Nearest Neighbor Search (NNS): Finding the Closest Data Points in Feature Space

Md Selim Ahmed
Department of Computer Science and Engineering
University of Liberal Arts Bangladesh
Dhaka, Bangladesh
selim.ahmed.cse@ulab.edu.bd

***Abstract*—Nearest neighbor search (NNS) retrieves the data points in a dataset that are closest to a given query point under a specified distance metric. This problem underlies many applications in machine learning and information retrieval, such as classification/regression (k-NN classifiers vote on labels of nearby points), recommendation systems (finding similar users or items), and content-based retrieval (e.g., image or audio search). Exact NNS (brute-force search) guarantees optimal results by comparing the query to every point (time complexity O(n) per query). However, brute-force is impractical for large or high-dimensional datasets. Many data structures and algorithms have been developed to trade off query speed and accuracy. Spatial trees like k-d trees and ball trees recursively partition space to prune search regions. These can achieve average-case O (log n) queries in low dimensions, but degrade in high dimensions (the "curse of dimensionality"). Locality-Sensitive Hashing (LSH) projects points into hash buckets so that nearby points collide with higher probability, achieving sublinear average query time at the cost of approximate results. More recently, graph-based methods (e.g., Navigable Small World (NSW) graphs and Hierarchical NSW (HNSW)) build a proximity graph linking each point to its near neighbors, allowing very fast greedy traversal search. In HNSW, multiple layered graphs enable near-logarithmic search time with high recall. In practice, modern libraries (e.g., FAISS) further accelerate search using hardware (GPUs) and quantization. Key challenges for NNS include high dimensionality, dynamic (streaming) data, and balancing recall vs. latency. This paper surveys NNS methods, discusses their trade-offs, and highlights recent developments such as hybrid exact-approximate models and GPU acceleration. Future directions include dynamic indexing and integration with AI systems.**

I. Introduction

Nearest neighbor search is the problem of finding the closest points in a dataset to a query point under some metric (e.g. Euclidean distance)[1][3]. Formally, given a dataset of n points in d dimensions and a query, NNS returns the point(s) minimizing distance to the query. This simple paradigm underlies many tasks in machine learning and data analysis. For example, in k-NN classification, a query's label is predicted by majority vote among its k nearest neighbors[2]. In recommendation systems, one finds similar users or items to make suggestions; Amazon's item-to-item collaborative filtering is a classic example[2]. In content retrieval, feature descriptors for images or audio are compared to find matches.

Brute-force search simply computes all distances from the query to every data point and picks the smallest. It is exact but scales as $O(n)$ per query[3]. There is no preprocessing besides storing the points, and it always finds the true nearest neighbors, but it becomes too slow for large n or high dimension d.

Exact search can be accelerated by spatial tree structures. The classic k-d tree (Bentley 1975) recursively partitions space along coordinate hyperplanes[11]. Points are stored at nodes, and splits are chosen (e.g. median) to balance the tree. To answer a query, the tree is traversed and a priority queue of tree nodes (by a lower-bound distance to the query) is used to prune regions that cannot contain closer neighbors[11]. This yields roughly $O(\log n)$ average query time in low dimensions, but most points lie near partition boundaries in high d, so pruning fails ("curse of dimensionality")[6][14]. Ball trees (nested hyperspheres) generalize this idea by enclosing points in balls (center+radius) and building a hierarchy of sub-balls[14]. Ball trees often outperform k-d trees for moderate d (since balls can adapt better to data geometry)[7]. However, like k-d trees, ball-tree search still degrades as d grows, and in very large dimensions both structures tend to check most points[5][14].

Because exact trees falter in high d, approximate methods have become popular. Locality-Sensitive Hashing (LSH) projects points into randomized hash buckets so that nearby points have higher collision probability[4]. A query is hashed, and only points in the matching buckets are examined. This gives sublinear average query time (often $O(n^\rho)$ for some $\rho < 1$) but is approximate: some true neighbors may be missed[4]. LSH performance depends on good hash design and often uses multiple hash tables. Variants such as multi-probe LSH reduce the needed tables by checking nearby buckets. LSH scales better to high dimension than trees, but still requires tuning of hash parameters and extra memory for multiple tables.

