ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА»

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА МАТЕМАТИЧЕСКОЙ ТЕОРИИ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(ДИПЛОМНАЯ РАБОТА)
магистра

МЕТОДЫ РАЗМЕЩЕНИЯ ГРАФОВ В ПАМЯТИ КОМПЬЮТЕРА ДЛЯ ЗАДАЧИ ПОИСКА БЛИЖАЙШИХ СОСЕДЕЙ

Выполнил студент М2-ой группы
Монгуш Чингиз Омарович

_____
подпись студента

Научный руководитель:
кандидат физико-математических наук, доцент
Алексеев Дмитрий Владимирович

_____
подпись научного руководителя

Москва
2026

1. Аннотация

Работа посвящена проблеме приближённого поиска ближайших соседей (ANNS) — фундаментальной задаче, востребованной в рекомендательных системах, компьютерном зрении и обработке естественного языка. Одним из наиболее эффективных подходов является построение графовых индексов, однако нерегулярный доступ к памяти при обходе графа приводит к низкой утилизации кэша и замедлению поиска.
В работе исследуются методы переупорядочивания вершин графового индекса (graph reordering) для улучшения пространственной локальности данных. Рассмотрены современные алгоритмы GOrder и Rabbit Order, а также предложена новая метрика качества размещения, основанная на подсчёте рёбер, замкнутых в одной кэш-линии. Доказана NP-полнота задачи максимизации этой метрики.

Содержание

2. Введение

Задачу поиска ближайших соседей (Nearest Neighbor Search) можно сформулировать следующим образом. Пусть дано множество $D = \{x_1, x_2, \ldots, x_N\}$, элементы которого — векторы из $\mathbb{R}^d$, и задана метрика близости $\rho(x, y)$. Для запроса $q \in \mathbb{R}^d$ требуется найти k ближайших к нему элементов D. Качество приближённого поиска принято оценивать метрикой

$$\text{Recall@}k = \frac{|A \cap A^*|}{k},$$

где A — множество вершин, возвращённых алгоритмом, а A^* — множество истинных ближайших соседей.

2.1. Обзор литературы

Проблема поиска ближайших соседей активно исследуется несколько десятилетий, и за это время было предложено множество подходов. Их можно разделить на четыре основных класса: методы на основе хеширования, деревьев, векторного квантования и графовые методы.

- **Методы на основе хеширования (LSH).** Локально-чувствительное хеширование (Locality-Sensitive Hashing, LSH) использует специальные хеш-функции, отображающие близкие векторы в одну корзину с высокой вероятностью. Ближайшие соседи запроса ищутся среди векторов, попавших в ту же корзину. Существует множество улучшений базового подхода, включая многозондовые методы, data-dependent LSH и др. [19, 2, 24]. Разработаны также методы, основанные на подсчёте числа коллизий без вычисления расстояний, что ускоряет отбор кандидатов [3].
- **Методы на основе деревьев (KD-дерево, Annoy).** Эти методы рекурсивно разбивают пространство гиперплоскостями. Их главный недостаток — резкое падение эффективности с ростом размерности данных. Существуют алгоритмы, способные решать задачу точно за $O(\text{poly}(c) \cdot N)$, где c характеризует внутреннюю размерность датасета [6, 15]. Однако на практике при $d > 20$ они часто вырождаются в линейный перебор. Поэтому, несмотря на успехи в отдельных приложениях [16], деревья уступают более современным методам.
- **Методы векторного квантования (IVF-PQ, FAISS).** Идея состоит в сжатии векторов с помощью квантования и построении инвертированных индексов. Базовый метод Product Quantization (PQ) [11] разбивает вектор на подвектора и для каждого строит

кодовое слово. Расстояние аппроксимируется по предварительно вычисленным таблицам, что позволяет быстро сканировать сжатые представления. Для миллиардных коллекций применяется инвертированный файл (IVF): пространство разбивается на крупные кластеры, и внутри каждого кластера используется PQ. Библиотека FAISS реализует эти техники. Главное преимущество — крайне низкое потребление памяти, однако по точности и скорости при заданном бюджете памяти методы квантования нередко уступают графовым, особенно при высоких требованиях к recall.

- **Графовые методы (HNSW, HCNNG)**. Графовые индексы строят разреженный граф ближайших соседей и выполняют по нему жадный или лучевой поиск. Они не требуют априорных предположений о распределении данных и работают в произвольных метрических пространствах. Одним из первых успешных подходов стал Navigable Small World (NSW) [17]. Дальнейшим развитием является Hierarchical Navigable Small World (HNSW) [1], в котором вводится иерархия слоёв, позволяющая быстро сужать область поиска. HNSW демонстрирует рекордные показатели Recall@k при минимальном числе вычислений расстояний и де-факто является стандартом в области. Альтернативный алгоритм HCNNG [21] строит граф с помощью многократной иерархической кластеризации и использует эвристику направленного обхода, уменьшающую количество просматриваемых соседей. Графовые методы стабильно лидируют в бенчмарках ann-benchmarks [5] по соотношению скорость/точность на данных высокой размерности.

Анализ литературы показывает, что именно графовые методы (прежде всего HNSW и его варианты) достигают лучших результатов в задаче приближённого поиска ближайших соседей, особенно в пространствах большой размерности. Однако их практическая производительность лимитируется не столько вычислительной сложностью, сколько паттернами доступа к памяти. Обход графа нерегулярен: переходы между вершинами определяются динамически, что приводит к частым кэш-промахам (cache misses). Согласно [23], до 70% времени запроса может уходить на ожидание данных из DRAM. Следовательно, даже оптимальный с алгоритмической точки зрения индекс может работать медленно из-за неэффективного размещения вершин графа в оперативной памяти.

Таким образом, актуальной задачей является оптимизация размещения графового индекса в памяти компьютера с целью повышения локальности данных и сокращения числа кэш-промахов при поиске. Малое число работ, непосредственно оценивающих качество переупорядочивания вершин графа с помощью формальных метрик, дополнительно мотивирует настоящее исследование.

3. Устройство компьютерной памяти и его влияние на графовые алгоритмы

3.1. Устройство компьютерной памяти

Эффективность алгоритмов поиска ближайших соседей по графовым индексам критически зависит не только от вычислительной сложности, но и от характера обращений к памяти. В современных процессорных архитектурах время доступа к данным может различаться на порядки в зависимости от уровня иерархии памяти, в котором они находятся. В данном подразделе мы рассмотрим ключевые аспекты организации памяти, влияющие на производительность графовых алгоритмов.

Иерархия памяти

Современные вычислительные системы используют многоуровневую иерархию памяти для балансировки между скоростью доступа, объёмом и стоимостью хранения данных. На рис. 1 представлена упрощённая схема памяти типичного процессора.

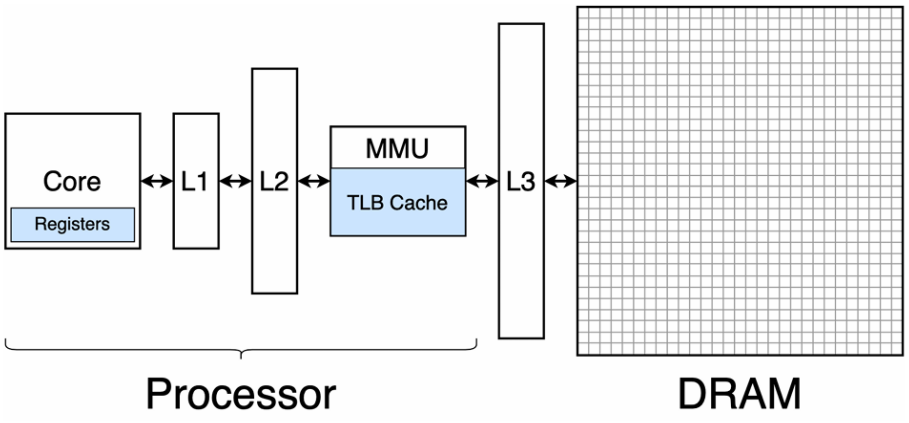

Рис. 1: Иерархия памяти современного процессора. Источник: адаптировано из [13].

