Carnegie Mellon University

Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World (HNSW) graphs

Hu, Tong; Liu, Jiarui; Ma, Christina

4/22/24

Motivation

- Similarity Search: applications in ML, retrieval, and with genAI -> RAG.
- KNN -> ANN: computational complexity vs. search accuracy.

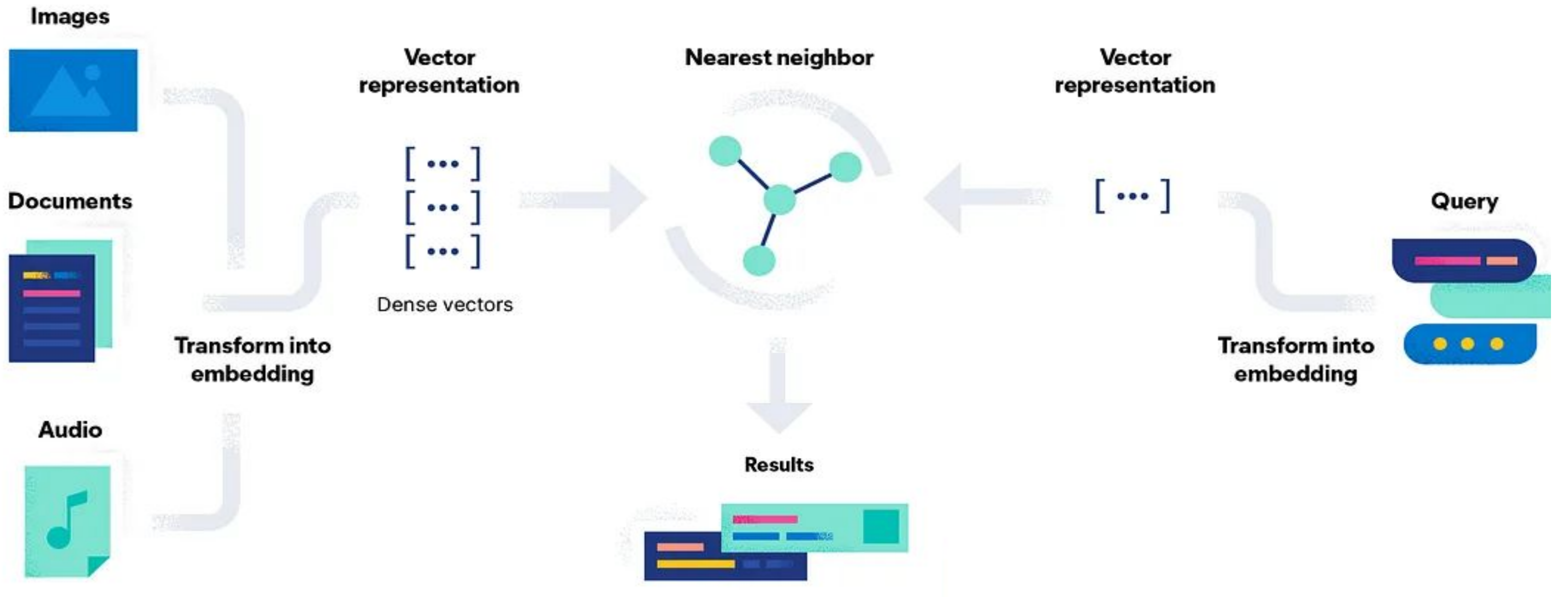
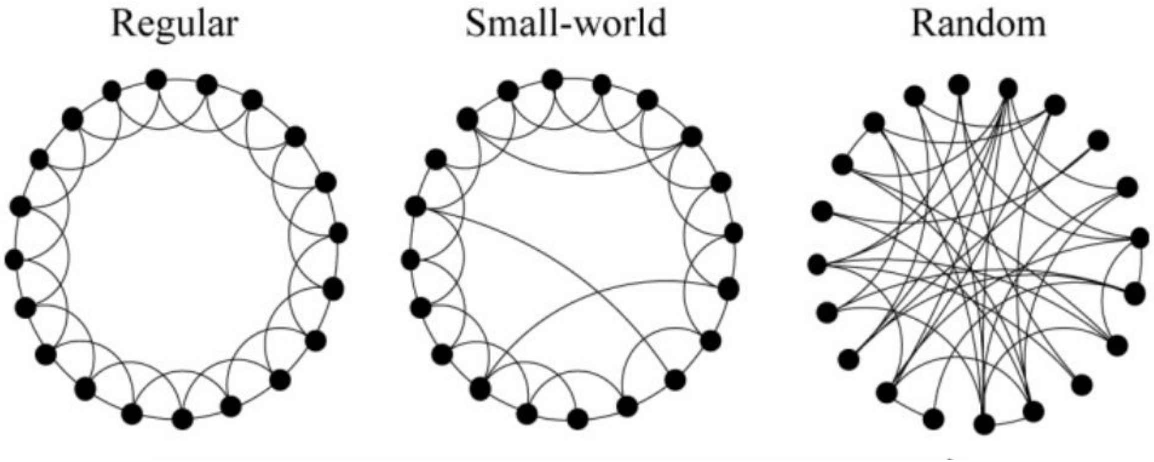


Figure: https://medium.com/@ma-korotkov/a-gentle-introduction-to-vector-search-dc6e54e34907

Motivation for Navigable Small Worlds (NSW)

Six degrees of separation experiments run by Milgram in the 1960s.



High clustering coefficient
High distance

High clustering coefficient
Low distance

Low clustering coefficient
Low distance

Figure: https://mooreniemi.github.io/2022/12/05/scenic-tour-of-hnsw.html

ANN algorithm: Navigable Small Worlds (NSW)

- **Polylogarithmic** search and insertion, better for high dimensional large dataset

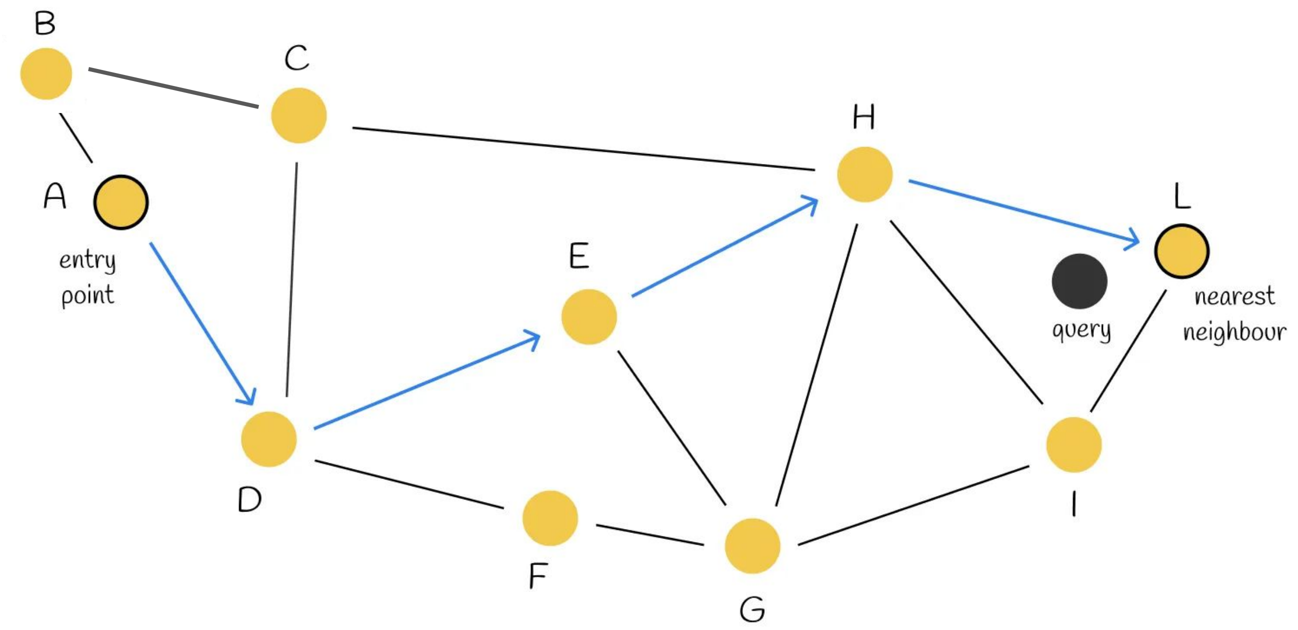


Figure: https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

ANN algorithm: Navigable Small Worlds (NSW)