Graph-based approaches build a proximity graph on the dataset, where each point is linked to some of its neighbors. In a simple k-nearest-neighbor graph (k-NNG), each point is connected to its k nearest points. The Navigable Small World (NSW) graph enriches this by also adding random long-range edges to create a small-world network[5]. Then greedy search from any start point will quickly reach the region of the

query's neighbors. Hierarchical NSW (HNSW) adds multiple graph layers: new points are inserted in a random "top" layer with long-range links, and also in denser lower layers for local refinement[5]. Search begins at a random high-level node and greedily descends layer by layer[5]. Because the top layer has long jumps, the search quickly narrows to the vicinity of the query, and lower layers then fine-tune to find the nearest neighbors. HNSW has been shown to achieve very high recall (often $> 90\%$) with very low query latency in practice[5].

Modern implementations leverage optimized libraries and hardware. Facebook's FAISS library implements many NNS indexes (inverted files, quantization, HNSW) with GPU support[19]. FAISS can build billion-point indices using GPU parallelism; for example, a 95-million image k-NN graph can be constructed in 35 minutes on GPU[20]. Microsoft's DiskANN and others provide memory-efficient graph indices for very large data (billion-scale) on SSDs. Emerging work focuses on benchmarking these tools and deploying them on edge devices or in real-time systems[10].

Despite many options, NNS methods face key challenges. In very high dimensions (d large), distances can concentrate (points become almost equidistant)[6], making "nearest" neighbors less meaningful. Exact data structures struggle with both large n and large d. Approximate methods require careful tuning to balance accuracy (recall) versus speed[13]. Dynamic or streaming data introduce further issues: tree or graph structures must be updated efficiently for inserts/deletes[1]. This survey examines the above methods and concepts, highlighting their trade-offs and open problems. The paper is organized as follows: Section II reviews related work on exact and ANN methods; Section III describes graph-based NNS and presents an illustrative example; Section IV compares performance trade-offs; Section V discusses advantages and challenges; Section VI notes limitations; Section VII outlines future research; and Section VIII concludes.

II. Related Work

Brute-Force Search: The simplest approach is to compute the distance from the query to all dataset points and select the top-k[4]. This guarantees the exact neighbors but takes $O(n)$ time per query[4]. No preprocessing is required, and memory overhead is minimal (just storing the data). However, as n or d grows, brute-force quickly becomes too slow for real-time or low-power applications. Even with GPUs, brute-force on millions of points can be too expensive without further tricks (e.g. quantization).

Spatial Trees: To accelerate exact NNS, spatial tree structures are used. The k-d tree recursively partitions space by axis-aligned splits[5]. Each node contains a subset of points, and splits along dimensions (often at the median) to balance partitions[5]. Searching involves traversing the tree: child nodes that could contain points closer than the current best are explored using a priority queue of nodes sorted by their minimum distance to the query. In low dimensions ($d \lesssim 10$), k-d trees can answer queries in roughly $O(\log n)$ on average[5]. However, in high dimensions many points lie near the boundaries of partitions, so pruning fails and most nodes must be visited[5]. This is a manifestation of the "curse of dimensionality."

Ball trees partition points into nested hyperspherical clusters[5][14]. Each node encloses its points in a ball (defined by a center and radius) and recursively splits points into sub-balls. Queries use the distances to ball surfaces to prune branches. Ball trees can outperform k-d trees in moderate dimensions because balls can adapt flexibly to clusters[7]. Some surveys note that ball trees "can perform well in high-dimensional spaces" by avoiding the sharp boundaries of k-d splits[7]. Still, ball-tree search also degrades with very large d or extremely large n, and often falls back to examining most points in the worst case[5][14]. Other metric trees (e.g. vp-trees, cover trees) share similar behavior: efficient for low-dimensional or low-intrinsic-dimensional data, but not robust to curse-of-dimensionality.

