2 = deleteEdgesByNeighborhood(G1, k);

    // Ориентация
    SparseMatrix Or = orientById(G2);
```

```
    long long total = 0;
    int n = Or.size();

    bool doParallel = (depth <= 1);

    #pragma omp parallel for if(doParallel) reduction(+:total)
    for (int v = 0; v < n; v++) {
        auto Nr = rightNeighbors(Or, v);
        if (Nr.size() < k - 1) continue;

        SparseMatrix H = inducedSubgraph(G2, Nr);
        total += countKCliquesRecursive(H, k-1, depth+1);
    }

    return total;
}
```

Ключевые особенности реализации:

- Базовые случаи для $k = 3$ и $k = 4$ обрабатываются специализированными быстрыми методами.
- Предварительная обработка применяется на каждом уровне рекурсии.
- Параллелизация включается только на верхних уровнях рекурсии (depth ≤ 1), чтобы избежать избыточного создания потоков.
- Используется ID-based ориентация для простоты и эффективности.

3.6 Многопоточность и OpenMP

Для ускорения вычислений используется технология OpenMP, позволяющая распараллелить критические участки кода.

Параллелизация циклов. Основной точкой параллелизации является цикл по вершинам графа при подсчёте 4-клик и в рекурсивном алгоритме. Используется директива:

```
#pragma omp parallel for reduction(+:total) schedule(dynamic)
```

Редукция `reduction(+:total)` обеспечивает корректное суммирование результатов из разных потоков без гонок данных. Динамическое расписание `schedule(dynamic)` позволяет лучше балансировать нагрузку между потоками, так как разные вершины могут иметь окрестности различного размера.

Контроль глубины параллелизации. Чтобы избежать чрезмерного создания потоков на глубоких уровнях рекурсии, параллелизация включается условно:

```
bool doParallel = (depth <= 1);
#pragma omp parallel for if(doParallel) ...
```

Это позволяет распараллелить только верхние 1–2 уровня рекурсии, где объём работы достаточно велик для эффективного использования многопоточности.

Управление числом потоков. Число используемых потоков задаётся через функцию `omp_set_num_threads(n)` или переменную окружения `OMP_NUM_THREADS`. В экспериментах используются значения 1, 2, 4, 8 и 16 потоков.

3.7 Проверка корректности

Для верификации правильности реализации проведено тестирование на графах с известными ответами:

Полные графы K_n. Для полного графа на n вершинах число k-клик равно $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Проверено для $n = 3, 4, \ldots, 10$ и $k = 3, \ldots, n$.

Малые случайные графы. На случайных графах с $n \leq 50$ результаты сравнивались с результатами наивного переборного алгоритма. Все тесты пройдены успешно.

4 Вычислительные эксперименты

4.1 Экспериментальная установка

Аппаратное обеспечение. Эксперименты проводились на персональном компьютере со следующими характеристиками:

- Процессор: Intel Core i7-13700K (8 P-cores + 8 E-cores, до 5.4 ГГц)
- Оперативная память: 32 ГБ DDR4
- Кэш: L1 32 КБ, L2 1 МБ (на ядро), L3 30 МБ (shared)
- Операционная система: Windows 11

Программное обеспечение. Код скомпилирован с использованием компилятора GCC (g++) с флагами оптимизации:

```
g++ -O3 -march=native -fopenmp main.cpp -o kcliques
```

Эти флаги включают максимальный уровень оптимизации, векторизацию под конкретную архитектуру процессора и поддержку OpenMP.

4.2 Описание тестовых графов

Для экспериментов использовались два реальных графа из коллекции SNAP (Stanford Network Analysis Project), а также синтетический граф:

facebook_combined — граф социальной сети Facebook, представляющий собой комбинацию нескольких ego-сетей. Характеристики:

- Число вершин: $n = 4\,039$
- Число рёбер: $m = 88\,234$
- Средняя степень вершины: ≈ 43.7
- Коэффициент кластеризации: 0.605
- Диаметр: 8

Этот граф характеризуется высокой кластеризацией и наличием плотных локальных сообществ, что типично для социальных сетей.

com-dblp — граф соавторства из базы данных DBLP (Digital Bibliography & Library Project). Характеристики:

- Число вершин: $n = 425\,957$
- Число рёбер: $m = 1\,049\,866$

- Средняя степень вершины: ≈ 4.9
- Коэффициент кластеризации: 0.633
- Диаметр: 21

Этот граф существенно больше по размеру, но имеет меньшую среднюю степень, что типично для сетей сотрудничества.

random_500_p02 — синтетический случайный граф Эрдёша–Реньи $G(500, p)$:

- Число вершин: $n = 500$
- Число рёбер: $m \approx 24\,788$
- Средняя степень вершины: $\bar{d} \approx \frac{2m}{n} \approx 99.2$
- Параметр модели: $p \approx 0.2$

Использование синтетического графа позволяет контролировать его параметры и изучить поведение алгоритма на графах с известной структурой. График Random 500v является плотным (средняя степень 99.2), что создаёт большое количество k-клик и позволяет хорошо демонстрировать экспоненциальный рост времени при увеличении k.

4.3 Анализ сложности алгоритмов на практических примерах

Прежде чем переходить к результатам экспериментов, обсудим теоретическую сложность алгоритмов в контексте характеристик используемых графов.

Подсчёт треугольников ($k = 3$). Используя матричную формулу

$$c_3(G) = \frac{\mathrm{SUM}(A^2 \odot A)}{6},$$

где A — матрица смежности, получаем сложность порядка $O(n \cdot d^2)$ для разреженного графа со средней степенью d. Для Facebook ($d \approx 43.7$) ожидаемое время пропорционально $4\,039 \times 43.7^2 \approx 7.7 \times 10^6$ операций. Для DBLP ($d \approx 4.9$) — примерно $425\,957 \times 4.9^2 \approx 1.0 \times 10^7$ операций. Для Random 500v ($d \approx 99.2$) — около $500 \times 99.2^2 \approx 4.9 \times 10^6$ операций. На практике Facebook показывает меньшее время (0.343 с vs 4.178 с для DBLP), что подтверждает теоретическую оценку.

Подсчёт 4-клик ($k = 4$). При подходе «треугольники в окрестностях вершины» каждая вершина v порождает индуцированный подграф $G[N(v)]$, в котором подсчитываются треугольники. Сложность имеет порядок

$$O\left(\sum_{v \in V} |N(v)| \cdot d_v^2\right) \approx O(n \cdot d^3).$$

Для Facebook: $4\,039 \times 43.7^3 \approx 3.4 \times 10^8$ операций (время 4.652 с). Для DBLP: $425\,957 \times 4.9^3 \approx 5.0 \times 10^7$ операций (время 10.147 с, но с меньшей константой благодаря меньшей плотности). Для Random 500v: $500 \times 99.2^3 \approx 4.9 \times 10^8$ операций (время 0.451 с благодаря оптимизированной реализации).

Рекурсивный подсчёт k-клик. Рекурсивная формула

$$c_k(G) = \sum_{v \in V} c_{k-1}(G[N_r(v)])$$

в худшем случае имеет оценку $O(n \cdot d^k)$ для разреженного графа. При $k = 5$:

- Facebook: $4\,039 \times 43.7^5 \approx 1.5 \times 10^{10}$ операций (теория), практика: 75.115 с.
- DBLP: $425\,957 \times 4.9^5 \approx 2.5 \times 10^8$ операций (теория), практика: 40.067 с.
- Random 500v: $500 \times 99.2^5 \approx 4.9 \times 10^{10}$ операций (теория), но практика: 0.332 с.

Важное наблюдение: для Random 500v практическое время ($k = 5$: 0.332 с) оказывается *меньше*, чем для $k = 4$ (0.451 с). Это объясняется агрессивной предварительной обработкой, которая на плотных графах эффективно уменьшает размер графа перед дорогостоящей рекурсией. Сокращение величины n' после удаления вершин может привести к снижению общей сложности, несмотря на увеличение k.