Когда процессор получает инструкцию загрузить данные по определённому адресу, эти данные должны быть перемещены из основной памяти (DRAM) в регистры CPU перед тем, как с ними можно будет выполнять операции. Для ускорения этого процесса между ядром процессора и оперативной памятью располагаются промежуточные уровни хранения — кэш-память уровней L1, L2 и L3.

Как видно из таблицы 1, доступ к данным в DRAM может быть в 100 раз медленнее, чем доступ к данным в кэше L1. Поскольку графовые алгоритмы характеризуются нерегулярным паттерном доступа к па-

Таблица 1: Относительная стоимость доступа к различным уровням памяти для процессора Intel Core i7. Данные из [13].

Уровень	**Такты процессора**	**Задержка (прибл.)**
L1 кэш	4	∼1 нс
L2 кэш	10	∼3 нс
L3 кэш	40	∼10 нс
Оперативная память (DRAM)	400	∼100 нс

мяти, минимизация промахов кэша становится критически важной для достижения высокой производительности.

Организация кэш-памяти

Кэш-память организована в виде набора *кэш-линий* — небольших блоков данных фиксированного размера (обычно 64 байта), которые всегда перемещаются между уровнями памяти совместно. Когда процессор запрашивает слово данных, в кэш загружается вся кэш-линия, содержащая это слово. Это свойство лежит в основе *пространственной локальности*: если программа последовательно обращается к соседним адресам памяти, вероятность кэш-попадания существенно возрастает.

При кэш-промахе (cache miss) одна из существующих кэш-линий вытесняется для освобождения места под новую. В большинстве систем используется политика вытеснения *Least Recently Used* (LRU) — удаляется линия, к которой дольше всего не обращались.

Определение 1 (Принцип локальности)**.** Алгоритм демонстрирует *пространственную локальность*, если обращается к элементам памяти, физически расположенным близко друг к другу. *Временная локальность* наблюдается, когда программа многократно обращается к одним и тем же адресам памяти. Алгоритмы, соблюдающие принцип локальности, получают максимальную выгоду от иерархии кэш-памяти.

Важно отметить, что пространственная локальность может ускорять выполнение программы даже в том случае, когда отдельные объекты данных превышают размер кэш-линии. Например, при обработке массива структур, даже если каждая структура занимает несколько килобайт, кэш может ускорить доступ к вспомогательным полям (таким как указатели на соседей в графе), которые помещаются в одну линию.

Модель идеального кэша

Для теоретического анализа алгоритмов, работающих с иерархией памяти, часто используется *модель идеального кэша* (Ideal Cache Model),

предложенная в классической работе по cache-oblivious алгоритмам [8]. В этой модели кэш состоит из M кэш-линий по B объектов в каждой (обычно слов). Программа взаимодействует только с кэшем; если запрошенный объект находится в кэше, происходит кэш-попадание, иначе — кэш-промах, вызывающий загрузку соответствующей линии из основной памяти. Вытеснение выполняется по оптимальной off-line стратегии Белади (вытесняется линия, доступ к которой произойдёт позже всех). Предполагается также, что кэш полностью ассоциативен и выполняется условие «высокого кэша» ($M = \Omega(B^2)$), которое обычно справедливо для реальных систем.

Эта модель позволяет анализировать *кэш-сложность* алгоритма — число кэш-промахов $Q(n; M, B)$ на задаче размера n. Для графового поиска ближайших соседей, например, важно, чтобы соседи вершины, к которым вероятен переход, располагались в одной или соседних кэш-линиях. Как показано в теоретической части (гл. 7), оптимизация размещения графа под модель идеального кэша эквивалентна максимизации числа рёбер, замкнутых внутри линий, и является NP-полной задачей. Это мотивирует разработку эвристических алгоритмов переупорядочивания, таких как Gorder и предлагаемый в работе CLAR, которые приближают оптимальное решение.

Аппаратное и программное предвыборка данных

Современные процессоры реализуют механизмы *предвыборки* (prefetching) данных для скрытия задержек доступа к памяти. Предвыборка может осуществляться как аппаратно, так и программно:

- **Аппаратная предвыборка** использует эвристики или простые модели обучения для автоматического определения кэш-линий, которые с высокой вероятностью понадобятся в ближайшем будущем, и заблаговременно загружает их в кэш.
- **Программная предвыборка** реализуется путём явной вставки в код специальных инструкций (compiler intrinsics), указывающих процессору загрузить данные по заданному адресу.

Хотя комбинация обоих подходов теоретически может дать наилучший результат, на практике взаимодействие между аппаратной и программной предвыборкой может быть сложным и даже антагонистическим [12]. Поэтому улучшение пространственной локальности доступа к памяти остаётся более надёжным способом повышения производительности, чем попытки тонкой настройки предвыборки.

Виртуальная память и TLB-кэш

При обращении к памяти процессор оперирует *виртуальными адресами*, которые должны быть транслированы в *физические адреса* оперативной памяти. Этот процесс осуществляется блоком управления памятью (Memory Management Unit, MMU). Поскольку полная трансляция для каждого обращения к памяти была бы слишком затратной, результаты преобразования кэшируются в специальном буфере — *Translation Lookaside Buffer* (TLB).

TLB, подобно кэшам данных, подчиняется принципу локальности: если программа обращается к адресам, принадлежащим одной и той же странице памяти, количество промахов TLB снижается. Для графовых алгоритмов это означает, что размещение смежных вершин графа в пределах одной или соседних страниц памяти может дополнительно ускорить выполнение за счёт снижения накладных расходов на трансляцию адресов.

NUMA-архитектуры

В многопроцессорных серверных системах широко используется архитектура с неравномерным доступом к памяти (Non-Uniform Memory Access, NUMA). Каждый процессор или группа ядер имеет локальную память, доступ к которой осуществляется быстро, тогда как обращение к памяти другого процессора происходит с большей задержкой. Такая топология делает локальность размещения данных ещё более критичной: вершины графа, расположенные в «чужом» банке памяти, будут загружаться в кэш с заметными потерями времени. Переупорядочивание вершин с учётом NUMA-границ (например, группировка вершин, часто посещаемых одним потоком, в одной NUMA-зоне) может дополнительно снизить среднюю задержку обхода графа.

Влияние на графовые алгоритмы

Графовые алгоритмы, включая поиск ближайших соседей по индексам HCNNG, HNSW, характеризуются следующими особенностями доступа к памяти:

1. **Нерегулярный паттерн обхода**: переходы между вершинами графа определяются структурой данных, а не последовательным доступом, что затрудняет предсказание следующих обращений.
2. **Косвенная адресация**: для получения данных вершины необходимо сначала прочитать указатель или индекс, что создаёт дополнительные зависимости по данным.

3. **Разнородный размер объектов**: вершины графа могут содержать как фиксированные метаданные, так и переменное количество рёбер, что усложняет эффективное размещение в памяти.

Эти особенности приводят к низкому коэффициенту попаданий в кэш и, как следствие, к существенным задержкам из-за обращений к медленной памяти. *Переупорядочивание графа* (graph reordering) — техника оптимизации размещения вершин в памяти, при которой смежные или часто совместно посещаемые вершины получают близкие адреса — позволяет существенно улучшить локальность доступа. Когда поиск переходит в новую вершину, высока вероятность, что её соседи (или хотя бы часть из них) уже находятся в кэше, поскольку их адреса расположены рядом. В работе [7] показано, что применение Gorder и Reverse Cuthill-McKee к графу HNSW снижает долю промахов L3-кэша с 13.5% до 8.3%, а TLB-промахов — с 3.85% до 2.14%, что даёт ускорение поиска до 40%. Таким образом, задача переупорядочивания напрямую связана с максимизацией числа рёбер, замкнутых в пределах кэш-линии, и исследование эффективных методов такого переупорядочивания для графов ближайших соседей составляет основное содержание данной работы.

4. Задача поиска ближайших соседей

4.1. Векторное представление данных

Векторное представление объектов (эмбеддинги) является фундаментальным инструментом во многих областях компьютерных наук, включая машинное обучение, компьютерное зрение, обработку естественного языка и рекомендательные системы. Каждый объект (изображение, текст, аудиозапись) отображается в точку многомерного евклидова пространства $\mathbb{R}^d$. Такое представление позволяет сравнивать объекты по их семантической близости с помощью метрик расстояния.

