

Оптимизация графового поиска к ближайших соседей посредством эффективизаций доступа к памяти

Ключевые слова: задача kNN, графовый алгоритм, эффективный доступ к памяти, комбинаторный алгоритм, локальная оптимизация

Аннотация: Задача поиска k ближайших соседей в последнее время привлекает повышенное внимание. Наиболее популярным для этой задачи является графовый алгоритм HNSW, который интегрируется в многие векторные базы данных. Оптимизация алгоритма HNSW является критической для бизнеса, так как напрямую влияет на скорость, стоимость и качество работы современных AI-решений. В данной работе производится анализ реализации HNSW из библиотеки `hnswlib`. На основе этого анализа выделяется её узкое место - неэффективный доступ к памяти - и предлагается методология оптимизации доступа к памяти, основанная на перенумерации вершин HNSW графа. Представляются три алгоритма перенумерации. Для апробации методологии реализована собственная версия HNSW, которая тестируется на наборах данных SIFT1M, GloVe100. В некоторых случаях демонстрируется улучшение метрики `qps` на 30% по сравнению с `hnswlib`. Отметим, что для редактирования текста работы использовались генеративные модели.

ACM CSS: Information systems \sim Information retrieval \sim Search engine architectures and scalability

Введение

Задача поиска k ближайших соседей (kNN) является фундаментальной проблемой в теоретической информатике и машинном обучении. Алгоритмы для решения данной задачи делятся на точные и приближенные. В работах [1–3] приводятся некоторые точные алгоритмы, использующие в себе k -d деревья. Однако, эти алгоритмы оказываются неэффективными по времени в пространствах большой размерности. Эффективные по времени алгоритмы известны для задачи приближенного поиска k ближайших соседей. Среди приближенных алгоритмов выделяют методы, основанные на идеях random projection trees [4], locality-sensitive hashing [5–7], product quantization [8; 9], графовых методов [10–12].

В последнее время задача поиска k ближайших соседей привлекает повышенное внимание. Этот всплеск интереса во многом обусловлен ростом популярности метода Retrieval Augmented Generation (RAG) в больших языковых моделях, ключевой этап которого заключается в поиске релевантных документов в векторных базах данных. В ответ на этот спрос ведущие разработчики СУБД, включая ClickHouse(R), PostgreSQL, Oracle, MongoDB Atlas, Redis Stack(TM) и Lucene(TM), за последние годы интегрировали в свои продукты функцию векторного поиска. При этом в качестве основного алгоритма большинство из этих реализаций используют графовый Hierarchical Navigable Small World (HNSW) [11]. В связи с этим, оптимизация алгоритма HNSW является критической для бизнеса, так как напрямую влияет на скорость, стоимость и качество работы современных AI-решений.

Задачей данной работы является создание методологии для повышения эффективности доступа к памяти в алгоритме HNSW. В качестве базового решения для сравнения при апробации методологии была выбрана реализация HNSW из библиотеки hnswlib [13].

1 Анализ реализации HNSW из библиотеки hnswlib

В алгоритме HNSW строится поисковый индекс набора данных, который описывается графом. Каждая вершина графа соответствует d -мерной точке в метрическом пространстве. В типичной реализации HNSW точки хранятся в массиве `float* data` размера $n * d$, где n – количество точек. В этом массиве ячейки `data[i * d]`, ... , `data[(i+1) * d - 1]` используются для хранения вектора i -й точки. Типичной операцией для HNSW является подсчет расстояний от вершины до её соседей. Описанный формат хранения точек приводит к произвольному доступу к памяти (random access), так как векторы соседей находятся в произвольных местах массива, что негативно сказывается на производительности.

Ниже представлены результаты профилирования реализации алгоритма HNSW из библиотеки hnswlib. Профилирование выполнялось с использованием Intel(R) VTune(TM) Profiler на вычислительной системе с процессором Intel(R) Core(TM) Ultra 7 155H и 16 ГБ ОЗУ. Алгоритм тестировался в режиме поиска $k = 10$ ближайших соседей на тестовых

вой части набора данных SIFT1M. При каждом значении параметра $ef = 200, 201, \dots, 219$ осуществлялся поиск ближайших соседей для всех запросов из тестового набора данных.