- Greedy search can be trapped in local optimum (early stopping)

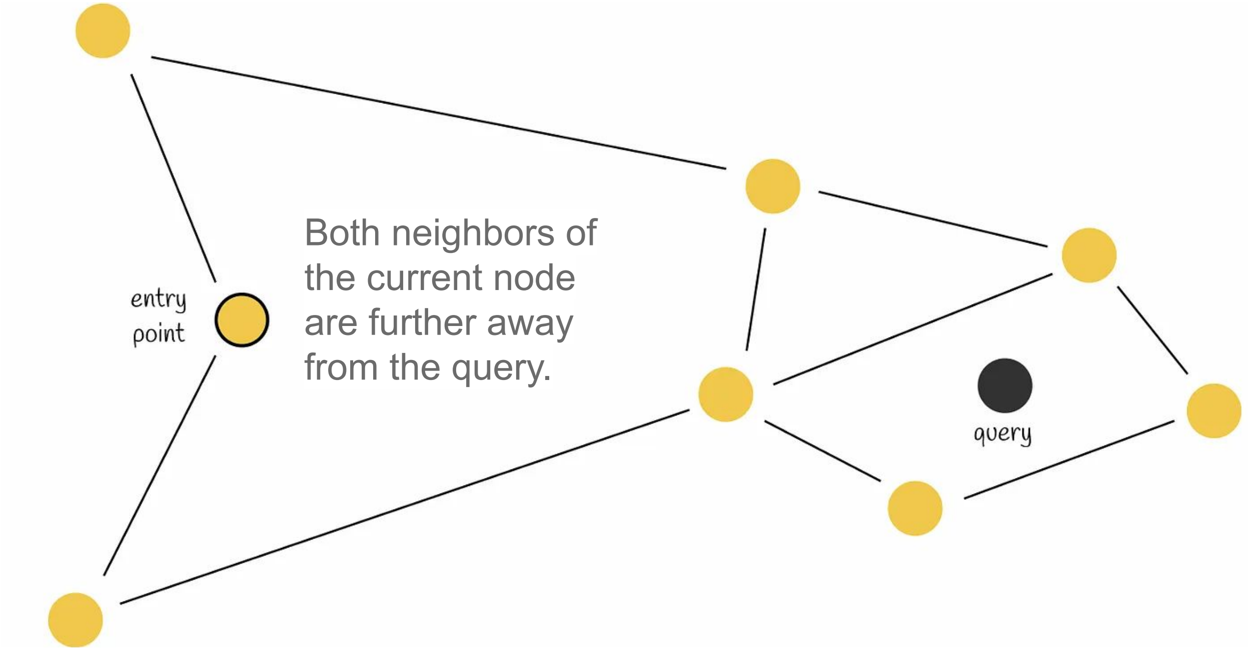


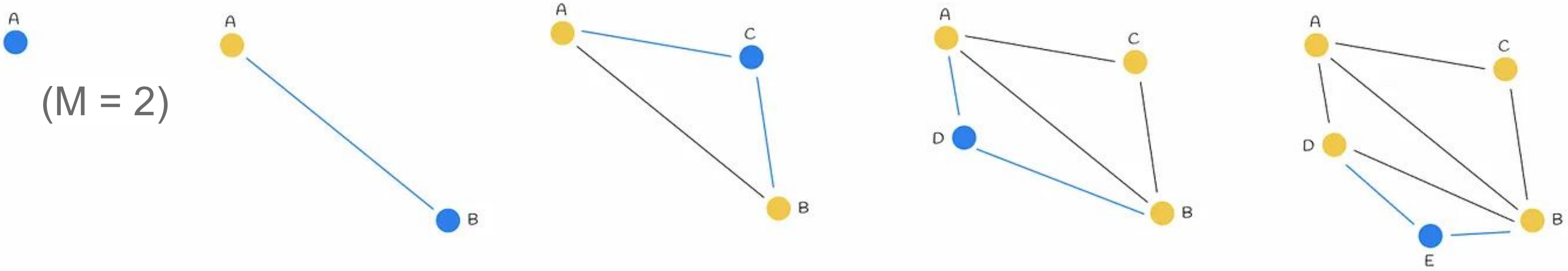
Figure: https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

NSW Graph Construction

- Insert random points and link edges to M nearest neighbors (search)
- Longer edges are likely created at the beginning phase of graph construction

– “later become bridges between the network hubs that keep the overall graph connectivity and allow the logarithmic scaling of the number of hops during greedy routing.”

(M = 2)



Data structure inspiration: Skip Lists

- **O(log n)** time complexity on average for both insertion and search
- Layered format with **longer** edges in the highest layers (for fast search) and **shorter** edges in the lower layers (for accurate search).

