Algorithms for Scalable, Accurate, and Efficient Approximate Nearest Neighbor Search

H. Yagiz Devre

Advisor: Professor Robert Tarjan

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Computer Science
Princeton University

April 2025

Abstract

Approximate Nearest Neighbor (ANN) search has become a fundamental component in large-scale machine learning and information retrieval systems. Among various ANN methods, Hierarchical Navigable Small World (HNSW) graphs have become as one of the most efficient and scalable solutions. This thesis presents both a rigorous theoretical analysis and practical enhancements to the HNSW algorithm.

We first revisit the problem of vector retrieval and approximate nearest neighbor search. Then we highlight data structures and algorithms that are particiularly popular in the field of ANNS. Then we detail theoretical underpinnings of HNSW, providing detailed proofs for level distribution, insertion complexity, and query performance. Building upon this foundation, we propose two core optimizations to our HNSW engine written in `C++`: distance-based robust pruning and level-based dimension reduction. Through a pruning strategy inspired by spanner theory, we show that the constructed graph retains a $(1+\varepsilon)$-spanner guarantee, achieving a tunable trade-off between sparsity and search accuracy. In addition, our dimension reduction optimization uses random matrix projections, resulting in faster but slightly less stable greedy searches.

Extensive experiments are conducted on both synthetic and real-world datasets, including GloVe, SIFT, NYTimes, and Fashion-MNIST benchmarks. The results demonstrate that our optimized methods improve returned query quality and reduce query times. This, in return maintains high recall rates with high queries per second(qps).

This work provides a pathway for further improving graph-based ANN methods and highlights how algorithmic design, informed by geometry and graph theory, can lead to tangible performance gains in practical systems.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, the legendary Prof. Robert Tarjan, for their invaluable guidance, continuous support, and insightful feedback throughout the course of this research. Their expertise in the field and encouragement on research have been instrumental in shaping this work.

I would also like to extend my sincere appreciation to my Junior Independet Work coordinator Mikki Hornstein for their support in coordinating the process and providing valuable advice that helped streamline my research efforts.

Additionally, I am grateful to the Princeton University SEAS for providing access to computational resources, which played a crucial role in conducting the experiments and evaluations necessary for this study. The allocation of compute credits significantly contributed to the feasibility and efficiency of this research.

Finally, I would like to thank my family for their encouragement, patience, and support during this entire journey. Their motivation and understanding have been invaluable.

To my grandmother

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's data-driven world, vector search has become an indispensable tool for managing and retrieving large volumes of high-dimensional data. With the advancements in deep learning and machine learning, specifically models that map forms of data such as images and texts to set of vectors called *embeddings*, gave rise to an increasing need for vector search methodologies. The need for efficient vector search algorithms was therefore primarily driven by the need to manage and search high-dimensional data spaces efficiently [12].

1.1 Similarity in Embeddings

Before discussing and formalizing the vector retrival problem, it is essential to understand the basics of embedding similarity. The embeddings encode essential and semantically meaningful features of the raw data in a compact vectorized form. They indirectly enable efficient and fast comparisons between the raw data objects through simple geometric operations applied directly to the embeddings themselves. More specifically, embedding models (whether learned via supervised, self-supervised, or contrastive training objectives) are constructed such that the distance between two embeddings reflects their semantic similarity in the original domain. For example, in a text embedding space, sentences with similar meanings are mapped to vectors that are close under the Euclidean or cosine distance. A simpler version of such a mapping can be seen in Figure 1.1, a *Bag of Words* model.

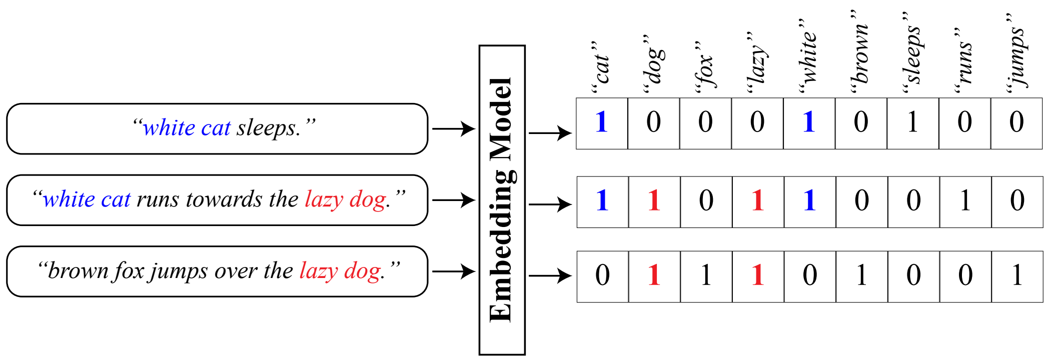

Figure 1.1: Embedding visualization: Similar tokens activate similar dimensions.

More formally, if $f : \mathcal{D} \to \mathbb{R}^d$ is an embedding function mapping a data instance $x \in \mathcal{D}$, such as an image, to a vector $\mathbf{v}_x \in \mathbb{R}^d$, then for two similar inputs $x, x' \in \mathcal{D}$, we expect

$$\|f(x) - f(x')\| \ll \|f(x) - f(y)\| \tag{1.1}$$

for a dissimilar object $y \in \mathcal{D}$.

This geometric structure that comes with the core of the embedding model makes it possible to compare, cluster, and retrieve items such as images and text using purely numerical vector operations [12].

Therefore, to compare data objects in $\mathcal{D}$ the task of identifying the k-nearest vector embeddings to a query in a large high-dimensional set $\mathcal{X} \subset \mathbb{R}^d$ becomes the central computational challenge. As modern applications frequently involve data sets where number of vectors $n \geq 10^6$ and number of dimensions per vector $d \in [128, 1024]$, this vector retrieval problem became a foundational component of many machine learning and recommendation systems. This growing reliance on embedding-based representations, therefore, directly contributes to the demand for scalable and accurate vector search methodologies.

1.2 Vector Retrival Problem

To understand the problem of vector retrieval, we begin by defining the task formally.

Let

$$\mathcal{X} = \{\, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \,\} \subset \mathbb{R}^d \tag{1.2}$$

denote a set of data consisting of n number of vectors in a d-dimensional real vector space. As described previously, in modern applications, typically we have large dimensions $d \in [128, 1024]$ and many vectors $n \geq 10^6$.

The k-nearest neighbor problem, therefore can be summarized as given a query vector in the real space $\mathbf{q} \in \mathbb{R}^d$, the goal of vector search is to find the subset

$$\mathcal{N}_k(\mathbf{q}) = \{\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_k}\} \subset \mathcal{X} \tag{1.3}$$

such that the vectors in $\mathcal{N}_k(\mathbf{q})$ are the k nearest neighbors to $\mathbf{q}$ under a distance function $\mathrm{dist}(\cdot, \cdot)$. This distance function is typically chosen based on the embedding space. For Euclidean spaces, it is often the ℓ_2 (Euclidean) distance, which we can define as

$$\mathrm{dist}_{\ell_2}(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2}, \tag{1.4}$$

On the other hand, while for angular embedding spaces, the cosine distance is commonly used, which we can define as

$$\mathrm{dist}_{\cos}(\mathbf{x}, \mathbf{y}) = 1 - \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2 \cdot \|\mathbf{y}\|_2} = \frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2} \cdot \sqrt{\sum_{i=1}^{d} y_i^2}}., \tag{1.5}$$

where $\langle \cdot, \cdot \rangle$ denotes the standard inner product in $\mathbb{R}^d$. The choice of the geometry of the embedding space is determined by the embedding model itself.

Formally, regardless of the distance function, minimizing the distance corresponds to solving the following optimization problem.

$$\mathcal{N}_k(\mathbf{q}) = \arg\min_{\substack{\mathcal{S} \subset \mathcal{X} \\ |\mathcal{S}| = k}} \sum_{\mathbf{x} \in \mathcal{S}} \mathrm{dist}(\mathbf{q}, \mathbf{x}). \tag{1.6}$$

In the special case of $k = 1$, this reduces to the classical, well known nearest neighbor (NN) problem.

$$\mathbf{x}^\star = \arg\min_{\mathbf{x} \in \mathcal{X}} \mathrm{dist}(\mathbf{q}, \mathbf{x}). \tag{1.7}$$

While the mathematical formulation of the vector retrieval problem is trivial, it becomes increasingly complex in high-dimensional settings with high volume of data. This complexity, also known as *Curse of Dimensionality* arises due to operating in large ambient spaces.

1.3 Curse of Dimensionality

Understanding the *curse of dimensionality* [23] is critical not only for analyzing the limitations of traditional nearest neighbor search methods but also highlighting the need for *approximate* nearest neighbor search algorithms.

As dimensionality d of the set of data $\mathcal{X}$ increases a phenomenon known as *concentration of measure* occurs and the distribution of distances between random points in high-dimensional space becomes sharply peaked [48].

More formally we can describe the *concentration of measure*, as the number of dimensions increase $d \to \infty$, the ratio between the distance to the nearest neighbor and the distance to the farthest neighbor approaches one.

$$\frac{\min_{\mathbf{x}\in\mathcal{X}} \|\mathbf{q}-\mathbf{x}\|}{\max_{\mathbf{x}\in\mathcal{X}} \|\mathbf{q}-\mathbf{x}\|} \to 1. \tag{1.8}$$

With $n = |\mathcal{X}|$ fixed and i.i.d. points drawn from a sub-Gaussian distribution, this can be more formally states as follows.

$$\forall \varepsilon > 0 : \lim_{d\to\infty} \Pr\Big[\frac{\min_{1\le i\le n} \|\mathbf{q}-\mathbf{x}_i\|}{\max_{1\le i\le n} \|\mathbf{q}-\mathbf{x}_i\|} \ge 1-\varepsilon\Big] = 1, \tag{1.9}$$

i.e. the nearest and farthest neighbours become indistinguishable in high dimensions [9, 48]

Therefore, in high-dimensional spaces, all points tend to appear roughly equidistant from any given query.

Consequently, regardless of the distance metric chosen, a vanilla distance-based ranking becomes unstable. Therefore the performance of any exact nearest neighbor algorithm suffers in terms of accuracy and speed [9]. Even under dimensions as low as $d = 15$, Beyer et al. concluded that this unstability is present, making a qquery "meaningless" [9].

As an example to the *Curse of Dimensionality*, tree-based exact nearest neighbor search methods such as k-d trees [8] rely on axis-aligned partitioning of the space. However, in high dimensions, such partitions tend to split very few data points due to sparsity, resulting in large subtrees being visited during queries. Theoretical analysis show that the expected query time of a balanced k-d tree becomes linear in $O(n)$ as dimension d increases [10]. Similarly, cover trees, which provide guarantees under bounded expansion conditions, still suffer in practice when applied to datasets with large dimensionality [10].

To capture the structure of high-dimensional datasets more precisely, the notion of *intrinsic dimensionality* c is introduced [10]. Unlike ambient dimensionality d, which simply counts the number of coordinates, intrinsic dimensionality shows the number of degrees of freedom of the data distribution.

To formalize the notion of *intrinsic dimensionality*, the concepts of *doubling dimension* and *expansion constant* [10, 23] were introduced. The doubling dimension[1] captures how metric volume scales with radius. Likewise the expansion constant[2] controls the growth rate of neighborhoods in a metric space. Even though many high-dimensional datasets has high ambient dimensions, they lie near low-dimensional manifolds due to their low intrinsic dimensionality caused by embedding models.

To summarize, the practical implication of the *curse of dimensionality* is clear. Any algorithm that depends heavily on absolute distance comparisons, space partitioning, or uniform metric properties for exact nearest neighbor search fails in both accuracy and speed as dimensionality grows. Therefore, instead of exact search methods, approximate methods becomes an alternative.

[1]The *doubling dimension* $\dim_{\text{dbl}}$ of a metric space is the smallest number d such that any ball of radius r can be covered by at most 2^d balls of radius $r/2$.

[2]The *expansion constant* c is defined such that for all $p \in \mathcal{X}$ and all radii $r > 0$, the inequality $|B(p, 2r)| \leq c \cdot |B(p, r)|$ holds, where $B(p, r)$ denotes the ball of radius r centered at p.

1.4 Approximate Nearest Neighbor Search

Approximate Nearest Neighbor Search (ANNS) methods attempt to overcome the *curse of dimensionality* by "relaxing" the strict optimality guarantees of exact search [23]. The compromise of exactness in exchange comes with significantly faster query times and lower memory overhead. Rather than requiring that the nearest neighbor retrieved be the absolute closest point to the query in terms of a distance metric, ANNS methods allow for a bounded approximation error. This relaxation on the criteria opens the door to more efficient algorithms that scale better with both dataset size n and ambient dimensionality d [1, 10, 12].

In Section 1.2 have previously showed that given a dataset $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, a distance function $\mathrm{dist} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geq 0}$, and a query point $\mathbf{q} \in \mathbb{R}^d$, the exact nearest neighbor problem with $k = 1$ criteria (that is, only returning the closest neighbor) seeks to find the following optimization.

$$\mathbf{x}^\star = \arg\min_{\mathbf{x} \in \mathcal{X}} \mathrm{dist}(\mathbf{q}, \mathbf{x}). \tag{1.10}$$

In contrast, an algorithm is said to solve the (c, R)-*approximate nearest neighbor* [23] problem for some approximation factor $c > 1$ and search radius $R > 0$ if, whenever there exists a point $\mathbf{x}^\star \in \mathcal{X}$ such that $\mathrm{dist}(\mathbf{q}, \mathbf{x}^\star) \leq R$, the algorithm returns a point $\mathbf{x}_{\mathrm{ann}} \in \mathcal{X}$ such that

$$\mathrm{dist}(\mathbf{q}, \mathbf{x}_{\mathrm{ann}}) \leq c \cdot R. \tag{1.11}$$

Likewise, in the $(1 + \epsilon)$-*approximate nearest neighbor* [1] setting, the algorithm seeks a point $\mathbf{x}_{\mathrm{ann}} \in \mathcal{X}$ such that

$$\mathrm{dist}(\mathbf{q}, \mathbf{x}_{\mathrm{ann}}) \leq (1 + \epsilon) \cdot \mathrm{dist}(\mathbf{q}, \mathbf{x}^\star), \tag{1.12}$$

for some small error parameter denoted as $\epsilon \geq 0$. As shown by the inequality, this approximation guarantees proximity up to a known multiplicative factor and is widely used [1, 23].

While the definition of ANNS is often introduced in the context of finding a single approximate neighbor, many practical applications require retrieving not just the closest point, but the top-k closest vectors to a query. This leads to the *approximate k-nearest neighbors* (k-ANN) problem. The objective is trivial to Section 1.2, that is return a subset of k vectors that are each close to the query point, though not necessarily the exact k closest vectors [36].

Formally, just like the exact *k-NN* approaches described previously, given a query $\mathbf{q} \in \mathbb{R}^d$, the goal is to return a set $\mathcal{N}_k^{\text{ann}}$

$$\mathcal{N}_k^{\text{ann}}(\mathbf{q}) = \{\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_k}\} \subset \mathcal{X} \tag{1.13}$$

such that each $\mathbf{x}_{i_j}$ is an approximate neighbor of $\mathbf{q}$ (instead of an exact neighbor), and for all $\mathbf{x}_{i_j} \in \mathcal{N}_k^{\text{ann}}(\mathbf{q})$, we have the inequality (1.14).

$$\text{dist}(\mathbf{q}, \mathbf{x}_{i_j}) \leq (1+\epsilon) \cdot \text{dist}(\mathbf{q}, \mathbf{x}_j^\star), \tag{1.14}$$

where $\mathbf{x}_j^\star$ is the j-th true nearest neighbor in the exact k-NN set, and $\epsilon \geq 0$ is the approximation parameter as also described in (1.12).

As one could notice, this formulation closely resembles the true nearest neighbors but with a controlled distance relaxation.

Because ANNS algorithms allow this relaxation, they can avoid the linear scan's $\mathcal{O}(nd)$ complexity even in high dimensions. Depending on the structure of the algorithm and assumptions on the distribution of data, sublinear time is often achievable. In the later sections, we will be discussing how Locality-Sensitive Hashing (LSH), for example, provides provable query times of $\mathcal{O}(dn^\rho)$ for some $\rho < 1$, under assumptions of metric structure [1]. Other approaches, such as graph-based methods, rely on navigating a proximity graph to reach good candidate neighbors quickly, often achieving logarithmic performance [41].

To summarize, the central idea behind ANNS is based on relazing the criteria. The assumption is that the small errors in distance often do not significantly affect the outcome. Especially since the embedding space itself is already an abstraction of the input domain, returning an approximate rather than an exact result barely has an effect while reducing the computational cost substantially.

1.5 Evaluation of Approximate Nearest Neighbor Methods

In Section 1.4 we discussed how unlike exact nearest neighbor search, *approximate* methods introduce intentional error to gain improvements in speed and scalability. This raises a natural and critical question:

How do we evaluate the effectiveness of an approximate algorithm when its answers are not guaranteed to be correct?

Since the core idea of ANNS is to trade perfect accuracy for efficiency, any fair comparison must assess both aspects [3, 36].

The primary evaluation criterion for ANNS methods is *retrieval quality*, which measures how closely the returned results approximate the true nearest neighbors. The quality of the retrieved results are quantified via the metric of *recall@k* [3].

Formally, we let $\mathcal{N}_k^{\text{exact}}(\mathbf{q})$ denote the exact top-k nearest neighbors of a query $\mathbf{q}$, and let $\mathcal{N}_k^{\text{ann}}(\mathbf{q})$ be the corresponding approximate result. Then the *recall@k* for the query is defined as in (1.15).

$$\text{Recall@k}(\mathbf{q}) = \frac{|\mathcal{N}_k^{\text{exact}}(\mathbf{q}) \cap \mathcal{N}_k^{\text{ann}}(\mathbf{q})|}{k}. \tag{1.15}$$

This ratio lies in the range $[0, 1]$, where a value of 1 indicates perfect agreement in all k neighbors retrieved between the approximate and exact results. Furthermore, in empirical studies, recall is typically averaged over a set of runs to yield an average performance score to highlight stable results [3, 41].

Other quality metrics include *precision@k* [3], which accounts for the relevance of returned neighbors, and $(1 + \epsilon)$-*recall*, which measures whether the approximate neighbors lie within a specified distance factor ϵ of the true nearest neighbor [1, 3].

Formally, *Precision@k* is defined as in (1.16).

$$\text{Precision@k}(\mathbf{q}) = \frac{|\mathcal{N}_k^{\text{ann}}(\mathbf{q}) \cap \mathcal{R}_k(\mathbf{q})|}{k}, \tag{1.16}$$

where $\mathcal{R}_k(\mathbf{q})$ is the set of ground-truth relevant results for the query $\mathbf{q}$. This metric is especially useful when not all ground truth neighbors are treated equally.

The $(1+\epsilon)$-*recall* [1] metric evaluates whether the retrieved approximate neighbors fall within an acceptable distance of the optimal. Formally, for each query $\mathbf{q} \in \mathbb{R}^d$, we define $r^\star = \min_{\mathbf{x} \in \mathcal{X}} \text{dist}(\mathbf{q}, \mathbf{x})$ as the true nearest neighbor distance. Then the $(1+\epsilon)$-*recall* can be formalized as in (1.17).

$$\text{Recall}_{1+\epsilon}(\mathbf{q}) = \frac{|\{\mathbf{x} \in \mathcal{N}_k^{\text{ann}}(\mathbf{q}) : \text{dist}(\mathbf{q}, \mathbf{x}) \leq (1+\epsilon) \cdot r^\star\}|}{k}, \tag{1.17}$$

which reflects the proportion of returned vectors that lie within the permitted multiplicative distance bound, which we denote as ϵ.

However, accuracy alone is insufficient. A second, equally important axis is *efficiency.* This axis is commonly measured by *indexing time*, *query time* and *insertion time.* The *indexing time* metrix reflects the computational cost of building the *index*, the data structure that holds the embedded vectors. The *query time*, commonly known as *queries per second (qps)*, shows the average time required to process a single query within an index. Likewise, the *insertion time*, as the name suggests, shows the average time required to insert a query to an index.

These performance metrics are therefore critical for real-time applications such as search engines or recommendation systems, where latencies above a certain treshold can degrade user experience. Memory usage, both during indexing and at query time, is also a practical concern for large-scale datasets [18, 41, 29].

To summarize the trade-off between recall and efficiency into a single metric, it is common to report a *recall-time Pareto frontier.* The *recall-time Pareto frontier* shows how much accuracy can be achieved at different levels of query speed [3, 36]. Typically, such *Pareto frontier* has *recall* values on the *x-axis* and *queries per second(qps)* on the *y-axis.* Algorithms that lie closer to the top-left corner of this plot (high recall, low time) are considered more effective than others. Benchmarking platforms such as *ANN-Benchmarks* [3] and *SISAP* [18] provide reproducible and stable frameworks for testing algorithms across different high dimensional datasets.

Ultimately, the choice of the "best" ANNS algorithm depends on the multiple factors. A well-rounded evaluation therefore should consider quality of the retreieved results through *recall@k*, time through *insertion*, *queries per second(qps)* and *indexing time*, and memory through emprical memory usage and *index size.* The remainder of this thesis follows this multi-metric approach when comparing enhanced versions of the HNSW algorithm to other state-of-the-art methods.

1.6 Motivation and Goal

The motivation for this research is refinement of *robust, high-speed* techniques for retrieving approximate neighbors under a wide variety of real-world conditions. We aim to further optimize the existing methods, specifically Hierarchical Navigable Small Worlds [41] developed algorithm by Malkov et al. by incorporating refined heuristics, exploring navigatiblity optimization, and quantifying how such designs perform in recognized benchmarks.

In the next chapter, Chapter 2, we present a deeper discussion of algorithmic principles behind graph-based and other popular ANNS strategies. We highlighting the primary techniques and their challenges. We further examine widely used evaluation methods and benchmark datasets. Ultimately, this framework sets the stage for our proposed methods of optimizations, Chapter 4, which aim to enhance indexing efficiency and retrieval accuracy in high-dimensional environments and experiments in Chapters 5 and 6.

Chapter 2

Background and Related Work

This chapter surveys the foundational concepts and algorithmic strategies behind ANNS, including an emphasis on the evolution of graph-based methods in Section 2.5, and particularly Hierarchical Navigable Small World [41], which serves as the baseline for the enhancements proposed in this thesis. The next chapter, Chapter 3 further details the theoretical bounds of HNSW.