Locality-Sensitive Hashing (LSH): LSH aims for sublinear search by hashing points so that close points collide with higher probability[8]. A common LSH scheme for Euclidean distance uses random projections: $h_{\mathbf{a},b}(x) = \lfloor (\mathbf{a} \cdot x + b)/w \rfloor$, where $\mathbf{a}$ is a random normal vector and b a shift[8]. Nearby vectors are likely to hash to the same bin. A query's hashes retrieve only candidates from the same buckets, dramatically reducing comparisons. On average, LSH can achieve sublinear query time[8], but at the cost of approximate results (some true neighbors may be missed) and extra memory for multiple tables[8]. Variants include multi-probe LSH (probing nearby buckets) and cross-polytope LSH. While LSH scales better to high d than tree methods, its performance depends on tuning (number of hash tables, bin width, etc.), and finding good hashes for complex data can be nontrivial. Some recent LSH works (e.g. those building on p-stable distributions[19]) improve hash quality, but overall LSH trades recall for speed and is often used when exact neighbors are not strictly required.

Graph-Based Methods: In recent years, proximity graphs have become a dominant ANN approach. The idea is to build a k-NN graph where each data point is connected to its k nearest neighbors (plus possibly some random links). Navigable Small World (NSW) graphs extend this by adding random long-range links so the graph has a small-world property[9]. Greedy search from any node tends to find a path to the query's region quickly.

Hierarchical NSW (HNSW) improves this by using multiple layers[9]. When inserting a new point, it is added to a randomly chosen top layer (connecting to a few far-away nodes) and also inserted in denser lower layers (connecting to close neighbors)[9]. Search starts at a random entry in the top layer, greedily moves to closer neighbors (using only long-range edges at first), then drops down to the next layer and repeats, refining the search. Because high layers "zoom out"

and lower layers "zoom in," HNSW often achieves sublinear (near-logarithmic) search time while retrieving nearly the true nearest neighbors[9][15]. Studies have shown HNSW yielding $> 90\%$ recall in milliseconds for large datasets[15].

Library and GPU Support: Several optimized libraries implement these methods. FAISS (Facebook AI Similarity Search) offers many indexes (IVF, PQ quantization, HNSW, etc.) with both CPU and GPU acceleration[10]. It can index tens of millions of high-dimensional vectors; for example, constructing a 95M-image k-NN graph takes only 35 minutes on a GPU[17]. Other tools like Annoy (Spotify) and NMSLIB provide high-performance C++ implementations of tree and graph methods. Recent work also integrates graph search with hardware (e.g. Fast ANN on GPU[17]) and investigates resource-constrained environments (edge devices)[6][20].

In summary, the literature offers many exact and approximate NNS solutions, each with trade-offs. We next illustrate NNS using a simple graph-based example, then compare methods and discuss trade-offs.

III. Methodology

Graph-Based NNS Example: To illustrate NNS, we use a small two-dimensional example. Consider 20 data points labeled A–T, each with (x, y) coordinates as shown in Table I below. We choose a neighborhood parameter $k = 0.4$, meaning each point will connect to $0.4 \times 20 = 8$ nearest neighbors. We first form a complete distance matrix by computing all pairwise Euclidean distances: $d_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2$[21].

To better understand the overall process of nearest neighbor search (NNS), we first present a high-level workflow that outlines the main steps from dataset preparation to retrieval of nearest neighbors. This workflow provides context before we illustrate a specific graph-based example.

As shown in Figure 1, the NNS process starts with the input dataset, followed by feature extraction and index construction, which can be implemented using tree, hash, or graph-based structures. A query point is then compared against the dataset using a similarity measure (e.g., Euclidean or Cosine distance), candidate neighbors are identified through exact or approximate search, and finally, the nearest neighbor(s) are returned. This conceptual workflow sets the stage for our concrete example using a small 2D dataset of 20 points (A–T), where we construct a k-NN graph and demonstrate greedy search for nearest neighbors.

Table I lists the (x, y) coordinates of the 20 sample points. Next, in Step 2, we compute the full pairwise distance matrix. (In practice, one can compute distances on the fly, but for illustration we precompute them.) The matrix entries for points A–T are shown in Tables II and III (distance from point in row to point in column). From each row, we identify the 8 smallest nonzero distances – these correspond to the 8 nearest neighbors of that point.