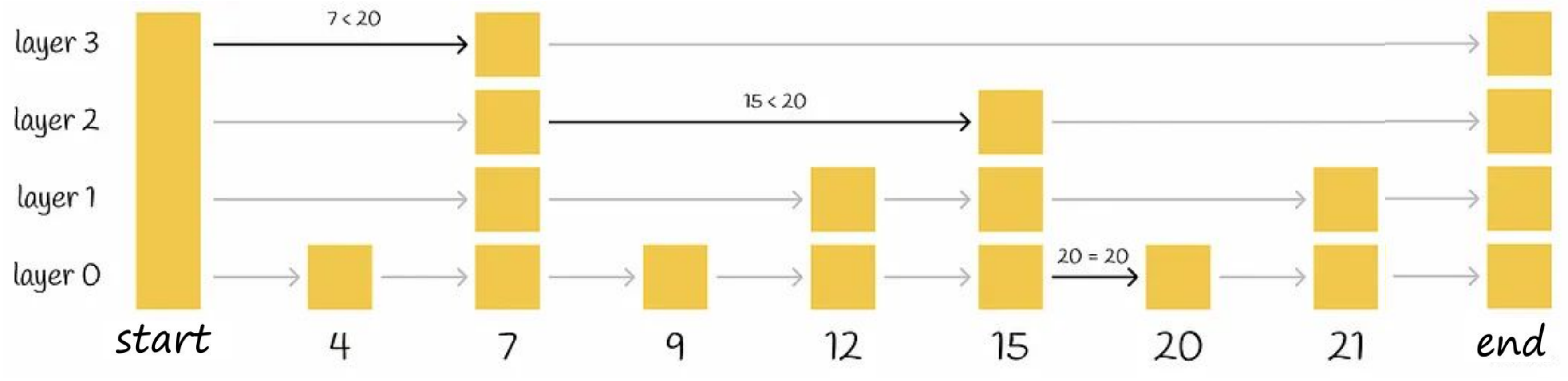
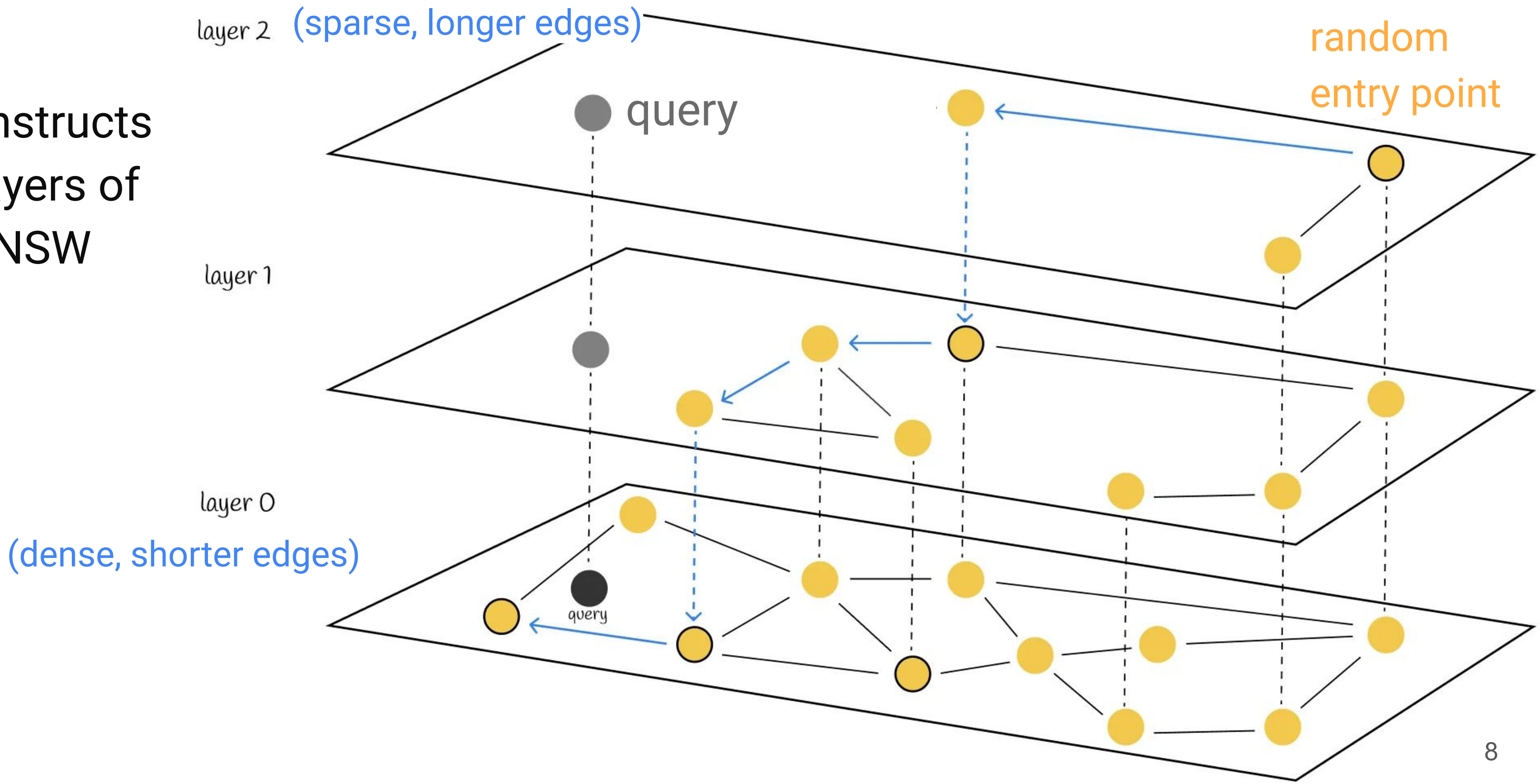


Figure: https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

HNSW: *Hierarchical* Navigable Small Worlds

HNSW constructs multiple layers of proximity NSW graphs.



HNSW: NSW + Skip List

From NSW:

- Zoom-out, then zoom-in (polylogarithmic) => zoom-in first in a graph (logarithmic)

From skip list:

- Separate the edges according to their length scale into different layers

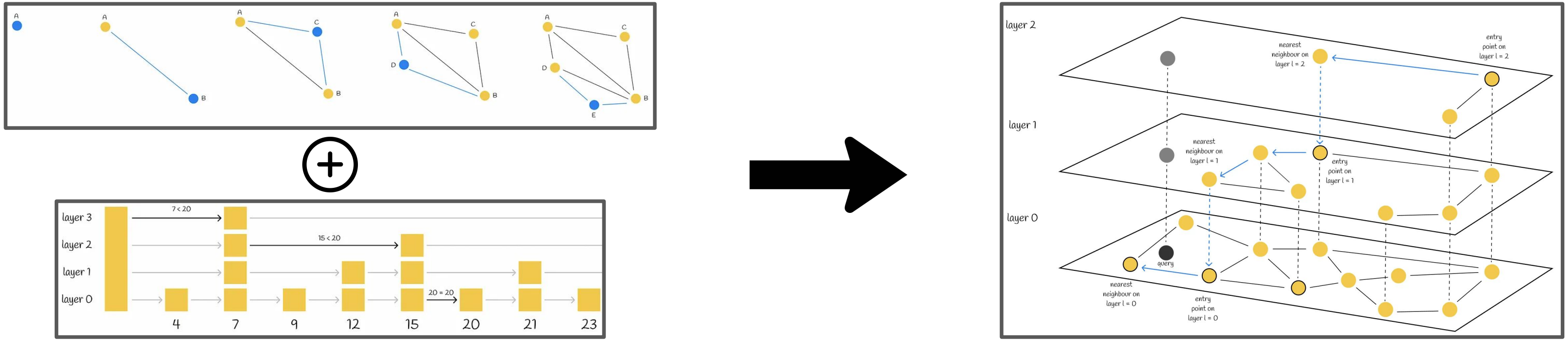


Figure: https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

HNSW Algorithm

1. Search
2. Insertion
3. Candidate selection heuristic

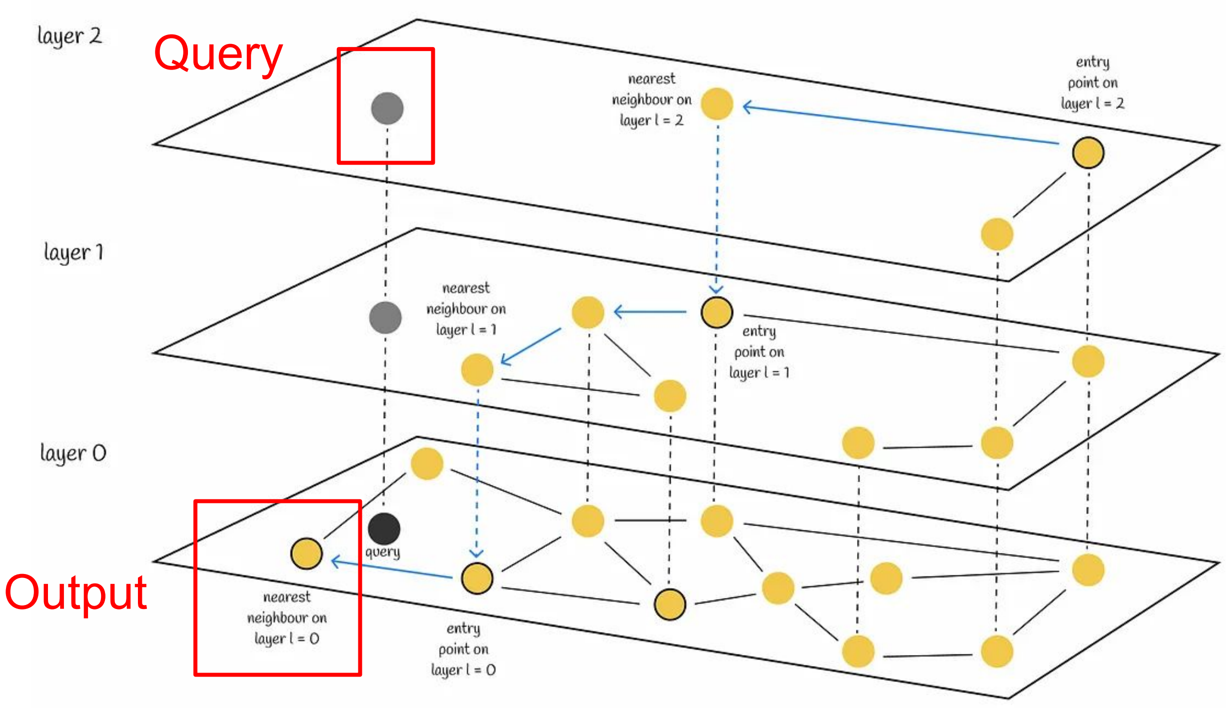
Search

Inputs:

1. A query
2. A constructed HNSW graph

Outputs:

- K nearest neighbors to the query



Search

1. Starts from the highest layer, by randomly choosing a starting enter point

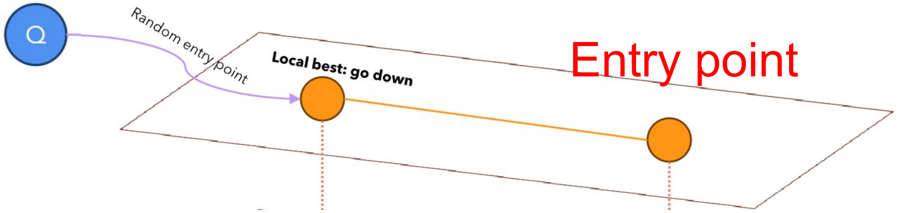
Algorithm 5

K-NN-SEARCH(*hnsw*, *q*, *K*, *ef*)

Input: multilayer graph *hnsw*, query element *q*, number of nearest neighbors to return *K*, size of the dynamic candidate list *ef*

Output: *K* nearest elements to *q*

```
W ← Ø  // set for the current nearest elements
ep ← get enter point for hnsw
L ← level of ep  // top layer for hnsw
for lc ← L … 1
   W ← SEARCH-LAYER(q, ep, ef=1, lc)
   ep ← get nearest element from W to q
W ← SEARCH-LAYER(q, ep, ef, lc =0)
return K nearest elements from W to q
```



Layer 3 (sparse)

Search

2. Proceeds to one level below each time, to find the local nearest neighbor among that layer nodes

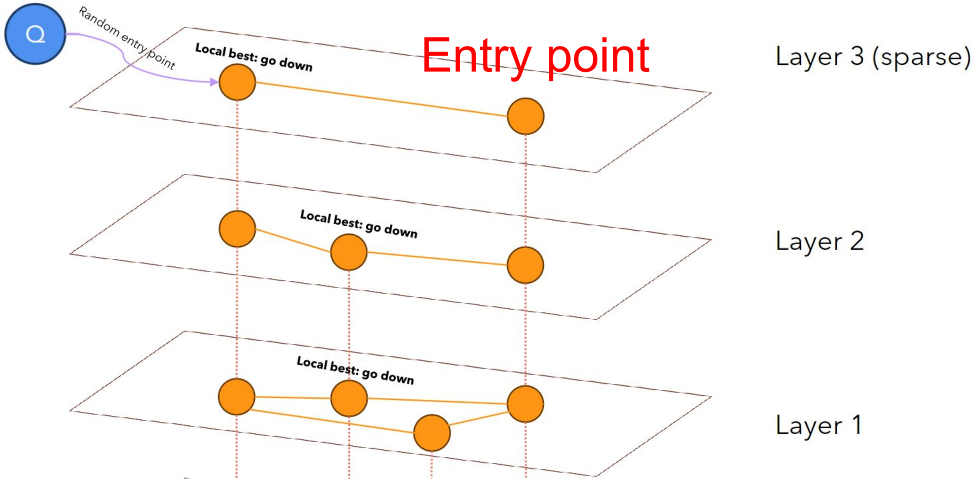
Algorithm 5

K-NN-SEARCH(*hnsw*, *q*, *K*, *ef*)

Input: multilayer graph *hnsw*, query element *q*, number of nearest neighbors to return *K*, size of the dynamic candidate list *ef*

Output: *K* nearest elements to *q*

```
W ← Ø   // set for the current nearest elements
ep ← get enter point for hnsw
L ← level of ep   // top layer for hnsw
for l_c ← L … 1
  W ← SEARCH-LAYER(q, ep, ef=1, l_c)
  ep ← get nearest element from W to q
W ← SEARCH-LAYER(q, ep, ef, l_c =0)
return K nearest elements from W to q
```

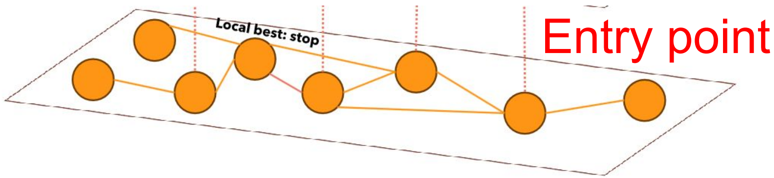


Search

2. Return K nearest neighbors found on the lowest layer

Algorithm 5

```
K-NN-SEARCH(hnsw, q, K, ef)
Input: multilayer graph hnsw, query element q, number of nearest
neighbors to return K, size of the dynamic candidate list ef
Output: K nearest elements to q
W ← ∅   // set for the current nearest elements
ep ← get enter point for hnsw
L ← level of ep   // top layer for hnsw
for lc ← L … 1
   W ← SEARCH-LAYER(q, ep, ef=1, lc)
   ep ← get nearest element from W to q
W ← SEARCH-LAYER(q, ep, ef, lc =0)
return K nearest elements from W to q
```



Layer 0 (dense)

Insertion

Insert nodes to the HNSW graph one-by-one

Inputs:

- HNSW
- Q, a new node
- efConstruction, size of the dynamic candidate list
- L, the number of layers
- mL, the normalization factor
- M, number of established edges
- Mmax: maximum number of edges for each element per layer

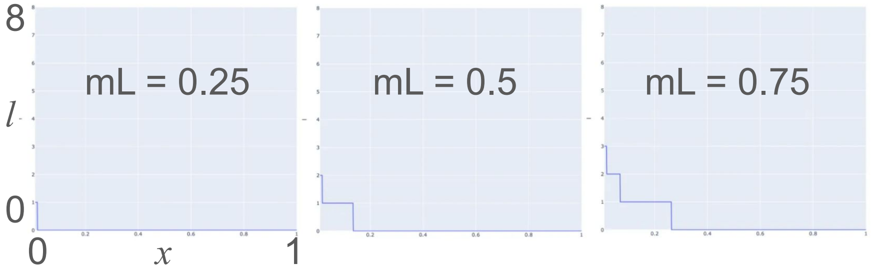
Insertion

Step 1: assign an integer l, the maximum layer where the node can present

- The number of layers l for every node is chosen randomly with exponentially decaying probability distribution