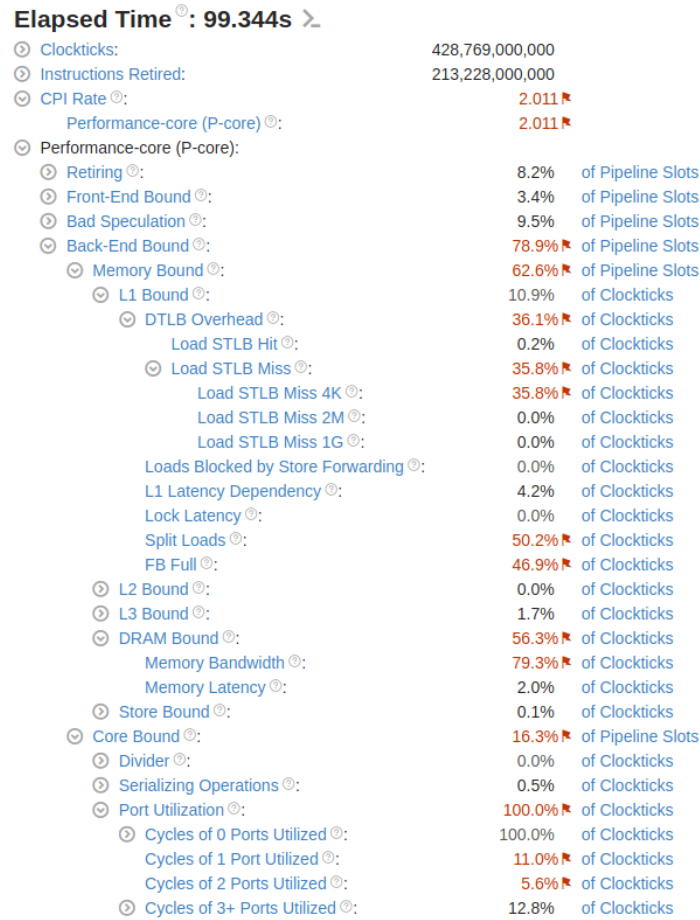





Рис. 1: Top-down дерево микроархитектурного анализа для реализации HNSW из библиотеки hnsplib.

На (рис. 1) представлено дерево микроархитектурного анализа, выполненное по методу top-down [14]. Позиции, отмеченные красными флажками, указывают на узкие места в реализации. Анализ показывает, что производительность реализации ограничена пропускной способностью памяти (Memory Bound).

Часть выявленных проблем может быть устранена с помощью низкоуровневых оптимизаций. Например, снизить количество промахов буфера трансляции (Load STLB Miss 4K) возможно за счет использования страниц памяти размером более 4 КБ. Количество операций чтения по невыровненным адресам (Split Loads) можно уменьшить путем выравнивания структур данных в памяти. В то же время, некоторые ограничения, такие как 'DRAM Bound' и 'Memory Bandwidth', являются фундаментальными и свидетельствуют о достижении предела аппаратных возможностей, что не может быть исправлено низкоуровневыми оптимизациями.

Top Hotspots 

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time 	% of CPU Time 
_Z13_mm256_sub_psDv8_fs_	example_search	64.065s	65.8%
hnsplib::L2SqrSIMD16ExtAVX	example_search	8.109s	8.3%
hnsplib::HierarchicalNSW<float>::searchBaseLayerST<(bool)1, (bool)0>	example_search	8.009s	8.2%
_Z13_mm256_mul_psDv8_fs_	example_search	4.840s	5.0%
_Z13_mm256_add_psDv8_fs_	example_search	2.550s	2.6%
[Others]	N/A*	9.820s	10.1%

*N/A is applied to non-summable metrics.

Рис. 2: Hotspot анализ для реализации HNSW из библиотеки hnsplib.