2.1 Tree Based Algorithms

In Chapter 1 we have outlined that the task of nearest neighbor (NN) search consists of, given a dataset $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ and a query point $\mathbf{q} \in \mathbb{R}^d$ to identify the point in $\mathcal{X}$ that is closest to $\mathbf{q}$ under some distance function dist.

Branch-and-bound (BnB) algorithms, also known as tree based algorithms is one of the earliest methods of solving the Nearest Neighbor search problem. These algorithms aim to achieve sublinear query time by hierarchically partitioning the space and pruning regions that cannot contain the true nearest neighbor. These methods use tree structures to recursively divide the dataset and use geometric properties to avoid exhaustive comparison [12].

2.1.1 k-d Trees

The k-d tree method, introduced by Bentley [8] in 1975, is one of the earliest indexing structures for Nearest Neighbor search. The idea is that a k-d tree structire partitions the data recursively using axis-aligned hyperplanes. At each node, a dimension $j \in \{1, \ldots, d\}$ and a threshold $t \in \mathbb{R}$ which represents the median in dimension j are selected. The dataset then is split into a left subset $\mathcal{X}_L$ and a right subset $\mathcal{X}_R$.

$$\mathcal{X}_L = \{\mathbf{x} \in \mathcal{X} \mid x_j \leq t\}, \quad \mathcal{X}_R = \{\mathbf{x} \in \mathcal{X} \mid x_j > t\}. \tag{2.1}$$

This results in a binary tree where each internal node corresponds to a hyperplane $x_j = t$ dividing the space into two half-spaces $\{x_j \leq t\}$ and $\{x_j > t\}$. Points are stored in leaves, and the tree has depth $O(\log |X|)$ if balanced.

Nearest neighbor search proceeds via depth-first traversal. At each node, the algorithm decides whether to explore the opposite branch based on whether the distance to the splitting hyperplane is less than the current best distance $\delta^\star$. If $|q_j - t| < \delta^\star$, the algorithm backtracks. Otherwise, it prunes that subtree [8, 12].

Under low dimension and well-spread data, k-d trees can answer NN queries in average time $O(\log n)$ [8], far faster than brute force implementation. In practice, k-d trees perform well for moderate dimensions and have storage cost $O(n)$. However, as d grows large, the efficiency of k-d trees rapidly degrades due to *the curse of dimensionality* discussed in Chapter 1.3. Formally, for d that grows, the expected cost approaches $O(n)$, so k-d trees fail to outperform linear scan.

2.1.2 Randomized Projection Trees (RP Trees)

Randomized Projection Trees [15], or commonly known as RP trees, is developed by Dasgupta et al. in 2008. The idea is to generalize k-d tree structure by using random directions rather than fixed axes.

At each node, a random unit vector $\mathbf{u} \in \mathbb{R}^d$ is drawn from the uniform distribution on the unit sphere. Then, we compute the scalar projections of all points onto $\mathbf{u}$. Therefore points are projected as $\langle \mathbf{x}, \mathbf{u} \rangle$. A threshold θ is selected that represents the quantile of these projected values. θ, is chosen at or very near the median of the projected values. This is done uniformly in a small interval around it. θ is then used to split the dataset.

$$\mathcal{X}_L = \{\mathbf{x} \mid \langle \mathbf{x}, \mathbf{u} \rangle \leq \theta\}, \quad \mathcal{X}_R = \{\mathbf{x} \mid \langle \mathbf{x}, \mathbf{u} \rangle > \theta\}. \tag{2.2}$$

This method of random projection defines a random hyperplane (orthogonal to $\mathbf{u}$) as the decision boundary, rather than a coordinate-aligned cut [12, 15].

To search on an RP tree one can use either exact search (with backtracking) or a defeatist approximate search [12, 15]. In an exact search, the same branch-and-bound approach as in a k-d tree, is used. However, the main advantage of RP trees is in approximate nearest neighbor search case. By design, a random split is unlikely to consistently cut off the nearest neighbor. A typical approach to perform a *defeatist search* is descending the tree following the random hyperplane decisions. This is done without backtracking, which runs in $O(\log n)$ time. The probability that such a search fails can be bounded probabilistically based on the number of points between $\mathbf{q}$ and $\mathbf{x}^\star$ in the projected direction [12, 15].

2.1.3 Spill Trees

A spill tree is another variation of a binary space-partitioning tree. A typical Spill Tree implementation aims to avoid costly backtracking by allowing "overlapping" splits. Proposed by Liu et al. [38], and further developed by others, spill trees, as the name suggests "spill" the points that lie near partition boundaries into both sides of the split. Therefore this structure modifies the partitioning strategy by overlapping the regions [12].

Instead of assigning each point to only one child node, a margin $\alpha \in [0, 0.5)$ is used to duplicate points near the splitting threshold. Letting the splitting be along dimension j, with bounds t_L and t_R, the partitions become the following.

$$\mathcal{X}_L = \{\mathbf{x} \mid x_j \leq t_L\} \cup \{\mathbf{x} \mid t_L < x_j \leq t_R\}, \quad \mathcal{X}_R = \{\mathbf{x} \mid x_j > t_R\} \cup \{\mathbf{x} \mid t_L < x_j \leq t_R\}. \tag{2.3}$$

This reduces the likelihood that a query's nearest neighbor is on the opposite side of a split. Therefore performance improvement for defeatist search is possible at the cost of increased tree size [12]. The degree of overlap α determines the space overhead. For small α value, this overhead is only slightly more than m; as $\alpha \to 0.5$, the exponent grows. Finally, in the extreme case $\alpha = 0.5$, the split overlaps entirely and no pruning is achieved [38].

2.1.4 Cover Trees

All the above structures (k-d, RP, spill trees) are binary trees that partition the data with hyperplanes. Therefore they typically assume an explicit vector space with coordinate axes or random projections.

Cover trees [10], developed by Beygelzimer et al. in 2006, operate in general metric spaces. This allows a cover tree structure to exploit hierarchical "covering" and "separation" properties. Formally, a cover tree is defined as a leveled tree with nodes at levels $i \in \mathbb{Z}$. The "covering" aspect is that every node at level $i-1$ is within distance 2^i of some node at level i. Likewise, the "seperation" property holds that any two nodes at level i are at least 2^i apart [10, 12]. More importantly, the cover tree is not binary. The nodes can have an arbitrary number of children. However, the separation condition limits how many children a node can have.

Mathematically, let C_i be the set of points at level i. These levels are infact, nested such that $C_i \subseteq C_{i-1}$. Then for all $p, q \in C_i$, $\text{dist}(p, q) > 2^i$, and for each $x \in C_{i-1}$, Cover Trees show that there exists $p \in C_i$ such that $\text{dist}(x, p) < 2^i$ [10, 12].

Querying a cover tree proceeds a top-down approach. At each node p, if $\text{dist}(\mathbf{q}, p) - 2^i \geq \delta^\star$, where $\delta^\star$ is the best distance found so far, then all children of p can be safely pruned due to the triangle inequality.

2.1.5 Limitations in High Dimensions

Despite their structural differences, all these methods suffer degradation in high-dimensional spaces due to the *curse of dimensionality* [12, 23]. As dimension $d \to \infty$, the distances between points concentrate [9].

As we transition into high-dimensional approximate methods, it is essential to understand these branch-and-bound approaches. They provide the theoretical backbone of NN search and work well in low to moderate dimensions. Their performance collapse in higher dimensions motivates the development of approximation techniques discussed in the following sections.

2.2 Hashing Based Algorithms

The fundamental idea behind hashing based methods is to design hash functions that preserve locality. Therefore, points that are close in the original space are more likely to hash to the same bucket [12]. This section covers classical and advanced hashing-based methods.

2.2.1 Locality-Sensitive Hashing (LSH)

To understand LSH algorithm we first need to define the hash function formally. Let h denote a hash function that belongs to a family of hash functions denoted as $\mathcal{H}$. Formally, an LSH family is defined with respect to a distance metric $D(\cdot, \cdot)$ in a given metric space $\mathbb{R}^d$. For two distance thresholds $r > 0$ and $c \cdot r$ (with $c > 1$ as the approximation factor), and probabilities $0 < p_1 \leq 1$ and $0 < p_2 < p_1$, a family of LSH hash functions $\mathcal{H} = h : \mathbb{R}^d \to U$ maps d-dimensional vectors to some universe U of buckets [1, 12].

Furthermore $\mathcal{H}$ is called (r, cr, p_1, p_2)-sensitive [1, 12] if for any two points $u, v \in \mathbb{R}^d$

$$\begin{cases} \text{If } D(u,v) \leq r, & \Pr[h(u) = h(v)] \geq p_1, \\ \text{If } D(u,v) > cr, & \Pr[h(u) = h(v)] \leq p_2, \end{cases} \tag{2.4}$$

where the hash function h is drawn uniformly at random from $\mathcal{H}$ and $p_1 > p_2$ [1, 25].

An LSH-based approximate nearest neighbor search (ANNS) algorithm, discussed by Andoni et al., constructs L number of hash tables with m number of hashes per table. The construction is done via using compound hash functions $g_i(x) = [h_{i,1}(x), \ldots, h_{i,m}(x)]$ where each hash function is part of the family $h_{i,j} \in \mathcal{H}$. These L number of g_i map $\mathbb{R}^d$ into a multi-dimensional bucket. One can think of such buckets as indexes in a table. To construct the structure, we insert each dataset point into bucket $g_i(x)$ in the ith hash table, for all $i = 1, \ldots, L$. This means that each table is a different random partition of the data [1, 12, 25].

To search for a query point $\mathbf{q}$, candidates are retrieved from all $g_i(\mathbf{q})$ buckets. This is done via computing $g_i(\mathbf{q})$ for all i and collecting all points found in those L buckets as candidate neighbors. In return, this approach reduces the search space significantly compared to exhaustive search.

By making p_1^m substantially larger than p_2^m, the probability gap between p_1 and p_2 is amplified [1, 12, 25]. Therefore $\Pr[g(x) = g(y)] = p_1^m$ holds true for nearby points and $\Pr[g(x) = g(y)] = p_2^m$ holds true for distant points. The number of tables L needed to ensure high recall is estimated to be $O(n^\rho)$, where the exponent ρ is defined as

$$\rho = \frac{\log(1/p_1)}{\log(1/p_2)}. \tag{2.5}$$

Smaller ρ implies better sub-linear performance, and the design of effective LSH families revolves around minimizing this exponent [1].

2.2.2 Hyperplane LSH

One of the earliest and most popular LSH families is hyperplane LSH [13] developed by Charikar et al. in Princeton. In the Hyperplane LSH case, each hash $h_{\mathbf{r}}(\cdot)$ function is the sign of the dot product with a random Gaussian vector $\mathbf{r}$. This means that $\mathbf{r} \sim \mathcal{N}(0, I_d)$ is a random vector. We define the hash function as

$$h_{\mathbf{r}}(\mathbf{x}) = \operatorname{sign}(\langle \mathbf{r}, \mathbf{x} \rangle). \tag{2.6}$$

This method splits space by a random hyperplane. For unit-norm vectors $\mathbf{u}, \mathbf{v}$, the collision probability under this hash becomes

$$\Pr[h_{\mathbf{r}}(\mathbf{u}) = h_{\mathbf{r}}(\mathbf{v})] = 1 - \frac{\theta(\mathbf{u}, \mathbf{v})}{\pi}, \tag{2.7}$$

where $\theta(\mathbf{u}, \mathbf{v})$ is the angle between $\mathbf{u}$ and $\mathbf{v}$ [13]. This formulation makes Hyperplane LSH very effective for cosine similarity search.

2.2.3 Multi-Probe LSH

The classical LSH requires a large number of hash tables to ensure that a nearby point appears in one of them. Multi-Probe LSH [40] developed by Lv et al. addresses this problem by probing multiple nearby buckets in Hamming or integer space. These buckets denoted as $\mathcal{N}_k(g(\mathbf{q}))$ are probed within each table during a query, rather than just the primary hash bucket $g(\mathbf{q})$.

This reduces memory usage by an order of magnitude, since fewer tables are needed. Empirical results show that for the same recall, Multi-Probe LSH can achieve $10\times$ to $15\times$ memory reduction compared to classical LSH [40].

2.2.4 Cross-Polytope LSH

Cross-Polytope LSH [2] developed by Adano et al. in 2015 improves the hash quality by partitioning the space more finely. The algorithm applies a random rotation $R \in \mathbb{R}^{d\times d}$ and maps a vector $\mathbf{x}$ to the index of the closest basis vector in the cross-polytope[1]. The hashing algorithm picks a random orthonormal rotation $R \in \mathbb{R}^{d\times d}$, applies it to the vector $\mathbf{x}$, and then output the index of the basis vector $\pm e_i$ to which

[1]In d dimensions, a cross polytope is the convex hull of the $2d$ vectors $\pm e_1, \pm e_2, \ldots, \pm e_d$, also known as the signed standard basis vectors.

$R\mathbf{x}$ is closest. Formally, this can be written as

$$h(\mathbf{x}) = \arg\max_i |(R\mathbf{x})_i|, \quad i \in \{\pm 1, \ldots, \pm d\}. \tag{2.8}$$

This method achieves near-optimal collision probabilities for angular distance and outperforms Hyperplane LSH both in theory and practice. The exponent ρ achieved by Cross-Polytope LSH approaches the lower bound $\rho = \frac{1}{c^2}$ for approximation factor c, making it one of the most efficient known hashing schemes.

2.2.5 Asymmetric LSH (ALSH)

Standard LSH techniques are designed for metric distances, for which collisions are more likely. But many applications such as recommendation systems require maximum inner product search (MIPS) [12].

MIPS is not a metric space problem and cannot be directly handled by vanilla LSH, as the collision probability condition does not naturally apply to unbounded inner products. ALSH [46], developed by Shrivastava et al. in 2014, transforms the problem of finding high inner product into an approximate nearest neighbor problem in a derived space. This transformation is achieved by asymmetrically modifying vectors.

Each data point $\mathbf{x}$ and query $\mathbf{q}$ are mapped via functions $f_D(x)$ and $f_Q(q)$ such that the inner product $\langle q, x\rangle$ correlates with a distance between $f_Q(q)$ and $f_D(x)$. Then one can apply an ordinary LSH on these transformed points. These transformation functions that converts maximizing $\langle q, x\rangle$ problem to minimizing Euclidean distance between in the augmented space can be denoted as

$$f_Q(\mathbf{q}) = [\mathbf{q}; 0], \quad f_D(\mathbf{x}) = [\mathbf{x}; \sqrt{K - \|\mathbf{x}\|^2}], \tag{2.9}$$

for some constant K hyperparameter [12, 46]. This transformation ensures that the inner product $\langle \mathbf{q}, \mathbf{x}\rangle$ corresponds to the squared Euclidean distance

$$\|f_Q(\mathbf{q}) - f_D(\mathbf{x})\|^2 = K - 2\langle \mathbf{q}, \mathbf{x}\rangle + \text{const}, \tag{2.10}$$

allowing LSH for L_2 distance to be applied to MIPS [46].

In summary, hashing-based ANN methods offer a theoretically grounded and practically effective class of algorithms for sublinear retrieval. From basic LSH to its more advanced variants, these methods exploit randomness and transformations to efficiently index high-dimensional data while providing guarantees on recall and query time.

2.3 Clustering & Quantization Based Algorithms

2.3.1 Cluster-Based Partitioning

An alternative strategy for approximate nearest neighbor (ANN) search problem is the use of clustering to partition the dataset into compact regions. The idea is to approximate the location of a query $\mathbf{x}$ by identifying and scanning only a small number of relevant regions.

Formally, from Chapter 1.2, we let $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^D$ be the dataset. For clustering, we need a new method called *vector quantizer.* A *vector quantizer* defines a mapping $q : \mathbb{R}^D \to C = \{c_1, \ldots, c_k\}$[2] from the continuous D-dimensional space of the vector $\mathbf{x}$ to a finite set of k cluster points. This mapping points the vectors to their nearest centroid, partitioning $\mathbb{R}^D$ into k disjoint Voronoi[3] cells defined as follows

$$V_i = \{\mathbf{x} \in \mathbb{R}^D \mid q(\mathbf{x}) = c_i\}. \tag{2.11}$$

Furthermore the quantization error of the quantizer q is measured via the mean squared error (MSE)

$$\mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[\|\mathbf{x} - q(\mathbf{x})\|^2\right]. \tag{2.12}$$

The *Inverted File Index* (IVF) [32] developed by Jegou et al in 2011, uses this quantization to organize the dataset into k lists. During query, the query vector $\mathbf{q}$ is quantized to the nearest centroids $q(\mathbf{q})$ and only the points in corresponding clusters

[2]Notice the quantizer is defined as q and the query vector is defined as $\mathbf{q}$.

[3]Vornoi cells are crucial for graph based methods as well, therefore a more detailed definition can be found in Chapter 2.5.

are scanned [12, 32]. Probing multiple closest clusters, let's say the top-w nearest centroids, allows tuning recall and speed [12].

A complicated schema such as the *Inverted Multi-Index (IMI)* [6] developed by Babenko et al. in 2012, extends this idea of clustering by factorizing the space into m orthogonal subspaces initially. This orthogonal factorization can be defined as $\mathbb{R}^D = \bigoplus_{i=1}^{m} \mathbb{R}^{D/m}$ [12].

Inverted Multi-Index (IMI) [6] then independently clusters each subspace. The product of centroid indices from each subspace defines a "virtual" centroid in the original space. This method in return produces exponentially many regions without clustering in $\mathbb{R}^D$ directly due to the subspace factorization [12, 6].

2.3.2 Product Quantization

Jégou et al. [32] introduced *Product Quantization* (PQ) in 2011. The key idea was to compress high-dimensional vectors into compact representations that enable fast approximate distance computations. Formally, this product quantization partitions a vector $\mathbf{x} \in \mathbb{R}^D$ into m number of sub-vectors. Formally, this can be shown as

$$\mathbf{x} = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}], \quad \mathbf{x}^{(i)} \in \mathbb{R}^{D/m}. \tag{2.13}$$

Then, for each subspace i, a sub-codebook of K centroids $C^{(i)} = \{\mu_1^{(i)}, \ldots, \mu_k^{(i)}\}$ is learned using K-means [12, 32]. $\mathbf{x}$ is quantized by independently mapping each sub-vector to its closest centroid

$$q(\mathbf{x}) = [\mu_{z_1}^{(1)}, \ldots, \mu_{z_m}^{(m)}], \quad z_i = \arg\min_j \|\mathbf{x}^{(i)} - \mu_j^{(i)}\|^2. \tag{2.14}$$

The encoded vector is stored as the tuple $(z_1, \ldots, z_m)$ [32]. These tuples can then be used to reproduce the quantized vector from $\mathbf{x}$ as $\tilde{\mathbf{x}} = [\mu_{z_1}^{(1)}, \mu_{z_2}^{(2)}, \ldots, \mu_{z_m}^{(m)}]$. This reproduction can be simplified to

$$\tilde{\mathbf{x}} = \sum_{i=1}^{m} \mu_{z_i}^{(i)}, \tag{2.15}$$

A key property of Product Quantization is that distances can be efficiently estimated additively from the codes [12]. Given a query $\mathbf{q}$, one can compute the distance to a PQ-coded vector $\mathbf{x}$ using the reproduced quantized vector $\tilde{\mathbf{x}}$ as

$$\|\mathbf{q} - \tilde{\mathbf{x}}\|^2 = \sum_{i=1}^{m} \|\mathbf{q}^{(i)} - \mu_{z_i}^{(i)}\|^2. \tag{2.16}$$

These distances are computed using precomputed lookup tables for each subspace, yielding efficient nearest neighbor queries in $O(m)$ time per candidate [12, 32].

2.3.3 Optimized Product Quantization

One limitation of standard PQ is the assumption of splitting the space into fixed subspaces. If the data's covariance is not uniform across coordinates, a poor choice of subspace partition can degrade performance. *Optimized Product Quantization (OPQ)* [20] developed by Ge et al. in 2013 improves on PQ by learning a rotation matrix $R \in \mathbb{R}^{D \times D}$. This matrix is then applied before quantization. The key idea is to find a mapping $R : \mathbb{R}^D \to \mathbb{R}^D$ such that when vectors are transformed by R, the resulting $R\mathbf{x}$ is better aligned for PQ, reducing correlation within subspaces [12, 20].

$$\min_{R, \{\mu^{(i)}\}} \mathbb{E}_{\mathbf{x} \sim \mathcal{X}} \left[\|R\mathbf{x} - q(R\mathbf{x})\|^2 \right]. \tag{2.17}$$

The matrix R is typically orthonormal. Thus, it aligns the subspaces with directions of higher variance or decorrelated components [20]. This improves quantization accuracy without increasing the code size.

2.3.4 Applications in GPU-Accelerated Systems

Quantization methods are widely deployed in GPU-based ANN systems such as (Facebook AI Similarity Search) FAISS [17, 30] developed by Facebook Research in 2019. The computations involve many independent table lookups and additions. GPUs are a perfect fit for this task since they can perform computations in parallel across thousands of threads. PQ codes therefore implemented to fit into GPU memory, allowing real-time billion-scale nearest neighbor search with sub-second latencies.

2.4 Sampling Based Algorithms

Sampling-based algorithms provide a different view for approximate nearest neighbor (ANN) search. A typical sampling based algorithm, as the name suggests, uses randomization to estimate similarity scores or ranks without exhaustively computing all distances.

Sampling methods fall into two main categories. *rank-approximation methods*, which estimate the rank of candidates via importance sampling, and *score-approximation methods*, which estimate similarity or distance scores directly via adaptive sampling [12]. Let's outline each of them one by one.