- *l = 1*: the node can only be found at layer 0 and layer 1
- *mL = 0*: the vectors are inserted at layer 0 only

$$l = \text{float}\left[-\ln(\text{uniform}(0, 1)) \cdot m_L\right]$$



Insertion: Step 1

"To achieve the optimum performance advantage of the controllable hierarchy, the overlap between neighbors on different layers has to be small."

mL value tradeoff:

- a smaller mL: more traversals on each layer
- a larger mL: more overlaps

Choose mL = 1/ln(M)

Insertion: Step 1

Algorithm 1

INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)

Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L

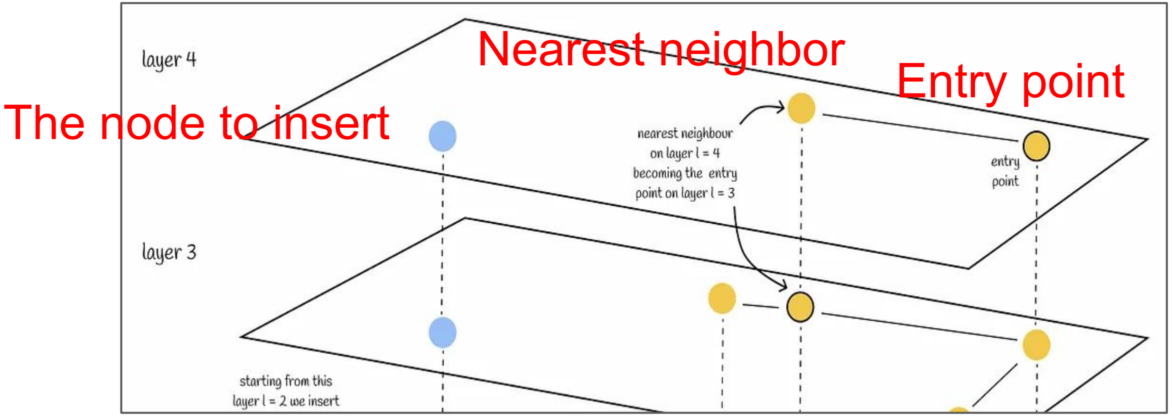
Output: update *hnsw* inserting element *q*

```
W ← ∅  // list for the currently found nearest elements
ep ← get enter point for hnsw
L ← level of ep  // top layer for hnsw
l ← ⌊-ln(unif(0..1))·m_L⌋  // new element's level
for l_c ← L … l+1
  W ← SEARCH-LAYER(q, ep, ef=1, l_c)
  ep ← get the nearest element from W to q
for l_c ← min(L, l) … 0
  W ← SEARCH-LAYER(q, ep, efConstruction, l_c)
  neighbors ← SELECT-NEIGHBORS(q, W, M, l_c) // alg. 3 or alg. 4
  add bidirectionall connections from neighbors to q at layer l_c
  for each e ∈ neighbors  // shrink connections if needed
    eConn ← neighbourhood(e) at layer l_c
    if |eConn| > M_max // shrink connections of e
                       // if l_c = 0 then M_max = M_max0
      eNewConn ← SELECT-NEIGHBORS(e, eConn, M_max, l_c)
                                              // alg. 3 or alg. 4
      set neighbourhood(e) at layer l_c to eNewConn
  ep ← W
if l > L
  set enter point for hnsw to q
```

Insertion

Step 2: greedy search

1. Greedily search for the nearest node from the upper layer (efConstruction=1)
2. Use it as an entry point to the next layer until reaching layer l



Insertion: Step 2

Algorithm 1

INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)

Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L

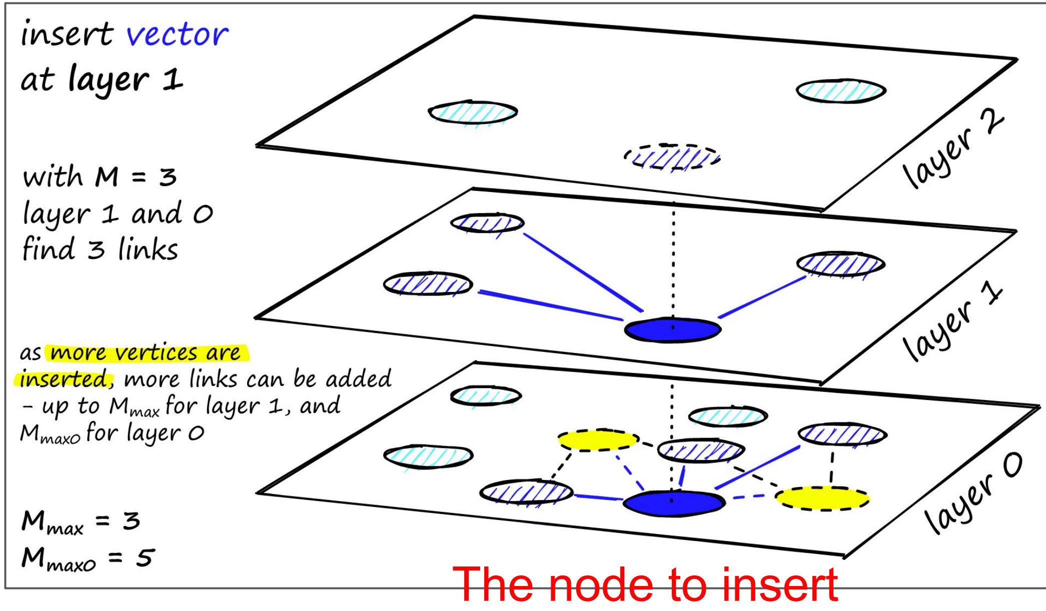
Output: update *hnsw* inserting element *q*

```
W ← ∅   // list for the currently found nearest elements
ep ← get enter point for hnsw
L ← level of ep   // top layer for hnsw
l ← ⌊-ln(unif(0..1))·mL⌋  // new element's level
for lc ← L … l+1
  W ← SEARCH-LAYER(q, ep, ef=1, lc)
  ep ← get the nearest element from W to q
for lc ← min(L, l) … 0
  W ← SEARCH-LAYER(q, ep, efConstruction, lc)
  neighbors ← SELECT-NEIGHBORS(q, W, M, lc) // alg. 3 or alg. 4
  add bidirectionall connections from neighbors to q at layer lc
  for each e ∈ neighbors   // shrink connections if needed
    eConn ← neighbourhood(e) at layer lc
    if |eConn| > Mmax // shrink connections of e
                       // if lc = 0 then Mmax = Mmax0
      eNewConn ← SELECT-NEIGHBORS(e, eConn, Mmax, lc)
                                               // alg. 3 or alg. 4
      set neighbourhood(e) at layer lc to eNewConn
  ep ← W
if l > L
  set enter point for hnsw to q
```

Insertion

Step 3: connect to the current graph

1. Insert the node starting from the layer l
2. Greedily search for efConstruction nearest neighbors
3. Select M nodes from the efConstruction node set and build edges