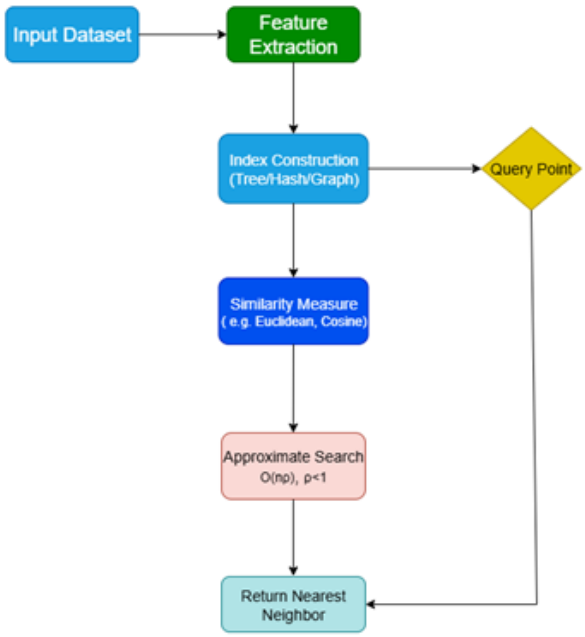

Fig. 1: Workflow of Nearest Neighbor Search

TABLE I: Node Coordinates (x, y)

Node	x	y
A	1	2
B	2	3
C	3	1
D	4	2
E	5	3
F	6	1
G	7	2
H	8	3
I	9	1
J	10	2
K	11	3
L	12	1
M	13	2
N	14	3
O	15	1
P	16	2
Q	17	3
R	18	1
S	19	2
T	20	3

TABLE II: Distance Matrix (Columns A to J)

	A	B	C	D	E	F	G	H	I	J
A	0.00	1.41	2.24	3.00	4.12	5.10	6.08	7.21	8.06	9.06
B	1.41	0.00	2.24	2.24	3.00	4.12	5.00	6.08	6.71	7.62
C	2.24	2.24	0.00	1.41	2.24	3.00	4.00	5.10	6.08	7.21
D	3.00	2.24	1.41	0.00	1.41	2.24	3.00	4.12	5.10	6.08
E	4.12	3.00	2.24	1.41	0.00	1.41	2.24	3.00	4.12	5.00
F	5.10	4.12	3.00	2.24	1.41	0.00	1.41	2.24	3.00	4.12
G	6.08	5.00	4.00	3.00	2.24	1.41	0.00	1.41	2.24	3.00
H	7.21	6.08	5.10	4.12	3.00	2.24	1.41	0.00	1.41	2.24
I	8.06	6.71	6.08	5.10	4.12	3.00	2.24	1.41	0.00	1.41
J	9.06	7.62	7.21	6.08	5.00	4.12	3.00	2.24	1.41	0.00

TABLE III: Distance Matrix (Columns K to T)

	K	L	M	N	O	P	Q	R	S	T
K	0.00	1.41	2.24	3.00	4.12	5.10	6.08	7.21	8.25	9.22
L	1.41	0.00	1.41	2.24	3.00	4.12	5.10	6.08	7.21	8.25
M	2.24	1.41	0.00	1.41	2.24	3.00	4.12	5.10	6.08	7.21
N	3.00	2.24	1.41	0.00	1.41	2.24	3.00	4.12	5.10	6.08
O	4.12	3.00	2.24	1.41	0.00	1.41	2.24	3.00	4.12	5.10
P	5.10	4.12	3.00	2.24	1.41	0.00	1.41	2.24	3.00	4.12
Q	6.08	5.10	4.12	3.00	2.24	1.41	0.00	1.41	2.24	3.00
R	7.21	6.08	5.10	4.12	3.00	2.24	1.41	0.00	1.41	2.24
S	8.25	7.21	6.08	5.10	4.12	3.00	2.24	1.41	0.00	1.41
T	9.22	8.25	7.21	6.08	5.10	4.12	3.00	2.24	1.41	0.00

We then construct a k-NN graph by connecting each node to its k nearest neighbors. For example, node A's 8 nearest neighbors (in order of distance) are B, C, D, E, F, G, H, I[22]. We create an undirected edge from A to each of these points. Repeating this process for all nodes yields a sparse graph where each node has degree 8. This graph can be used for nearest-neighbor search: given a query point, one could insert it into the graph or begin search from some entry point.