Определение 2 (Векторная база данных)**.** Пусть $\mathcal{D} \subset \mathbb{R}^d$ — конечное множество векторов, называемое векторной базой данных, где $d \in \mathbb{N}$ — размерность пространства, а $N = |\mathcal{D}|$ — количество векторов в базе. Каждый элемент $\mathbf{v} \in \mathcal{D}$ представляет собой вектор $\mathbf{v} = [v_1, v_2, \ldots, v_d]^\top$, где $v_i \in \mathbb{R}$.

4.2. Точный поиск ближайших соседей

Задача поиска ближайших соседей (Nearest Neighbor Search, NNS) заключается в нахождении в базе данных $\mathcal{D}$ векторов, наиболее близких к заданному запросу в смысле некоторой функции расстояния.

Определение 3 (Функция расстояния)**.** Функция $dist : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ называется метрикой расстояния, если для любых $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^d$ выполняются свойства:

- $dist(\mathbf{x}, \mathbf{y}) = 0 \iff \mathbf{x} = \mathbf{y}$ (тождественность),
- $dist(\mathbf{x}, \mathbf{y}) = dist(\mathbf{y}, \mathbf{x})$ (симметричность),
- $dist(\mathbf{x}, \mathbf{z}) \leq dist(\mathbf{x}, \mathbf{y}) + dist(\mathbf{y}, \mathbf{z})$ (неравенство треугольника).

Один из самых распространенных расстояний является стандартная евклидовая метрика:

$$dist(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}. \tag{1}$$

Определение 4 (Поиск k ближайших соседей (k-NN))**.** Для заданного запроса $\mathbf{q} \in \mathbb{R}^d$ и целого $k \geq 1$ задача k-NN состоит в нахождении множества $\mathcal{A}^* \subset \mathcal{D}$ такого, что $|\mathcal{A}^*| = k$ и

$$\forall \mathbf{v} \in \mathcal{A}^*,\ \forall \mathbf{u} \in \mathcal{D} \setminus \mathcal{A}^* : \quad dist(\mathbf{q}, \mathbf{v}) \leq dist(\mathbf{q}, \mathbf{u}). \tag{2}$$

Наивный алгоритм решения k-NN — полный перебор всех векторов базы с вычислением расстояний до запроса. Его временная сложность составляет $O(d \cdot N)$, что для современных масштабов данных (миллионы и миллиарды векторов в сотнях измерений) делает его неприменимым на практике. Более того, известное явление «проклятия размерности» [10] приводит к тому, что даже продвинутые структуры данных (например, KD-деревья) при $d > 20$ вырождаются до линейного сканирования.

4.3. Приближенный поиск ближайших соседей

Для преодоления вычислительных ограничений точного поиска применяются методы приближенного поиска ближайших соседей (Approximate Nearest Neighbor Search, ANNS), которые жертвуют гарантией точности ради значительного ускорения выполнения запросов.

Определение 5 (Приближенный поиск k ближайших соседей)**.** Для заданного запроса $\mathbf{q} \in \mathbb{R}^d$, целого $k \geq 1$ и построенного индекса $\mathcal{I}$ над $\mathcal{D}$ задача ANNS состоит в нахождении множества $\mathcal{A} \subset \mathcal{D}$, $|\mathcal{A}| = k$, которое аппроксимирует точное решение $\mathcal{A}^*$. Качество аппроксимации оценивается метрикой $Recall@k$:

$$Recall@k = \frac{|\mathcal{A} \cap \mathcal{A}^*|}{k}. \tag{3}$$

Цель алгоритма — максимизировать $Recall@k$ при минимальном количестве вычислений расстояний.

Современные методы ANNS можно разделить на несколько классов [22, 14]:

- **Методы на основе хеширования (LSH)**: используют семейства локально-чувствительных хеш-функций для отображения близких векторов в одни и те же корзины с высокой вероятностью.
- **Методы на основе деревьев (KD-tree, Annoy)**: рекурсивно разбивают пространство гиперплоскостями или кластерами, сужая область поиска.
- **Методы векторного квантования (IVF-PQ, OPQ)**: кластеризуют пространство и сжимают векторы с помощью кодовых книг для быстрого вычисления приближенных расстояний.

4.4. Графовые методы ANNS

Графовые методы на сегодняшний день демонстрируют наилучший компромисс между точностью и скоростью поиска, особенно на данных высокой размерности [9, 1]. В основе графового подхода лежит построение

ориентированного или неориентированного графа $G = (V, E)$, где вершины V соответствуют векторам из $\mathcal{D}$, а рёбра E соединяют вершины, которые считаются «близкими» согласно выбранной метрике. Как правило, такими рёбрами являются k ближайших соседей каждой вершины, прошедшие процедуры прореживания и диверсификации для улучшения навигационных свойств графа.

Определение 6 (Графовый индекс для ANNS)**.** Графовый индекс представляет собой граф $G = (V, E)$, построенный на множестве векторов $\mathcal{D}$, где:

- $V = \mathcal{D}$ — каждая вершина соответствует уникальному вектору данных;
- $E \subseteq V \times V$ — множество рёбер, определяющих отношение соседства (например, k ближайших соседей или рёбра, удовлетворяющие условию разреженности).

Поиск по графу выполняется с помощью *лучевого поиска* (beam search) — алгоритма, который поддерживает динамический список кандидатов и итеративно расширяет наиболее перспективные направления.

Лучевой поиск (beam search)

В отличие от простого *жадного поиска* (greedy search), который на каждом шаге переходит к лучшему соседу текущей вершины и останавливается в локальном минимуме, лучевой поиск поддерживает ограниченное множество кандидатов. Формально, пусть $\mathbf{q}$ — запрос. Алгоритм использует очередь с приоритетом $\mathcal{Q}$ и множество уже обработанных вершин. На каждом шаге из $\mathcal{Q}$ извлекается вершина $\mathbf{v}$ с минимальным расстоянием $dist(\mathbf{v}, \mathbf{q})$, после чего все её непосещённые соседи $\mathbf{u} \in N(\mathbf{v})$ добавляются в очередь. Параллельно поддерживается множество лучших ef кандидатов, где ef задаёт ширину луча. Поиск завершается, когда очередь исчерпана или когда лучший кандидат в очереди уже не может улучшить текущее множество результатов. В качестве ответа возвращаются k вершин с наименьшим расстоянием до запроса среди поддерживаемых кандидатов.

В индексе HNSW лучевой поиск используется на базовом слое и управляется параметром $efSearch$, который задаёт размер очереди кандидатов и тем самым регулирует компромисс между точностью поиска и временем выполнения. Увеличение $efSearch$ обычно повышает Recall@k, но требует большего числа вычислений расстояний и случайных обращений к памяти.

Для HCNNG используется другой механизм поиска: он опирается на иерархическую кластеризацию, соединение точек внутри кластеров с

помощью MST и получение стартовых вершин через глобальные KD-tree, поэтому его не следует описывать как прямое применение beam search.

Основные представители графовых методов

- **HCNNG** (Hierarchical Clustering-based Nearest Neighbor Graph) [21] — строит граф путём многократной иерархической кластеризации и слияния MST-подграфов; при поиске применяет эвристику направленного обхода, сокращающую число вычислений расстояний.
- **HNSW** (Hierarchical Navigable Small World) [1] — многоуровневый граф, где верхние уровни содержат мало вершин и длинные рёбра, позволяя быстро приблизиться к окрестности запроса; нижние уровни уточняют результат.
- **NSG** (Navigating Spreading-out Graph) [9] — строит граф с ограничением на максимальную степень вершин и контролем углового разнообразия соседей.

Производительность и проблема размещения графа в памяти

Несмотря на алгоритмическую эффективность, практическое время выполнения запросов в графовых индексах существенно зависит от паттернов доступа к памяти. При лучевом поиске переходы между вершинами определяются динамически, что порождает нерегулярный доступ в оперативную память. Такие случайные обращения вызывают высокую частоту кэш-промахов (cache misses) и промахов TLB, что, как было показано в главе 3, может замедлять поиск вплоть до того, что до 70% времени уходит на ожидание данных из DRAM [23].