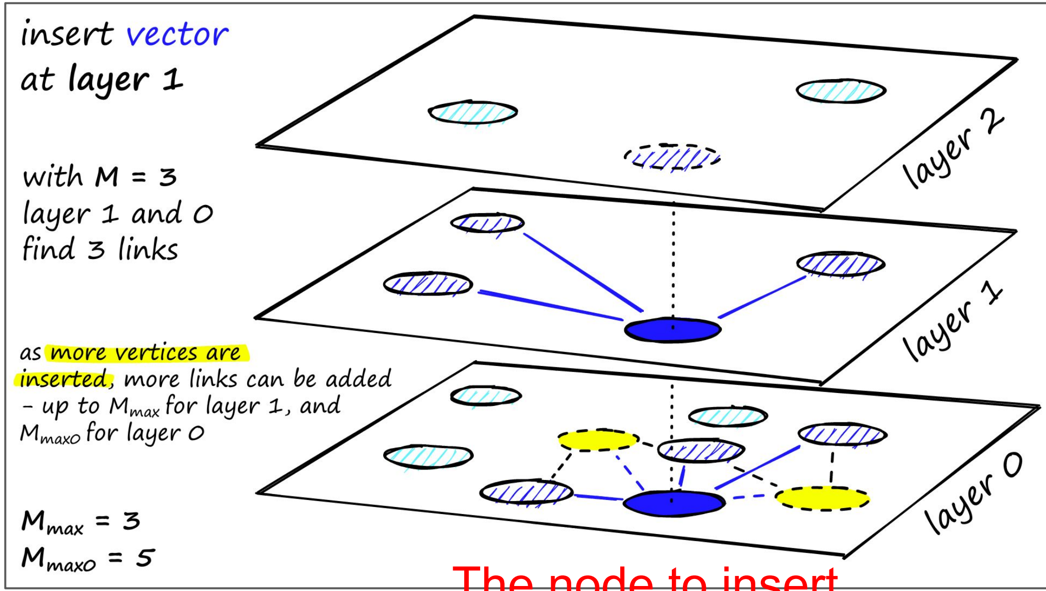
The edge connection is constrained by Mmax in each layer

Insertion

Step 3: connect to the current graph

4. Each of found efConstruction nodes acts as an entry point

5. Terminate after building edges in layer 0



The node to insert

The edge connection is constrained by Mmax in each layer

Insertion: Step 3

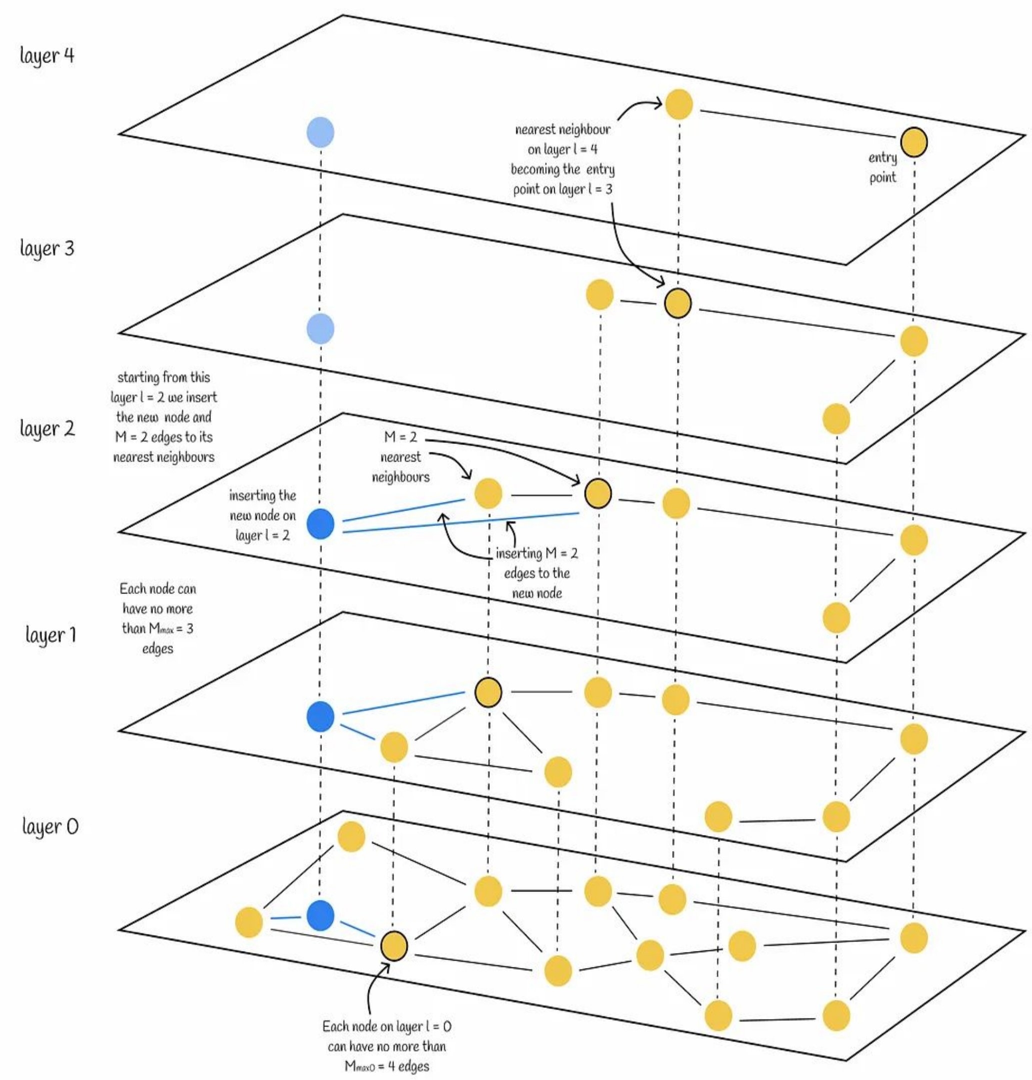
Algorithm 1

INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)

Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L

Output: update *hnsw* inserting element *q*

```
W ← ∅   // list for the currently found nearest elements
ep ← get enter point for hnsw
L ← level of ep   // top layer for hnsw
l ← ⌊-ln(unif(0..1))·mL⌋  // new element's level
for lc ← L … l+1
  W ← SEARCH-LAYER(q, ep, ef=1, lc)
  ep ← get the nearest element from W to q
for lc ← min(L, l) … 0
  W ← SEARCH-LAYER(q, ep, efConstruction, lc)
  neighbors ← SELECT-NEIGHBORS(q, W, M, lc) // alg. 3 or alg. 4
  add bidirectionall connections from neighbors to q at layer lc
  for each e ∈ neighbors   // shrink connections if needed
    eConn ← neighbourhood(e) at layer lc
    if |eConn| > Mmax // shrink connections of e
                      // if lc = 0 then Mmax = Mmax0
      eNewConn ← SELECT-NEIGHBORS(e, eConn, Mmax, lc)
                                               // alg. 3 or alg. 4
      set neighbourhood(e) at layer lc to eNewConn
  ep ← W
if l > L
  set enter point for hnsw to q
```



Algorithm 1

INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)

Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L

Output: update *hnsw* inserting element *q*

$W \leftarrow \emptyset$ // list for the currently found nearest elements
ep ← get enter point for *hnsw*
$L \leftarrow$ level of *ep* // top layer for *hnsw*
$l \leftarrow \lfloor -\ln(unif(0..1)) \cdot m_L \rfloor$ // new element's level
for $l_c \leftarrow L \ldots l+1$
    $W \leftarrow$ SEARCH-LAYER(q, ep, ef=1, l_c)
    ep ← get the nearest element from *W* to *q*
for $l_c \leftarrow \min(L, l) \ldots 0$
    $W \leftarrow$ SEARCH-LAYER(q, ep, *efConstruction*, l_c)
    neighbors ← SELECT-NEIGHBORS(q, W, M, l_c) // alg. 3 or alg. 4
    add bidirectionall connections from *neighbors* to *q* at layer l_c
    for each $e \in$ *neighbors* // shrink connections if needed
        eConn ← *neighbourhood*(*e*) at layer l_c
        if $|eConn| > M_{max}$ // shrink connections of *e*
            // if $l_c = 0$ then $M_{max} = M_{max0}$
            eNewConn ← SELECT-NEIGHBORS(e, *eConn*, M_{max}, l_c) // alg. 3 or alg. 4
            set *neighbourhood*(*e*) at layer l_c to *eNewConn*
    $ep \leftarrow W$
if $l > L$
    set enter point for *hnsw* to *q*

Search Layer

Obtain the approximate ef nearest neighbors in layer lc

- Used in NSW
- Allow discarding candidates for evaluation

Algorithm 2

SEARCH-LAYER(q, ep, ef, l_c)

Input: query element q, enter points ep, number of nearest to q elements to return ef, layer number l_c

Output: ef closest neighbors to q

```
v ← ep    // set of visited elements
C ← ep    // set of candidates
W ← ep    // dynamic list of found nearest neighbors
while │C│ > 0
  c ← extract nearest element from C to q
  f ← get furthest element from W to q
  if distance(c, q) > distance(f, q)
    break   // all elements in W are evaluated
  for each e ∈ neighbourhood(c) at layer lc   // update C and W
    if e ∉ v
      v ← v ∪ e
      f ← get furthest element from W to q
      if distance(e, q) < distance(f, q) or │W│ < ef
        C ← C ∪ e
        W ← W ∪ e
        if │W│ > ef
          remove furthest element from W to q
return W
```

Candidate Selection Simple

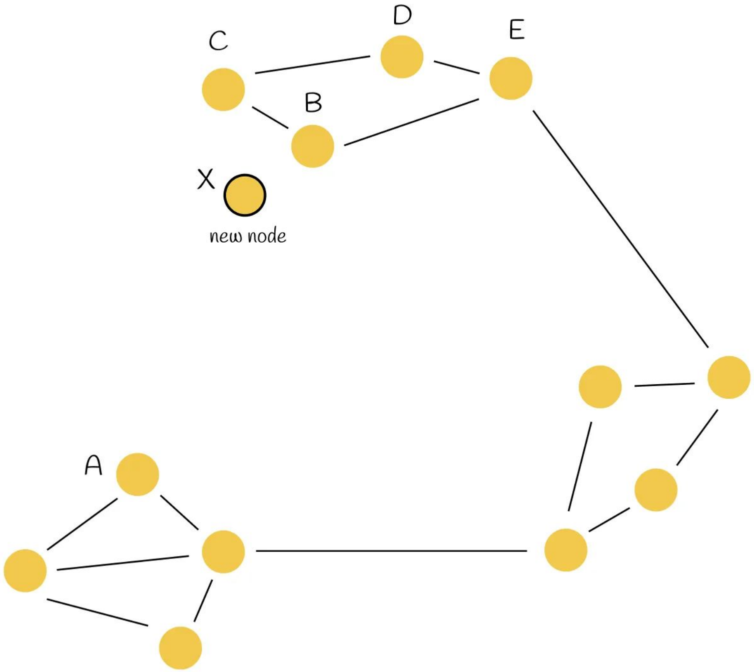
Q: Which M nodes to take out of efConstruction candidates?

A: Naive way – take M closest candidates

Here X will be connected to B and C if M = 2.

However, ideally it can be better for navigation if the region A and B can be connected.

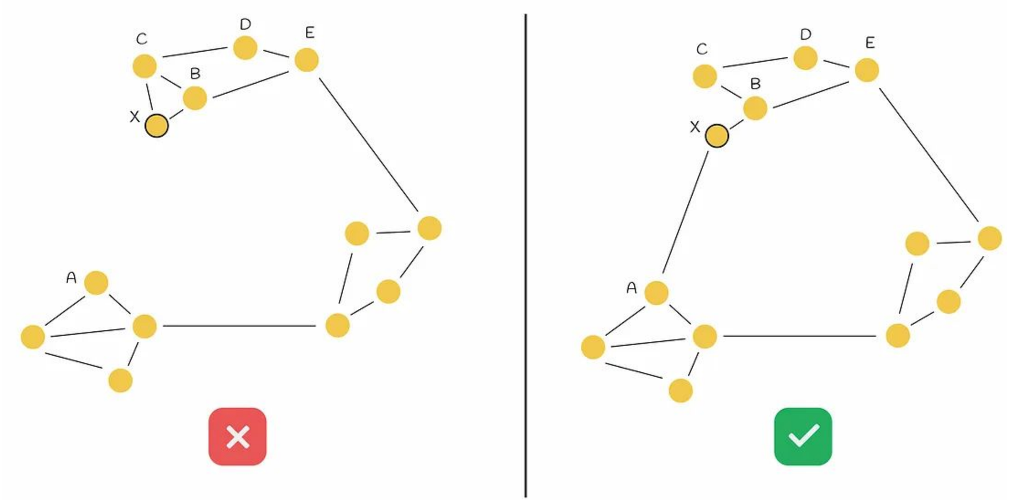
Algorithm 3
SELECT-NEIGHBORS-SIMPLE(q, C, M)
Input: base element q, candidate elements C, number of neighbors to return M
Output: M nearest elements to q
return M nearest elements from C to q



Candidate Selection Heuristic

The heuristic considers both:

- The closest distances between nodes
- The connectivity of different regions on the graph



Algorithm 4

SELECT-NEIGHBORS-HEURISTIC(q, C, M, l_c, *extendCandidates*, *keepPrunedConnections*)

Input: base element q, candidate elements C, number of neighbors to return M, layer number l_c, flag indicating whether or not to extend candidate list *extendCandidates*, flag indicating whether or not to add discarded elements *keepPrunedConnections*

Output: M elements selected by the heuristic

$R \leftarrow \emptyset$
$W \leftarrow C$ // working queue for the candidates
if *extendCandidates* // extend candidates by their neighbors
  for each $e \in C$
    for each $e_{adj} \in$ *neighbourhood*(e) *at layer* l_c
      if $e_{adj} \notin W$
        $W \leftarrow W \cup e_{adj}$
$W_d \leftarrow \emptyset$ // queue for the discarded candidates
while $|W| > 0$ and $|R| < M$
  $e \leftarrow$ extract nearest element from W to q
  if e is closer to q compared to any element from R
    $R \leftarrow R \cup e$
  else
    $W_d \leftarrow W_d \cup e$
if *keepPrunedConnections* // add some of the discarded
                // connections from W_d
  while $|W_d| > 0$ *and* $|R| < M$
    $R \leftarrow R \cup$ extract nearest element from W_d to q
return R

Complexity Analysis

Search takes *O(logn)* time in total

Insertion of a single vertex: *O(logn)*

HNSW construction requires *O(n * logn)* time in total

Implementation

```
# Initializing index - the maximum number of elements should be known beforehand
p.init_index(max_elements=num_elements, ef_construction=200, M=16)

# Element insertion (can be called several times):
p.add_items(data, ids)

# Controlling the recall by setting ef:
p.set_ef(50)  # ef should always be > k

# Query dataset, k - number of the closest elements (returns 2 numpy arrays)
labels, distances = p.knn_query(data, k=1)
```

https://github.com/nmslib/hnswlib/blob/3f3429661187e4c24a490a0f148fc6bc89042b3d/examples/python/example_search.py

Evaluation - Implementation

- HNSW implementation uses custom distance functions together with C-style memory management.