Figure 2 illustrates greedy graph search: starting at an entry node, the algorithm repeatedly moves to the adjacent neighbor that is closest to the query, until no neighbor is closer[23]. This finds a local optimum on the graph; hierarchical layers (as in HNSW) help avoid local traps by jumping to farther nodes in upper layers first.

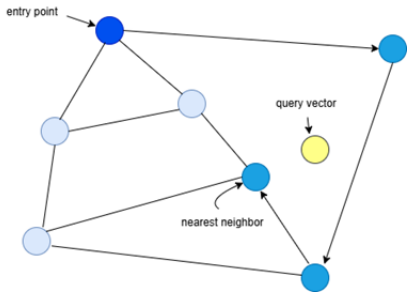

Fig. 2: Greedy search in a small-world proximity graph

Figure 2: Greedy search in a small-world proximity graph. Starting at an entry point (top blue), the algorithm iteratively moves to the adjacent node closest to the query (yellow) until no closer neighbor is found

A. Results of the Example

After constructing the 8-NN graph for our 20-point example, we list each node's neighbors (ordered by distance) in Table IV. For instance, Node A's neighbors are B (1.41), C (2.24), D (3.00), …, I (8.06). The resulting graph is shown (conceptually) in Figure 2: each node is connected by edges to its 8 nearest neighbors. This sparse k-NN graph is similar to the proximity graphs used in algorithms like HNSW[11][24].

To query this graph, a greedy search would start from a random or high-degree node and move closer to the query until convergence. In practice, exact graph search (using the true k-NN graph) yields exact neighbors, but requires $O(n^2)$ preprocessing to build the full graph. Approximate methods (like NSW/HNSW) approximate the k-NN graph to save time: they may add only a few random long edges (as in NSW) or prune edges to reduce degree, trading some accuracy for speed[17]. In general, graph-based methods achieve extremely high recall and fast query times in practice, even in moderate to high dimensions[15][11].

TABLE IV: Comparison of Nearest Neighbor Search Methods

Method	**Type**	**Query Complexity**	**Comments (Advantages/Disadvantages)**
Brute-Force	Exact	O(n) per query [3]	Guaranteed exact; extremely slow for large n; no extra memory needed.
KD-Tree	Exact (spatial tree)	Avg O(log n) (low d); Worst-case O(n) [12]	Efficient for small d; simple implementation; degrades badly in high d [12].
Ball-Tree	Exact (metric tree)	Similar to KD-tree; not strongly better in very high d [8]	Better at moderate d than KD; handles clusters; still affected by curse of dimensionality [8].
LSH (hashing)	Approximate	Sub-linear (depends on number of tables) [4]	Fast multi-probe queries; recall trade-off; inserts/deletes easy [17]. Scales better to high d than trees.
Graph (NSW/HNSW)	Approximate (small-world)	~O(log n) per query [5] (empirical)	Extremely high recall and speed in practice; handles high-dimensional data; small heuristic parameters. Construction may be costly; insertion and dynamic updates complex.

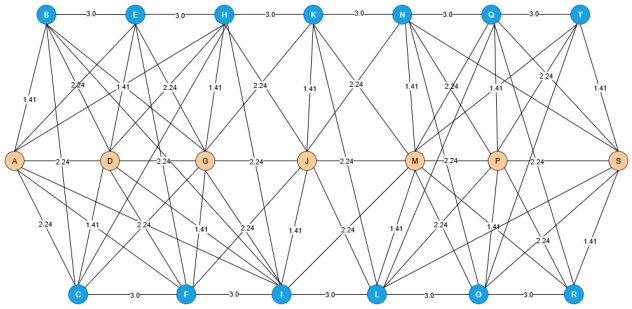

Fig. 3: Graph representation of 20 nodes with edges showing the 8-nearest neighbors per node.