Hotspot-анализ реализации, представленный на (рис. 2), демонстрирует, что на SIMD-инструкцию `_Z13_mm256_sub_psDv8_fs_`, используемую для вычисления расстояний и связанную с первичной загрузкой векторов точек из памяти, приходится 65,8% времени выполнения. Таким образом, ключевым направлением для повышения производительности алгоритма HNSW является оптимизация доступа к векторам точек.

2 Оптимизация доступа к памяти

Предлагаемый в данной работе метод оптимизации доступа к памяти основан на переупорядочивании данных. Предлагается перенумеровать точки в соответствии с структурой HNSW графа, а затем перезаписать все данные, в том числе массив точек, в соответствии с новой нумерацией. Более формально, пусть $G = (V, E)$ – граф нулевого уровня HNSW с множеством вершин $V = \{1, 2, \dots, N\}$. Необходимо выбрать перестановку $\pi : V \rightarrow \{1, 2, \dots, N\}$, а затем для всех вершин $v \in V$ задать новые номера по правилу $v \mapsto \pi(v)$. При этом предполагается, что π такая перестановка, перенумерация графа с которой приводит к повышению эффективности работы алгоритма при взаимодействии с памятью. Далее рассматриваются критерии выбора подобных перестановок.

2.1 Перенумерация с помощью укладки остовного дерева

Найти перестановку для перенумерации можно как результат решения следующей оптимизационной задачи:

$$\sum_{i=1}^N \max_{j: (i,j) \in E} |\pi(i) - \pi(j)| \rightarrow \min_{\pi} \quad (2.1.1)$$

Функция из этой оптимизационной задачи в некоем смысле формализует принцип пространственной локальности данных. Для произвольных графов эта задача NP-трудна, и

остается NP-трудной даже для деревьев специального вида. Однако для корневых деревьев известен линейный алгоритм укладки минимальной длины [15], который решает задачу (2.1.1) с дополнительным ограничением:

$$\sum_{i=1}^N \max_{j:(i,j) \in E} |\pi(i) - \pi(j)| \rightarrow \min_{\pi}$$

$$s.t. \pi(i) > \pi(j) \forall (i, j) \in E$$

Используя этот алгоритм, можно перенумеровать остовное дерево графа и использовать найденную нумерацию для исходного графа. Нами предлагаются алгоритмы 1 и 2 для перенумерации BFS и MST (Minimum Spanning Tree) деревьев графа нулевого уровня HNSW.

Algorithm 1 Перенумерация с помощью укладки BFS дерева

Require: Граф $G = (V, E)$ и массив точек, которые соответствуют вершинам.

Ensure: перестановка π

- 1: Найти вершину i , точка которой наиболее близка к средней точке набора данных.
 - 2: Построить из вершины i BFS дерево графа G .
 - 3: Найти перестановку π как нумерацию минимальной длины BFS-дерева.
-

Algorithm 2 Перенумерация с помощью укладки MST дерева

Require: Граф $G = (V, E)$ и массив точек, которые соответствуют вершинам.

Ensure: перестановка π

- 1: Построить граф $G' = (V, E')$, в котором $(i, j) \in E'$, если $(i, j) \in E$ или $(j, i) \in E$
 - 2: Найти вершину i , точка которой наиболее близка к средней точке набора данных.
 - 3: Построить из вершины i MST дерево графа G' , используя в качестве весов ребер расстояния между соответствующими точками.
 - 4: Найти перестановку π как нумерацию минимальной длины MST дерева.
-

2.2 Перенумерация с помощью локального поиска

Решать задачи, похожие на (2.1.1), можно также с помощью методов дискретной оптимизации. Например, с помощью локального поиска. Однако, (2.1.1) содержит в себе функцию максимума, которая несколько затрудняет оптимизацию. В данной работе для нахождения перестановки предлагается локальным поиском решать задачу:

$$\varphi(\pi) = \sum_{(i,j) \in E} |\pi(i) - \pi(j)| \rightarrow \min_{\pi}$$

Ниже приведен алгоритм 3 для одной итерации локального поиска (под итерацией понимается один проход по графу).