2.4.1 Rank-Approximation via Importance Sampling

The goal of rank-approximation methods is to sample data points from the dataset $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. However, this sampling is not uniform. The frequency with which a point $\mathbf{x}$ is sampled approximates the probability that $\mathbf{x}$ is among the top-k most similar points to a query vector $\mathbf{q} \in \mathbb{R}^d$ [7, 12]. Therefore, the ideal distribution for sampling is one where each point is chosen with probability proportional to its *similarity score* with respect to the query. Formally

$$P(\mathbf{x} \mid \mathbf{q}) \propto \langle \mathbf{q}, \mathbf{x} \rangle. \tag{2.18}$$

One might point out that optimal sampling requires explicitly computing all inner products. To solve this computationally intensive step, the linearity of the inner product and use a two-stage sampling strategy [7, 14] is leveraged such that

$$P(\mathbf{x} \mid \mathbf{q}) = \sum_{t=1}^{d} P(t \mid \mathbf{q}) \cdot P(\mathbf{x} \mid t, \mathbf{q}), \tag{2.19}$$

where $P(t \mid \mathbf{q})$ is a dimension-based sampling distribution. Likewise, $P(\mathbf{x} \mid t, \mathbf{q})$ is the probability of sampling point $\mathbf{x}$ conditioned on dimension t. These conditional probibilities can be calculates as

$$P(t \mid \mathbf{q}) = \frac{q_t \cdot \sum_{\mathbf{x} \in X} x_t}{\sum_{s=1}^{d} q_s \cdot \sum_{\mathbf{x} \in X} x_s}, \quad P(\mathbf{x} \mid t) = \frac{x_t}{\sum_{\mathbf{y} \in X} y_t}. \tag{2.20}$$

This approach makes sampling a coordinate t followed by a point $\mathbf{x}$ yield an overall sampling distribution proportional to the inner product $\langle \mathbf{q}, \mathbf{x} \rangle$ [7, 14, 12]. Furthermore, this sampling scheme can be implemented efficiently using the alias method [50]. This method yields $O(1)$ sampling time per draw after an $O(n)$ preprocessing cost per dimension.

Finally, after S samples are drawn, we obtain a multiset of points. The sampling frequencies approximate the importance within the multiset. The top-k most frequently sampled points are returned as approximate nearest neighbors. Theoretical studies show that with $S = O\left(\frac{1}{\epsilon^2} \log \frac{n}{\delta}\right)$ samples, we recover an ϵ-approximate top-k set with probability at least $1 - \delta$ [12, 7].

2.4.2 Score-Approximation via Adaptive Sampling

An alternative class of algorithms for ANN uses sampling to approximate the scores (distances or similarities) themselves, rather than directly sampling points. The idea is to estimate each data point $\mathbf{x}$'s similarity to the query $\mathbf{q}$ by looking at a random subset of the coordinates. After a few samples, many points can be identified as very unlikely to be among the top-k and safely pruned from consideration [12].

To illustrate the basic intuition, suppose we have a query $\mathbf{q}$ and data point $\mathbf{x}$ in $\mathbb{R}^d$ and we consider the inner product similarity. Just like in the previous example, the similarity score is $\langle \mathbf{q}, \mathbf{x} \rangle = \sum_{t=1}^{d} q_t x_t$, a dot product between vectors. If we randomly sample a subset of r dimensions $D_r \subset 1, \ldots, d$ and compute the partial sum

$$\hat{s}_{\mathbf{x},\mathbf{q}}^{(r)} = \sum_{t \in D_r} q_t x_t, \quad D_r \subset \{1, \ldots, d\}, \ |D_r| = r, \tag{2.21}$$

this becomes which an unbiased estimator of the full similarity score $\langle \mathbf{q}, \mathbf{x} \rangle$.

Moreover, if the variance of contributions across dimensions is not uniform, a small subset may already give a very good approximation. For instance, if the query $\mathbf{q}$ and data point $\mathbf{x}$ have most of their energy in a few coordinates (say one or two coordinates dominate the rest), then just those coordinates are enough to distinguish the nearest neighbors. If the partial scores $\hat{s}^{(r)}_{\mathbf{x},\mathbf{q}}$ of two points differ significantly after a small number of samples, then with high probability, their full scores will differ similarly, allowing confident pruning [37, 47].

In each round, new coordinates are sampled for surviving candidates. Partial scores are updated, and statistical confidence bounds are computed using concentration inequalities such as Hoeffding's [24] or Bernstein's [11] bounds. Points whose upper bounds fall below the lower bound of the current top-k threshold are eliminated. The process continues until the top-k candidates are identified.

2.4.3 Theoretical Guarantees and Extensions

Both rank-based and score-based sampling methods provide (ϵ, δ)-guarantees. This means that with probability at least $1 - \delta$, the returned set of candidates is within ϵ of the true top-k. For rank-approximation, the number of samples required depends on the minimum margin between scores of adjacent-ranked points. For score-approximation, the number of dimensions required per candidate depends on the variance of coordinate contributions and the margin between candidates [12].

Recent work has extended these approaches to inner product retrieval, Euclidean distance, and kernel similarity search. These methods allow non-uniform and adaptive sampling strategies improving performance significantly [21, 47].

2.5 Graph Based Algorithms

Graph-based methods have emerged as one of the most effective solutions for the Approximate Nearest Neighbor Search (ANNS) problem [3, 19, **?**, 29]. These algorithms are particularly effective in high-dimensional settings where the performance of tree-based or hashing methods decreases significantly in both query latency and memory efficiency [10, 23].

To formalize the graph based methods, we will be starting with the mathematical definitions on how to represent a set of vectors as a graph structure. Similar to Chapter 1.2, let $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\} \subset \mathbb{R}^d$ be the set of data, where each $\mathbf{x}_i$ is a vector in d-dimensional space. A graph-based indexing method constructs a graph $G = (V, E)$ where each node in V represents a vector in $\mathcal{X}$, thus $V = \mathcal{X}$, and each edge in E denotes a set of proximity edges between points based on a distance function $\text{dist} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geq 0}$.

As also described in Chapter 1.2, the goal of ANN search is, given a query point $\mathbf{q} \in \mathbb{R}^d$, to find a set of k points $\mathcal{N}_k(\mathbf{q}) \subset \mathcal{X}$ such that each $\mathbf{x}_i \in \mathcal{N}_k(\mathbf{q})$ approximately minimizes $\text{dist}(\mathbf{q}, \mathbf{x})$. Unlike exhaustive search, which computes all n distances with $\text{dist}(\mathbf{q}, \mathbf{x}_i)$, *proximity graph* traversal strategies aim to identify near neighbors by visiting only a logarithmic or sublinear number of points in G.

2.5.1 Greedy Routing

To understand the inner dynamics of *proximity graphs*, it is necessary to define the key feature of such structures. An ideal proximity graph supports *greedy routing*, which is a local search procedure to approximate nearest neighbor search. Formally, this means that for any query $\mathbf{q}$ and any starting node $\mathbf{x}_0 \in \mathcal{X}$, a path $\mathcal{X}_T : \{\mathbf{x}_0, \dots, \mathbf{x}_T\}$ exists such that $\text{dist}(\mathbf{x}_{t+1}, \mathbf{q}) < \text{dist}(\mathbf{x}_t, \mathbf{q})$ for all $t < T$, and $\mathbf{x}_T$ is close to the true nearest neighbor $\mathbf{x}^*$. The algorithm repeatedly moves to a neighboring node that is closer to the query point $\mathbf{q}$, i.e.,

$$\mathbf{x}_{t+1} = \arg \min_{\mathbf{y} \in \mathcal{N}(\mathbf{x}_t)} \text{dist}(\mathbf{q}, \mathbf{y}), \tag{2.22}$$

until no neighbor is found that is closer than the current node. In other words, the search terminates when a local minimum with respect to the query distance is reached.

This procedure requires only local information about the current node's neighborhood, making it extremely efficient in terms of computational and memory overhead. In ideal proximity graphs, greedy routing is guaranteed to find the true nearest neighbor. However, in graphs without global navigability guarantees, greedy routing may find an approximate neighbor instead of the exact nearest neighbor. This makes the greedy search converge to a near-optimal result instead of the exact nearest neighbor [12, 33, 41].

2.5.2 Proximity Graphs

There are many types of *proximity graphs* that support *greedy routing*. This subsection outlines the most popular and well known types of *proximity graphs*.

The *Delaunay Graph* [4, 12, 16], denoted as $\mathcal{D}(\mathcal{X})$, provides a theoretically optimal proximity structure in Euclidean space. The *Delaunay Graph* contains edges between nodes whose *Voronoi Cells* [49] are adjacent. Given a finite set of points $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, the *Voronoi cell* of a point $\mathbf{x}_i \in \mathcal{X}$ is defined as the region of space closer to $\mathbf{x}_i$ than to any other point in the dataset. Formally, Voronoi cells for a given point $\mathbf{x}_i$ can be written as

$$\mathrm{Vor}(\mathbf{x}_i) = \left\{ \mathbf{z} \in \mathbb{R}^d \;\middle|\; \forall j \neq i,\ \mathrm{dist}(\mathbf{z}, \mathbf{x}_i) \leq \mathrm{dist}(\mathbf{z}, \mathbf{x}_j) \right\}. \tag{2.23}$$

The collection of all such Voronoi cells, denoted $\{\mathrm{Vor}(\mathbf{x}_i)\}_{i=1}^{n}$, partitions the space $\mathbb{R}^d$ into convex polytopes known as the *Voronoi Diagram* [4, 5, 12]. All points inside a cell share the same nearest neighbor under the chosen distance metric.

The *Delaunay graph* is defined as the dual of the Voronoi diagram. If, for instance, $\mathrm{Vor}(\mathbf{x}_i)$ denotes the Voronoi region of $\mathbf{x}_i$, then edge e_{ij} exists in $\mathcal{D}$ if $\mathrm{Vor}(\mathbf{x}_i) \cap \mathrm{Vor}(\mathbf{x}_j) \neq \emptyset$. Delaunay graphs are connected and support greedy routing to the true nearest neighbor, but are expensive to construct in high dimensions [45].

The *k-Nearest Neighbor Graph (k-NNG)*, where each node is connected to its k nearest neighbors, is an alternative to Delaunay graphs. To formalize this algorithm, we let $\mathcal{N}_k(\mathbf{x}_i)$ denote the k closest points to $\mathbf{x}_i$ under a distance metric $\text{dist}(\cdot,\cdot)$ as

$$\mathcal{N}_k(\mathbf{x}_i) = \text{argmin}_{\substack{\mathcal{S}\subset\mathcal{X}\\|\mathcal{S}|=k}} \sum_{\mathbf{x}_j\in\mathcal{S}} \text{dist}(\mathbf{x}_i,\mathbf{x}_j). \tag{2.24}$$

This gives us the nodes in V for the graph. To construct the graph, we need a criterion to build the edges in E as

$$E = \{(\mathbf{x}_i,\mathbf{x}_j) : \mathbf{x}_j \in N_k(\mathbf{x}_i)\}. \tag{2.25}$$

This structure approximates Delaunay and supports greedy search, though naive construction may include redundant edges [45, 12, 19]. While easy to build, k-NNGs do not guarantee the existence of monotonic paths, and therefore the search may get stuck in local minima.

To mitigate redundant edges, the *Relative Neighborhood Graph* (RNG) [28] includes edge $(\mathbf{x}_i,\mathbf{x}_j)$ only if:

$$\forall \mathbf{x}_k \in \mathcal{X}\setminus\{\mathbf{x}_i,\mathbf{x}_j\}, \quad \text{dist}(\mathbf{x}_i,\mathbf{x}_j) < \max\left(\text{dist}(\mathbf{x}_i,\mathbf{x}_k), \text{dist}(\mathbf{x}_j,\mathbf{x}_k)\right).$$

Verbally, this means that RNG based methods do not include the edge between two given nodes if there is a third node that is closer to both of them, then they are to each other. This method removes edges that are included by closer intermediaries and ensures the presence of the Minimum Spanning Tree. However, RNG can be too sparse for robust search.

The *Monotonic Relative Neighborhood Graph* (MRNG) extends RNG by ensuring the existence of monotonic descent paths toward any query. While theoretically appealing, MRNG construction remains computationally intractable in high-dimensional spaces [19].

2.5.3 Small-World and Hierarchical Graphs

Small-world graphs, such as those constructed in the Navigable Small World (NSW) approach [41] are another form of graph based solutions. The key idea is to maintain a low graph diameter by combining short-range (local) edges with a sparse set of long-range connections. This structure enables fast navigation and efficient greedy search over the graph [12]. These graphs, as the name suggests, have a property called *small-world property* [33]. That is, the expected number of hops between any two nodes grows polylogarithmically with the number of nodes:

$$\mathbb{E}[\text{path length}] = \mathcal{O}(\log^k n),$$

for a small constant $k > 0$ under typical assumptions of navigability and sparsity [41].

Formally, let $G = (V, E)$ be the resulting graph constructed over dataset $\mathcal{X} \subset \mathbb{R}^d$, where the nodes are $V = \mathcal{X}$. When inserting a new point $\mathbf{x} \in \mathcal{X}$, the algorithm selects a small subset, call it $C \subset \mathcal{X}$, of candidate connection points. These candidates are selected by performing a greedy or beam search on the current graph. The new point $\mathbf{x}$ is then connected to its M nearest points in C

$$\mathcal{N}_M(\mathbf{x}) = \arg \min_{\substack{\mathcal{S} \subset C \\ |\mathcal{S}| = M}} \sum_{\mathbf{y} \in \mathcal{S}} \text{dist}(\mathbf{x}, \mathbf{y}),$$

where $\text{dist}(\cdot, \cdot)$ is a predefined distance function.

By inserting nodes in random order and using a combination of local neighbor selection and approximate search among previously inserted nodes, the NSW algorithm ensures that the resulting graph exhibits the *small-world property* [12, 41].

Hierarchical Navigable Small World Graphs (HNSW) [41] extend the NSW model by introducing a multi-layer graph architecture. This multi-layer graph structure supports logarithmic search complexity and better theoretical navigability [52]. The idea is to assign each node $\mathbf{x} \in \mathcal{X}$ a maximum level $\ell \in \mathbb{N}$ drawn independently from the geometric distribution and thus the layer selection probility can be found as

$$\mathbb{P}(\ell = k) = (1 - p)p^k, \quad \text{for } k \geq 0, \tag{2.26}$$

where $p \in (0, 1)$ is the level decay parameter. Formally, the derivation is presented in Chapter 3 in Lemma 3.1.1 (Level Assignment Distribution).

Each point appears in all layers from 0 up to its assigned ℓ. In Chapter 3, Lemma

3.1.4 (Maximum Level) we show that the highest level L is logarithmic in n. This means that the maximum number of layers could be estimated as $L = \mathcal{O}(\log_{1/p} n)$.

The layering in HNSW forms a pyramid of graphs, with sparse connectivity at higher levels and denser local links at lower levels [41].

At each layer ℓ, the graph maintains a proximity structure where each node $\mathbf{x}$ stores a set of connections $\mathcal{N}_M^{(\ell)}(\mathbf{x}) \subseteq G_\ell$ of fixed size M other nodes. Here G_ℓ denotes the graph layer at level ℓ. These neighbors are selected greedily from candidate nodes found via approximate search

$$\mathcal{N}_M^{(\ell)}(\mathbf{x}) = \text{Select}(\text{Candidates}(\mathbf{x}), M), \tag{2.27}$$

where the selection is designed to maximize spread and coverage [41].

Query search in HNSW proceeds in a top-down fashion. Starting from a randomly chosen entry point $\mathbf{c}_L$ in the top layer G_L, the algorithm performs greedy descent through layers. Formally:

$$\mathbf{c}_{\ell-1} = \arg \min_{\mathbf{x} \in \mathcal{N}^{(\ell)}(\mathbf{c}_\ell)} \text{dist}(\mathbf{q}, \mathbf{x}), \tag{2.28}$$

repeating this process for each l until reaching the base layer G_0. At the bottom level, a greedy search is performed to retrieve the final approximate neighbors.

This structure, including the construction procedure and optimization heuristics, will be analyzed depth in the next chapter, where in this thesis we present our enhancements to its design.

2.5.4 Single-Layer Pruned Graphs: NSG and PANNG

The *Navigating Spreading-out Graph* (NSG) [19] is a graph-based indexing method. The aim is to approximate the navigability guarantees of monotonic proximity graphs. Unlike HNSW approaches, NSG based approaches work in a single-layer structure with significantly reduced construction complexity.

The algorithm begins by constructing a dense approximate k-nearest neighbor graph (k-NNG) [19]. Let $\mathbf{x}_i \in \mathcal{X}$ be a data point, and let $\mathcal{C}_i \subseteq \mathcal{X}$ denote a candidate set of neighbors for $\mathbf{x}_i$, typically containing more than k points. A selection step is then applied to $\mathcal{C}_i$ to retain a subset $\mathcal{N}_i$ satisfying:

$$\mathcal{N}_i = \text{Select}(\mathcal{C}_i,\ \text{monotonicity criterion}), \tag{2.29}$$

where the selection procedure attempts to ensure that for any query $\mathbf{q}$ and any $\mathbf{x} \in \mathcal{X}$, there exists a greedy routing [12, 19]. This monotonic condition guarantees that a greedy search will always progress toward the nearest neighbor of the query.

To prevent edge redundancy and improve sparsity, NSG enforces *Triangle inequality pruning* and *Angular pruning.* In *Triangle inequality pruning* for candidate neighbors $\mathbf{y}, \mathbf{z} \in \mathcal{C}_i$, edge $(\mathbf{x}_i, \mathbf{z})$ is removed if

$$\text{dist}(\mathbf{x}_i, \mathbf{z}) \geq \text{dist}(\mathbf{x}_i, \mathbf{y}) + \text{dist}(\mathbf{y}, \mathbf{z}), \tag{2.30}$$

thus removing indirect or redundant paths.

In *Angular pruning*, candidates are filtered to retain neighbors that provide coverage in diverse directions. This maximizes angular spread. The resulting NSG after two pruning mechanisms remains connected with high probability, and supports greedy routing from any node to any other node [12, 19].

The *Pruned Approximate Nearest Neighbor Graph* (PANNG) [27, 26] follows a similar idea. It starts from a symmetric bidirectional k-NNG and prunes edges that do not significantly contribute to routing quality. For a directed edge $(\mathbf{x}_i, \mathbf{x}_j)$ to be retained, PANNG ensures that:

$$\exists \mathbf{x}_k \in \mathcal{N}(\mathbf{x}_i) : \text{dist}(\mathbf{x}_i, \mathbf{x}_j) < \text{dist}(\mathbf{x}_k, \mathbf{x}_j), \tag{2.31}$$

ensuring that neighbors contribute to progress toward unexplored regions of the space.

Both NSG and PANNG produce sparse, navigable graphs. Furthermore, their single-layer nature makes them attractive for hardware-efficient implementations.

2.5.5 External-Memory Graphs: DiskANN and Vamana

DiskANN [29], developed by Microsoft Research, is a graph-based approximate nearest neighbor search. What makes DiskANN unique is that it is designed to work efficiently over very large datasets[4] that cannot fit entirely in memory. The central idea is to store most of the index on SSD while keeping a small amount of data in RAM to guide the search process. This dual memory approach DiskANN to scale far beyond what purely in-memory algorithms like HNSW or NSG can handle, while still maintaining low latency and high accuracy.

DiskANN can be labeled as graph based method. Infact, the underlying structure used by DiskANN proximity graph called *Vamana* [29]. It's a directed graph $G = (V, E)$ where each node $\mathbf{x}_i \in \mathcal{X} \subset \mathbb{R}^d$ is connected to a set of neighbors $\mathcal{N}(\mathbf{x}_i)$. The number of neighbors is bounded. That is, for each node, the out-degree satisfies

$$|\mathcal{N}(\mathbf{x}_i)| \leq M, \tag{2.32}$$

where M is a fixed parameter chosen during index construction. This bounded degree helps reduce memory usage and limits the number of SSD reads during search.

Unlike other approaches we discussed, Vamana is designed with disk access in mind. During index construction, neighbors of each node are stored close together on disk. This minimizes the number of random page accesses required when traversing the graph [12, 29]. This is important since random I/O on SSDs is orders of magnitude slower than sequential access on RAM.

When a query $\mathbf{q} \in \mathbb{R}^d$ arrives, the search starts from a small number of *entry points* that are kept in RAM. The *entry points* are well-distributed "pivot" nodes selected during indexing. From there, the algorithm performs *beam search* across the disk-resident graph. At each step t, a beam (priority queue) of the B most promising

[4]On the order of billions of vectors

candidate nodes is maintained. The beam is updated using:

$$\mathcal{B}_{t+1} = \text{Top-}B\left(\bigcup_{\mathbf{x}\in\mathcal{B}_t} \mathcal{N}(\mathbf{x}) \cup \mathcal{B}_t,\ \text{by dist}(\mathbf{q}, \cdot)\right), \tag{2.33}$$

The search continues until no better candidates are found or until a fixed number of SSD page reads have been performed.

The expected number of SSD accesses per query scales roughly as

$$\mathcal{O}(B \cdot \log n), \tag{2.34}$$

which is acceptable for real-world applications. In practice, when paired with parallel SSD access, DiskANN achieves high recall with only a few milliseconds of latency even on datasets with over a billion vectors.

2.5.6 Conclusion

Graph-based ANN algorithms are effective methods capable of delivering fast and accurate nearest neighbor search. HNSW offers logarithmic complexity $\mathcal{O}(\log n)$, while well-pruned graphs such as NSG and PANNG perform in $\mathcal{O}(\log^2 n)$ empirically. Their mathematical theory supports efficient navigation through greedy routing and beam search. Their empirical success shows their importance in modern vector search systems.

Chapter 3

Hierarchical Navigable Small World Algorithm

In Chapter 2, we have outlined the methods and different data structures used to solve Approximate Nearest Neighbor Search problem. This chapter will specifically focus on *Hierarchical Navigable Small World* (HNSW) [41] graphs described previously. Since the purpose of this thesis is to provide scalable and fast enhancements on the HNSW algorithm, it is crucial to detail how the algorithm works and its computational complexity.

Hierarchical Navigable Small World (HNSW) graphs were introduced by Malkov and Yashunin in 2018 as a highly efficient approach to approximate nearest neighbor search in high-dimensional spaces [41]. HNSW builds upon earlier *small-world graph* methods [41, 22], incorporating a multi-layer skip-list-inspired structure with greedy graph traversal. It has become one of the most widely adopted ANN algorithms in both academic and industrial systems.

3.1 Graph Structure and Theoretical Foundation

Let $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ be a set of n high-dimensional vectors. The HNSW algorithm constructs a hierarchical set of proximity graphs $\{G_\ell\}_{\ell=0}^{L_{\max}}$ such that each graph $G_\ell = (V_\ell, E_\ell)$ represents a subset of $\mathcal{X}$ connected by similarity edges under some distance function $\mathrm{dist}(\cdot, \cdot)$.