5. Графовые индексы

Графовые индексы являются одним из наиболее эффективных классов структур для приближённого поиска ближайших соседей в пространствах высокой размерности. Их основная идея состоит в том, чтобы представить множество объектов в виде разреженного графа, в котором вершины соответствуют векторам признаков, а рёбра соединяют «близкие» объекты. При поиске новый запрос не сравнивается со всеми элементами коллекции; вместо этого алгоритм перемещается по графу, последовательно переходя к более перспективным вершинам.

В контексте данной работы особый интерес представляют два индекса: HNSW (Hierarchical Navigable Small World) и HCNNG. Оба метода используют графовую структуру и ориентированы на эффективный приближённый поиск ближайших соседей, однако отличаются принципом построения, организацией обхода и требованиями к памяти. HNSW строит многослойный навигационный граф и обеспечивает очень высокое качество поиска при умеренных затратах на построение. HCNNG, в свою очередь, опирается на иерархическую кластеризацию и направленный обход.

5.1. HCNNG

Индекс HCNNG представляет собой графовый метод приближённого поиска ближайших соседей (ANNS), сочетающий иерархическую кластеризацию и эвристический направленный обход. Его ключевая идея заключается в построении разреженного навигационного графа, в котором каждая вершина соединена лишь с ограниченным числом соседей. Благодаря этому поиск может быстро приближаться к области, содержащей ответ, не рассматривая всю коллекцию объектов.

5.1.1. Построение графа

Построение HCNNG состоит из двух основных этапов [21]:

1. многократное применение процедуры иерархической кластеризации к исходному множеству векторов;
2. объединение локальных графов, полученных внутри кластеров, в единый глобальный граф.

Процедура иерархической кластеризации. Рекурсивная процедура `HierarchicalClustering` принимает на вход множество точек P и минимальный размер кластера n. Если $|P| < n$, рекурсия завершается, а точки внутри кластера соединяются с помощью модифицирован-

ного минимального остовного дерева с ограничением степени вершин (MST3). Ограничение степени тремя ребрами позволяет сохранить разреженность графа и при этом избежать чрезмерной потери связности.

Если размер множества превышает n, случайным образом выбираются две точки p_1 и p_2, после чего множество P разбивается на два подмножества:

$$P_1 = \{p \in P \mid D(p, p_1) < D(p, p_2)\}, \quad P_2 = \{p \in P \mid D(p, p_1) \geq D(p, p_2)\},$$

где D — функция расстояния, обычно евклидова метрика. Далее процедура рекурсивно применяется к P_1 и P_2. Такой подход не требует заранее задавать число кластеров и хорошо адаптируется к структуре данных.

Слияние графов. Однократное выполнение иерархической кластеризации даёт сильно несвязный граф, поскольку рёбра формируются только внутри конечных кластеров. Чтобы повысить связность, кластеризация выполняется m раз, после чего все рёбра объединяются. Формально, пусть $C = \{c_1, c_2, \ldots, c_k\}$ — множество кластеров, полученных в результате одного запуска процедуры. Для каждого c_i строится локальный граф $G_i = (V_i, E_i)$, где V_i — вершины кластера, а E_i — рёбра, сформированные алгоритмом MST3. Итоговый граф определяется как

$$V' = \bigcup_{i=1}^{k} V_i, \qquad E' = \bigcup_{i=1}^{k} E_i.$$

Параметр m определяет, насколько плотным и связным получится итоговый граф: увеличение числа запусков повышает вероятность появления дополнительных «мостов» между областями данных, но одновременно увеличивает время построения и объём памяти. Параметр n обычно выбирается пропорционально $\sqrt{N}$, где N — размер коллекции.

С точки зрения вычислительной сложности построение HCNNG является затратной процедурой, однако получаемый граф имеет малую среднюю степень вершин, что обеспечивает линейную сложность хранения по числу объектов. Это особенно важно для больших коллекций, где именно объём памяти и характер доступа к ней часто становятся основным ограничением производительности.

5.1.2. Направленный поиск (Guided Search)

В классическом жадном поиске на графе на каждом шаге приходится вычислять расстояния от запроса до всех соседей текущей вершины. При высокой степени вершин это приводит к значительным накладным расходам. Алгоритм направленного поиска в HCNNG снижает число вычислений расстояний за счёт геометрической эвристики, позволяющей отсеивать заведомо неперспективных соседей.

Геометрическая эвристика. Находясь в вершине p, поиск рассматривает только тех соседей $v \in N(p)$, которые лежат в том же направлении относительно p, что и запрос q. Формально, сосед v считается перспективным, если

$$(v - p) \cdot (q - p) > 0.$$

Это означает, что направление к соседу образует с направлением к запросу острый угол, то есть переход к нему с высокой вероятностью приближает поиск к цели. Если среди отобранных соседей не находится более близкой вершины, алгоритм расширяет область просмотра, включая соседние направления и возвращаясь к ранее посещённым вершинам.

Выбор начальной вершины. Эффективность поиска существенно зависит от стартовой точки. Вместо случайного выбора HCNNG использует ансамбль KD-деревьев, построенных на случайных подвыборках данных. Для запроса q выполняется спуск по каждому дереву до листа, после чего среди полученных кандидатов выбирается ближайшая точка, которая и становится начальной вершиной графового обхода. Такая инициализация повышает вероятность старта вблизи истинного ближайшего соседа и уменьшает число требуемых сравнений [21].

Алгоритм обхода. Поиск выполняется с использованием очереди с приоритетами и механизма backtracking. На каждом шаге из текущей вершины рассматриваются только те соседи, которые удовлетворяют геометрической эвристике. Если найден более близкий кандидат, алгоритм переходит к нему и продолжает обход. Если перспективных переходов нет, происходит возврат к ранее посещённым вершинам и расширение области поиска до достижения локального минимума [21].

Влияние параметров поиска. Ключевым параметром является максимальное число вычислений расстояний T, которое задаёт компромисс между скоростью и полнотой поиска. Увеличение T обычно повышает recall, но приводит к росту времени ответа. В экспериментальных исследованиях HCNNG демонстрирует более высокое качество поиска по сравнению с рядом близких графовых методов при фиксированном бюджете вычислений, особенно в задачах 1-NN и 10-NN.

Таким образом, HCNNG сочетает компактный разреженный граф и геометрически направленный обход. Для данной работы он важен прежде всего как пример индекса, в котором структура графа и порядок размещения вершин в памяти тесно связаны с производительностью поиска.

5.2. HNSW

Алгоритм HNSW (Hierarchical Navigable Small World), предложенный Мальковым и Яшуниным [1], является одним из наиболее эффективных методов приближённого поиска K ближайших соседей в пространствах высокой размерности. В отличие от классических деревьев поиска, HNSW полностью основан на графовой структуре и не требует отдельного этапа грубого отбора кандидатов. Его эффективность достигается за счёт многослойной организации графа: верхние уровни содержат немного вершин и служат для быстрого перемещения между удалёнными областями пространства, а нижний уровень обеспечивает точное локальное уточнение ответа.

5.2.1. Структура индекса

Индекс HNSW состоит из слоёв $l = 0, 1, \ldots, L$, где L — максимальный уровень графа. Нижний слой $l = 0$ содержит все элементы коллекции, а на более высоких уровнях количество вершин экспоненциально уменьшается. Для каждого объекта при вставке случайно выбирается максимальный уровень, на котором он будет присутствовать. Такое распределение уровней позволяет сформировать структуру с хорошими навигационными свойствами и небольшим числом длинных переходов.

На каждом слое строится разреженный proximity graph, который приближает граф ближайших соседей на соответствующем масштабе. Верхние слои содержат более длинные рёбра и помогают быстро перемещаться по графу, а нижние слои уточняют ответ в локальной окрестности запроса [1].