Algorithm 3 Итерация локального поиска

Require: Граф $G = (V, E)$, N – количество вершин, π и π^{-1} – массивы перестановок, $\varphi(\pi)$ – целевая функция, L – гиперпараметр алгоритма

Ensure: обновленные массивы перестановок π и π^{-1}

```

1: for  $i = 1, 2, \dots, N$  do
2:   for  $j = \pi^{-1}(\pi(i) + 1), \pi^{-1}(\pi(i) + 2), \dots, \pi^{-1}(\min(\pi(i) + L, N))$  do
3:     Попытаться выбрать такую вершину  $k \in Neighbors(i)$ , для которой
        $swap(\pi(k), \pi(j))$  дает наибольшее уменьшение целевой функции  $\varphi(\pi)$ .
4:     В случае, если вершина выбрана, обновить массивы перестановок, выполнив опе-
       рации  $swap(\pi(k), \pi(j))$  и  $swap(\pi^{-1}(\pi(k)), \pi^{-1}(\pi(j)))$ .
5:   end for
6: end for

```

Заметим, что в алгоритме 3 для подсчета изменения целевой функции $\varphi(\pi)$ достаточно вычислить сумму

$$\sum_{l:(k,l) \in E} |\pi(k) - \pi(l)| + \sum_{l:(l,k) \in E} |\pi(l) - \pi(k)| + \sum_{l:(j,l) \in E} |\pi(j) - \pi(l)| + \sum_{l:(l,j) \in E} |\pi(l) - \pi(j)|$$

до $swap$ и после $swap$, а затем найти разность вычисленных сумм.

3 Описание программной реализации

Для данной работы на языке программирования C++ был реализован алгоритм HNSW с описанными улучшениями. Программная реализация доступна в GitHub репозитории [16]. Компиляция кода производится компилятором g++ 13.3.0 с опциями: -Ofast -lrt -DHAVE_CXX0X -march=native -fpic -w -fopenmp -ftree-vectorize -ftree-vectorizer-verbose=0.

Приведем основные детали реализации структуры HNSWInference из заголовочного файла hnsw_inference.h.

- Для хранения окрестностей на нулевом уровне HNSW используется динамический массив `int* list0` размера $(\text{maxM0} + 1) * n$, где maxM0 – максимальный допустимый размер окрестности на нулевом уровне, n – число вершин. В этом массиве `list0[(maxM0 + 1) * i]` соответствует количеству вершин в окрестности вершины i , а ячейки `list0[(maxM0 + 1) * i + 1], \dots, list0[(maxM0 + 1) * i + maxM0]` предназначены для хранения окрестности вершины i . Данный массив выделяется с помощью оператора `aligned_alloc` с параметром выравнивания по границе 64 байт.
- Для хранения окрестностей на ненулевых уровнях HNSW используется динамический массив указателей `int** list` размера n . В нем `list[i]` это указатель на динамический массив размера $(\text{maxM} + 1) * (\text{max_level}[i] - 1)$, где maxM – максимальное допустимое

количество вершин в окрестности на ненулевом уровне, $\text{max_level}[i]$ – максимальный уровень вершины i . Элемент массива $\text{list}[i][(j-1) * (\text{maxM} + 1)]$ соответствует размеру окрестности вершины i на уровне j , а в ячейках $\text{list}[i][(j-1) * (\text{maxM} + 1) + 1], \dots, \text{list}[i][(j-1) * (\text{maxM} + 1) + \text{maxM}]$ хранится сама окрестность.