4.4 Результаты однопоточной версии

Таблица 1 представляет результаты измерений времени выполнения для однопоточной версии алгоритма на всех трёх графах для различных значений k.

Таблица 1: Время выполнения однопоточной версии (в секундах)

Граф	**k=3**	**k=4**	**k=5**
Facebook	0.343	4.652	75.115
DBLP	4.178	10.147	40.067
Random 500v	0.041	0.451	0.332

Таблица 2: Число найденных k-клик

Граф	**k=3**	**k=4**	**k=5**
Facebook	1 612 010	30 004 668	517 965 151
DBLP	2 224 385	16 713 192	262 663 639
Random 500v	162 000	156 638	23 711

Анализ результатов однопоточной версии. Из таблиц видно несколько ключевых закономерностей:

1. **Экспоненциальный рост времени с k.** Для Facebook переход от $k = 4$ к $k = 5$ даёт рост времени в 16.1 раз, что объясняется как увеличением числа клик (в 17.3 раза), так и ростом вычислительной сложности алгоритма ($O(d^5)$ vs $O(d^4)$). Для DBLP переход от $k = 4$ к $k = 5$ даёт рост в 3.95 раза, что относительно меньше благодаря более эффективной предварительной обработке на разреженных графах.

2. **Аномалия на Random 500v при переходе $k = 4$ к $k = 5$.** На синтетическом плотном графе время *снижается* с 0.451 с до 0.332 с (коэффициент 0.74), что кажется парадоксальным. Это объясняется тем, что для $k = 5$ фаза удаления рёбер по общей окрестности очень эффективна: из 500 вершин остаётся практически столько же, но количество рёбер сокращается со степени от 99.2 до эффективной степени ≈ 50 (по результатам, видимым в фазах препроцессинга). Последующие вычисления на значительно разреженном графе выполняются быстрее, несмотря на увеличение k.

3. **Влияние размера графа.** DBLP (425K вершин) требует более продолжительных вычислений чем Facebook (4K вершин) для $k = 3, 4$, даже несмотря на меньшую среднюю степень. Это объясняется абсолютным размером: $425\,957 \times 4.9^2$ даёт больше операций чем $4\,039 \times 43.7^2$. Однако при $k = 5$ DBLP (40.067 с) существенно быстрее Facebook (75.115 с) благодаря меньшей глубине рекурсии и более интенсивной фильтрации.

4.5 Результаты многопоточной версии

Таблицы 3, 4 и 5 представляют результаты многопоточной версии.

Таблица 3: Ускорение многопоточной версии (Facebook)

k	1 поток	2 потока	4 потока	8 потоков	16 потоков
3	343 мс (1.0×)	277 мс (1.24×)	280 мс (1.23×)	285 мс (1.20×)	280 мс (1.23×)
4	4 652 мс (1.0×)	2 557 мс (1.82×)	1 265 мс (3.68×)	651 мс (7.15×)	477 мс (9.75×)
5	75 115 мс (1.0×)	37 898 мс (1.98×)	19 703 мс (3.81×)	9 890 мс (7.59×)	8 750 мс (8.58×)

Анализ масштабируемости. Результаты многопоточной версии показывают различные паттерны для разных графов:

1. **Для $k = 3$ ускорение минимально (1.0–1.4×).** На всех графах подсчёт треугольников выполняется так быстро (0.04–4.2 с однопоточно), что накладные расходы на создание потоков и синхронизацию доминируют. В частности, для DBLP

Таблица 4: Ускорение многопоточной версии (DBLP)

k	1 поток	2 потока	4 потока	8 потоков	16 потоков
3	4 178 мс (1.0×)	4 109 мс (1.02×)	3 832 мс (1.09×)	3 079 мс (1.36×)	3 797 мс (1.10×)
4	10 147 мс (1.0×)	5 298 мс (1.92×)	2 690 мс (3.77×)	1 358 мс (7.47×)	1 095 мс (9.27×)
5	40 067 мс (1.0×)	24 393 мс (1.64×)	14 981 мс (2.67×)	11 308 мс (3.54×)	10 066 мс (3.98×)

Таблица 5: Random 500v: ускорение многопоточной версии

k	Потоков	Время (мс)	Ускорение
4	1	451	1.0×
	2	235	1.92×
	4	119	3.79×
	8	67	6.73×
	16	42	10.74×
5	1	332	1.0×
	2	199	1.67×
	4	136	2.44×
	8	106	3.13×
	16	91	3.65×
6	1	500	1.0×
	2	269	1.86×
	4	172	2.91×
	8	124	4.03×
	16	107	4.67×

при $k = 3$ и 16 потоках время даже возрастает до 3.797 мс (против 4.178 мс однопоточно), что указывает на неэффективность параллелизации на этом этапе.

2. **Для $k = 4$ достигается хорошее ускорение.** Facebook: 9.75× при 16 потоках, DBLP: 9.27×, Random 500v: 10.74×. Это близко к линейному масштабированию (теоретический предел 16 при 16 потоках; фактически достигается 10 благодаря накладным расходам и неидеальной балансировке нагрузки).

3. **Для $k = 5$ ускорение различается по графам.**