5.2.2. Построение индекса

Построение HNSW реализуется как последовательная вставка элементов. Для каждого нового объекта:

1. случайным образом выбирается его максимальный уровень;
2. начиная с верхнего уровня, выполняется жадный поиск ближайшей вершины, которая используется как точка входа на следующий слой;
3. на слоях ниже максимального выполняется более тщательный поиск кандидатов с использованием параметра `efConstruction`;
4. из найденных кандидатов выбирается ограниченное число соседей, с которыми устанавливаются двунаправленные связи;
5. при превышении допустимой степени у соседей выполняется обрезка списка связей.

Эта схема позволяет поддерживать граф достаточно разреженным и в то же время сохранять высокое качество навигации.

5.2.3. Поиск по индексу

Поиск K ближайших соседей в HNSW начинается с точки входа на верхнем уровне. На каждом уровне выше нулевого выполняется жадный спуск до локального минимума, после чего точка входа обновляется. На нижнем уровне применяется лучевой поиск с параметром `efSearch`, который определяет число кандидатов, рассматриваемых одновременно.

Интуитивно HNSW сначала быстро приближается к нужной области пространства, а затем аккуратно уточняет ответ на нижнем слое. Благодаря этому поиск остаётся быстрым даже при больших объёмах данных. На практике сложность запроса обычно растёт значительно медленнее, чем линейно, хотя точные оценки зависят от размерности данных, выбора параметров и структуры коллекции.

5.2.4. Параметры HNSW

Ключевыми параметрами алгоритма являются:

- M — максимальное число связей, устанавливаемых для вершины на слое. Он влияет на степень графа, объём памяти и скорость поиска;
- `efConstruction` — размер динамического списка кандидатов при построении индекса. Чем он больше, тем выше качество графа, но тем дороже построение;
- `efSearch` — размер списка кандидатов при выполнении запроса. Увеличение этого параметра повышает recall, но увеличивает время ответа;
- m_L — параметр распределения уровней вершин;
- $M_{\max}$ и $M_{\max 0}$ — ограничения на число рёбер у вершины на верхних уровнях и на базовом слое.

6. Переупорядочивание вершин графа

Одним из основных препятствий на пути повышения производительности графовых алгоритмов является низкая локальность обращений к памяти, обусловленная нерегулярной структурой графов [4]. При обходе графа, например, в процессе поиска ближайших соседей, доступ к вершинам и их рёбрам часто приводит к промахам кэша и трансляции адресов (TLB misses), что существенно замедляет выполнение запросов [7]. Переупорядочивание вершин (graph reordering) представляет собой технику оптимизации размещения данных в памяти, при которой близкие в графе вершины получают близкие идентификаторы и, следовательно, хранятся в смежных ячейках памяти. Это улучшает пространственную и временную локальность, позволяя эффективнее использовать иерархию кэшей центрального процессора.

Существует множество подходов к переупорядочиванию графов: от простых эвристик (сортировка по степени) до методов, основанных на разрезании гиперграфов или спектральной кластеризации. Однако многие из них либо не обеспечивают достаточного качества упорядочивания для графов ближайших соседей, либо требуют неприемлемо большого времени на перестроение индекса. Алгоритм **Rabbit Order** [4] был предложен как первый метод *just-in-time* параллельного переупорядочивания, способный одновременно достигать высокой локальности и малого времени работы, что критически важно для динамически обновляемых графов и сценариев поиска ближайших соседей, где индекс может перестраиваться с каждым изменением модели.

6.1. Rabbit Order

Rabbit Order решает задачу переупорядочивания вершин графа с помощью двух ключевых идей: (1) *иерархическое упорядочивание на основе сообществ* (hierarchical community-based ordering) для максимизации локальности и (2) *параллельная инкрементальная агрегация* (parallel incremental aggregation) для быстрого извлечения сообществ.

6.1.1. Иерархическое упорядочивание на основе сообществ

Реальные графы, в том числе графы ближайших соседей, часто обладают иерархической сообщественной структурой: плотные группы вершин объединяются в сообщества, которые, в свою очередь, формируют более крупные сообщества. Внутри сообщества вершины имеют высокую связность, и при обходе графа алгоритм поиска будет часто переходить между вершинами одного сообщества. Rabbit Order эксплуатирует это свойство, рекурсивно размещая вершины одного сообщества в смежных

участках памяти, причём вложенные сообщества отображаются на соответствующие уровни кэш-памяти (L1, L2, L3). Такой подход гарантирует, что при обращении к одной вершине сообщества её соседи с высокой вероятностью уже будут находиться в кэше, что снижает количество промахов и межпроцессорных коммуникаций.

6.1.2. Параллельная инкрементальная агрегация

Для быстрого построения иерархии сообществ Rabbit Order использует технику инкрементальной агрегации, изначально разработанную для детектирования сообществ по модулярности [20]. Алгоритм последовательно выбирает вершины (в порядке возрастания степени) и сливает каждую с тем соседом, который даёт наибольший прирост модулярности $\Delta Q(u, v)$, вычисляемый как

$$\Delta Q(u, v) = 2 \left\{ \frac{w_{uv}}{2m} - \frac{d(u)d(v)}{(2m)^2} \right\},$$

где w_{uv} — вес ребра между u и v, $d(\cdot)$ — взвешенная степень вершины, а m — общее число рёбер в исходном графе. Если $\Delta Q > 0$, вершина u поглощается вершиной v, которая становится представителем сообщества; в противном случае u остаётся самостоятельной вершиной верхнего уровня. Процесс повторяется для всех вершин, причём уже объединённые сообщества могут далее сливаться друг с другом. В результате строится дендрограмма сообществ, где листья соответствуют исходным вершинам, а корни — сообществам верхнего уровня.

Для ускорения работы на многоядерных процессорах Rabbit Order реализует *ленивую агрегацию* (lazy aggregation) и использует атомарные операции вместо блокировок. При слиянии вершины u в v информация о поглощении записывается в специальный массив *dest*, а фактическое обновление рёбер откладывается до момента, когда эти данные потребуются при вычислении прироста модулярности для других вершин. Это позволяет минимизировать объём данных, модифицируемых атомарно, и обеспечивает высокую масштабируемость (ускорение до 17.4× на 48 потоках).

6.1.3. Генерация финального порядка

После построения дендрограммы сообществ Rabbit Order генерирует новую нумерацию вершин путём обхода дендрограммы в глубину (DFS) начиная с каждого корневого сообщества. Вершины, принадлежащие одному сообществу и его подсообществам, получают последовательные идентификаторы, что отражает иерархическую близость. Полученный порядок используется для переупорядочивания массивов CSR-представления графа.

6.1.4. Применимость к графам ближайших соседей

Хотя Rabbit Order изначально оценивался на задачах PageRank, BFS и других классических графовых алгоритмах [4], его свойства делают его привлекательным и для графов ближайших соседей. Графы HNSW, PANNG и другие, используемые в поиске, обладают выраженной кластерной структурой, соответствующей областям пространства признаков с высокой плотностью. Иерархическое упорядочивание Rabbit Order естественным образом группирует вершины одного кластера, улучшая локальность при beam search. Прямые эксперименты с Rabbit Order на индексах ближайших соседей в литературе пока отсутствуют, однако его способность переупорядочивать графы с миллиардом рёбер за десятки секунд и достигнутое ускорение PageRank в среднем в 2,2 раза [4] позволяют рассматривать его как перспективный метод. Для сравнения, алгоритм Gorder (рассмотренный ниже) уже был успешно применён к HNSW и дал сокращение времени запроса до 40% [7], что подтверждает эффективность объектно-ориентированного переупорядочивания в задаче поиска ближайших соседей.

Таким образом, Rabbit Order представляет собой эффективный и масштабируемый метод переупорядочивания, который может быть интегрирован в системы поиска ближайших соседей для снижения задержек и повышения пропускной способности.

6.2. Gorder

Одним из наиболее эффективных подходов к оптимизации размещения графа в памяти является алгоритм **Gorder** (Graph Ordering), предложенный Вэй и др. [23]. В отличие от легковесных методов, основанных только на локальных характеристиках вершин (например, сортировка по степени), Gorder явно формулирует задачу максимизации кэш-локальности и строит упорядочивание, приближающее глобально оптимальное решение.