The base layer, layer 0 contains all the nodes. We denote the base layer graph as

$G_0 = (V_0, E_0)$. Therefore the vertexes in this graph can be written as

$$V_0 = \mathcal{X} \tag{3.1}$$

All the graph layers above the base layer contains a subset of the nodes of the layer bellow. This means that the graphs are nested. Each point appears in all layers from 0 up to its assigned ℓ. We denote the maximum number of layers a HNSW structure has as $L_{\max}$. This means that the following condition holds for each layer

$$V_0 \supseteq V_1 \supseteq \cdots \supseteq V_{L_{\max}} \tag{3.2}$$

Each data point $\mathbf{x}_i$ is assigned a maximum level ℓ_i sampled from a geometric distribution:

$$\mathbb{P}(\ell_i \geq k) = p^k, \quad 0 < p < 1. \tag{3.3}$$

This means that the probability a point is assigned to at least level k in the HNSW hierarchy decreases exponentially with k. In other words, most points will only appear in the lower levels, and only a few points will be present in the upper layers of the graph. The higher levels act as a coarse index, allowing fast navigation across the nodes. The parameter p controls the decay rate. Smaller values of p lead to taller hierarchies with fewer high-level nodes. Given a point $\mathbf{x}_i$, using Eq. 3.1, we can find the probability this point being assigned to level k as in the following lemma.

Lemma 3.1.1 (Level Assignment Distribution)**.** *Let ℓ_i be the randomly assigned level of a node $\mathbf{x}_i$ in HNSW, where $\mathbb{P}[\ell_i \geq k] = p^k$ per Eq. 3.1. Then the probability mass function, $\mathbb{P}[\ell_i = k]$ is:*

$$\mathbb{P}[\ell_i = k] = (1-p) \cdot p^k, \quad \ell \in \mathbb{N}_0.$$

Proof. This follows from the memoryless property of the geometric distribution. Given $\mathbb{P}[\ell_i = k] = p^k$, we compute:

$$\begin{aligned} \mathbb{P}[\ell_i = k] &= \mathbb{P}[\ell_i \geq k] - \mathbb{P}[\ell_i \geq k+1] \\ &= p^k - p^{k+1} = (1-p)p^k \end{aligned}$$

□

Utilizing the Lemma 3.1.1, we can get the expected level of a node as in Corollary 3.1.2.

Corollary 3.1.2 (Expected Level of a Node)**.** *The expected value of ℓ_i for a given node $\mathbf{x}_i$ is:*

$$\mathbb{E}[\ell_i] = \frac{p}{1-p}.$$

Proof. By definition, the expected value of ℓ_i is:

$$\mathbb{E}[\ell_i] = \sum_{k=0}^{\infty} k\,\mathbb{P}[\ell_i = k].$$

Substituting the probability mass function from Lemma 3.1.1, we have:

$$\mathbb{E}[\ell_i] = \sum_{k=0}^{\infty} k\,(1-p)p^k.$$

Factoring out the constant $(1-p)$:

$$\mathbb{E}[\ell_i] = (1-p)\sum_{k=0}^{\infty} kp^k.$$

We recognize that:

$$\sum_{k=0}^{\infty} kp^k = \frac{p}{(1-p)^2},$$

which is a standard identity for the expectation of a geometric distribution shifted to start at 0. Thus:

$$\mathbb{E}[\ell_i] = (1-p)\cdot\frac{p}{(1-p)^2} = \frac{p}{1-p}.$$

□

Utilizing the expected value of the level from Corollary 3.1.2, we can now analyze the expected population of nodes at each layer. Since each node independently samples its level according to the same geometric distribution, the expected number of nodes appearing at or above a given level ℓ follows directly by linearity of expectation, as formalized in the following lemma.

Lemma 3.1.3 (Expected Number of Nodes at Level ℓ)**.** *Given N total data points inserted into HNSW, and each point independently assigned a level according to Eq. 3.1, the expected number of nodes at level ℓ is:*

$$\mathbb{E}[|V_\ell|] = N \cdot p^\ell.$$

where N is the total number of nodes inserted into the HNSW graph, hence $|\mathcal{X}|$.

Proof. Let X_i^ℓ be the indicator random variable for point $\mathbf{x}_i$, defined as:

$$X_i^\ell = \begin{cases} 1, & \text{if } \ell_i \geq \ell, \\ 0, & \text{otherwise.} \end{cases}$$

Then the total number of nodes at level ℓ is:

$$|V_\ell| = \sum_{i=1}^{N} X_i^\ell.$$

Taking the expectation and using linearity of expectation:

$$\mathbb{E}[|V_\ell|] = \sum_{i=1}^{N} \mathbb{E}[X_i^\ell].$$

Since all points are independent and identically distributed, and $\mathbb{P}[\ell_i \geq \ell] = p^\ell$ by Eq. 3.1, we have:

$$\mathbb{E}[X_i^\ell] = p^\ell.$$

Thus:

$$\mathbb{E}[|V_\ell|] = N \cdot p^\ell.$$

□

Having established the expected number of nodes at each level, we can now ask how high the hierarchy can extend. In particular, we are interested in bounding the maximum level $L_{\max}$ achieved across all inserted points. The following lemma addresses this question by providing a high-probability bound on $L_{\max}$.

Lemma 3.1.4 (Maximum Level $L_{\max}$)**.** *Let N be the total number of inserted points, and assume each point's level ℓ_i is sampled independently according to Eq. 3.1:*

$$\mathbb{P}(\ell_i \geq k) = p^k,$$

for some $0 < p < 1$. Then, with high probability, the maximum level $L_{\max}$ among all points satisfies:

$$L_{\max} = \mathcal{O}\left(\log_{1/p} N\right).$$

Proof. Fix any level $\ell \geq 0$. For a single point $\mathbf{x}_i$, the probability that $\ell_i \geq \ell$ is:

$$\mathbb{P}[\ell_i \geq \ell] = p^\ell.$$

Define the indicator random variable:

$$X_i^\ell = \begin{cases} 1, & \text{if } \ell_i \geq \ell, \\ 0, & \text{otherwise.} \end{cases}$$

The total number of points reaching level ℓ is:

$$S_\ell = \sum_{i=1}^{N} X_i^\ell,$$

and by Lemma 3.1.3 of expectation:

$$\mathbb{E}[S_\ell] = Np^\ell.$$

We are interested in bounding $\mathbb{P}[S_\ell \geq 1]$, i.e., the probability that at least one node reaches level ℓ. Applying the Chernoff bound for sums of independent Bernoulli random variables, for any $t > 0$,

$$\mathbb{P}[S_\ell \geq t] \leq \exp\left(-t \ln\left(\frac{t}{\mathbb{E}[S_\ell]}\right) + t - \mathbb{E}[S_\ell]\right).$$

Setting $t = 1$, we get:

$$\begin{aligned}\mathbb{P}[S_\ell \geq 1] &\leq \exp\left(-\ln\left(\frac{1}{\mathbb{E}[S_\ell]}\right) + 1 - \mathbb{E}[S_\ell]\right)\\ &= \exp\left(\ln(\mathbb{E}[S_\ell]) + 1 - \mathbb{E}[S_\ell]\right)\\ &= \mathbb{E}[S_\ell] \cdot e^{1-\mathbb{E}[S_\ell]}.\end{aligned}$$

Thus, the probability that at least one node exists at level ℓ is exactly:

$$\mathbb{P}[S_\ell \geq 1] = \mathbb{E}[S_\ell] \cdot e^{1-\mathbb{E}[S_\ell]}.$$

Now, we choose ℓ such that $\mathbb{E}[S_\ell]$ is sufficiently small. Specifically, require that:

$$\mathbb{E}[S_\ell] = Np^\ell = \frac{1}{N}.$$

Thus:

$$p^\ell = \frac{1}{N^2}.$$

Taking the natural logarithm of both sides:

$$\ell \ln p = -2 \ln N,$$

and since $\ln p < 0$, dividing through gives:

$$\ell = \frac{2 \ln N}{\ln(1/p)} = 2 \log_{1/p} N.$$

Therefore, at level $\ell = 2 \log_{1/p} N$, we have:

$$\mathbb{E}[S_\ell] = \frac{1}{N}.$$

Substituting into the probability bound:

$$\begin{aligned}\mathbb{P}[S_\ell \geq 1] &= \mathbb{E}[S_\ell] \cdot e^{1-\mathbb{E}[S_\ell]}\\ &= \frac{1}{N} \cdot e^{1-1/N}\\ &\leq \frac{e}{N},\end{aligned}$$

where we used $e^{1-1/N} \leq e$ since $1/N$ is very small for large N. Thus, the probability that there is any point at level $\ell = 2 \log_{1/p} N$ or higher is at most $\frac{e}{N}$, which is negligible. Hence, with high probability, no points exist at levels larger than $2 \log_{1/p} N$, and we conclude:

$$L_{\max} = \mathcal{O}(\log_{1/p} N).$$

□

In summary, the height of the HNSW hierarchy grows only logarithmically with the dataset size, ensuring that even for large N, the graph remains shallow and search procedures remain efficient. Furthermore, the probability of any single point achieving a significantly higher level decays exponentially fast, guaranteeing that extreme deviations are exceedingly rare.

Having rigorously characterized the distribution of levels and the growth behavior of the maximum level $L_{\max}$, we are now prepared to describe the construction of the HNSW graph in detail. The insertion procedure for new points builds upon the established layered structure and ensures that the graph maintains its small-world and navigability properties across all levels.

3.2 Index Construction

Having described the hierarchical structure of the HNSW graph, we now turn to a detailed exposition of the index construction process. This section formalizes the steps of level assignment, greedy descent, local search, neighbor selection, and graph updating, along with theoretical guarantees on correctness and efficiency.

At each layer ℓ, the graph G_ℓ connects every node $\mathbf{x}_i \in V_\ell$ to at most M neighbors based on proximity. An edge $(\mathbf{x}_i, \mathbf{x}_j) \in E_\ell$ is present if $\mathbf{x}_j$ is among the selected neighbors of $\mathbf{x}_i$ according to a local selection heuristic. The bottom layer G_0 may allow a larger maximum degree $M_0 > M$ to improve recall and graph robustness.

Given a new point $\mathbf{q}$, the insertion algorithm proceeds in several phases. First, let's analyze how a point is assigned to a layer.

3.2.1 Level Assignment

Each new point $\mathbf{q}$ is assigned a random level ℓ_q as follows:

$$\ell_q = \left\lfloor \frac{\ln U}{\ln(1-p)} \right\rfloor, \quad U \sim \text{Uniform}(0,1), \tag{3.4}$$

where p is the parameter controlling the exponential decay of level assignment. We can prove that this sampling is the exact same idea of Lemma 3.1.1, and therefore correct.

Lemma 3.2.1 (Level Sampling Correctness)**.** *The procedure above ensures that:*

$$\mathbb{P}(\ell_q = \ell) = (1-p)p^{\ell}, \quad \ell \in \mathbb{N}_0.$$

Proof. Let $U \sim \text{Uniform}(0,1)$. By definition:

$$\ell_q = \left\lfloor \frac{\ln U}{\ln(1-p)} \right\rfloor.$$

For a fixed ℓ,

$$\begin{aligned}\mathbb{P}(\ell_q = \ell) &= \mathbb{P}\left(\ell \leq \frac{\ln U}{\ln(1-p)} < \ell + 1\right) \\ &= \mathbb{P}\left((1-p)^{\ell+1} \leq U < (1-p)^{\ell}\right) \\ &= (1-p)^{\ell} - (1-p)^{\ell+1} \\ &= (1-p)p^{\ell},\end{aligned}$$

as required. □

One key idea here is that if the assigned level ℓ_q exceeds the current maximum level $L_{\max}$, then $L_{\max}$ is updated and $\mathbf{q}$ becomes the new entry point into the graph. Now, after assigning a point a level, we have to find a way to get to that layer, and this is where *Greedy Layer Descent* is particularly useful.

3.2.2 Greedy Layer Descent

The insertion process starts from the topmost layer $L_{\max}$ and proceeds down to $\ell_q + 1$.

At each layer ℓ, starting from an entry node e, the algorithm performs greedy search: at each step, it moves to the neighbor closest to the query $\mathbf{q}$, until no closer neighbor is found.

Lemma 3.2.2 (Greedy Descent Correctness)**.** *At each level $\ell > \ell_q$, the greedy descent ensures that the algorithm finds a local minimum of the distance to $\mathbf{q}$ among all nodes reachable from the starting entry point.*

Proof. Let the current node at time step t be $\mathbf{x}_t$. Define the distance to the query point as:

$$d_t = \text{dist}(\mathbf{x}_t, \mathbf{q}).$$

At each step, the algorithm selects the next node:

$$\mathbf{x}_{t+1} = \arg \min_{\mathbf{y} \in \mathcal{N}(\mathbf{x}_t)} \text{dist}(\mathbf{q}, \mathbf{y}),$$

where $\mathcal{N}(\mathbf{x}_t)$ denotes the neighbors of $\mathbf{x}_t$ at level ℓ. The algorithm proceeds if and only if:

$$\text{dist}(\mathbf{x}_{t+1}, \mathbf{q}) < \text{dist}(\mathbf{x}_t, \mathbf{q}),$$

i.e., if a strictly closer neighbor is found. Thus, the sequence $(d_t)_{t=0}^T$ is strictly decreasing:

$$d_0 > d_1 > d_2 > \cdots > d_T,$$

where T is the final step when no closer neighbor exists. Since the set of nodes is finite and the distances are lower bounded (by 0), the descent must terminate in a finite number of steps. At termination, for the final node $\mathbf{x}_T$, we have:

$$\text{dist}(\mathbf{x}_T, \mathbf{q}) \leq \text{dist}(\mathbf{y}, \mathbf{q}) \quad \forall \mathbf{y} \in \mathcal{N}(\mathbf{x}_T).$$

Thus, $\mathbf{x}_T$ is a local minimum of distance to $\mathbf{q}$ within the connected component reachable via greedy moves. □

Formally, the traversal at layer ℓ can be written as a trivial greedy routing, just like in Eq. 2.22:

$$\mathbf{x}_{t+1} = \arg \min_{\mathbf{y} \in \mathcal{N}(\mathbf{x}_t)} \text{dist}(\mathbf{q}, \mathbf{y}). \tag{3.5}$$

One other key feature of the hierarchical layers is that the final node at level ℓ becomes the entry point for level $\ell - 1$ and the search continues through the lower levels.

3.2.3 Local Search and Candidate Neighbor Selection

At the target layer ℓ_q and all lower layers down to 0, a local best-first search is performed to identify candidate neighbors.

Specifically, starting from the previously identified entry point, a priority queue is maintained with up to `efConstruction` elements, sorted by distance to $\mathbf{q}$. To further understand the selection of neighbors we define a new set as follows:

Definition 3.2.3 (Candidate Neighbor Set). *Let $C_\ell(\mathbf{q})$ denote the set of candidate neighbors found at level ℓ for point $\mathbf{q}$. It is obtained by expanding up to `efConstruction` nodes in a best-first manner.*

After collecting candidate neighbors through the best-first local search, a heuristic pruning step is performed to select a final, well-distributed subset of neighbors. Formally, the algorithm considers all nodes encountered during the search, aggregates their neighbors, and selects the most promising ones:

$$C_\ell(\mathbf{q}) \leftarrow \text{Top-}M\left(\bigcup_{i=1}^{\texttt{efConstruction}} \mathcal{N}(\mathbf{x}_i)\right), \tag{3.6}$$

where Top-M denotes selecting the M closest nodes to $\mathbf{q}$, while additionally applying a diversity heuristic to avoid selecting nodes that are too close to each other.

The pruning heuristic proceeds greedily: nodes are considered in order of increasing distance to $\mathbf{q}$, and each new candidate is added to $C_\ell(\mathbf{q})$ only if it is sufficiently far from all already selected nodes. This ensures that the resulting neighborhood is not only close to the query but also spatially well-separated. We can formally show this as in Lemma 3.2.4.

Lemma 3.2.4 (Neighbor Diversity Guarantee)**.** *The pruning heuristic ensures that the selected neighbors are not only close to* $\mathbf{q}$ *but also well-separated among themselves.*

Proof. Let $C_\ell(\mathbf{q})$ denote the final set of selected neighbors at level ℓ. During the pruning procedure, a candidate node $\mathbf{y}$ is considered for addition into $C_\ell(\mathbf{q})$ only if, for all already selected nodes $\mathbf{z} \in C_\ell(\mathbf{q})$, it satisfies:

$$\text{dist}(\mathbf{q}, \mathbf{y}) < \text{dist}(\mathbf{y}, \mathbf{z}).$$

That is, the candidate must be closer to the query $\mathbf{q}$ than to any previously selected neighbor. This condition prevents adding candidates that are too close to existing neighbors, enforcing a lower bound on the mutual distances between nodes in $C_\ell(\mathbf{q})$. As a result, the selected neighbors are not only near $\mathbf{q}$ but are also spatially diverse, avoiding redundant or colinear placements. This spatial diversity improves the navigability of the graph and its overall connectivity properties. $\square$

3.2.4 Graph Update

After selecting the final set of neighbors $C_\ell(\mathbf{q})$ through local search and diversity pruning at each level, the algorithm updates the graph structure by inserting bidirectional edges between $\mathbf{q}$ and its selected neighbors:

$$E_\ell \leftarrow E_\ell \cup \{(\mathbf{q}, \mathbf{x}_j), (\mathbf{x}_j, \mathbf{q}) \mid \mathbf{x}_j \in C_\ell(\mathbf{q})\}. \tag{3.7}$$

This way both the query point $\mathbf{q}$ and each neighbor $\mathbf{x}_j$ recognize each other as adjacent vertices.

The graph update process is repeated sequentially from the assigned level ℓ_q down to the base layer 0, incrementally integrating $\mathbf{q}$ into each corresponding G_ℓ.

Lemma 3.2.5 (Connectivity Maintenance)**.** *At each level ℓ, the insertion of* $\mathbf{q}$ *preserves the local connectivity and the small-world properties of the graph.*

Proof. The greedy search performed during insertion ensures that $\mathbf{q}$ connects to an existing node that is locally optimal with respect to distance minimization.

Moreover, the local best-first search with parameter `efConstruction` expands the neighborhood exploration, finding a diverse set of nearby nodes. The diversity pruning heuristic ensures that the selected neighbors $C_\ell(\mathbf{q})$ are spatially well-distributed around $\mathbf{q}$, preventing clustering and promoting coverage.

Since all selected neighbors are connected bidirectionally to $\mathbf{q}$, the graph remains undirected at each level.

Thus, the resulting graph after insertion continues to satisfy the small-world property meaning high local clustering combined with short global navigation paths. □

3.2.5 Insertion Complexity Analysis

Having detailed the structural updates during insertion, we now analyze the computational complexity involved in inserting a new point into the HNSW graph.

Theorem 3.2.6 (Insertion Complexity)**.** *The expected time complexity for inserting a single point* $\mathbf{q}$ *into an HNSW graph with N points is:*

$$\mathcal{O}(\log N).$$

Proof. The insertion procedure consists of two major phases

Greedy Descent through Upper Layers: The number of layers traversed is proportional to the maximum level $L_{\max}$. By previous analysis, $L_{\max} = \mathcal{O}(\log_{1/p} N)$, thus traversal requires $\mathcal{O}(\log N)$ steps. At each layer, the greedy descent examines a small constant number of nodes due to the small-world structure.

Local Search and Neighbor Updates: At each level $\ell \leq \ell_q$, a local best-first search is performed with parameter `efConstruction`, which is typically set as a small constant relative to N. Searching and updating neighbors require $\mathcal{O}(\texttt{efConstruction} \cdot M)$ operations, which are constant with respect to N.

Thus, the dominant contribution to insertion time comes from the $\mathcal{O}(\log N)$ greedy traversal across levels, leading to an overall expected insertion complexity of $\mathcal{O}(\log N)$. □

Corollary 3.2.7 (Total Index Construction Time)**.** *Building an HNSW graph on a dataset of N points requires:*

$$\mathcal{O}(N \log N)$$

expected total time.

Proof. Each point is inserted independently into the graph with expected cost $\mathcal{O}(\log N)$ (by Theorem 3.2.6). Summing over all N points yields a total expected construction time of:

$$\sum_{i=1}^{N} \mathcal{O}(\log i) = \mathcal{O}(N \log N),$$

where we used the fact that $\sum_{i=1}^{N} \log i = \Theta(N \log N)$ by standard integral approximations. $\square$

In conclusion, the hierarchical and navigable structure of HNSW enables highly efficient index construction, with near-linear scaling in the number of points inserted. Each new point requires only logarithmic effort relative to the current graph size, maintaining both scalability and fast query performance as the dataset grows.

Having now established how the HNSW graph is built and updated, we turn to the second fundamental operation. Querying for approximate nearest neighbors. The HNSW search algorithm leverages the layered graph structure to quickly locate good candidates by first performing coarse navigation at higher levels and then refining the search at the base layer.

3.3 Search Algorithm

The search process for a query point $\mathbf{q} \in \mathbb{R}^d$ consists of two main phases, which together combine global exploration with local exploitation.

3.3.1 Greedy Hierarchical Descent

The first phase is a fast greedy descent from the topmost layer $L_{\max}$ down to the base layer 0.

The algorithm begins at the designated entry point $\mathbf{x}_{L_{\max}} \in V_{L_{\max}}$, which is the highest-level node in the structure. At each level ℓ, given the current node $\mathbf{x}_{\ell+1}$, the next node $\mathbf{x}_\ell$ is selected by minimizing the distance to the query over the neighbors of $\mathbf{x}_{\ell+1}$:

$$\mathbf{x}_\ell = \arg \min_{\mathbf{y} \in \mathcal{N}(\mathbf{x}_{\ell+1})} \operatorname{dist}(\mathbf{q}, \mathbf{y}), \tag{3.8}$$

where $\mathcal{N}(\mathbf{x}_{\ell+1})$ denotes the set of neighbors of $\mathbf{x}_{\ell+1}$ at level ℓ.

This greedy step is repeated until no closer neighbor is found at the current level, at which point the algorithm descends one level lower. Since each move guarantees a reduction in distance to $\mathbf{q}$, and the number of levels is logarithmic in N, the hierarchical descent phase requires only $\mathcal{O}(\log N)$ steps.

At the end of this phase, the search has located a promising candidate $\mathbf{x}_0 \in V_0$ at the bottom layer, close to the query point. We can reuse the Lemma 3.2.2 Greedy Descent Correctness into the following lemma

Lemma 3.3.1 (Greedy Descent Monotonicity)**.** *During hierarchical descent, the distance to the query* $\mathbf{q}$ *is non-increasing at every step.*

Proof. At each level ℓ, the algorithm selects a neighbor $\mathbf{y}$ satisfying

$$\text{dist}(\mathbf{q}, \mathbf{y}) \leq \text{dist}(\mathbf{q}, \mathbf{x}_{\ell+1}).$$

Thus, the distance to $\mathbf{q}$ does not increase as we move down the hierarchy. Since the number of candidate neighbors is finite and each transition either decreases or preserves distance, the process must terminate at a local minimum at each level. $\square$

3.3.2 Best-First Search at Layer 0

Once at layer 0, the algorithm switches from greedy descent to a more exhaustive but bounded search strategy. Specifically, starting from $\mathbf{x}_0$, a best-first search is performed using a priority queue of size up to `ef` (the "exploration factor").