- Для хранения векторов точек используется динамический массив $\text{float}^* \text{data}$ размера $n * d$, где d – размерность векторного пространства. В этом массиве ячейки $\text{data}[i * d], \dots, \text{data}[(i+1) * d - 1]$ используются для хранения вектора i -й точки. Данный массив выделяется оператором `mmap` на страницах памяти размером в 2 мегабайта. Также массив выравнивается по границе 64 байт.
- В функции `HNSWInference::SearchLayer` на этапе сканирования окрестности осуществляется префетчинг вектора точки за 1 итерацию до того как она понадобится. Для этого используется инструкция `_mm_prefetch` с параметром `_MM_HINT_T0`.
- Для подсчета расстояний используются функции `SpaceL2::Distance` и `SpaceCosine::Distance` из файла `primitives.h`. На процессоре Intel(R) Core(TM) Ultra 7 155H они автоматически векторизуются при компиляции с использованием SIMD инструкций из набора AVX. Пример векторизации приведен на рисунке 3.

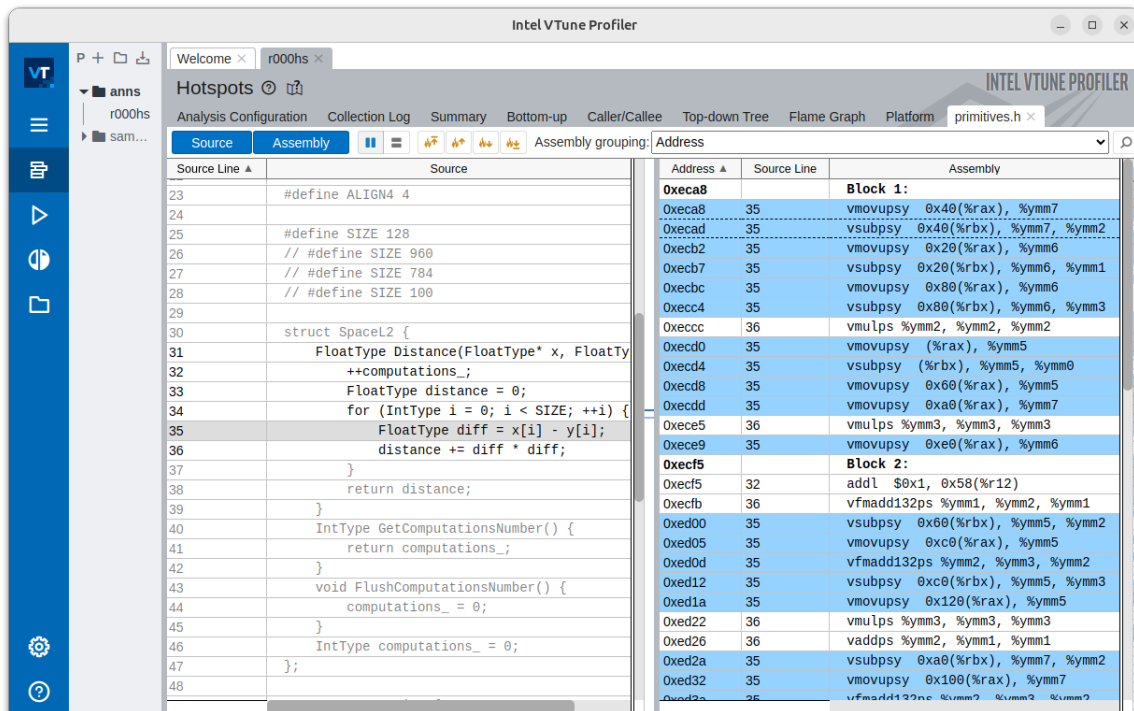


Рис. 3: Ассемблерный код для функции `SpaceL2::Distance` с SIMD инструкциями.

4 Вычислительные эксперименты

4.1 Характеристики вычислительной системы

Эксперименты производились на вычислительной системе с процессором Intel(R) Core(TM) Ultra 7 155H, 16 ГБ оперативной памяти под управлением операционной системы Ubuntu 24.04.2 LTS. На время эксперимента исполняемый процесс закреплялся за производительным ядром (P-core) процессора с помощью команды `taskset -c 3 <executable>`.