6.2.1. Формальная постановка

Пусть дан ориентированный или неориентированный граф $G = (V, E)$, где V — множество вершин, E — множество рёбер. Переупорядочивание состоит в нахождении биекции $P : V \to \{1, 2, \ldots, |V|\}$, которая каждой вершине $v \in V$ ставит в соответствие уникальный индекс (адрес в памяти). Цель — расположить вершины, которые часто используются совместно в ходе графовых алгоритмов, в близких позициях, чтобы при загрузке кэш-линии в процессор вместе с запрошенной вершиной в кэш попадали и её потенциальные соседи.

Gorder оптимизирует следующую целевую функцию:

$$P_{\mathrm{GO}} = \arg\max_{P} \sum_{\substack{u,v \in V \\ |P(u)-P(v)|<w}} \big(S_s(u,v) + S_n(u,v)\big), \tag{4}$$

где w — размер окна (гиперпараметр алгоритма), а слагаемые S_s и S_n определены как:

- $S_s(u,v)$ — количество общих рёбер между u и v (для ориентированного графа — сумма прямых связей в обоих направлениях);
- $S_n(u,v) = |N_{\mathrm{in}}(u) \cap N_{\mathrm{in}}(v)|$ — число общих входящих соседей.

Таким образом, Gorder поощряет размещение рядом вершин, которые либо соединены ребром, либо имеют много общих «родителей», что характерно для графов со значительным перекрытием окрестностей.

6.2.2. Жадный алгоритм построения

Поскольку задача максимизации (4) является NP-трудной, авторы Gorder предлагают эффективный жадный эвристический алгоритм, гарантирующий аппроксимацию с фактором $1/(2w)$ [23]. Процедура итеративно строит упорядочивание, на каждом шаге выбирая следующую вершину для размещения.

Пусть на текущий момент уже упорядочено подмножество $U \subset V$ и им присвоены первые $|U|$ индексов. Для каждой ещё не размещённой вершины $v \in V \setminus U$ вычисляется *показатель близости* к уже упорядоченным вершинам, попадающим в окно длиной w от конца построенной последовательности:

$$\mathrm{score}(v) = \sum_{u \in U_{\mathrm{window}}} \big(S_s(u,v) + S_n(u,v)\big), \tag{5}$$

где U_{window} — последние $\min(w, |U|)$ вершин, уже получивших индексы. Вершина с максимальным значением $\mathrm{score}(v)$ выбирается следующей и получает очередной порядковый номер. При равенстве очков предпочтение отдаётся вершине с большей степенью. Начальная вершина выбирается как вершина с максимальной степенью.

Вычислительная сложность алгоритма составляет $O\big((|V| + |E|) \cdot w \cdot \log |V|\big)$ при использовании очереди с приоритетом для отслеживания наилучшего кандидата, что на практике приемлемо для графов с миллионами вершин. Время работы Gorder растёт с увеличением w, однако качество результирующего упорядочивания при этом улучшается лишь до определённого предела. Эксперименты показывают, что значения w

в диапазоне от 64 до 2048 дают наилучший компромисс между скоростью вычисления и достигаемым ускорением графовых алгоритмов [23, 7]. При использовании меньшего окна, например $w = 5$, алгоритм работает быстрее и имеет лучшую гарантию аппроксимации, но может уступать более широким окнам по итоговой локальности.

6.2.3. Применение к графам ближайших соседей

В контексте поиска ближайших соседей графы (например, HNSW или k-NN графы) обладают специфической структурой: они относительно плотны, но благодаря эвристикам прореживания распределение степеней далеко от регулярного [7]. Gorder, в отличие от методов, опирающихся только на степень вершины, способен улавливать нетривиальные закономерности совместной встречаемости вершин в ходе поискового луча (beam search). Как показано в работе [7], применение Gorder к HNSW-графу на наборах данных масштаба SIFT100M и DEEP100M приводит к сокращению времени запроса на 20–40%, причём выигрыш растёт с увеличением размера набора и требуемой полноты (recall). Дополнительные измерения с помощью аппаратных счётчиков (perf) подтверждают, что ускорение обусловлено значительным снижением доли кэш-промахов на уровнях L1, L2 и L3, а также промахов TLB [7].

Таким образом, Gorder представляет собой мощный и хорошо обоснованный метод переупорядочивания, который особенно эффективен для задач с интенсивным обходом графа, таких как поиск ближайших соседей. Несмотря на более высокую вычислительную стоимость по сравнению с легковесными эвристиками, его применение оправдано тем, что время переупорядочивания на порядок меньше времени построения самого индекса, а ускорение запросов окупает эти затраты при многократном использовании индекса.

7. Теоретический анализ переупорядочивания вершин графа

В данном разделе мы формализуем задачу оптимизации переупорядочивания графа с точки зрения эффективности использования кэш-памяти, докажем её NP-полноту и проведём теоретический анализ существующих алгоритмов, включая Gorder и Rabbit Order. Основное внимание уделяется связи между комбинаторной метрикой $M(P)$ (число рёбер, попадающих в одну кэш-линию) и более тонкой характеристикой $\mathrm{CE}_{\mathrm{DFS}}(P)$, возникающей в идеализированной модели кэша. Этот анализ позволяет строго обосновать высокую эффективность Gorder и одновременно объяснить, почему ряд других эвристик не даёт гарантированного ускорения.

7.1. Формальная постановка задачи и метрика качества

Пусть задан неориентированный граф $G = (V, E)$, представляющий индекс поиска ближайших соседей, где $|V| = N$. Переупорядочивание вершин задаётся биекцией

$$P : V \to \{1, 2, \ldots, N\},$$

которая определяет порядок размещения вершин в памяти: вершина v размещается по адресу $P(v)$.

Современные процессоры загружают данные из основной памяти в кэш блоками фиксированного размера, называемыми *кэш-линиями*. Пусть размер кэш-линии составляет B вершин.

Определение 7 (Номер кэш-линии)**.** Для вершины $v \in V$ определим номер кэш-линии, в которую она попадает при перестановке P:

$$L_P(v) = \left\lfloor \frac{P(v) - 1}{B} \right\rfloor.$$

При обходе графа (например, при поиске в ширину или beam search) процессор загружает в кэш не только запрошенную вершину, но и всю кэш-линию, содержащую эту вершину. Если следующая посещаемая вершина находится в той же кэш-линии, происходит *попадание в кэш* (cache hit), что значительно быстрее *промаха* (cache miss).

Определение 8 (Индикатор кэш-попадания)**.** Для ребра $e = (u, v) \in E$ определим индикатор того, что обе вершины попадают в одну кэш-

линию:

$$m_P(u,v) = \mathbf{1}\{L_P(u) = L_P(v)\} = \begin{cases} 1, & \text{если } \left\lfloor \frac{P(u)-1}{B} \right\rfloor = \left\lfloor \frac{P(v)-1}{B} \right\rfloor, \\ 0, & \text{иначе.} \end{cases}$$

Определение 9 (Функция эффективности кэша)**.** Эффективность перестановки P характеризуется числом рёбер, концы которых попадают в одну кэш-линию:

$$M(P) = \sum_{(u,v)\in E} m_P(u,v). \tag{6}$$

Определение 10 (Задача MCLE)**.** Задача Maximum Cache-Line Edges (MCLE): для заданного графа $G = (V, E)$ и размера кэш-линии B найти перестановку P^*, максимизирующую $M(P)$:

$$P^* = \arg\max_P M(P).$$

Задача MCLE естественным образом возникает при оптимизации графовых алгоритмов: максимизация $M(P)$ эквивалентна минимизации числа промахов кэша при обходе графа, так как каждое ребро, попавшее в одну кэш-линию, потенциально избегает одного промаха.

7.2. NP-полнота задачи MCLE

Докажем, что задача MCLE является вычислительно трудной уже при небольших значениях B.

Теорема 1. *Задача разрешения для MCLE является NP-полной при* $B = 3$.

Доказательство. Рассмотрим соответствующую задачу разрешения:

Cache-Reordering
Вход: граф $G = (V, E)$, размер кэш-линии B, целое число K.
Вопрос: существует ли перестановка P такая, что $M(P) \geq K$?