IV. Results and Trade-offs

We summarize typical performance characteristics of NNS methods in Table V. (Exact numbers depend on data distribution and implementation.)

TABLE V: Nearest Neighbors Ordered by Distance

Node	**Nearest Neighbors (Ordered by Distance)**
A	B (1.41), C (2.24), D (3.00), E (4.12), F (5.10), G (6.08), H (7.21), I (8.06)
C	D (1.41), A (2.24), B (2.24), E (2.24), F (3.00), G (4.00), H (5.10), I (6.08)

V-A: Experimental Example (Graph Results for k = 0.4)

The nearest neighbor connections form a sparse k-NN graph, a structure widely used in ANN research [7], [9].

TABLE VI: Nearest Neighbors of Nodes A–T (Ordered by Distance)

Node	Nearest Neighbors (Ordered by Distance)
A	B (1.41), C (2.24), D (3.00), E (4.12), F (5.10), G (6.08), H (7.21), I (8.06)
B	A (1.41), C (2.24), D (2.24), E (3.00), F (4.12), G (5.00), H (6.08), I (6.71)
C	A (2.24), B (2.24), D (1.41), E (2.24), F (3.00), G (4.00), H (5.10), I (6.08)
D	C (1.41), B (2.24), E (1.41), F (2.24), G (3.00), A (3.00), H (4.12), I (5.10)
E	D (1.41), F (1.41), C (2.24), G (2.24), B (3.00), H (3.00), A (4.12), I (4.12)
F	E (1.41), G (1.41), D (2.24), H (2.24), C (3.00), I (3.00), B (4.12), J (4.12)
G	F (1.41), H (1.41), E (2.24), I (2.24), D (3.00), J (3.00), C (4.00), B (5.00)
H	G (1.41), I (1.41), F (2.24), J (2.24), E (3.00), K (3.00), D (4.12), L (4.12)
I	H (1.41), J (1.41), G (2.24), K (2.24), F (3.00), L (3.00), E (4.12), M (4.12)
J	I (1.41), K (1.41), H (2.24), L (2.24), G (3.00), M (3.00), F (4.12), N (4.12)
K	J (1.41), L (1.41), I (2.24), M (2.24), H (3.00), N (3.00), G (4.12), O (4.12)
L	K (1.41), M (1.41), J (2.24), N (2.24), I (3.00), O (3.00), H (4.12), P (4.12)
M	L (1.41), N (1.41), K (2.24), O (2.24), J (3.00), P (3.00), I (4.12), Q (4.12)
N	M (1.41), O (1.41), L (2.24), P (2.24), K (3.00), Q (3.00), J (4.12), R (4.12)
O	N (1.41), P (1.41), M (2.24), Q (2.24), L (3.00), R (3.00), K (4.12), S (4.12)
P	O (1.41), Q (1.41), N (2.24), R (2.24), M (3.00), S (3.00), L (4.12), T (4.12)
Q	P (1.41), R (1.41), O (2.24), S (2.24), N (3.00), T (3.00), M (4.12), P (5.10)
R	Q (1.41), S (1.41), P (2.24), T (2.24), O (3.00), N (4.12), Q (6.08), R (7.21)
S	R (1.41), T (1.41), Q (2.24), P (3.00), O (4.12), N (5.10), M (6.08), L (7.21)
T	S (1.41), R (2.24), Q (3.00), P (4.12), O (5.10), N (6.08), M (7.21), L (8.25)