4.2 Испытуемые реализации HNSW

В экспериментах сравнивались следующие реализации HNSW:

- 'hnswnlib' – реализация из hnswnlib [13]
- 'our version (basic)' – наша реализация без использования перенумераций
- 'our version + reordering (bfs)' – наша реализация с перенумерацией с помощью укладки BFS дерева (алгоритм 1)
- 'our version + reordering (mst)' – наша реализация с перенумерацией с помощью укладки MST дерева (алгоритм 2)
- 'our version + reordering (local search)' – наша реализация с перенумерацией с помощью локального поиска (алгоритм 3). В алгоритме локального поиска используется гиперпараметр $L = 30$, а также проводятся 100 итераций локального поиска.
- 'our version + reordering (local search on kNN graph)' – наша реализация с перенумерацией с помощью локального поиска (алгоритм 3). Однако, в алгоритме вместо графа нулевого уровня HNSW использовался kNN граф (при $k = 50$). В алгоритме локального поиска использовался гиперпараметр $L = 30$, а также проводились 100 итераций локального поиска.

Во всех реализациях при фиксированном наборе данных использовался одинаковый поисковый индекс.

4.3 Наборы данных

В качестве наборов данных использовались SIFT1M, GloVe100 из репозитория [17].

4.4 Метрики

Для оценки kNN алгоритмов классически используются следующие метрики:

- *qps* (queries per second) – количество запросов в секунду, которые обрабатывает алгоритм

- $recall@k$ – доля из найденных алгоритмом k точек, которые входят в множество k ближайших соседей

В нашей работе используется метрика:

$$speedup = \frac{qps_B}{qps_A}$$

которая показывает во сколько раз больше алгоритм B обрабатывает запросов в секунду чем алгоритм A (полагаем, что $qps_B > qps_A$).

4.5 Результаты

Эксперименты проводились для случая поиска $k = 10$ ближайших соседей.