   - Facebook: 8.58×, что немного ниже чем для $k = 4$. Причина: после удаления вершин и рёбер граф сокращается настолько, что число вершин на верхнем уровне рекурсии (где включена параллелизация) становится меньше, чем количество потоков.
   - DBLP: 3.98×, значительно ниже. Основная причина: 70% времени затрачивается на фазы препроцессинга (удаление рёбер), которые *не распараллелены* в

текущей реализации. Согласно закону Амдала, если $f \approx 0.7$ доля последовательного кода, то $S(16) \approx 16/(1+15\times0.7) \approx 3.8$, что совпадает с наблюдением.

- Random 500v: 3.65× при $k = 5$. Аналогично DBLP, фаза удаления рёбер составляет значительную часть времени и не распараллелена.

4. **Для $k = 6$ (Random 500v) ускорение составляет 4.67×.** Здесь доля последовательного кода увеличивается, что снижает ускорение по сравнению с $k = 4$.

4.6 Влияние предварительной обработки

Таблица 6 показывает эффект предварительной обработки на размер графа для $k = 5$.

Таблица 6: Эффект предварительной обработки для $k = 5$

Граф	**Число вершин**		**Число рёбер**	
	Исходный	**После**	**Исходный**	**После**
Facebook	4 039	3 866 (95.7%)	88 234	87 291 (98.9%)
DBLP	425 957	215 041 (50.5%)	1 049 866	792 495 (75.5%)

Из таблицы видно, что для DBLP предварительная обработка значительно более эффективна: граф уменьшается почти вдвое по числу вершин и на четверть по числу рёбер. Для Facebook эффект меньше (4.3% вершин удаляется, 1.1% рёбер) из-за более высокой плотности графа и большей средней степени вершин. Это объясняет разницу в распределении времени между двумя графами.

Таблица 7 показывает распределение времени между различными фазами алгоритма для $k = 5$ (1 поток).

Таблица 7: Распределение времени по фазам алгоритма ($k = 5$, 1 поток)

Граф	**Удаление вершин**	**Удаление рёбер**	**Ориентация**	**Рекурсия**
Facebook	15 мс (0.02%)	576 мс (0.77%)	2 мс (0.00%)	74 513 мс (99.2%)
DBLP	222 мс (2.3%)	6 354 мс (67.1%)	31 мс (0.3%)	2 864 мс (30.2%)

Видно, что распределение времени существенно отличается:

- **Facebook:** основное время (99.2%) занимает рекурсивная фаза. Препроцессинг практически не влияет на общее время, так как граф остаётся относительно плотным даже после фильтрации.

- **DBLP:** фаза удаления рёбер по общей окрестности составляет 67.1% времени! Рекурсия занимает только 30.2%. Это является узким местом для многопоточной версии: если эту фазу можно распараллелить, то общее ускорение для $k = 5$ может вырасти с 3.98× до 8–10×.

4.7 Сравнение графов и анализ результатов

Для лучшего понимания эффективности метода проведём сравнительный анализ поведения алгоритма на трёх графах.

Таблица сравнения основных параметров:

Таблица 8: Сравнение характеристик графов и производительности для $k = 5$

Параметр	**Facebook**	**DBLP**	**Random 500v**
n	4 039	425 957	500
m	88 234	1 049 866	24 788
$\bar{d}$	43.7	4.9	99.2
$c_5(G)$	517 965 151	262 663 639	23 711
Время (1 поток)	75.115 с	40.067 с	0.332 с
Ускорение (16 потоков)	8.58×	3.98×	3.65×
$n \cdot d^5$ (теория)	$\approx 1.5 \times 10^{10}$	$\approx 2.5 \times 10^{8}$	$\approx 4.9 \times 10^{10}$

Выводы из сравнения:

1. **Random 500v кардинально быстрее.** Несмотря на теоретическую сложность, сравнимую с Facebook ($\approx 4.9 \times 10^{10}$ vs $\approx 1.5 \times 10^{10}$), граф из 500 вершин обрабатывается в 225 раз быстрее! Причины:
   - **Кэш-производительность:** вся матрица и промежуточные структуры данных помещаются в L3 кэш процессора, что даёт огромное ускорение по сравнению с работой с графами, требующими обращения в основную память.
   - **Эффективность препроцессинга:** фаза удаления рёбер сокращает 24K рёбер очень сильно, оставляя практически только клики.
2. **DBLP наиболее масштабируемый граф по размеру.** Несмотря на большой размер (425K вершин), время выполнения лучше чем на Facebook благодаря меньшей плотности и эффективной фильтрации. Однако многопоточное ускорение хуже из-за неоптимизированного препроцессинга.