Принадлежность классу NP. Если перестановка P дана в качестве сертификата, значение $M(P)$ вычисляется за полиномиальное время $O(|E|)$ путём однократного просмотра всех рёбер графа и проверки условия $L_P(u) = L_P(v)$ для каждого ребра. Следовательно, задача принадлежит классу NP.

NP-трудность. Выполним полиномиальное сведение от классической NP-полной задачи Triangle Partition [18].

Triangle Partition:

Вход: граф $H = (V_H, E_H)$, где $|V_H| = 3q$ для некоторого $q \in \mathbb{N}$.
Вопрос: можно ли разбить V_H на q попарно непересекающихся множеств $T_1, \ldots, T_q$, каждое из которых индуцирует полный подграф K_3 (треугольник)?

Построение экземпляра Cache-Reordering. По данному экземпляру H задачи Triangle Partition построим экземпляр задачи Cache-Reordering:

- $G := H$ (тот же граф);
- $B := 3$ (размер кэш-линии);
- $K := 3q$ (целевое значение).

Заметим, что построение выполняется за время $O(1)$. Поскольку $|V| = 3q$ и $B = 3$, при любой перестановке P вершины разбиваются ровно на q кэш-линий по 3 вершины в каждой.

Прямое направление. Предположим, что H допускает разбиение на q непересекающихся треугольников $T_1, \ldots, T_q$. Построим перестановку P, размещая вершины каждого треугольника T_i подряд в одной кэш-линии. Поскольку каждый треугольник содержит ровно 3 ребра, получаем:

$$M(P) = \sum_{i=1}^{q} |E(T_i)| = 3q = K.$$

Следовательно, ответ на экземпляр Cache-Reordering положителен.

Обратное направление. Предположим, что существует перестановка P, для которой $M(P) \geq 3q$. При $B = 3$ множество вершин разбивается ровно на q кэш-линий по 3 вершины. Внутри одной кэш-линии на трёх вершинах может находиться не более 3 рёбер (максимум достигается на полном графе K_3). Следовательно,

$$M(P) \leq \sum_{\text{линий } \ell} |E(G[\ell])| \leq 3q.$$

Из условия $M(P) \geq 3q$ вытекает $M(P) = 3q$, а значит, каждая кэш-линия индуцирует ровно 3 ребра, то есть полный граф K_3. Таким образом, вершины H разбиваются на q непересекающихся треугольников.

Тем самым, Triangle Partition $\leq_p$ Cache-Reordering, и задача Cache-Reordering NP-трудна. Вместе с принадлежностью классу NP получаем NP-полноту. □

Следствие 1. *Задача оптимизации MCLE является NP-трудной.*

Замечание 1. Теорема 1 показывает, что точное решение задачи MCLE нецелесообразно уже при $B = 3$. Поскольку реальный размер кэш-линии обычно больше, следует применять эвристические алгоритмы, не гарантирующие оптимальности, но хорошо работающие на практике.

8. Численные эксперименты

Подробное исследование влияния различных техник перестановки вершин графа на качество работы HNSW содержится в работе [7]. В этом разделе мы сосредоточимся на индексе HCNNG.

Будем использовать датасет **SIFT1M**, который является популярным в задачах поиска ближайших соседей. Данный датасет содержит $1,000,000$ базовых векторов (base) и $10,000$ векторов поиска (query). Все векторы имеют размерность 128.

Основными метриками качества работы алгоритма поиска будут выступать:

- Среднее время получения ответа на запрос. Считаем среднее время получения ответа на запрос до и после переупорядочивания вершин графа.
- Время переупорядочивания. Время, затрачиваемое непосредственно на процесс генерации новой перестановки вершин графа
- Recall@k для запросов.

Эксперимент будет выстроен следующим образом. На первом этапе будет проведена серия контрольных тестов количеством равным 10, в которых HCNNG-индекс используется в своей стандартной конфигурации, без применения Rabbit Order и Gorder. Целью этой стадии станет установление базового уровня производительности — времени, затрачиваемого на выполнение запросов приближенного поиска в отсутствие каких-либо дополнительных оптимизаций, связанных с улучшением локальности данных. Это позволит получить репрезентативные данные о базовой эффективности алгоритма поиска. Далее, индекс будет переупорядочен с помощью Rabbit Order и Gorder, и заново посчитаны метрики.

Характеристики системы, на которой будут проводиться тесты:

- Центральный процессор Intel(R) Core (TM) i3-8130U CPU @ 2.20GHz 2.21 GHz
- Оперативная память: 4,00 Гб
- Операционная система: x64 Windows, WSL2 Debian
- L1 cache: 128 КБ
- L2 cache: 512 КБ
- L3 cache: 4,0 МБ
- 2 ядра, 4 логических процессора

9. Результаты экспериментов

Полные численные результаты экспериментов (время выполнения, ускорение, метрики качества) приведены в **Приложении A**.

Как видно из таблиц Приложения A, применение алгоритмов упорядочивания вершин графа позволяет достичь ускорения поиска на 8–9 % при незначительном снижении качества (Recall@5 остаётся выше 99.7 %).

На рисунке 2 показано сравнение времени первоначального построения индекса HCNNG и времени, затрачиваемого на переупорядочивание вершин с помощью Rabbit Order и Gorder. Как видно, алгоритмы переупорядочивания вершин требуют на порядок меньше времени по сравнению с полным построением индекса, что делает их применение практичным даже при необходимости периодического обновления графа. Так как в системах машинного обучения требуется часто обращаться к решению задачи поиска ближайших соседей, то предобработка индекса быстро окупит себя.

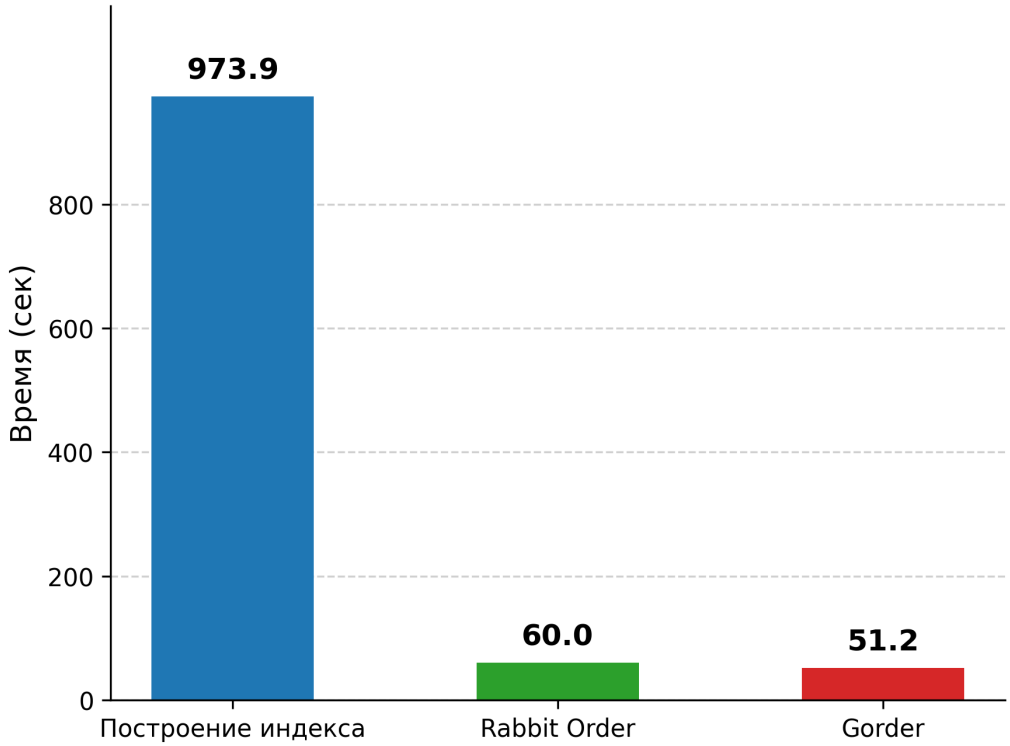

Рис. 2: Сравнение времени построения индекса HCNNG и переупорядочивания вершин

10. Выводы

В данной работе были рассмотрены современные подходы к решению задачи поиска ближайших соседей на основе построения графовых индексов. Основное внимание уделено методу переупорядочивания вершин графа как способу улучшения пространственно-временной локальности доступа к памяти и, как следствие, повышения производительности поисковых запросов.