The search proceeds with the initialization of the heap with $\mathbf{x}_0$. While the heap is not empty and a stopping condition is not met the algorithm pops the node $\mathbf{x}_t$ with the smallest distance to $\mathbf{q}$. During the same while loop, for each neighbor $\mathbf{y} \in \mathcal{N}(\mathbf{x}_t)$, $\mathbf{y}$ is inserted to the heap if $\mathbf{y}$ has not been visited and if $\mathbf{y}$ is closer than the farthest candidate recorded. This means that

$$\text{dist}(\mathbf{q}, \mathbf{y}) < \text{dist}(\mathbf{q}, \texttt{farthest candidate}), \tag{3.9}$$

needs to hold.

The best-first search continues expanding nodes, updating the candidate heap with closer neighbors whenever possible [12, 41]. Once the heap stabilizes, meaning no new candidates improve the distance to the query, the algorithm terminates and returns the top-k approximate nearest neighbors. Formally, this is done by returning the k closest nodes maintained in the candidate heap:

$$\texttt{CandidateHeap} \leftarrow \text{Top-}k\left(\text{beam\_search}(\mathbf{x}_0, \texttt{ef})\right), \tag{3.10}$$

where $\texttt{beam\_search}(\mathbf{x}_0, \texttt{ef})$ denotes the best-first traversal starting from $\mathbf{x}_0$ with a maximum of `ef` nodes kept in the active exploration list.

The parameter `ef` (exploration factor) controls the breadth of the search. The larger values of `ef` allow the algorithm to explore a wider region of the graph around $\mathbf{q}$, leading to more accurate results but at a higher computational cost. Conversely, small values of `ef` make the search faster but more greedy and locally confined.

Lemma 3.3.2 (Search Completeness vs Exploration Factor)**.** *The probability that the returned top-k set contains a true nearest neighbor increases monotonically with* `ef`*. Moreover, the expected recall improves as* `ef` *increases, while the time complexity grows linearly with* `ef`*.*

Proof. At each expansion step during `beam_search`, the algorithm considers all neighbors of the currently best candidate and inserts closer nodes into the candidate heap, maintaining at most `ef` active candidates.

Let $R(\texttt{ef})$ denote the expected recall, this means the probability that a true nearest neighbor is contained in the final Top-k list.

By construction:

$$R(\texttt{ef}_1) \leq R(\texttt{ef}_2) \quad \text{for} \quad \texttt{ef}_1 < \texttt{ef}_2,$$

because a search with more active candidates explores a larger neighborhood and thus has a higher chance of encountering true neighbors.

Similarly, let $T(\texttt{ef})$ denote the expected search time, dominated by:

$$T(\texttt{ef}) = \mathcal{O}(\texttt{ef} \log \texttt{ef}),$$

due to heap operations (insertions and deletions) performed on a structure of size `ef`.

Thus, there is a direct trade-off: larger `ef` increases recall but also increases computational cost approximately linearly in `ef` (up to a logarithmic factor from heap maintenance). □

In practical settings, for small datasets, `ef` values around 10-50 may be sufficient. For larger datasets or when high accuracy is critical, values up to 200 or higher may be used. Tuning `ef` is thus a crucial part of optimizing HNSW performance.

3.3.3 Overall Search Complexity

Having described the search procedure in detail, we now analyze the expected computational complexity of a single query.

We distinguish two phases of the query:

Lemma 3.3.3 (Hierarchical Greedy Descent Complexity). *The greedy traversal from the top layer $L_{\max}$ down to layer 0 requires $\mathcal{O}(\log N)$ steps in expectation.*

Proof. From the earlier analysis, we know that with high probability, the maximum level $L_{\max}$ satisfies:

$$L_{\max} = \mathcal{O}(\log_{1/p} N),$$

where $p \in (0, 1)$ is the level decay probability.

At each level ℓ, the algorithm performs a greedy search starting from the entry point found at the higher layer. Formally, at level ℓ, the current node $\mathbf{x}_\ell$ is updated by:

$$\mathbf{x}_\ell = \arg \min_{\mathbf{y} \in \mathcal{N}(\mathbf{x}_{\ell+1})} \text{dist}(\mathbf{q}, \mathbf{y}),$$

where $\mathcal{N}(\mathbf{x}_{\ell+1})$ denotes the neighborhood of the starting point at level ℓ.

Due to the small-world property of each graph G_ℓ, the expected number of hops needed to reach a local minimum at each level is bounded by a constant $C > 0$, independent of N. That is:

$$\mathbb{E}[\text{hops per level}] \leq C.$$

Thus, the total number of node-to-node hops across all levels is bounded by:

$$\mathcal{O}(L_{\max} \cdot C) = \mathcal{O}(L_{\max}).$$

Substituting $L_{\max} = \mathcal{O}(\log_{1/p} N)$, we conclude:

$$\mathcal{O}(L_{\max}) = \mathcal{O}(\log_{1/p} N) = \mathcal{O}(\log N),$$

where the base of the logarithm is absorbed into the $\mathcal{O}(\cdot)$ notation. Therefore, the greedy hierarchical descent requires $\mathcal{O}(\log N)$ expected time. □

Lemma 3.3.4 (Layer-0 Best-First Search Complexity). *The best-first search at the base layer requires $\mathcal{O}(\texttt{ef} \log \texttt{ef})$ operations.*

Proof. During the base layer search (at level 0), the algorithm maintains a candidate heap H, a priority queue ordered by distance to the query point $\mathbf{q}$.

At any moment, the size of the heap H is bounded above by `ef`, where `ef` is the exploration factor parameter.

Each basic heap operation such as insertion, extraction of the minimum, or update costs:

$$\mathcal{O}(\log \texttt{ef})$$

time per operation. Since the best-first search expands at most `ef` nodes (each node added to or removed from the heap once), the total number of heap operations is $\mathcal{O}(\texttt{ef})$.

Thus, the overall time complexity of managing the heap is:

$$\mathcal{O}(\texttt{ef} \cdot \log \texttt{ef}).$$

This term dominates the complexity of the layer-0 search, since distance computations and neighborhood checks are also bounded per candidate expansion. Hence, the best-first search at the base layer requires $\mathcal{O}(\texttt{ef} \log \texttt{ef})$ time. □

Combining both phases, we can deduce our final theorem for this section.

Theorem 3.3.5 (Query Time Complexity). *The expected search time for a query in HNSW is sublinear in the dataset size N, scaling as:*

$$\mathcal{O}(\log N + \mathtt{ef} \log \mathtt{ef}).$$

Proof. The total search cost is the sum of:

- $\mathcal{O}(\log N)$ steps for hierarchical greedy descent, and
- $\mathcal{O}(\mathtt{ef} \log \mathtt{ef})$ operations for the best-first search at the base layer.

Note that typically $\mathtt{ef} \ll N$. This means tat the dominant term is logarithmic in N, ensuring highly scalable search performance. □

3.4 Conclusion

In conclusion, the HNSW [41] algorithm combines probabilistic modeling with hierarchical graph design to achieve high-speed and accurate approximate nearest neighbor search. Its layered structure enables logarithmic search depth, while bounded-degree local connectivity ensures memory efficiency and robustness. In the following chapters, we will explore how this core framework can be enhanced and extended for real-time updates, memory efficiency, and high-recall regimes.

Chapter 4

Optimizations

In the preceding chapters, specifically Chapter 3, we have detailed the structure, construction, and search algorithms of Hierarchical Navigable Small World (HNSW) [41] graphs. In this chapter, we explore two optimization strategies aimed at enhancing HNSW's performance.

The principal objectives of these optimizations are to minimize both query and indexing times, and to achieve higher recall-efficiency trade-offs while maintaining memory consumption within practical limits. In the following chapters, specifically Chapter 5 and 6, we will be implementing and evaluating these optimizations

Many of the ideas discussed here or are inspired by prior work [18, 21, 35, 41, 53], and adapted to the specific properties of HNSW graph-based search structures. We also propose some novel enhancements.

4.1 Robust Pruning via Distance-Based Filtering

One common problem in HNSW graphs is that in densely populated regions, nodes can end up with many redundant neighbors. As the algorithm has to expand more neighbors at every step, this leads to high out-degrees per node. Therefore, increasing the memory consumption and slowing down search.

To address this, we introduce a simple but effective pruning method inspired by DiskANN [29], which filters out redundant neighbors based on relative distance comparisons.

Given a node $\mathbf{v}$ and a set of candidate neighbors $\mathcal{C}(\mathbf{v})$ found during local search, for each candidate neighbor $\mathbf{u} \in \mathcal{C}(\mathbf{v})$, we check if there is another candidate $\mathbf{w} \in \mathcal{C}(\mathbf{v})$ that is a "better" connection. We check this using the following:

$$\|\mathbf{v} - \mathbf{w}\| < \frac{1}{\alpha}\|\mathbf{u} - \mathbf{w}\|, \tag{4.1}$$

where $\alpha > 1$ is a tunable parameter controlling the aggressiveness of pruning. If such a $\mathbf{w}$ exists, we prune the candidate $\mathbf{u}$.

If $\mathbf{w}$ is much closer to $\mathbf{v}$ than $\mathbf{u}$ is to $\mathbf{w}$ (scaled by $1/\alpha$) then we prefer to connect to $\mathbf{w}$ instead of $\mathbf{u}$. This is because $\mathbf{w}$ is already "better positioned" to reach towards $\mathbf{u}$ if needed later. The Pseudocode implementation of the idea is as follows.

Algorithm 1 GreedySpannerPrune($v, C(v), \alpha, M$)

```
Output: neighbor set $N(v)$
Sort $C(v)$ in ascending order of $d(v, \cdot)$;                // closest first
$N(v) \leftarrow \emptyset$  foreach $u \in C(v)$ do
    if $|N(v)| \geq M$ then
        break
    accept ← true  foreach $w \in N(v)$ do
        if $d(v, w) < \frac{d(u,w)}{\alpha}$ then
            accept ← false  break
    if accept then
        $N(v) \leftarrow N(v) \cup \{u\}$
return $N(v)$
```

Now, we formalize that this pruning schema forms a spanner of the original candidate graph.

Theorem 4.1.1 (Stretch factor of distance–ratio pruning). *Let $\alpha > 1$ and let G' be the graph obtained from a candidate Euclidean graph G by distance–ratio pruning: an edge $(\mathbf{v}, \mathbf{u})$ is deleted iff there exists a vertex $\mathbf{w}$ such that*

$$\|\mathbf{v} - \mathbf{w}\| \; < \; \frac{1}{\alpha} \, \|\mathbf{u} - \mathbf{w}\|. \tag{4.2}$$

Then G' is a $t(\alpha)$-spanner of G with

$$t(\alpha) \; = \; \frac{\alpha + 1}{\alpha - 1} \, . \tag{4.3}$$

In particular, for any accuracy parameter $\varepsilon \in (0, 1)$, choosing $\alpha := 1 + \frac{2}{\varepsilon}$ yields a

$(1+\varepsilon)$*-spanner:*

$$d_{G'}(\mathbf{v},\mathbf{u}) \;\le\; (1+\varepsilon)\,\|\mathbf{v}-\mathbf{u}\| \qquad \textit{for every pair of vertices } \mathbf{v},\mathbf{u}. \tag{4.4}$$

Proof. We fix two vertices $\mathbf{v},\mathbf{u}$ and consider the first edge $(\mathbf{v},\mathbf{u})$ that is discarded when the candidate edges are examined in non-decreasing order of length. By the pruning rule there exists a witness vertex $\mathbf{w}$ with

$$\|\mathbf{v}-\mathbf{w}\| \;<\; \frac{1}{\alpha}\,\|\mathbf{u}-\mathbf{w}\| \implies \|\mathbf{u}-\mathbf{w}\| > \alpha\|\mathbf{v}-\mathbf{w}\|. \tag{4.5}$$

We denote $a=\|\mathbf{v}-\mathbf{w}\|,\; b=\|\mathbf{u}-\mathbf{w}\|,\; c=\|\mathbf{v}-\mathbf{u}\|$. From (4.8) we have $b>\alpha a$. The triangle inequality gives $c \ge b-a$, hence

$$a < \frac{1}{\alpha-1}\,c \quad\text{and}\quad b < \frac{\alpha}{\alpha-1}\,c.$$

Therefore the length of the two-hop detour $v \to w \to u$ is

$$a+b < \left(\tfrac{1}{\alpha-1}+\tfrac{\alpha}{\alpha-1}\right)c = \frac{\alpha+1}{\alpha-1}\,c. \tag{4.6}$$

Because edges are processed shortest first, every surviving edge is no longer than (v,u). Replacing each discarded edge along a path by its two-hop detour and summing the bound (4.9) inductively along that path yields

$$d_{G'}(v,u) \;\le\; \frac{\alpha+1}{\alpha-1}\,\|v-u\|, \tag{4.7}$$

establishing the stated stretch factor $t(\alpha)$. Setting $\alpha := 1+\frac{2}{\varepsilon}$ makes $t(\alpha)=1+\varepsilon$, completing the proof. □

This distance-based pruning strategy for graph construction and proved that it yields a $(1+\varepsilon)$-spanner when the pruning parameter is set to $\alpha = 1+\frac{2}{\varepsilon}$. Our analysis, based purely on local triangle inequalities, shows that each removed edge can be replaced by a two-hop path with controlled stretch. The parameter α directly governs the sparsity-accuracy trade-off. Larger α values (smaller ε) preserve distances more tightly at the cost of retaining more edges, while smaller α values allow more aggressive pruning with slightly higher path distortions. Thus, by adjusting α, the algorithm offers a tunable balance between graph compactness and path quality, backed by formal spanner theory.

4.2 Layer-wise Dimension Reduction via Random Projections

The coarse layers of an HNSW index contain only a small subset of all points. The distance evaluations in those layers are still performed in the full ambient dimension d. A classical tool for lowering that cost is the *Johnson–Lindenstrauss (JL) lemma [31]*, which states that a short random projection already preserves all pairwise distances up to a tiny, controllable error. For the purposes of this thesis, we will not prove the *Johnson–Lindenstrauss (JL) lemma*, instead we will give the lemmas as follows.

Lemma 4.2.1 (Johnson–Lindenstrauss [31])**.** *Fix $\varepsilon \in (0, 1/2)$ and let $P = \{x_1, \dots, x_m\} \subset \mathbb{R}^d$ with $m \geq 2$. Choose a random matrix $R \in \mathbb{R}^{k \times d}$ whose entries are i.i.d. $\mathcal{N}(0, 1/k)$. If*

$$k \;\geq\; \left\lceil 8\,\varepsilon^{-2} \ln m \right\rceil,$$

then with probability at least $1 - m^{-2}$ the linear map $f(x) = Rx$ satisfies

$$(1-\varepsilon)\,\|x_i - x_j\|_2^2 \;\leq\; \|f(x_i) - f(x_j)\|_2^2 \;\leq\; (1+\varepsilon)\,\|x_i - x_j\|_2^2 \quad \text{for all } 1 \leq i < j \leq m.$$

To implement this lemma the HNSW setup, let n_ℓ be the number of vertices that "survive" up to layer ℓ ($n_0 = N = |\mathcal{X}|$ at the base).

We pick a fixed distortion $\varepsilon = 0.1$ and define

$$k_\ell \;=\; \left\lceil 8\,\varepsilon^{-2} \ln n_\ell \right\rceil. \tag{4.8}$$

In the implementation, independently for each layer, we sample a Gaussian matrix $R^{(\ell)} \in \mathbb{R}^{k_\ell \times d}$ and store every vector x in that layer as the shorter vector $x^{(\ell)} = R^{(\ell)}x$. Distance evaluations performed while searching in layer ℓ are therefore done in $\mathbb{R}^{k_\ell}$.

Under Lemma 4.2.1 this means that for every pair of points x_i, x_j in the layer we have the two–sided inequality

$$(1-\varepsilon)\,\left\|x_i - x_j\right\|_2^2 \;\leq\; \left\|R^{(\ell)}x_i - R^{(\ell)}x_j\right\|_2^2 \;\leq\; (1+\varepsilon)\,\left\|x_i - x_j\right\|_2^2, \tag{4.9}$$

where $R^{(\ell)}$ is the random projection matrix used in layer ℓ. In our evaluation, we will modify this εin (4.9). For example, substituding $\varepsilon = 0.1$i n (4.9) yields

$$0.9\,\left\|x_i - x_j\right\|_2^2 \;\leq\; \left\|R^{(\ell)}x_i - R^{(\ell)}x_j\right\|_2^2 \;\leq\; 1.1\,\left\|x_i - x_j\right\|_2^2. \tag{4.10}$$

Lemma 4.2.1 (with $m = n_\ell$ and our choice of k_ℓ) guarantees that *all* pairwise distances inside layer ℓ deviate by at most a small percentage. Such a small perturbation leaves the ordering of the M closest neighbours at every node unchanged. Therefore greedy search in the compressed layers walks along similarly to the vertex sequence as in the full space.

Notice that the new traversal in the compressed layers is precisely the same as the ambient dimension walk. However this tradeoff for quality of queries comes with an improvement in speed. Each dot product in layer ℓ now costs $k_\ell = O(\ln n_\ell)$ multiplications, instead of d. Because n_ℓ shrinks exponentially with ℓ, these k_ℓ values quickly fall below a few dozen, yielding substential reduction in arithmetic work above the base layer.

By compressing every coarse layer with its own Johnson–Lindenstrauss projection we keep the routing quality essentially similar while making distance computations much cheaper and the upper-layer storage much smaller.

Chapter 5
Approach

In the previous chapters, we rigorously analyzed the theoretical foundations of the HNSW algorithm, including its layer construction, insertion complexity, and expected query performance. Now, we shift focus from theory to practice. In this chapter, we outline how we will evaluate our proposed enhancements to the HNSW algorithm through practical experimentation. Our evaluation consists of two main components. First, we implement and test the base algorithm for its expected logarithmic behavior. Second, we benchmark its performance against existing HNSW implementations on standard datasets.

Throughout these experiments, we gave careful attention to the testing environments. The reason is to ensure reproducibility and fairness in our implementation choices, and evaluation methodology. Therefore, we need to start with detailing the implementation environments and the choices we have made for the purpose of this thesis.

5.1 Implementation Environment

All of our algorithms and enhancements are fully implemented in `C++`, with minimal external dependencies. We also utilized the `Python` bindings[1] to the `C++` code to ensure experimentation and access to the source code through a well known interface.

[1]The `Python` bindings were heavily inspired by the existing the HNSW libraries. Since the algorithm itself and the enhancements are not dependent on the bindings used, altered versions of the `Python` bindings from the `hnswlib`, `FAISS-HNSW`, and `Vespa-HNSW` were used to ensure stability and reproducibility across runs

First, given the nature of approximate nearest neighbor search, where performance is highly sensitive to low-level operations like distance computations and graph updates, `C++` allows direct control over memory layout, concurrency, and fine-tuning of critical code paths. Second, leading HNSW libraries such as `hnswlib`, `FAISS-HNSW`, and `Vespa-HNSW` are also implemented in `C++`, making it essential to ensure a fair and meaningful comparison. Lastly, modern `C++` standards (`C++17` and `C++20`) provide access to parallelism and efficient low-level operations, which are crucial for scaling our algorithm to large datasets. Thirdly, since `C++` is a low level language compared to other forms of implementations such as `Python`, the `C++` implementation allows orders of magnitude higher number of operations on the same set of data with the same amount of time, thus a higher qps(queries per second) is achievable using a low level language.

We use standard build tools including `CMake` and modern compilers such as `g++` and `clang++`. All development and testing are containerized using Docker to maintain consistency across different machines and seperate the experiments on dedicated CPUs such that no external process running on the same machine interferes with the results.

For hardware, we use two Amazon Web Services (AWS) EC2 instances of the `r6i` family powered by 3rd Generation Intel Xeon Platinum 8375C CPU 2.90GHz Scalable processors. The first environment features 16 vCPUs and 128 GiB of RAM, while the second provides 64 vCPUs and 512 GiB of RAM. Both run Amazon Linux 2023.7 release with `x86-64` architecture and are optimized for high-memory applications. The first environemnt with lower number of VCPUs is used for rapid development of implementation whereas the 64 vCPU machine was dedicated to run the second phase of evaluations with real-world datasets for production. Therefore these environments allow us to test both moderate and large-scale behaviors of the algorithm.

5.2 Evaluation Strategy

Our evaluation proceeds in two phases to test specific properties of the algorithm and its enhancements. Therefore, it is crucial to outline each with detail.

5.2.1 Phase 1: Verifying Logarithmic Behavior

The first phase aims to validate the theoretical logarithmic scaling of insertion and search times derived in Chapter 3. In order to work the enhancements proposed on Chapter4, we first have to write our own `C++` implementation for the HNSW algorithm. This phase of experimentation is therefore necessary to validate the `C++` implementation written is valid and works without error. To validate the algorithm, we propose to run our `C++` implementation, that is `my-algorithm-base` directory and measure the computational complexity experimentally. In the case that the algorithm supports the analysis of Chapter 3, we will then optimize this implementation with Chapter 4 in mind. For fast development and easy debugging, this phase uses the 16 vCPU environment the runs the experiments on parallel for faster results.

To test, we generate a synthetic random datasets with a set seed for reproducibility in the feature. This dataset is made up of uniformly distributed vectors in a high-dimensional space, specifically in $d = 128$. The dataset sizes vary from 10^4 up to 10^7 points to replicate the real world data. For each dataset size, we measure three quantities: the average insertion time per point, the average query time per point and the index construction time.

If the theoretical analysis in Chapter 3 holds, we expect that the insertion time $T_{\text{insert}}(N)$ and the search time $T_{\text{query}}(N)$ scale approximately as $\mathcal{O}(\log N)$.

Formally, for insertion,

$$T_{\text{insert}}(N) = \mathcal{O}(\log N),$$

and for querying,

$$T_{\text{query}}(N) = \mathcal{O}(\log N + \texttt{ef} \log \texttt{ef}),$$

where `ef` is the exploration factor defined in Chapter 3.