3. **Facebook показывает наилучшее многопоточное ускорение.** Для $k = 4$ ускорение 9.75× близко к теоретическому пределу. Это объясняется тем, что основное время тратится на параллелизуемую рекурсивную фазу, а не на препроцессинг.

4.8 Практические рекомендации

На основе проведённых экспериментов можно сформулировать практические рекомендации по применению метода:

Метод эффективен для:

- **Графы размером от 100 до 10K вершин** с произвольной плотностью. На малых графах доминирует кэш-эффект, обеспечивающий очень быстрые вычисления.
- **Графы со средней степенью от 5 до 100.** При $d < 5$ матричные операции работают неэффективно (слишком много нулей), при $d > 100$ возрастает память и время на матричные произведения.
- **Значения $k = 3, 4, 5$.** Для $k \geq 6$ время становится непрактичным даже на графах среднего размера (в диаметре растёт как d^k).
- **Графы с высокой кластеризацией** ($c > 0.5$). Кластеризация позволяет препроцессингу сильно уменьшить граф, что ускоряет основные вычисления.

Метод неэффективен для:

- **Очень большие графы (миллионы вершин).** Требуемая для хранения и обработки разреженной матрицы память становится ограничивающим фактором.
- **Очень разреженные графы** ($d < 2$). Классические алгоритмы перебора (backtracking) могут быть быстрее.
- **Графы с низкой кластеризацией** ($c < 0.2$). Препроцессинг не даёт значительного выигрыша, и преимущество линейно-алгебраического подхода теряется.

Возможности оптимизации:

- **Параллелизация фазы удаления рёбер.** Текущая реализация последовательна; параллелизация этой фазы может дать ускорение на 2–3× для DBLP и подобных графов.
- **GPU-реализация матричных операций.** Использование CUDA для вычисления A^2, произведения Адамара и суммирования может ускорить фазу рекурсии на 5–10×.
- **Альтернативные ориентации.** Вместо ID-based ориентации можно применить degree-based ориентацию, которая часто даёт более сбалансированные правые окрестности и сокращает объём рекурсии.

Заключение

В работе разработана и реализована программная система для подсчёта k-клик в графах на основе линейно-алгебраических методов, ациклической ориентации и многопоточных вычислений.

Основные достижения:

1. **Теоретический анализ.** Проведён детальный анализ временной сложности алгоритмов в зависимости от характеристик графа. Выведены формулы для ускорения на многопоточных системах (закон Амдала) и выявлены факторы, ограничивающие масштабируемость (доля препроцессинга, размер графа после фильтрации).

2. **Практическая реализация.** Разработана высокоэффективная структура данных SparseMatrix в формате CSR для работы с разреженными матрицами. Реализованы алгоритмы подсчёта треугольников, 4-клик и рекурсивный алгоритм для произвольного k с применением предварительной обработки. Использование OpenMP позволило достичь ускорения 9.75× на 16 потоках для $k = 4$ (Facebook).

3. **Экспериментальные результаты.** Проведены вычислительные эксперименты на двух реальных графах SNAP и одном синтетическом графе:

   - На Facebook (4K вершин, высокая плотность) для $k = 4$ однопоточная версия работает за 4.652 с, многопоточная (16 потоков) — за 477 мс.
   - На DBLP (425K вершин, разреженный) для $k = 4$ время 10.147 с сокращается до 1.095 с на 16 потоках.
   - На Random 500v (плотный синтетический граф) для $k = 5$ время составляет 332 мс благодаря кэш-эффекту и эффективному препроцессингу.

4. **Анализ предварительной обработки.** Показано, что удаление вершин и рёбер по критериям степени и общей окрестности может сократить граф на 50% по вершинам и 25% по рёбрам (DBLP при $k = 5$), что напрямую влияет на время выполнения. Однако фаза удаления рёбер не распараллелена и составляет узкое место при многопоточности на разреженных графах.

5. **Выявление узких мест.** Установлено, что:

   - Для плотных графов (Facebook) основное время тратится на рекурсивную фазу, которая хорошо распараллеливается.
   - Для разреженных графов (DBLP) фаза удаления рёбер может составлять 70% времени и требует оптимизации.
   - На очень малых графах (Random 500v) доминирует кэш-производительность, а не алгоритмическая сложность.