В теоретической части работы была формально поставлена задача Maximum Cache-Line Edges (MCLE), в которой требуется найти такую перестановку вершин графа $P : V \to \{1, \ldots, N\}$, чтобы максимизировать число рёбер, оба конца которых попадают в одну кэш-линию размера B. Доказано, что соответствующая задача разрешения является NP-полной уже при $B = 3$, путём сведения к классической NP-полной проблеме Triangle Partition. Этот результат обосновывает необходимость разработки эвристических алгоритмов переупорядочивания, способных находить близкие к оптимальным решения за приемлемое время.

На основе идеализированной модели кэша показана связь введённой метрики $M(P)$ со сложностью кэша C_{DFS} при обходе графа: максимизация $M(P)$ ведёт к сокращению нижней границы числа кэш-промахов. Проведённый анализ позволил теоретически объяснить успех существующих эвристик, в частности алгоритма Gorder, чья оконная целевая функция выступает прокси для указанной метрики, а также выявить ограничения community-ориентированных подходов (Rabbit Order), не имеющих, на данный момент, формальных гарантий улучшения кэш-эффективности.

Список литературы

[1] Malkov Yu A и Yashunin Dmitriy A. “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs”. В: *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018).

[2] Andoni A. и Razenshteyn I. “Optimal data-dependent hashing for approximate near neighbors”. В: *Proceedings of 47th annual ACM symposium on Theory of computing*. 2015.

[3] Shrivastava A., Wang Y. и Ryu J. “Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search”. В: *Proceedings of the 2018 International Conference on Management of Data*. 2018.

[4] Junya Arai и др. “Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis”. В: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), с. 22—31. URL: https://api.semanticscholar.org/CorpusID:1164828.

[5] Martin Aumüller, Erik Bernhardsson и Alexander Faithfull. “ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms”. В: *Inf. Syst.* 87.C (янв. 2020). ISSN: 0306-4379. DOI: 10.1016/j.is.2019.02.006. URL: https://doi.org/10.1016/j.is.2019.02.006.

[6] A. Beygelzimer, S. Kakade и J. Langford. “Cover trees for nearest neighbors”. В: *Proceedings of the 23rd international conference on Machine Learning*. 2006.

[7] Benjamin Coleman и др. “Graph Reordering for Cache-Efficient Near Neighbor Search”. В: *Advances in Neural Information Processing Systems*. Под ред. S. Koyejo и др. Т. 35. Curran Associates, Inc., 2022, с. 38488—38500. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/fb44a668c2d4bc984e9d6ca261262cbb-Paper-Conference.pdf.

[8] Matteo Frigo и др. “Cache-Oblivious Algorithms”. В: *ACM Transactions on Algorithms* 8 (янв. 2012), с. 4. DOI: 10.1145/2071379.2071383.

[9] Cong Fu и др. “Fast approximate nearest neighbor search with the navigating spreading-out graph”. В: *Proc. VLDB Endow.* 12.5 (янв. 2019), с. 461—474. ISSN: 2150-8097. DOI: 10.14778/3303753.3303754. URL: https://doi.org/10.14778/3303753.3303754.

[10] Piotr Indyk и Rajeev Motwani. “Approximate nearest neighbors: towards removing the curse of dimensionality”. В: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC ’98. Dallas, Texas, USA: Association for Computing Machinery, 1998, с. 604—613. ISBN: 0897919629. DOI: 10.1145/276698.276876. URL: https://doi.org/10.1145/276698.276876.

[11] Hervé Jégou, Matthijs Douze и Cordelia Schmid. “Product Quantization for Nearest Neighbor Search”. В: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (янв. 2011), с. 117—128. DOI: 10.1109/TPAMI.2010.57. URL: https://inria.hal.science/inria-00514462.

[12] Jaekyu Lee, Hyesoon Kim и Richard Vuduc. “When Prefetching Works, When It Doesn’t, and Why”. В: *ACM Trans. Archit. Code Optim.* 9.1 (март 2012). ISSN: 1544-3566. DOI: 10.1145/2133382.2133384. URL: https://doi.org/10.1145/2133382.2133384.

[13] D. Levinthal. *Performance analysis guide for intelr core (tm) i7 processor and intel xeon (tm) 5500 processors*. Тех. отч. Intel Corporation, 2009.

[14] Wen Li и др. “Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement (v1.0)”. В: *IEEE Transactions on Knowledge and Data Engineering* PP (окт. 2016). DOI: 10.1109/TKDE.2019.2909204.

[15] Izbicki M. и Shelton C. “Faster cover trees”. В: *International Conference on Machine Learning*. 2015.

[16] Zaheer M. и др. “Canopy fast sampling with cover trees”. В: *International Conference on Machine Learning*. 2017.

[17] Yury Malkov и др. “Approximate nearest neighbor algorithm based on navigable small world graphs”. В: *Information Systems* 45 (2014).

[18] David S. Johnson Michael R. Garey. *Computers and Intractability*. W. H. Freeman & Co., 1990.

[19] Lv Q., Wang Z. и Li K. Charikar M. “Multi-probe lsh: efficient indexing for high-dimensional similarity search”. В: *33rd International Conference on Very Large Data Bases*. 2007.

[20] Hiroaki Shiokawa, Yasuhiro Fujiwara и Makoto Onizuka. “Fast algorithm for modularity-based graph clustering”. В: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI’13. Bellevue, Washington: AAAI Press, 2013, с. 1170—1176.

[21] Javier Vargas Muñoz и др. “Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search”. В: *Pattern Recognition* 96 (2019), с. 106970. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2019.106970. URL: https://www.sciencedirect.com/science/article/pii/S0031320319302730.

[22] Mengzhao Wang и др. “A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search”. В: *Proc. VLDB Endow.* 14.11 (июль 2021), с. 1964—1978. ISSN: 2150-8097. DOI: 10.14778/3476249.3476255. URL: https://doi.org/10.14778/3476249.3476255.

[23] Hao Wei и др. “Speedup Graph Processing by Graph Ordering”. В: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, с. 1813—1828. ISBN: 9781450335317. DOI: 10.1145/2882903.2915220. URL: https://doi.org/10.1145/2882903.2915220.

[24] Dong Y., Razenshteyn I. и Wagner T. “Learning space partitions for nearest neighbors search”. В: *International Conference on Learning Representations*. 2019.

Приложение A. Результаты численных экспериментов

В данном приложении приведены полные результаты бенчмаркинга индекса HCNNG на датасете SIFT1M. Поиск осуществлялся для $k = 5$ ближайших соседей, каждое измерение повторено 10 раз.

Таблица 2: Общее время обработки 10 запросов (сек)

Базовый поиск	Rabbit Order	Gorder
1090.98	1016.54	1005.79
1089.07	1007.51	997.94
1090.80	1008.09	996.27
1090.83	1011.75	999.63
1096.44	1009.34	997.33
1127.41	1009.80	1002.35
1090.01	1003.41	996.38
1102.71	1005.17	997.02
1090.75	1005.69	997.75
1091.46	1009.82	997.79
1096.05	**1008.71**	**998.82**

Таблица 3: Время обработки одного запроса (сек)

Базовый поиск	Rabbit Order	Gorder
1.09098	1.01654	1.00579
1.08907	1.00751	0.99794
1.09080	1.00809	0.99627
1.09083	1.01175	0.99963
1.09644	1.00934	0.99732
1.12741	1.00980	1.00235
1.09001	1.00341	0.99638
1.10271	1.00517	0.99702
1.09075	1.00569	0.99775
1.09146	1.00982	0.99779
1.0960	**1.0087**	**0.9988**

Таблица 4: Ускорение и затраты на переупорядочивание

	Rabbit Order	**Gorder**
Ускорение среднего времени запроса	8.0 %	8.9 %
Время переупорядочивания вершин (сек)	20.512	36.391

Таблица 5: Средний Recall@5 по 10 запускам

Метод	Recall@5
Оригинальный индекс	99.82
После Rabbit Order	99.80
После Gorder	99.74