Furthermore, we also test the effect of the parameters `ef` and `M` in this randomly generated set of vectors.

To verify these relations, we plot the measured times against $\log n$ and compare them to the theoretical guarantees. Each measurement will be averaged over 5 trials with different seed to minimize the noise and effect of randomness.

5.2.2 Phase 2: Benchmarking Against Standard Datasets

The second phase of our experiments evaluates the practical performance of our implementation against existing HNSW libraries. This phase uses the 64 vCPU environment to handle larger datasets and more expensive computations and uses dedicated Docker containers of 16 vCPUs for each algorithm. The runs are made with the `--parallelism 15` flag to run the operations of a given algorithm in paralell to reduce the time per each algorithm.

We benchmark our implementation against three well-known HNSW implementations: `hnswlib`, `FAISS-HNSW`, and `Vespa-HNSW`. `hnswlib`[2] is developed by the authors of the original HNSW paper by Yu. A. Malkov and D. A. Yashunin. `FAISS-HNSW`[3] is an alternative HNSW implementation developed by Facebook Research. Finally, `Vespa-HNSW`[4] is another implementation of HNSW by Vespa.ai.

Each implementation is configured with the same hyperparameters for a fair comparison. Specifically, all algorithms are run with `efConstruction: 500`. To explore the trade-off between search quality and efficiency, we vary the graph connectivity parameter `M` across the set `[4, 8, 12, 16, 24, 36, 48, 64, 96]`. Similarly, to control the breadth of the search, we vary the exploration factor `ef` over the set `[10, 20, 40, 80, 120, 200, 400, 600, 800]`.

Varying both M and `ef` allows us to generate a large collection of points with dif-

[2]https://github.com/nmslib/hnswlib/tree/denorm
[3]https://github.com/facebookresearch/faiss
[4]https://github.com/vespa-engine/vespa-ann-benchmark

ferent speed-accuracy trade-offs. This is essential for constructing the Pareto Frontier discussed in Chapter 1 Section 5, where we identify configurations that are non-dominated in both recall and query time. By using common M and `ef` lists across all algorithms, we ensure that the Pareto curves are directly comparable and that no algorithm is unfairly favored by parameter tuning.

We evaluate across six widely used datasets in ANN-Benchmarks repository[3]:

- *GloVe-25-angular*: Word embeddings of dimension 25 with angular distance.
- *GloVe-100-angular*: Word embeddings of dimension 100 with angular distance.
- *SIFT-128-euclidean*: Local image descriptors in 128 dimensions with Euclidean distance.
- *NYTimes-256-angular*: Bag-of-words embeddings of New York Times articles, 256-dimensional, with angular distance.
- *Fashion-MNIST-784-euclidean*: Flattened pixel embeddings of fashion item images, 784-dimensional, using Euclidean distance.
- *GIST-960-euclidean*: High-dimensional (960D) global image descriptors for image retrieval tasks, using Euclidean distance.

Each of these datasets presents different challenges in terms of dimensionality and distance structure. This setup allow a comprehensive analysis for evaluating the quality and scalability of nearest neighbor search methods. Next section gives brief descriptions of the datasets.

5.2.3 Dataset Descriptions

GloVe

The *GloVe[44]* datasets, both *GloVe-25-angular[3]* and *GloVe-100-angular[3]*, are pre-trained word vectors derived from large corpora such as Wikipedia and Common Crawl [44]. They capture semantic similarities between words and serve as a benchmark for low and medium dimensional nearest neighbor search.

The *GloVe-25-angular[3]* dataset consists of 1.18 million vectors, each of 25 dimensions, while *GloVe-100-angular[3]* contains the same number of vectors but with 100 dimensions. In both cases, similarity search is performed using the angular (cosine) distance metric.

SIFT

The *SIFT*[39] dataset consists of feature vectors extracted from real-world images. It is widely used in computer vision benchmarks [32]. Each vector encodes local image descriptors, making the dataset a benchmark for high-dimensional nearest neighbor search. The *SIFT-128-euclidean[3]* variant contains 1 million vectors, each of 128 dimensions. In this benchmark, the Euclidean distance metric is used to measure similarity between image descriptors.

NYTimes

The *NYTimes[42]* dataset represents a real-world application of dense document embeddings. It is often used in recommendation systems and search engines [3]. Each vector corresponds to a news article represented in a semantic embedding space. The *NYTimes-256-angular[3]* benchmark contains approximately 290,000 vectors, each of 256 dimensions. In this dataset, the angular (cosine) distance metric is used to evaluate similarity between document embeddings.

Fashion-MNIST

The *Fashion-MNIST[51]* dataset is a modern replacement for the classic MNIST[34] handwritten digit dataset, featuring grayscale images of various fashion products such as shirts, shoes, and bags [51]. Each image is flattened into a 784-dimensional vector, corresponding to its 28×28 pixel grid. The benchmark *fashion-mnist-784-euclidean[3]* consists of approximately 70,000 vectors, and the Euclidean distance is used to measure similarity between the embedded representations of the fashion items.

GIST

Finally, the *GIST[43]* dataset consists of high-dimensional global image descriptors, hence the name GIST, that capture both the spatial structure and textural properties of images [43]. In the *gist-960-euclidean[3]* benchmark, each image is represented as a 960-dimensional feature vector. The dataset contains approximately 1 million vectors, and similarity is measured using the Euclidean distance. GIST is often used to evaluate the scalability and robustness of nearest neighbor search algorithms under very high-dimensional settings.

5.2.4 Evaluation Metrics

As introduced in Chapter 1 Section 5, evaluating Approximate Nearest Neighbor Search (ANNS) methods requires multiple forms of assessment. Therefore a "good" evaluation metrics needs to asses both retrieval quality and computational efficiency. In this section, we briefly summarize the specific metrics used in our experiments that were already detailed in Chapter 1 Section 5, following the same formal definitions and evaluation protocols established earlier.

First, we measure the *search quality* using the widely adopted metric of *Recall@k* [3], defined formally in Chapter 1 Equation (1.14). *Recall@k* measures the proportion of true nearest neighbors successfully retrieved among the top-k results:

$$\text{Recall@k} = \frac{|\mathcal{N}_k^{\text{exact}}(\mathbf{q}) \cap \mathcal{N}_k^{\text{ann}}(\mathbf{q})|}{k}.$$

Second, we evaluate *efficiency* through the following metrics:

Query Time (ms), defined as the average time taken to process a single query. This metric corresponds to the efficiency discussions in Chapter 1 Section 5.

Build Time (s), the total wall-clock time required to construct the index structure over all data points.

Memory Usage (MB), the maximum resident set size (RSS) measured during both indexing and querying phases.

All experimental evaluations closely follow the standardized benchmarking methodology proposed by the `ANN-Benchmarks` framework [3], ensuring fair, reproducible, and comparable results across different datasets and methods.

Furthermore, to summarize the trade-off between accuracy and efficiency, we report *Recall-Time Pareto Frontiers* as introduced in Chapter 1 Section 5. These plots visualize how much search quality (*recall*) can be achieved for different levels of query speed (*qps*). The Pareto frontier, as mentioned in the Introduction, plots recall on the x-axis and queries per second (qps) on the y-axis. This method highlights the inherent trade-offs in ANN algorithms. Methods closer to the top-left corner (higher recall, faster queries) are considered superior according to this composite evaluation [3, 36].

5.3 Summary

In conclusion, the two-phase evaluation strategy provides a complete and rigorous assessment of our algorithm. The first phase verifies theoretical complexity claims under controlled settings, while the second phase benchmarks practical performance on large, diverse datasets against strong baselines. This methodology shows both the mathematical properties and real-world applicability of our approach are thoroughly validated.

The next chapter will present detailed experimental results, comparisons, and analysis based on this evaluation framework.

Chapter 6
Evaluation

In Chapter 3 we rigorously analyzed the theoretical foundations of the HNSW algorithm[41]. In Chapter 4 detailed the proposed enhancements aimed at improving its efficiency. In Chapter 5 we outlined the experimental framework and implementation environment in which these methods would be validated.

In this chapter, we present the empirical evaluation of our enhancements against the baseline HNSW implementation. The goal for this chapter is verifying both theoretical predictions and practical performance gains.

The structure of this chapter follows the plan outlined in Chapter 5. First, we present the experimental results validating our own implementation of HNSW using randomly generated a set of vectors. Secondly, we conduct a comparative study of the proposed optimizations and analyze their impact on various search and indexing performance metrics in real world datasets.

6.1 Baseline Validation

Before evaluating the effectiveness of the proposed enhancements, it is essential to verify that our baseline HNSW implementation accurately reproduces the theoretically expected behaviors outlined in Chapter 3. This baseline validation ensures that any additional improvements observed can be attributed solely to the algorithmic optimizations, and not some systematic error.

To this end, we conducted a detailed study using synthetically generated datasets consisting of uniformly random vectors in $\mathbb{R}^{128}$, varying the dataset sizes from 10^4 to 10^7 points. Each dataset was generated with a fixed random seed to ensure reproducibility across multiple experimental runs. For each dataset size, we measured the average insertion time per point, the average query time per point, and the total index construction time. These metrics are then compared against the theoretical scaling predictions derived in Chapter 3.

To make these evaluations more consistent, a Docker container was used. Furtermore, to run the evaluation faster, the algorithm was ran on all `16vCPUs`. To ensure full reproducibility of the experimental setup, we implemented a custom Python script with `seed=42` to automate the generation, indexing, querying, and measurement processes.

The synthetic datasets were generated by drawing independent random vectors from a uniform distribution over $[0,1]^d$, normalized appropriately to fit into a bounded domain if needed. The following function was used.

```python
def generate_random_vectors(count, dim):
    return np.random.random((count, dim)).astype(np.float32)
```

To construct the index, we needed to access our HNSW engine implemented in `C++` through a `Python` interface. Since the purposes of this thesis is on implementation and enhancements of the algorithm, the existing `Python` bindings in the HNSW repository has been used to access our own engine. Since the choice of `Python` bindings does not change the source implementation in `C++`, this choice has been made to spend more time on algorithm development instead of focusing on building an interface.

To install our `C++` HNSW engine implementation to the `Python` interface, the following Docker file has been used.

```dockerfile
FROM python:3.9-slim AS builder

# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    git \
    python3-dev \
    python3-pip \
    g++ \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*
# Set working directory
WORKDIR /build

# Copy only what's needed for building our hnswlib implementation
COPY src/ /build/src/

# Install build dependencies
RUN pip install pybind11 wheel setuptools numpy matplotlib tqdm psutil

# Build our underlying hnswlib engine
WORKDIR /build/src/python_bindings
RUN python3 setup.py install
```

For the complexity analysis of the code, we have carefully chosen parameters `M=48`, `efConstruction=500`, and `efSearch=500`. We specifically chose these parameters to make the transition from baseline validation to real world scnerios since the same parameters were used in the *ANN-Benchmarks* repository. Therefore we initialized our index using the following code.

```python
index = hnswlib.Index(space='l2', dim=128)
index.init_index(max_elements=10**7, ef_construction=500, M=48)
```

Insertion times were recorded by measuring the elapsed wall-clock time between successive calls to `add_items` during batch insertions, divided by the number of inserted points to get a fair average. Search times were measured by issuing k-nearest neighbor queries for randomly generated queries, using the following timing framework:

```python
index.set_ef(500)  # Set efSearch parameter
start_time = time.time()
for query in queries:
    index.knn_query(query.reshape(1, -1), k=10)
end_time = time.time()
average_query_time = (end_time - start_time) / len(queries)
```

To validate theoretical scaling, we varied the dataset size N from 10^4 to 10^7 points, measured insertion and query times, and plotted them.

6.1.1 Insertion Time Results

The experimental results for insertion time are shown in Figure 6.1.

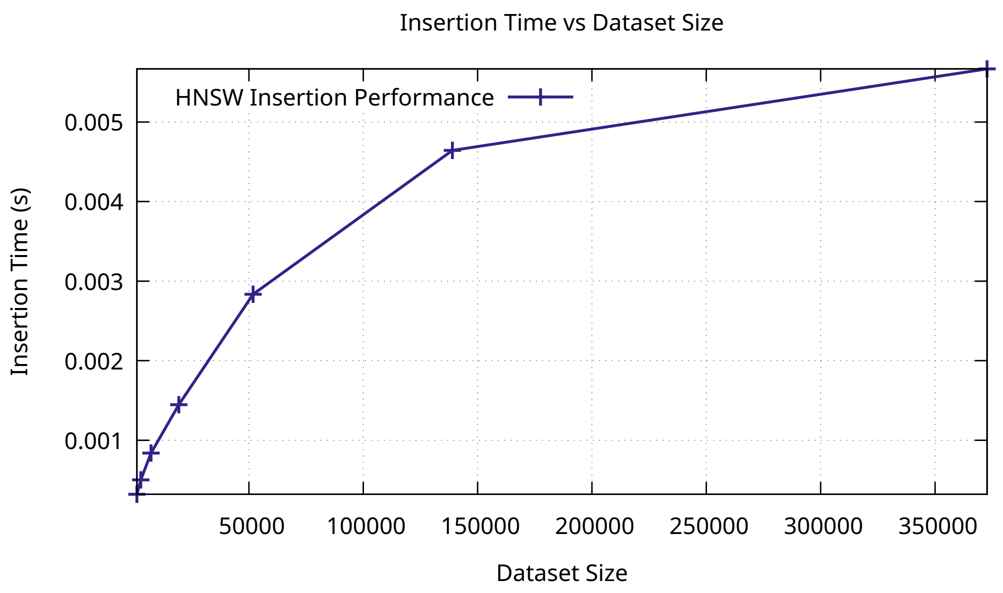

Figure 6.1: The average insertion time(s) $T_{\text{insert}}(N)$ as a function of number of vectors N in linear axis scaling.

As observed, the average insertion time per point grows logarithmically with N.

Overall, Figure 6.1 confirms that our implementation is the matching the $\mathcal{O}(\log N)$ complexity derived in Chapter 3 theoretically. This means that our HNSW engine written in `C++` is in agreement with the theoretical bounds of insertion.

6.1.2 Search Time Results

Similarly, Figure 6.2 shows the average query time per point plotted against N.

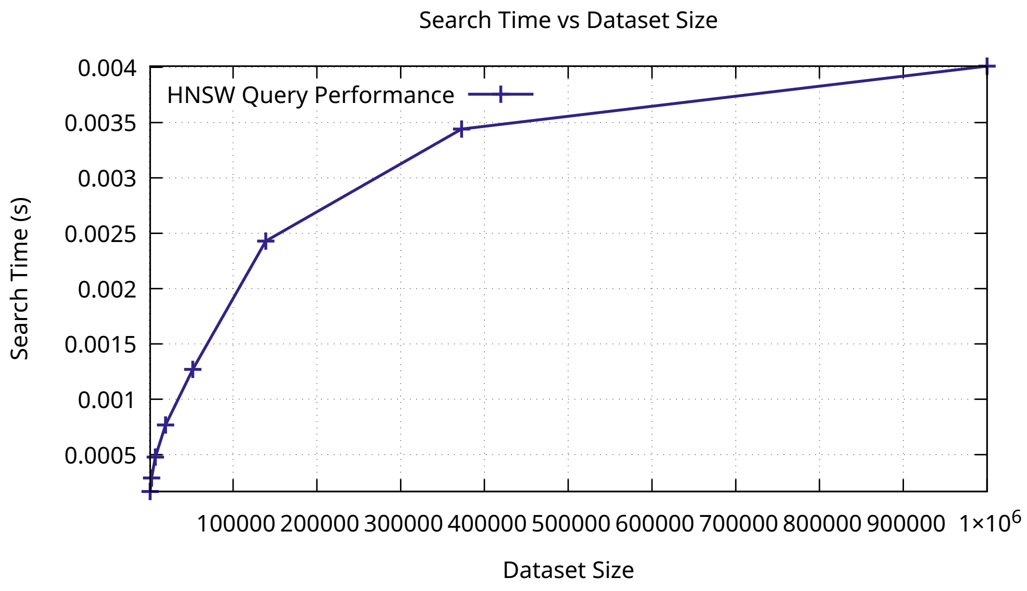

Figure 6.2: The average search time(s) $T_{\text{search}}(N)$ as a function of number of vectors N in linear axis scaling.

Here too, the trend is consistent with a logarithmic growth model.

Figure 6.2 therefore shows a trend with respect to $\log N$, confirming that the average search time grows logarithmically with the dataset size. Minor deviations from perfect linearity are again observed, which can be attributed to system-level factors as described previously.

Thus, the empirical behavior of the search procedure aligns with the theoretical complexity of $\mathcal{O}(\log N)$ for search operations, as derived in Chapter 3. The results validate that our HNSW implementation maintains its expected efficiency across both insertion and search phases, even at large scales.

6.1.3 Memory Usage

In addition to insertion and search times, memory consumption is also very critical. This metric is particulary useful for us to evaluate the practical scalability of our HNSW engine written in `C++`. Figure 6.3 shows the total memory footprint of the HNSW index as a function of the number of inserted vectors N.

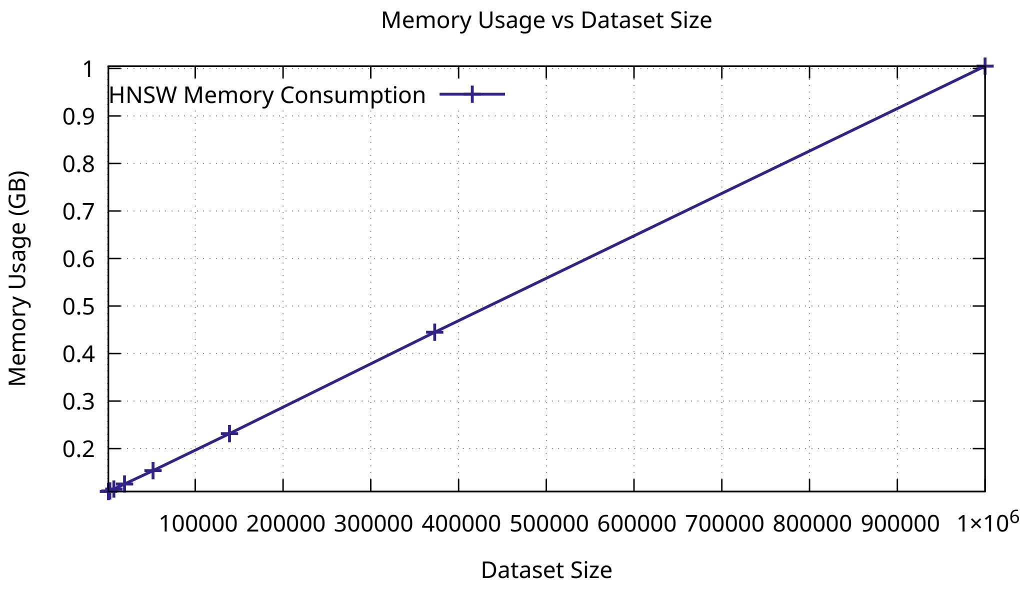

Figure 6.3: The total memory usage of RAM (in GB) of the HNSW index as a function of number of vectors N.

As depicted in Figure 6.3, the memory usage grows approximately linearly with N. This behavior is expected since each node maintains a fixed-size set of connections determined by the maximum degree parameters M and $M_{\max}$, as discussed in Chapter 3. Therefore, the per-node memory overhead remains constant. This leads to an overall linear memory growth relative to the number of vectors.

6.1.4 Index Construction Time

Finally, as we discussed in Corollary 3.2.7 in Chapter 3, the total index construction time is $O(N \log N)$. To prove this hypotesis and show that our implementation achieves this bound we will use Figure 6.4:

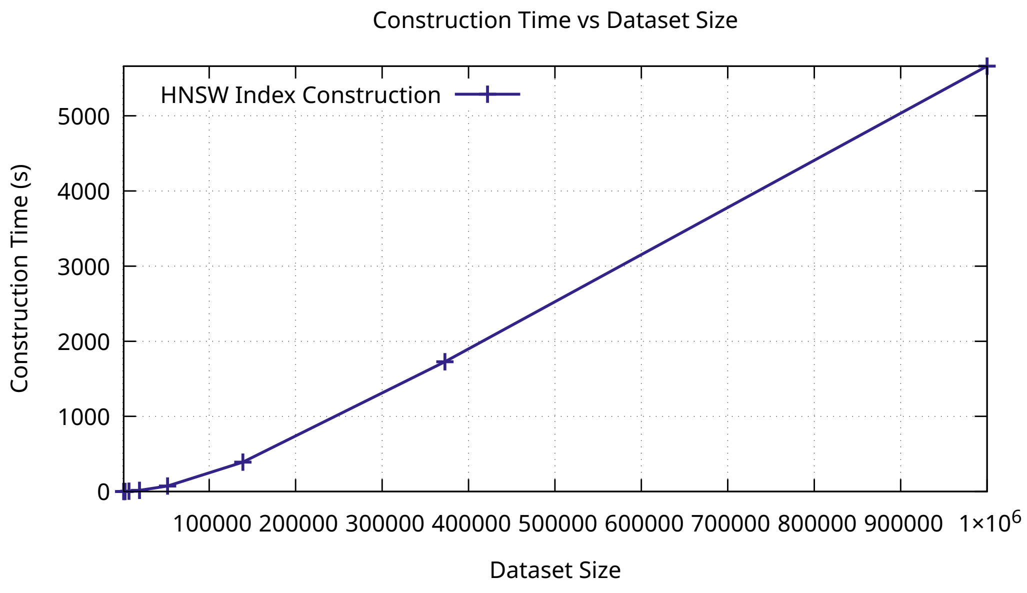

Figure 6.4: The total index construction time(s) of the HNSW index as a function of number of vectors N.

Even though this figure shows that total index construction time scales with $O(N \log N)$, in order to formalize this, we need to alter the *x-axis* and make it $N \log N$ instead of N. After doing this alteration, if the algorithm is scaling with $O(N \log N)$, then we should see a linear relation. This is shown in Figure 6.4 and supports our idea of $O(N \log N)$, also theorized in Corollary 3.2.7 in Chapter 3. Even though we see minor deviations from perfect linearity[1], this supports our theory and implementation.