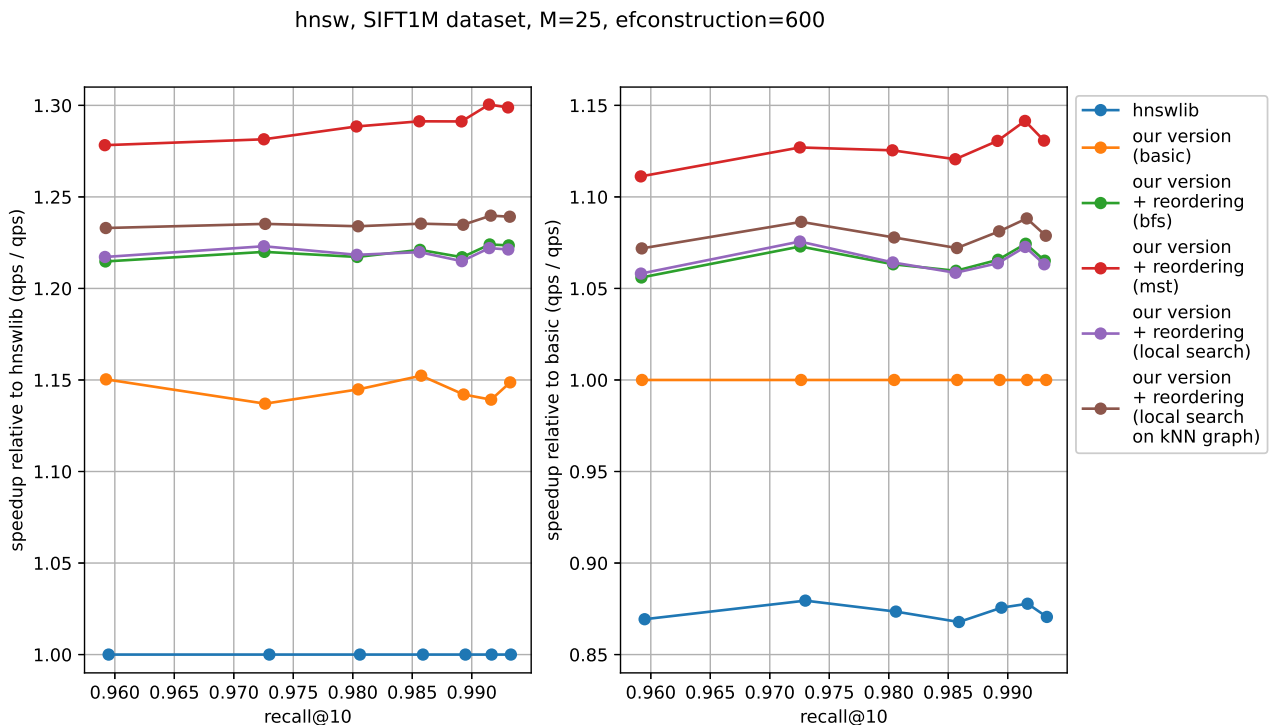


Рис. 4: Результат эксперимента. Набор данных SIFT1M. Задача поиска $k = 10$ ближайших соседей.

Точки на графике строились путем перебора $ef = 40, 50, \dots, 100$.

На (рис. 4) представлены результаты для набора данных SIFT1M. Видно, что одна из наших реализаций показывает прирост qps до 30% по сравнению с реализацией из hnswnlib. А также, показывает прирост qps до 15% по сравнению с нашей базовой реализацией посредством перенумерации HNSW графа.

На (рис. 5) представлены результаты для набора данных GloVe100. Одна из наших реализаций показывает прирост qps до 26% по сравнению с реализацией из hnswnlib. А также, показывает прирост qps до 15% по сравнению с нашей базовой реализацией посредством перенумерации HNSW графа.

hnsw, GloVe100 dataset, M=25, efconstruction=2500

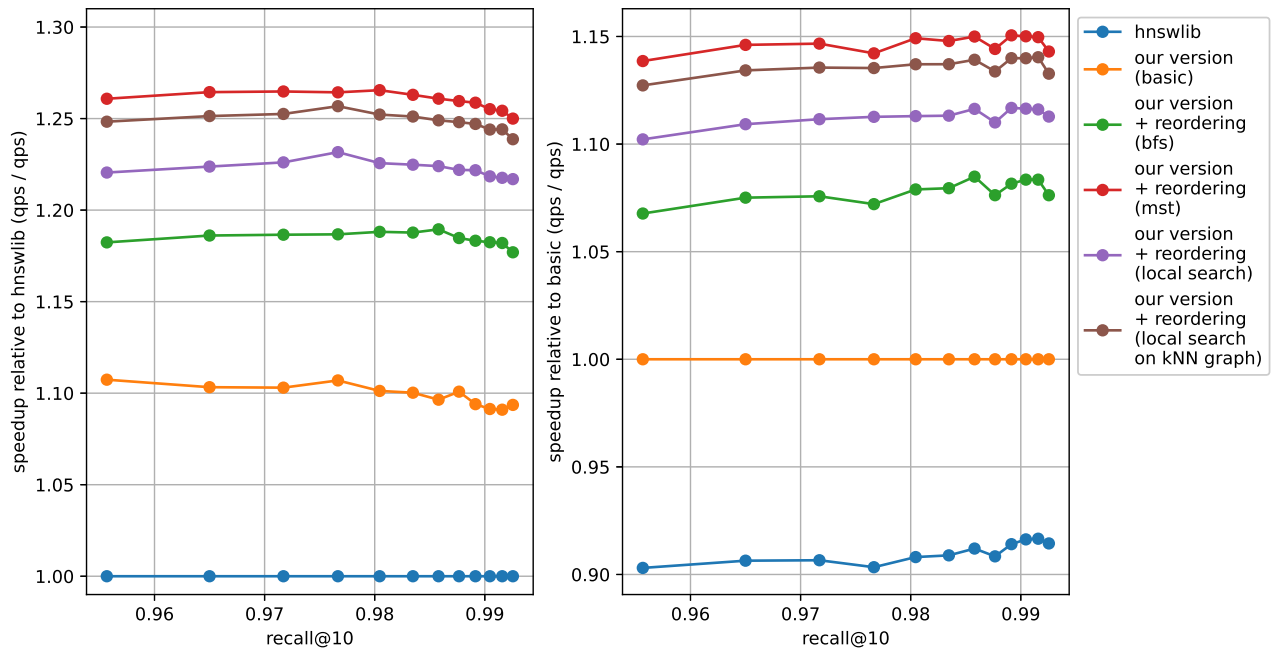


Рис. 5: Результат эксперимента. Набор данных GloVe100. Задача поиска $k = 10$ ближайших соседей.