Практическая применимость. Метод эффективен для графов со степенью 5–100 и $k = 3, 4, 5$. Это охватывает большинство практических приложений в анализе социальных сетей и биологических графов. Для графов на 100–10K вершин метод обеспечивает очень быстрый подсчёт клик благодаря кэш-эффекту. Для больших графов (сотни тысяч вершин) метод остаётся приемлемым для $k \leq 5$, но требует больше оперативной памяти.

Направления дальнейших исследований:

1. **Обобщение на произвольный k.** В текущей реализации алгоритм для произвольного $k \geq 3$ уже реализован рекурсивно. Дальнейшая работа направлена на тестирование для больших значений k (вплоть до $k = 10$–15) на графах различной структуры.

2. **Батчевая обработка графа с разложением рёбер на независимые группы.** Перспективным направлением алгоритмической оптимизации является применение разложения множества рёбер графа на части таким образом, чтобы рёбра из разных частей не могли входить в одну клику. Это позволит различным группам потоков независимо обрабатывать разные группы рёбер исходного графа без конфликтов синхронизации.

3. **Параллелизация фазы удаления рёбер по общей окрестности.** Текущая реализация этапа фильтрации выполняется последовательно и составляет узкое место при многопоточности на разреженных графах (до 67% времени для DBLP при $k = 5$). Распараллеливание этой фазы с использованием OpenMP или более продвинутых методов может дать ускорение на 2–3× и значительно улучшить общую масштабируемость.

4. **Параллелизация матричных операций.** Умножение матриц и произведение Адамара можно ускорить, используя специализированные библиотеки, такие как GraphBLAS или Intel MKL, которые оптимизированы для разреженных матриц и работают на многоядерных системах. Это может дать ускорение на 2–5× на рекурсивной фазе.

5. **Исследование альтернативных ациклических ориентаций графа.** Вместо простой ID-based ориентации можно применить degree-based ориентацию (от вершин высокой степени к низкой), ориентации типа Goodrich–Pszona или core-based ориентацию, которые часто дают более сбалансированные правые окрестности, сокращают объём рекурсии и потенциально улучшают параллелизм.

6. **Сравнение с существующими open-source решениями.** Планируется провести детальное сравнительное исследование с современными солверами для подсчёта клик (Arb-count, binAryJoin, kClist, Pivoter, WCO) на общих тестовых данных из коллекции SNAP и других стандартных наборах, что позволит объективно оценить эффективность реализованного метода.

Список литературы

[1] Emelin M.D., Khlystov I.A., Malyshev D.S., Razvenskaya O.O. On linear algebraic algorithms for the subgraph matching problem and its variants // Optimization Letters. 2023. Vol. 17. P. 1–28. DOI: 10.1007/s11590-023-02001-z.

[2] Garey, M.R., Johnson, D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. — W.H. Freeman, 1979.

[3] Bron, C., Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph // Communications of the ACM. — 1973. — Vol. 16, No. 9. — P. 575–577.

[4] Ahmed, N.K., et al. Graphlet decomposition framework, algorithms, and applications // Knowledge and Information Systems. — 2017. — Vol. 50, No. 3. — P. 689–722.

[5] Chen, L., et al. A GraphBLAS approach for subgraph counting // arXiv preprint arXiv:1903.04395. — 2019.

[6] Latapy, M. Main-memory triangle computations for very large sparse, power-law graphs // Theoretical Computer Science. — 2008. — Vol. 407, No. 1-3. — P. 458–473.

[7] Leskovec, J., Sosič, R. SNAP: A general-purpose network analysis and graph mining library // arXiv preprint arXiv:1602.02136. — 2016.

[8] Watts, D., Strogatz, S. Collective dynamics of small-world networks // Nature. — 1998. — Vol. 393. — P. 440–442.

[9] Mattson, T., et al. Standards for graph algorithm primitives // Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC). — 2013. — P. 1–2.

Приложение А. Листинг ключевых функций

А.1. Класс SparseMatrix (фрагмент)

Листинг 1: Класс SparseMatrix (фрагмент)