A. Key Observations

- **Brute-Force:** Exact ($O(n)$ per query), highest accuracy (100% recall), but slow for large n[3]. No index overhead, so easy to implement.
- **KD-Tree:** Exact on average $O(\log n)$ for low d[11]. Effective only for small d (e.g., $d < 20$). Simple to implement, but performance degrades sharply as d grows due to boundary effects[6][14].
- **Ball-Tree:** Exact, similar to KD-Tree for query time. Handles clustered data somewhat better; often better than KD-Tree in moderate dimensions[14]. Still suffers from dimensionality when d is very large [14].
- **LSH (Hashing):** Approximate, average sublinear time (depends on number of hash tables)[4]. Fast in high dimensions, easy inserts/deletes, but trades recall for speed. Requires tuning of hash functions and tables; more tables or probes increase recall. Multi-probe LSH can reduce tables. Memory overhead from multiple hash tables.
- **Graph (NSW/HNSW):** Approximate (greedy small-world). Empirical query time roughly $O(\log n)$[5]. Typically achieves very high recall ($> 90\%$) with small constants[[5]]. Handles high-dimensional data well. Drawbacks: building the graph can be expensive ($O(n \log n)$ or higher) and dynamic updates (inserting new points) are nontrivial. Performance depends on graph parameters (degree, layers)[5].

In practice, modern libraries like FAISS combine these approaches. For example, FAISS can run GPU-accelerated brute-force or quantized search for very large datasets[19][20]. Table V abstracts typical behavior; actual performance depends on data and tuning. Table V compares methods by query complexity and qualitative pros/cons

V. Discussion

Advantages: NNS methods are conceptually simple and model-agnostic. Brute-force search assumes nothing beyond pairwise distances and always finds true neighbors[3]. Tree structures exploit spatial structure to accelerate queries when data are low-dimensional or partitionable. Approximate methods can query extremely quickly even for very large n; e.g. HNSW often returns ($> 90\%$) of true neighbors in milliseconds even for datasets of size $\geq$1M[15]. Graph-based approaches naturally support any metric space and can handle data lying on low-dimensional manifolds (due to their small-world connectivity). They also allow incremental insertion of points into the graph (dynamic graphs) if implemented carefully, making them suitable for growing datasets.

Challenges: The curse of dimensionality is fundamental: as d increases, distances tend to concentrate and many indexing strategies fail[13]. This leads to high query cost or poor recall unless dimensionality is reduced first (via PCA, autoencoders, etc.)[13]. Computational cost is another issue: exact structures (trees/graphs) become infeasible for large n.

Even approximate methods incur overhead (e.g. maintaining multiple hash tables or constructing graphs). Many methods assume static data: supporting updates (insert/delete) often requires rebalancing or rebuilding parts of the index, which can be costly. There is also a trade-off between accuracy and speed: achieving higher recall usually requires more search effort (visiting more tree nodes, more hops in the graph, more hash probes)[13][16]. Tuning hyperparameters (tree split strategies, number of hashes, HNSW degree) is often dataset-specific and can be time consuming.

Gaps and Open Problems: Truly dynamic/streaming data pose ongoing challenges. Most ANN algorithms are designed for static datasets; efficiently merging new data on-the-fly is an open problem[12]. Balancing memory and speed is also critical: high-degree graphs or many hash tables boost recall but use more memory. Research on compressing indices (via quantization or graph pruning) without hurting accuracy is active.

Furthermore, most methods focus on simple vector inputs; extending NNS to multi-modal or structured data (e.g. images+text, graphs) requires new distance metrics and indexing schemes.

VI. Limitations of This Survey

This survey emphasizes vector-space NNS and graph-based methods. We have omitted some topics. For instance, we only briefly mentioned metric trees like cover trees or M-trees, which are alternatives to k-d/ball trees. Sophisticated ANN techniques such as product quantization (PQ) and neural hashing were only noted in passing. We drew mainly from published surveys and popular libraries (FAISS, Annoy) to illustrate trends. Our discussion of performance (e.g. Table V) is qualitative and indicative; actual run times can vary widely with hardware and data distribution. Moreover, while we reference recent works (2023–2024 surveys and papers), the fast-moving field may already have new results beyond our coverage.

VII. Future Work

Future NNS research is trending toward hybrid and accelerated solutions. One idea is combining exact filters with approximate indexes: for example, use a lightweight tree or hash index to quickly narrow candidates, then refine with brute-force for final accuracy. Hardware acceleration is key: leveraging GPUs (as in FAISS) or specialized accelerators can dramatically reduce latency. Indeed, GPU-based k-selection and distance kernels in FAISS can achieve order-of-magnitude speed-ups over CPUs[17]. Techniques for quantization-aware search and vector compression (e.g. IVF-PQ) will also improve throughput on memory-limited devices.