[1]Which can be attributed to hardware-level effects such as cache hierarchy transitions and memory bandwidth saturation at higher N

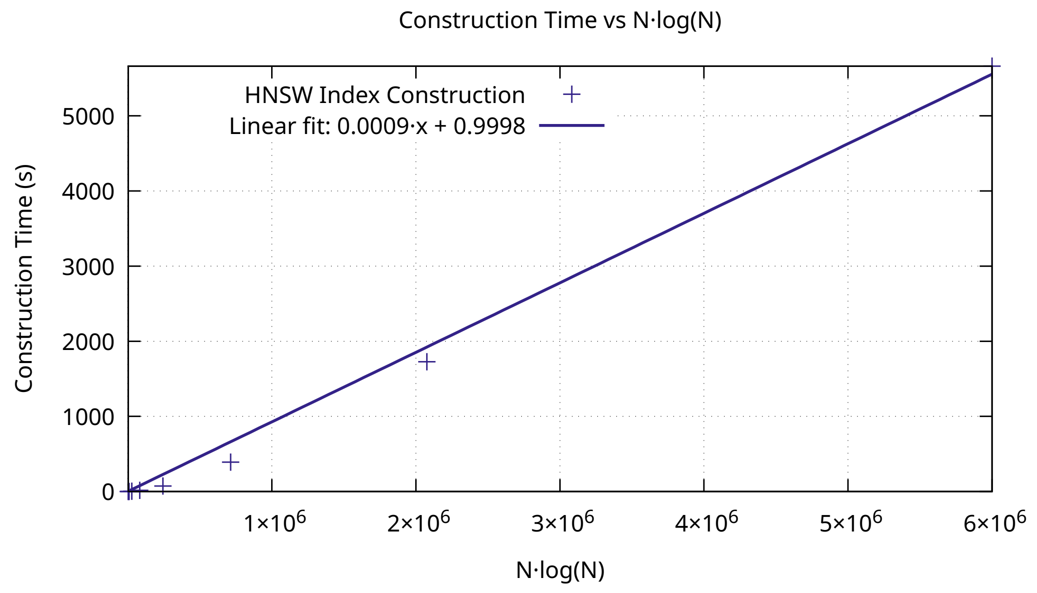

Figure 6.5: The total index construction time(s) of the HNSW index as a function of number of vectors N modified axis.

Overall, these results confirm that our HNSW implementation achieves the theoretically expected computational and memory efficiencies, making it ready for large-scale, real-world data and possible enhancements detailed in Chapter 4.

6.2 Optimizations and Real World Results

In the previous section, we validated our baseline implementation of HNSW and confirmed its logarithmic behavior under controlled synthetic datasets. We now turn to evaluating the practical benefits of the proposed optimizations introduced in Chapter 4. In this section, we benchmark the enhanced version of the algorithm against the baseline HNSW across a variety of real-world datasets.

The primary objective of this phase is to assess whether the enhancements, specifically distance-based robust pruning and dimension reduction, translate into measurable improvements. Our primary analysis will query performance when deployed on non-synthetic, high-dimensional data. In particular, we are interested in whether the optimized algorithms can improve the trade-off between search speed and recall, effectively shifting the Pareto frontier introduced in Chapter 1. Our goal is achieving either higher recall at the same query time, or lower query time at the same recall, compared to the baseline HNSW and other HNSW implementations.

6.2.1 Datasets

Unlike random vectors, real datasets often exhibit complex structures such as clustering and low-dimensional manifold embeddings. As a reitteration of Chapter 5, lets summarize the datasets we will be using with Table 6.1.

Table 6.1: Summary of Datasets Used in Evaluation.

Dataset	**Vectors**	**Dimensions**	**Metric**
GloVe-25-angular	1.18M	25	Angular
GloVe-100-angular	1.18M	100	Angular
SIFT-128-euclidean	1M	128	Euclidean
NYTimes-256-angular	290K	256	Angular
Fashion-MNIST-784-euclidean	70K	784	Euclidean
GIST-960-euclidean	1M	960	Euclidean

6.3 Empirical Effects of Distance-Based Robust Pruning

The first optimization we evaluated is distance-based robust pruning applied during neighbor selection. As discussed in Chapter 4, in classical HNSW graphs, nodes often accumulate many redundant neighbors, especially in dense regions of the space. This redundancy increases both memory consumption and query times.

To address this, we implemented a distance-based pruning method inspired by DiskANN [29]. Given a candidate neighbor set $\mathcal{C}(\mathbf{v})$ for a node $\mathbf{v}$, a neighbor $\mathbf{u}$ is pruned if there exists another candidate $\mathbf{w}$ satisfying

$$\|\mathbf{v} - \mathbf{w}\| < \frac{1}{\alpha}\|\mathbf{u} - \mathbf{w}\|,$$

where $\alpha > 1$ controls the pruning.

This pruning method ensures that redundant connections are removed while still preserving the overall connectivity of the graph. In Chapter 4, Theorem 4.2.1 we showed that the resulting graph G' after pruning forms a $t(\alpha)$-spanner of the original graph G with $t = \frac{\alpha+1}{\alpha-1}$. This means that with setting $\alpha := 1 + \frac{2}{\epsilon}$ we can effectively achieve an $(1+\epsilon)$-spanner graph of the original graph G. Therefore, with preserving the distances almost the same across vectors, we are in fact reducing the number of edges substantially. In other words, we are preserving the shortest-path distances approximately with only a small multiplicative stretch but also reducing the number of redundant edges.

In our evaluation, we varied α to analyze its impact. Since higher values of α mean lower values for ϵ, we ran our optimized algorithm on multiple levels of α to find an optimal value. We hypothesize that in lower values of α closer to 1, since ϵ increases, more aggressive pruning will occur sparsing the graph. However, in high values of α, since ϵ is inversely correlated, lower values of ϵ relax the pruning schema, result in a denser graph with still removing redundant edges. To find the optimum point, we have used the values of α shown in Table 6.2.

Table 6.2: Summary of Datasets Used in Evaluation.

α	1.25	2.5	5	7.5	10
ϵ	8	0.8	0.4	0.2667	0.2

The following `C++` function illustrates how the pruning strategy based on the $(1/\alpha)$-criterion is implemented in practice:

```
// ---------------------------------------------------------------------------
//  Greedy neighbour selection that enforces the
//  (1+epsilon)-spanner pruning rule
//  An edge (v,u) is kept unless there exists a witness w such that
//  dist(v,w) < (1/alpha) * dist(u,w)      with  alpha = 1 + 2/epsilon.
// ---------------------------------------------------------------------------
void HierarchicalNSW<dist_t>::getNeighborsByHeuristic2(
        std::priority_queue<
              std::pair<dist_t, tableint>,
                std::vector<std::pair<dist_t, tableint>>,
                CompareByFirst> &top_candidates,
        const size_t M)
{
    // keep everything if we have  M candidates
    if (top_candidates.size() <= M) return;

    //-----------------------------------------------------------------
    // 1. Re-heap into ascending order of distance to the query
    //-----------------------------------------------------------------
    std::priority_queue<std::pair<dist_t, tableint>> queue_closest; // min-heap
    while (!top_candidates.empty()) {
        queue_closest.emplace(
            -top_candidates.top().first,  // negate to get min-heap
             top_candidates.top().second);
        top_candidates.pop();
    }

    //-----------------------------------------------------------------
    // 2. Greedy pruning: enforce      d(v,w) < (1/alpha) * d(u,w)
    //-----------------------------------------------------------------
    std::vector<std::pair<dist_t, tableint>> return_list;
    while (!queue_closest.empty() && return_list.size() < M) {

        const auto current_pair = queue_closest.top();
        queue_closest.pop();

        const dist_t vu = -current_pair.first; // d(v,u)
        bool accept = true;

        for (const auto &chosen_pair : return_list) {
            const dist_t vw = -chosen_pair.first; // d(v,w)

            const dist_t uw = fstdistfunc_(
                    getDataByInternalId(chosen_pair.second), // w
                    getDataByInternalId(current_pair.second),// u
                    dist_func_param_);

            // theorem condition:    d(v,w) < (1/alpha) * d(u,w)
            if (vw < uw / alpha_) { // reject u
                accept = false;
                break;
            }
        }
        if (accept) return_list.emplace_back(current_pair);
    }

    //-----------------------------------------------------------------
    // 3. Push survivors back (restore max-heap convention of HNSW)
    //-----------------------------------------------------------------
    for (const auto &p : return_list) {
        top_candidates.emplace(-p.first, p.second); // re-negate
    }
}
```

6.4 Graphical Analysis of Distance-Based Robust Pruning

To better understand this optimization and the pruning process, we conducted a graphical analysis over our available sets of data. This illustrates how varying α

impacts the resulting stretch factor $(1 + \epsilon)$.

Figure 6.6 visualizes the effect for the *Glove-25-Angular* dataset.

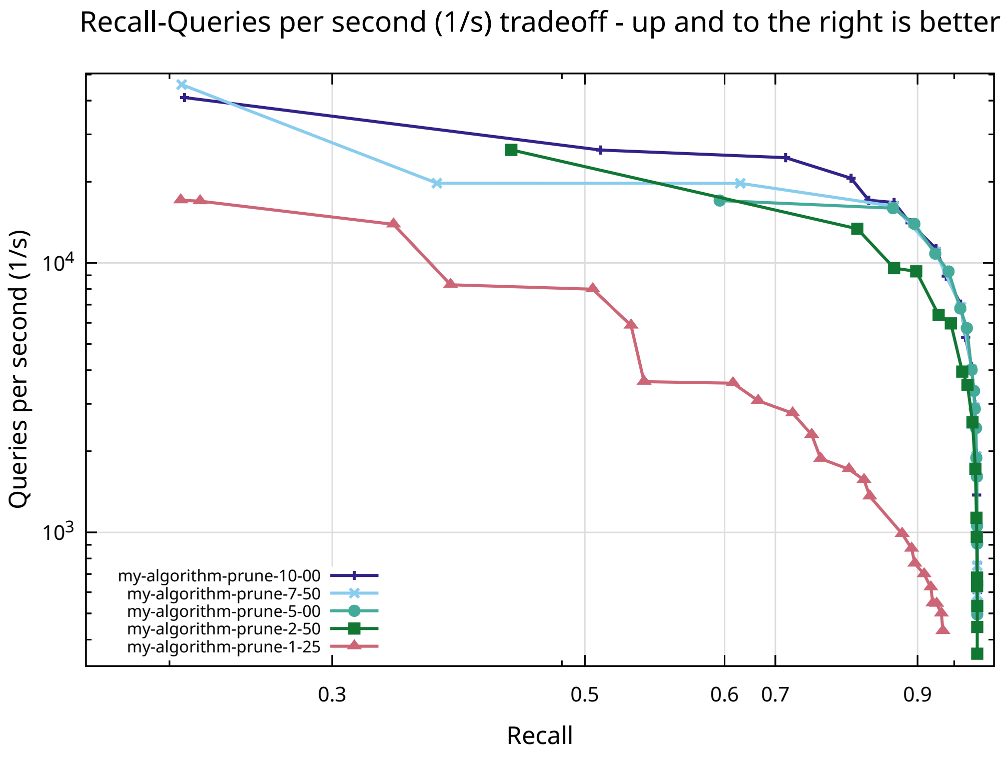

Figure 6.6: The Pareto frontier of varying values of α within *Glove-25-Angular* dataset with the same configuration.

We observe that smaller values of α, such as $\alpha = 1.25$, result in highly aggressive pruning. While this leads to very high query throughput, it severely compromises recall, making the search less reliable.

The curve corresponding to $\alpha = 1.25$ lies significantly below the others. This demonstrates a steep drop in recall even at moderate throughput levels. As α increases to moderate values around 2.5 and 5.0, the curves shift upward and rightward, indicating a much better balance between recall and speed. In particular, $\alpha = 2.5$ and $\alpha = 5.0$ maintain high recall levels (above 0.9) while sustaining high query rates. This suggests that these configurations provide a practical balance between pruning aggressiveness and distance preservation.

Further increasing α to 7.5 and 10.0 yields only marginal gains in recall, while slightly decreasing query throughput, indicating diminishing returns. Higher α values lead to less aggressive pruning, producing denser graphs that increase opperation cost without significant improvements in result quality.

To see the optimization effects of setting a large α, we will be using benchmarking the base algorithm we implemented and evaluated in the previous section of this chapter. The analysis can be shown in Figure 6.7.

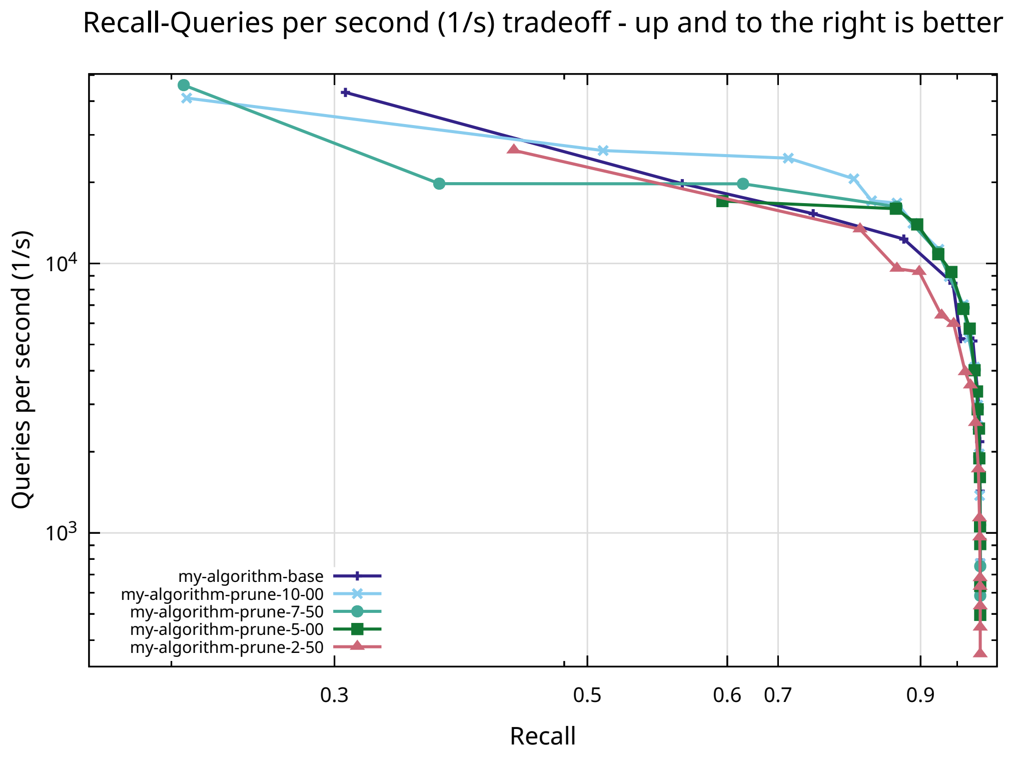

Figure 6.7: The Pareto frontier of varying values of α within *Glove-25-Angular* dataset with the base algorithm.

As shown in Figure 6.7, the optimization effect of increasing α becomes clear when compared to the base version of the algorithm ran on the same configuration. The base implementation, represented by the dark blue curve, maintains high recall across a wide range but at a moderate query throughput. Introducing pruning with larger α values, such as $\alpha = 10.0$, $\alpha = 7.5$, and $\alpha = 5.0$ results in substantial improvements while retaining comparable levels of recall to the base method. This demonstrates that pruning based on a higher α parameter can significantly accelerate search without major degradation in search quality.

Finally, benchmarking with other implementations of HNSW such as HNSW(faiss), HNSW(vespa) and hnswlib shows the effect of optimization even further within the same configuration and dataset. This can be shown as in Figure 6.8.

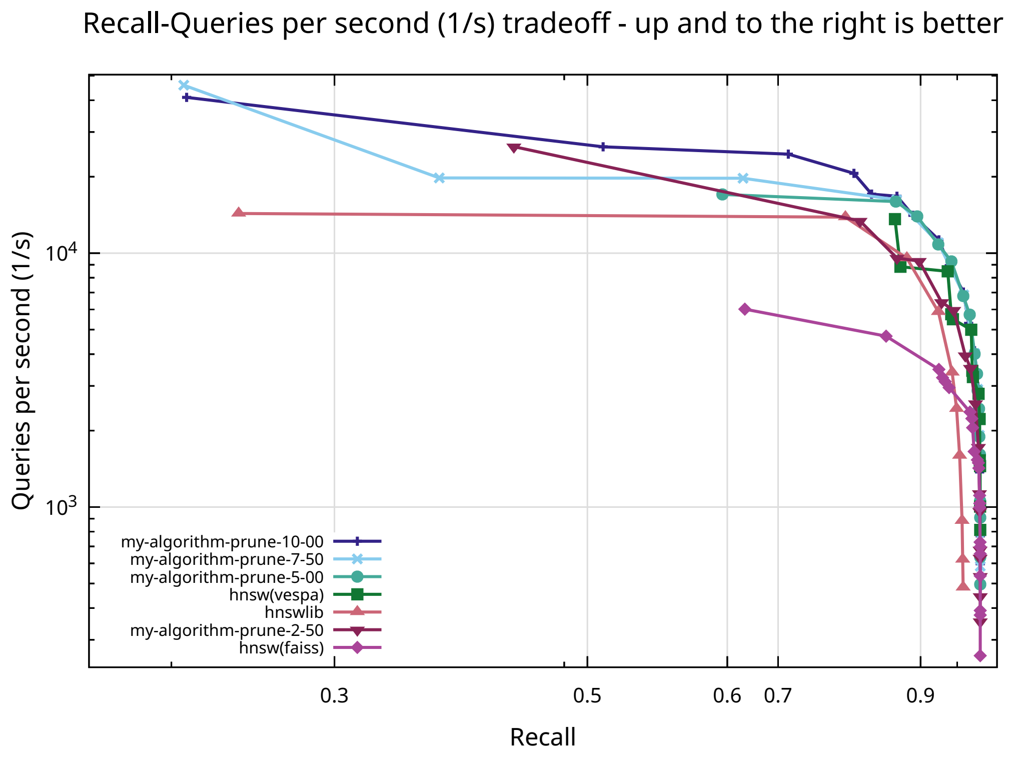

Figure 6.8: The Pareto frontier of HNSW benchmarks within *Glove-25-Angular* dataset with the same configuration algorithm.

Selecting $\alpha = 5.0$ case, due its effective pruning we can effectively show that this pruning methodology works across all benchmarks and beats the existing methods. The following figures in this section highlight the optimized behaviour of distance based robust pruning across all of number of dimensions and the distance metrics.

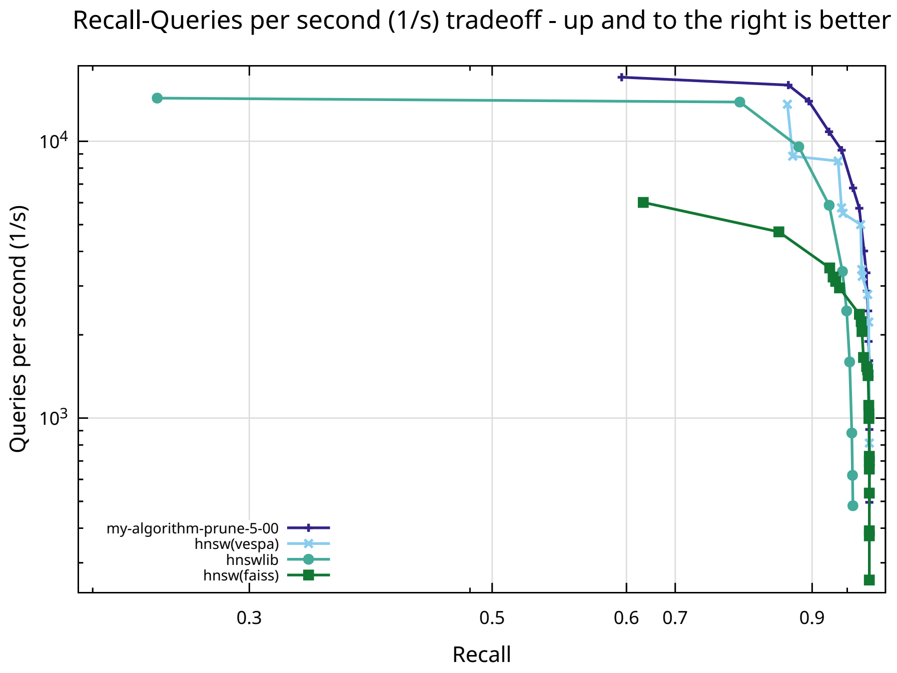

Figure 6.9: The Pareto frontier of HNSW benchmarks within *Glove-25-Angular* dataset with $\alpha = 5.0$.

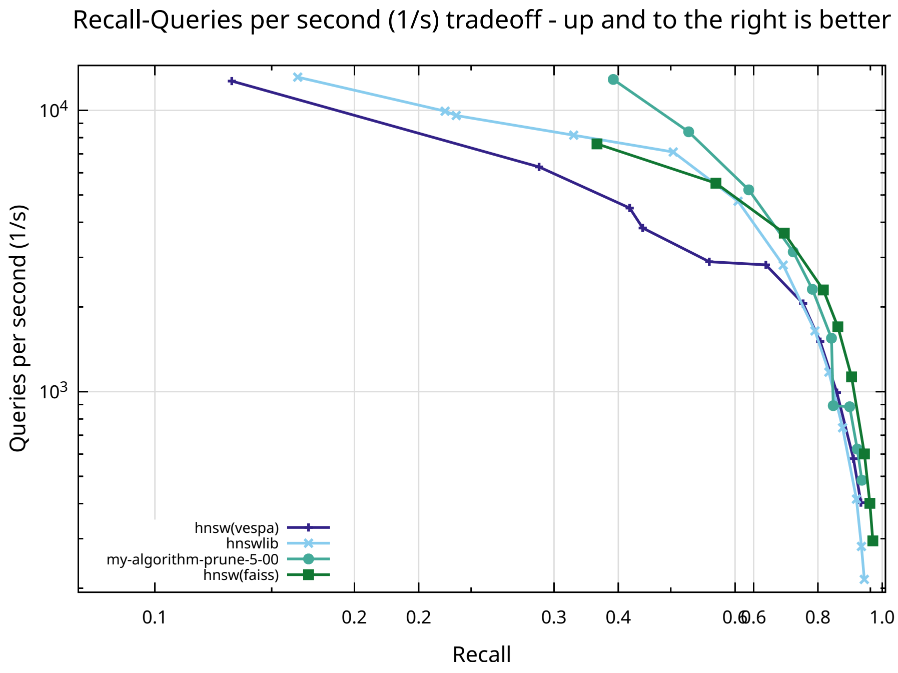

Figure 6.10: The Pareto frontier of HNSW benchmarks within *Glove-100-Angular* dataset with $\alpha = 5.0$.