Точки на графике строились путем перебора $ef = 400, 500, \dots, 1500$.

Список литературы

1. *Bentley J. L.* Multidimensional binary search trees used for associative searching // Communications of the ACM. — 1975. — Т. 18, № 9. — С. 509–517 ; — [Статья в журнале].
2. *Omhundro S. M.* Five balltree construction algorithms. — 1989 ; — [Статья в журнале].
3. *Yianilos P. N.* Data structures and algorithms for nearest neighbor search in general metric spaces // Soda. Т. 93. — 1993. — С. 311–21 ; — [Статья в журнале].
4. *Dasgupta S., Freund Y.* Random projection trees and low dimensional manifolds // Proceedings of the fortieth annual ACM symposium on Theory of computing. — 2008. — С. 537–546 ; — [Статья в журнале].
5. *Indyk P., Motwani R.* Approximate nearest neighbors: towards removing the curse of dimensionality // Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. — Dallas, Texas, USA : Association for Computing Machinery, 1998. — С. 604–613. — (STOC '98) ; — [Статья в журнале].
6. Locality-sensitive hashing scheme based on p-stable distributions / М. Datar [и др.] // Proceedings of the twentieth annual symposium on Computational geometry. — 2004. — С. 253–262 ; — [Статья в журнале].
7. *Charikar M. S.* Similarity estimation techniques from rounding algorithms // Proceedings of the thirty-fourth annual ACM symposium on Theory of computing. — 2002. — С. 380–388 ; — [Статья в журнале].
8. *Jégou H., Douze M., Schmid C.* Product Quantization for Nearest Neighbor Search // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2011. — Т. 33, № 1. — С. 117–128 ; — [Статья в журнале].
9. *Johnson J., Douze M., Jégou H.* Billion-Scale Similarity Search with GPUs // IEEE Transactions on Big Data. — 2021. — Т. 7, № 3. — С. 535–547 ; — [Статья в журнале].
10. Approximate nearest neighbor algorithm based on navigable small world graphs / Y. Malkov [и др.] // Information Systems. — 2014. — Т. 45. — С. 61–68 ; — [Статья в журнале].
11. *Malkov Y. A., Yashunin D. A.* Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2020. — Т. 42, № 4. — С. 824–836 ; — [Статья в журнале].
12. *Fu C., Wang C., Cai D.* High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2022. — Т. 44, № 8. — С. 4139–4150 ; — [Статья в журнале].

13. hnsplib / Ю. А. Мальков [и др.]. — 2024. — URL: <https://github.com/nmslib/hnsplib> ; GitHub [Электронный ресурс] (дата обращения: 06.09.2025).
14. *Yasin A.* A Top-Down method for performance analysis and counters architecture // 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). — 2014. — С. 35—44 ; — [Статья в журнале].
15. *Соболевская Е. П., Котов В. М.* Укладка деревьев // Ежеквартальный научно-методический журнал "Информатизация образования". — 2012. — [Статья в журнале].
16. *nirs-participant-2025.* fluffy-invention. — 2025. — URL: <https://github.com/nirs-participant-2025/fluffy-invention> ; GitHub [Электронный ресурс] (дата обращения: 25.09.2025).
17. ann-benchmarks / Е. Bernhardsson [и др.]. — 2025. — URL: <https://github.com/erikbern/ann-benchmarks> ; GitHub [Электронный ресурс] (дата обращения: 06.09.2025).