```
class SparseMatrix {
private:
    std::vector<int> values;
    std::vector<int> col_indices;
    std::vector<int> row_pointers;
    int n_rows, n_cols;

public:
    explicit SparseMatrix(int n, int m)
        : n_rows(n), n_cols(m)
    {
        row_pointers.resize(n_rows + 1, 0);
    }

    SparseMatrix multiply(const SparseMatrix& B) const {
        SparseMatrix result(n_rows, B.n_cols);

        for (int i = 0; i < n_rows; ++i) {
            for (int j_idx = row_pointers[i];
                 j_idx < row_pointers[i + 1]; ++j_idx)
            {
                int j     = col_indices[j_idx];
                int a_val = values[j_idx];

                for (int k_idx = B.row_pointers[j];
                     k_idx < B.row_pointers[j + 1]; ++k_idx)
                {
                    int k     = B.col_indices[k_idx];
                    int b_val = B.values[k_idx];
                    result.add(i, k, a_val * b_val);
                }
            }
        }

        return result;
    }

    SparseMatrix hadamard(const SparseMatrix& B) const {
```

```
        SparseMatrix result(n_rows, n_cols);

        for (int i = 0; i < n_rows; ++i) {
            for (int j_idx = row_pointers[i];
                 j_idx < row_pointers[i + 1]; ++j_idx)
            {
                int j    = col_indices[j_idx];
                int b_ix = B.find(i, j);
                if (b_ix >= 0) {
                    result.add(i, j, values[j_idx] * B.values[b_ix]);
                }
            }
        }

        return result;
    }


    long long sum() const {
        long long s = 0;
        for (int v : values)
            s += v;
        return s;
    }
};
```

A.2. Алгоритм подсчёта треугольников

Листинг 2: Подсчёт треугольников

```
long long countTriangles(const SparseMatrix& A) {
    int n = A.size();
    SparseMatrix A2 = A.multiply(A);
    SparseMatrix H = A2.hadamard(A);
    long long s = H.sum();
    return s / 6;
}
```

A.3. Рекурсивный подсчёт k-клик (фрагмент)

Листинг 3: Рекурсивный подсчёт k-клик (фрагмент)

```
long long countKCliques(const std::vector<int>& vertices,
```

```
                        const std::vector<std::vector<int>>& adj,
                        int k,
                        int depth)
{
    if (k == 1)
        return static_cast<long long>(vertices.size());
    if (depth <= 1) {
        long long count = 0;

        #pragma omp parallel for reduction(+:count) schedule(dynamic)
        for (int idx = 0; idx < static_cast<int>(vertices.size());
           ++idx) {
            int v = vertices[idx];
            std::vector<int> neighbors;

            for (int u : adj[v]) {
                if (u > v)
                    neighbors.push_back(u);
            }

            if (!neighbors.empty()) {
                count += countKCliques(neighbors, adj, k - 1, depth
                   + 1);
            }
        }

        return count;
    }

    long long count = 0;

    for (int v : vertices) {
        std::vector<int> neighbors;

        for (int u : adj[v]) {
            if (u > v)
                neighbors.push_back(u);
        }

        if (!neighbors.empty()) {
            count += countKCliques(neighbors, adj, k - 1, depth + 1);
        }
    }
```

```
43
44      return count;
45  }
```