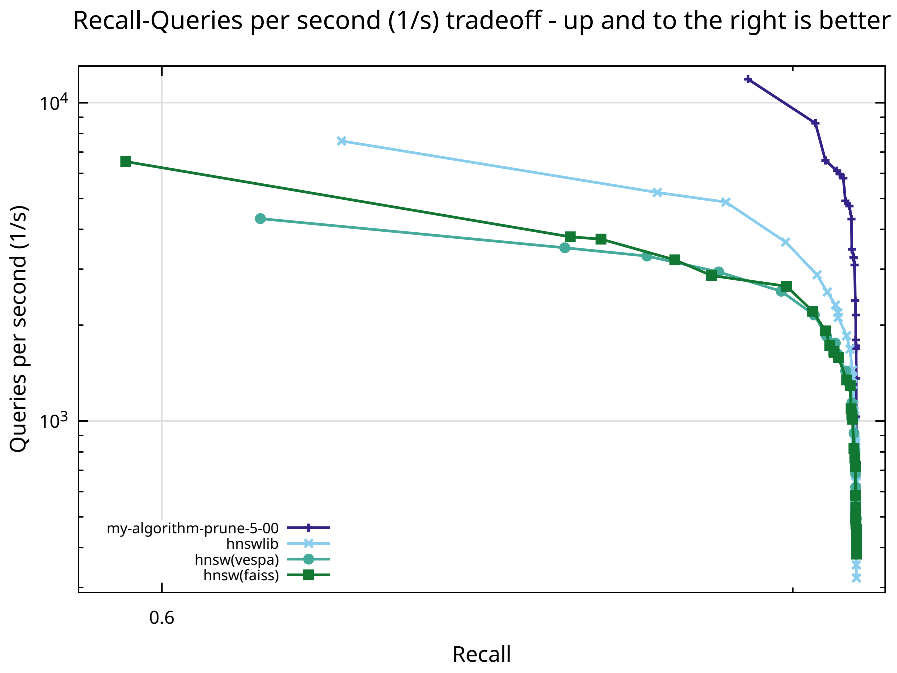

Figure 6.11: The Pareto frontier of HNSW benchmarks within *Fashion-MNIST-784-Euclidian* dataset with $\alpha = 5.0$.

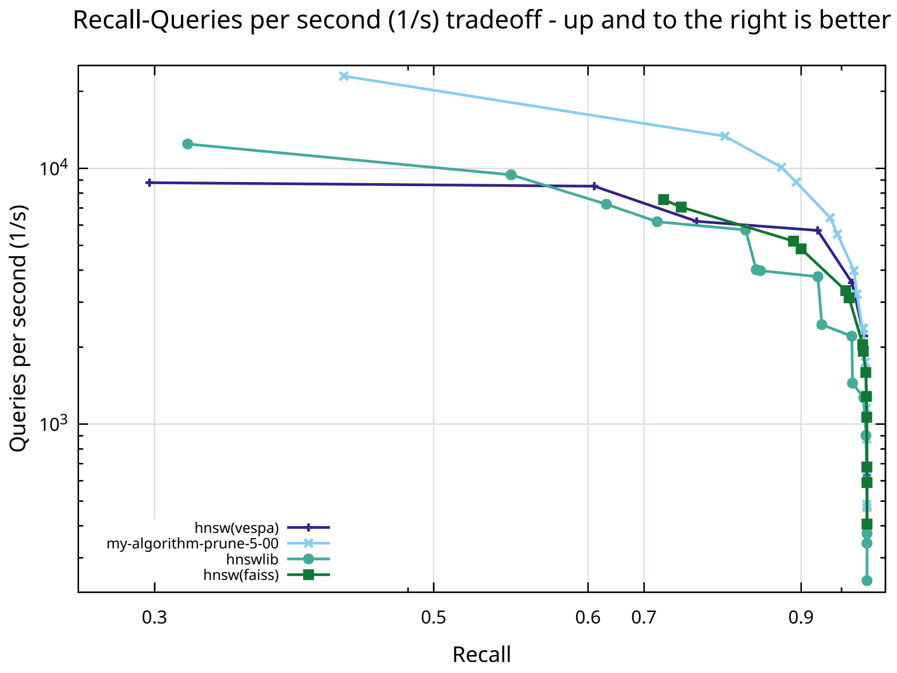

Figure 6.12: The Pareto frontier of HNSW benchmarks within *SIFT-128-Euclidian* dataset with $\alpha = 5.0$.

6.5 Empirical Effects of Layer-wise Dimension Reduction

In Chapter 4, we introduced a layer-wise dimension reduction scheme based on random projections according to the Johnson–Lindenstrauss lemma.

In this section, we evaluate the practical effects of applying this dimension reduction across the layers of the hierarchy. We compare the performance of the base algorithm without projection against the version incorporating layer-specific random projections.

The main question we seek to answer is in this evaluation is that whether the distortion introduced by the Johnson–Lindenstrauss transform (specifically with a distortion parameter ε) has any noticeable negative effect on the quality of nearest neighbor retrieval.

For the empirical evaluation of dimension reduction, we focus on the *Fashion-MNIST-784-Euclidian* dataset. Among the datasets considered, Fashion-MNIST-784-Euclidian has one of the highest input dimensions ($d = 784$), second only to the GIST-960-Euclidian dataset. However, running experiments on GIST-960-Euclidian proved infeasible on our hardware setup, which consisted of 16 virtual CPUs distributed per algorithm. As a result, Fashion-MNIST-784-Euclidian was selected as the next most appropriate alternative for dimension reduction experiments. It offers sufficiently high dimensionality of 784 to meaningfully assess the practical benefits of layer-wise random projection while remaining compatible with our computational constraints.

To systematically study the impact of the distortion introduced by the Johnson–Lindenstrauss projection, we conducted experiments using 3 ε values: specifically, $\varepsilon \in \{0.4, 0.6, 0.8\}$. These values were selected to cover a meaningful spectrum of distortion levels, from very conservative (small ε) to more aggressive compression (larger ε).

For each setting, a separate HNSW index was built and queried, allowing us to observe how the choice of ε affects search accuracy, query throughput, and computational cost. This enables a detailed understanding of the trade-offs between precision and efficiency when applying random projections in hierarchical approximate nearest neighbor search.

Finally, the following `C++` code demonstrates how the projections are initialized in our HNSW engine with the Johnson-Lindenstrauss dimension reduction lemma in mind using a Mersenne Twister random generator for the projection matrix.

```cpp
    // Initialize projection matrices for JL dimension reduction
template <typename dist_t>
void HierarchicalNSW<dist_t>::initProjections(size_t max_level,
                                              float target_eps /* =0.10f */,
                                              size_t min_dim   /* =32    */) {
    level0_dim_ = data_size_ / sizeof(float);  // store the original d

    proj_dim_.assign(max_level + 1, level0_dim_); // default=no compression
    proj_mat_.resize(max_level + 1);              // empty for uncompressed
    proj_data_.resize(max_level + 1);

    std::mt19937 gen(42);
    std::normal_distribution<float> g(0.0f, 1.0f);

    for (size_t l = 1; l <= max_level; ++l) {
        /* rough layer size   n_l  N / 2^l */
        size_t n_l = std::max<size_t>(1, max_elements_ >> l);

        /* theoretical JL dimension */
        double jl_dim = 8.0 / (target_eps * target_eps) *
                        std::log(static_cast<double>(n_l));
        size_t k_l    = static_cast<size_t>(std::ceil(jl_dim));

        /* hard caps:
         *   • never go below min_dim (accuracy safeguard)
         *   • never go above original d (makes no sense)                */
        if (k_l >= level0_dim_ || k_l < min_dim) {
            proj_dim_[l] = level0_dim_;   // mark as "no projection"
            continue;
        }

        proj_dim_[l] = k_l;               // we *do* project this layer

        /* sample R^(l)  | rows are normal(0,1/k) */
        std::vector<float> R(k_l * level0_dim_);
        float scale = 1.0f / std::sqrt(static_cast<float>(k_l));
        for (float& c : R) c = g(gen) * scale;
        proj_mat_[l].swap(R);
    }
}
```

Our experiments show that applying dimension reduction yields slight efficiency gains with negligible degradation in recall. In the following subsections, we present detailed benchmarking results to quantify these effects and to validate the theoretical advantages discussed previously.

6.6 Graphical Analysis for Layer-wise Dimension Reduction

In this section, across the three parameters we have chosen, we expect to see that $\varepsilon = 0.6$ outperforms the other two values. The main reasoning is that at $\varepsilon = 0.4$ level, due to the $k \geq \lceil 8\,\varepsilon^{-2} \ln m \rceil$, requirement bound for dimension reduction, and due to the logarithmic decrease of number of nodes per each higher level, the bound $k \geq \lceil 8\,\varepsilon^{-2} \ln m \rceil$, will not be sufficient enough for $\varepsilon = 0.4$ to drastically reduce dimensions. In the $\varepsilon = 0.8$ case the opposite effect will be experienced, where many of the 784 dimensions will be reduced, therefore it will be harder to navigate the reduced graph. Therefore we propose that $\varepsilon = 0.6$ value will be the optimal value for reduction in 6000 vectors each with 784 dimensions.

The following Pareto Frontier on Figure 6.13 done on Fashion-MNIST-784-Euclidian supports this claim.

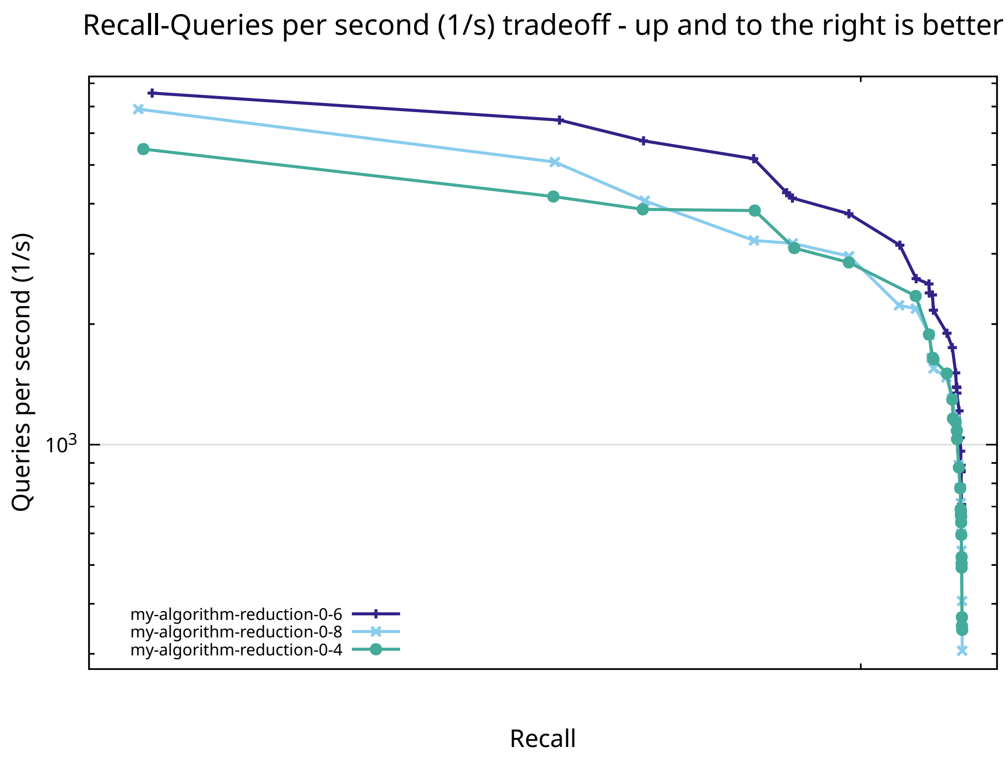

Figure 6.13: The Pareto frontier of varying values of ε within *Fashion-MNIST-784-Euclidian* dataset for same configurations.

Figure6.13 shows the impact of varying the JL projection distortion parameter ε on the search speed and recall trade-off in the FASHION-MNIST-784-EUCLIDIAN dataset.

As expected, allowing a larger distortion ($\varepsilon = 0.6$) results in much faster query processing across almost the entire recall range, compared to other distortions ($\varepsilon = 0.8$ and $\varepsilon = 0.4$). As expected, $\varepsilon = 0.8$ performed better than $\varepsilon = 0.4$ in low-recall high qps regimes due to the stronger and more aggressive dimension reduction. This is because higher ε values lead to stronger dimensionality reduction, making each distance computation cheaper. This fact is also the reason why $\varepsilon = 0.4$ started to perform better than $\varepsilon = 0.8$ as the Pareto frontier approached higher recall and lower qps regimes. The reason is, the speed-up of $\varepsilon = 0.8$ comes at a cost which is the quality of the returned vectors. Overall the $\varepsilon = 0.6$ value becomes an optimal choice for dimension reduction using JL projection.

To finalize this optimization, we have to evaluate other benchmark HNSW implementations without the dimension reduction. For the graphical analysis we select the best candidate $\varepsilon = 0.6$. Figure6.14 illustrates how the dimension reduced algorithm compare with other state of the art HNSW algorithms on the same benchmark.

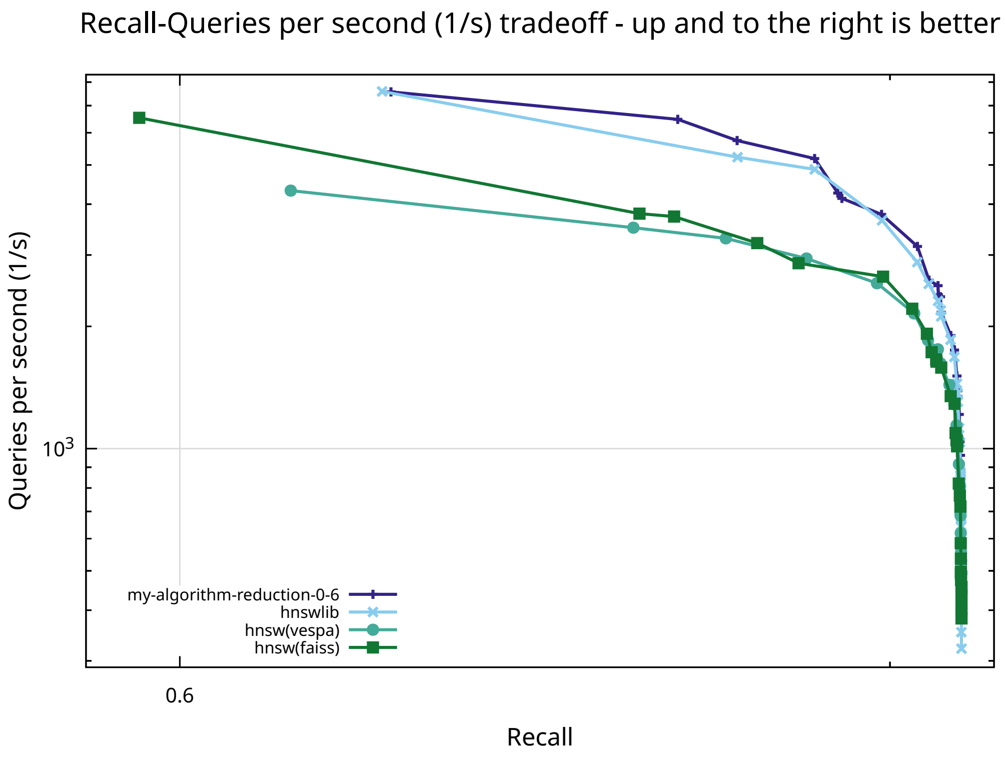

Figure 6.14: The Pareto frontier of state of the art HNSW benchmarks for *Fashion-MNIST-784-Euclidian* dataset for same configurations.

Finally, Figure6.14 shows that the optimization of reduction using the JL Lemma improves our algorithm on medium recall and high qps regumes. However due to the dimensionality reduction, and loss of information across layers, as we approach to higher recall regimes the algorithm struggles to beat the benchmarks. Therefore, we can conclude that for medium recall and high qps requirements and criteria, optimization via dimension reduction is a great alternative and enhancement to the HNSW implementation.

Chapter 7
Conclusions and Future Work

In this thesis, we have analyzed the theoretical foundations of the Hierarchical Navigable Small World (HNSW) algorithm. We then proposed targeted optimizations to improve its performance, and validated these enhancements through rigorous empirical evaluation. Our key contributions include a detailed complexity analysis of the standard HNSW structure, the introduction of distance-based robust pruning inspired by spanner theory to reduce memory and search time, and layer wise dimension reduction through the JL Lemma.

Looking forward, there are several promising directions to build upon this work. One natural extension would be to develop adaptive pruning strategies, where the parameter $\mathrm{ef}_{\mathrm{search}}$is dynamically adjusted based on local graph density or query requirements.

Another possible optimization lies in combining an NSG inspired diversity-based neighbor selection with learning-based techniques that predict optimal connectivity patterns based on data distribution characteristics. Additionally, exploring the compressed indexing for memory-constrained environments could also yield substantial gains for the literature, especially for billion-scale vector datasets.

Moreover, we believe this thesis can serve as a foundation for others interested in pushing HNSW and graph-based approximate nearest neighbor search even further. Researchers could investigate the theoretical limits of graph sparsification. This would be especially useful in dynamic settings, where nodes are continuously inserted or deleted. Developers might adapt the proposed optimizations to specialized hardware platforms, such as GPUs. This way parallelism based optimization also become critical.

In closing, the results of this thesis demonstrate that optimization of graph construction and neighbor selection strategies can meaningfully advance both the theoretical guarantees and the practical performance of approximate nearest neighbor search systems. We hope that these contributions not only strengthen the understanding of HNSW but also inspire continued innovation in the broader field of approximate nearest neighbor search.

Appendix A
Code

The code for this Junior Independent Work has been uploaded to the following GitHub repository: https://github.com/ygzdvr/Junior-Independent-Work

Bibliography

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, page 459–468, USA, 2006. IEEE Computer Society.

[2] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal lsh for angular distance, 2015.

[3] M. Aumüller, E. Bernhardsson, and A. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, 2018.

[4] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, Sept. 1991.

[5] F. Aurenhammer, R. Klein, and D.-T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, Singapore, 2013.

[6] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(6):1247–1260, June 2015.

[7] G. Ballard, T. G. Kolda, A. Pinar, and C. Seshadhri. Diamond sampling for approximate maximum all-pairs dot-product (mad) search. In *2015 IEEE International Conference on Data Mining*, page 11–20. IEEE, Nov. 2015.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. volume 18, page 509–517, New York, NY, USA, Sept. 1975. Association for Computing Machinery.

[9] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *International conference on database theory*, pages 217–235, 1999.

[10] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 97–104, New York, NY, USA, 2006. Association for Computing Machinery.

[11] S. Boucheron, G. Lugosi, and P. Massart. *Concentration Inequalities: A Nonasymptotic Theory of Independence.* Oxford University Press, 2013.

[12] S. Bruch. *Foundations of Vector Retrieval.* Springer Nature Switzerland, 2024.

[13] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 380–388, New York, NY, USA, 2002. Association for Computing Machinery.

[14] E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 225–234, New York, NY, USA, 2007. Association for Computing Machinery.

[15] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 537–546, New York, NY, USA, 2008. Association for Computing Machinery.

[16] B. Delaunay. Sur la sphère vide. *Bulletin de l'Académie des Sciences de l'URSS, Classe des sciences mathématiques et naturelles*, 6:793–800, 1934.

[17] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. The faiss library, 2025.

[18] C. Foster and B. Kimia. Computational enhancements of hnsw targeted to very large datasets. In *International Conference on Similarity Search and Applications (SISAP)*, 2023.

[19] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph, 2018.

[20] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, page 2946–2953, USA, 2013. IEEE Computer Society.

[21] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. Accelerating large-scale inference with anisotropic vector quantization, 2020.

[22] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 1312–1317. AAAI Press, 2011.

[23] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *Theory of computing*, pages 321–350, 2012.

[24] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[25] P. Indyk. Approximate proximity problems in high dimensions via locality-sensitive hashing. Lecture notes, MIT Department of Electrical Engineering and Computer Science, 2007. Course slides.

[26] M. Iwasaki. Neighborhood graph and tree for indexing high-dimensional data. `https://github.com/yahoojapan/NGT`, 2015. Yahoo Japan Corporation, Retrieved August 22, 2020.

[27] M. Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications (SISAP)*, pages 20–33. Springer, 2016.

[28] J. Jaromczyk and G. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, 1992.

[29] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[30] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[31] W. B. Johnson and J. Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[32] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[33] J. Kleinberg. The small-world phenomenon: an algorithmic perspective. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 163–170, New York, NY, USA, 2000. Association for Computing Machinery.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition, 1998.

[35] W. Li, Y. Zhang, Y. Sun, W. Wang, and X. Lin. Spann: Highly-efficient billion-scale approximate nearest neighbor search. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2021.

[36] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement (v1.0), 2016.

[37] R. Liu, T. Wu, and B. Mozafari. A bandit approach to maximum inner product search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4376–4383, Jul. 2019.

[38] T. Liu, A. Moore, K. Yang, and A. Gray. An investigation of practical approximate nearest neighbor algorithms. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004.

[39] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[40] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd*

International Conference on Very Large Data Bases, VLDB '07, page 950–961. VLDB Endowment, 2007.

[41] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[42] D. Newman. Bag of words [dataset]. UCI Machine Learning Repository, 2008.

[43] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001.

[44] J. Pennington, R. Socher, and C. Manning. GloVe: Global vectors for word representation. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[45] L. Prokhorenkova and A. Shekhovtsov. Graph-based nearest neighbor search: from practice to theory. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[46] A. Shrivastava and P. Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips), 2014.

[47] M. Tiwari, R. Kang, J.-Y. Lee, D. Lee, C. Piech, S. Thrun, I. Shomorony, and M. J. Zhang. Faster maximum inner product search in high dimensions, 2023.

[48] R. Vershynin. *High-Dimensional Probability: An Introduction with Applications in Data Science.* Cambridge University Press, 2018.

[49] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. première partie: Sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die Reine und Angewandte Mathematik*, 133:97–178, 1908.

[50] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256, Sept. 1977.

[51] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[52] W. Xiao, Y. Zhan, R. Xi, M. Hou, and J. Liao. Enhancing hnsw index for real-time updates: Addressing unreachable points and performance degradation, 2024.

[53] X. Zhao, Y. Tian, K. Huang, B. Zheng, and X. Zhou. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proc. VLDB Endow.*, 16(8):1979–1991, Apr. 2023.