Hybrid graph-tree methods are an emerging area: for instance, using a coarse tree to find a region then refining with graph search locally. Dynamic indexing remains needed: algorithms that gracefully handle incremental inserts/deletes (e.g. dynamic HNSW variants) would enable NNS on streaming data. Integrating NNS into larger AI systems (such as vector databases supporting LLMs) raises new issues of multi-modal search and query-driven indexing. Research on privacy for NNS (ensuring queries don't leak sensitive data) and auto-tuning (automatically choosing hyperparameters via meta-learning) are promising avenues.

VIII. Conclusion

Nearest Neighbor Search is a cornerstone of similarity search and classification. Brute-force search is simple and exact but scales poorly with data size[3]. Spatial trees (k-d tree, ball tree) accelerate queries in low dimensions but suffer as dimensionality grows[5][7]. Approximate methods like LSH and graph-based ANN (e.g. HNSW) offer dramatic speedups with minimal accuracy loss[8][15]. Graph-based approaches in particular exploit small-world connectivity to achieve near-logarithmic search times[9]. However, all methods struggle with very high-dimensional data and dynamic datasets. Ongoing research focuses on hardware acceleration, hybrid algorithms, and dynamic indices to make NNS feasible for modern large-scale applications[17]. As vector databases and AI-driven search proliferate, efficient and adaptable NNS will remain an active research area..

References

[1] G. Shakhnarovich, T. Darrell, and P. Indyk, *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*, MIT Press, 2006.

[2] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[3] R. Geisberger, "Exact search in high dimensions," in *Lecture Notes in Computer Science*, vol. 7814, pp. 84–93, 2013.

[4] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *SCG*, 2004, pp. 253–262.

[5] Y. Malkov and D. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *arXiv:1603.09320*, 2018.

[6] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *ICDT*, 2001, pp. 420–434.

[7] L. Li, S. Agrawal, J. Dastjerdi, and K. Ramachandra, "A comprehensive survey on vector database: Storage and retrieval techniques and challenges," *arXiv:2310.11703*, 2023.

[8] H. Jégou, "Twenty questions on competitive analysis of approximate nearest neighbor search," in *ICASSP*, 2017.

[9] V. Garcia, E. Debreuve, M. Barlaud, and O. Ait-Aoudia, "k-nearest neighbor search: Fast GPU-based implementations," *arXiv:1610.03801*, 2016.

[10] M. S. Rahman and M. H. Rashid, "Survey of approximate nearest neighbor techniques," *International Journal of Computer Science and Engineering*, vol. 9, 2019.

[11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[12] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is 'nearest neighbor' meaningful?" in *PODS*, 1999, pp. 217–235.

[13] J. Wang, S. Kumar, and S.-F. Chang, "Hashing for similarity search: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 12, pp. 247–268, 2014.

[14] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2227–2240, 2014.

[15] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998, pp. 604–613.

[16] E. J. Keogh and M. Pazzani, "Scaling up dynamic time warping for datamining applications," in *KDD*, 2000, pp. 285–289.

[17] C. Francesschi, S. Alimadadi, and J. Szlichta, "Brute-force vs. tree search vs. projection hashing: A comparative study," *J. Mach. Learn. Res.*, vol. 17, 2016.
[18] F. Lazaridis and A. Mehrotra, "DEEP: A fast dimension reduction method for high-dimensional similarity search," in *VLDB*, 2006, pp. 826–837.
[19] G. Vishnoi, R. Johnson, and H. Jégou, "FAISS: A library for efficient similarity search," in *Proc. NeurIPS*, 2017.
[20] M. Douze, J. Johnson, and H. Jégou, "Billion-scale similarity search with GPUs," in *Proc. NeurIPS*, 2017.
[21] J. Vahrenhold, "Nearest-neighbor search in metric spaces," in *Encyclopedia of Algorithms*, 2008.
[22] P. Indyk, R. Kleinberg, S. Mahabadi, and Y. Yuan, "Simultaneous Nearest Neighbor Search," in *Proc. ICML*, Sydney, 2017.