- Utilized nmslib implementation of sw-graph for NSW.

- Compare with the most up-to-date SOTA.

- Compare with the SOTA in Euclid Spaces with open-source implementation.

Evaluation - Method

- Comparison with Baseline NSW
- Comparison in Euclid Spaces
- Comparison in General Space
- Comparison with product quantization based algorithms.

Evaluation - HNSW vs. Baseline NSW

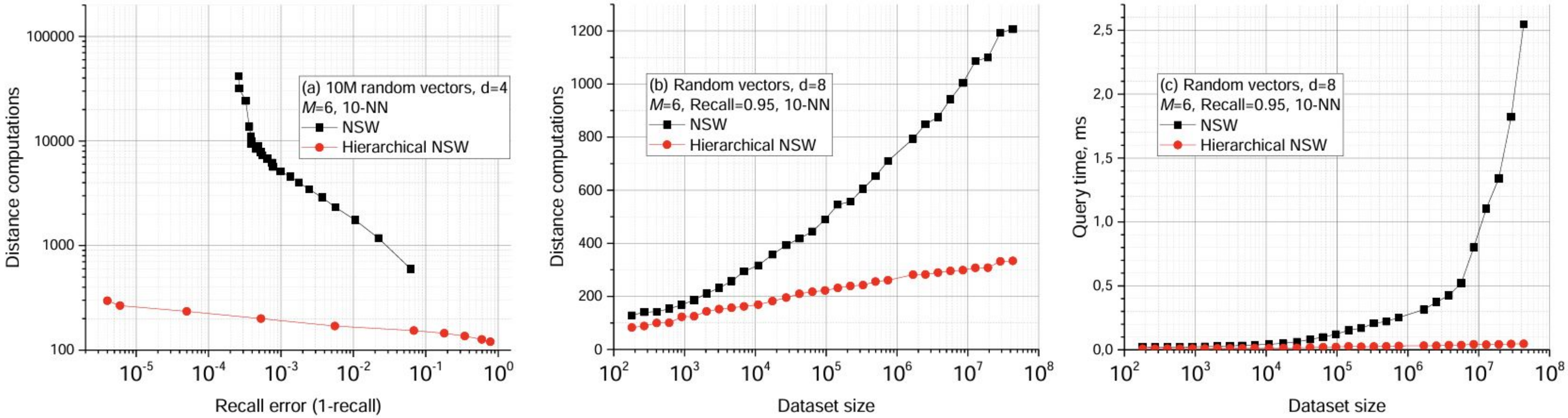


Fig. 12. Comparison between NSW and Hierarchical NSW: (a) distance calculation number vs accuracy tradeoff for a 10 million 4-dimensional random vectors dataset; (b-c) performance scaling in terms of number of distance calculations (b) and raw query(c) time on a 8-dimensional random vectors dataset.

Evaluation - Euclid Spaces - Algorithms to Compare

- Baseline NSW Algorithm
- FLANN
- Annoy
- VP-tree
- FALCONN

Evaluation - Euclid Spaces - Datasets

TABLE 1
Parameters of the used datasets on vector spaces benchmark.

Dataset	Description	Size	d	BF time	Space
SIFT	Image feature vectors [13]	1M	128	94 ms	L_2
GloVe	Word embeddings trained on tweets [52]	1.2M	100	95 ms	cosine
CoPhIR	MPEG-7 features extracted from the images [53]	2M	272	370 ms	L_2
Random vectors	Random vectors in hypercube	30M	4	590 ms	L_2
DEEP	One million subset of the billion deep image features dataset [14]	1M	96	60 ms	L_2
MNIST	Handwritten digit images [54]	60k	784	22 ms	L_2

Evaluation - Euclid Spaces

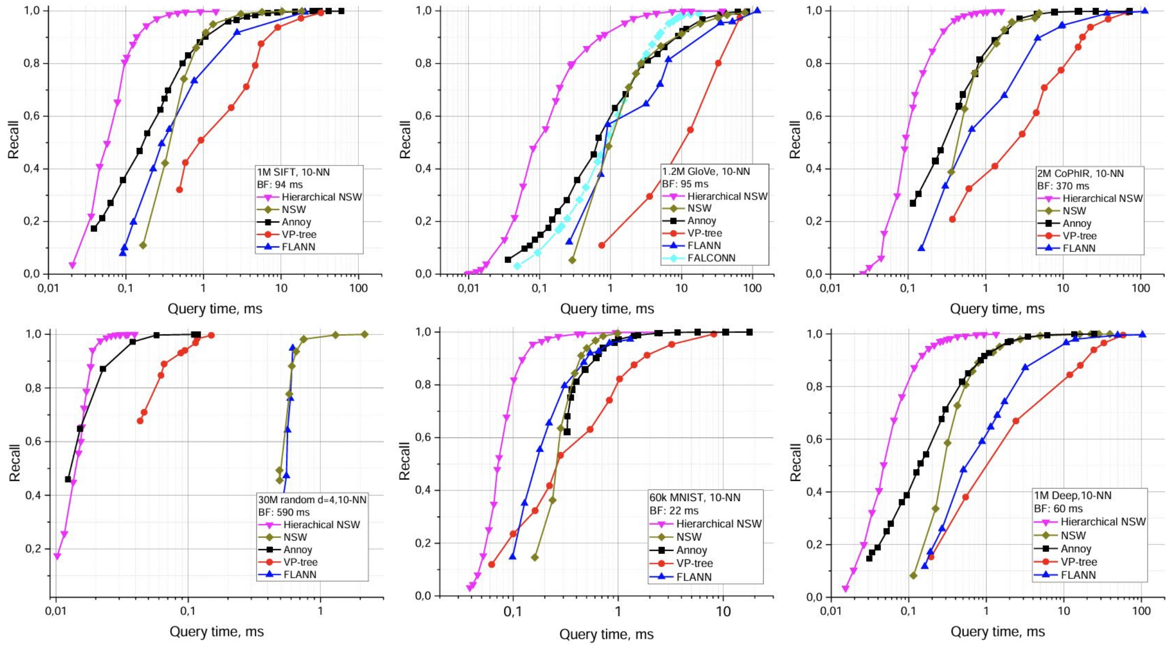


Fig. 13. Results of the comparison of Hierarchical NSW with open source implementations of K-ANNS algorithms on five datasets for 10-NN searches. The time of a brute-force search is denoted as the BF.

Evaluation - General Spaces - Purpose & Algorithms

- Baseline NSW algorithm has several problems on low dimensional datasets as suggested in the paper "Permutation search methods are efficient, yet faster search is possible."
- VP-tree
- Permutation Techniques (NAPP & Brute Force Filtering)
- Baseline NSW Algorithm
- NNDescent-produced proximity graphs

Evaluation - General Spaces - Datasets

TABLE 2.
Used datasets for repetition of the Non-Metric data tests subset.

Dataset	Description	Size	d	BF time	Distance
Wiki-sparse	TF-IDF (term frequency–inverse document frequency) vectors (created via GENSIM [58])	4M	10^5	5.9 s	Sparse cosine
Wiki-8	Topic histograms created from sparse TF-IDF vectors of the wiki-sparse dataset (created via GENSIM [58])	2M	8	-	Jensen–Shannon (JS) divergence
Wiki-128	Topic histograms created from sparse TF-IDF vectors of the wiki-sparse dataset (created via GENSIM [58])	2M	128	1.17 s	Jensen–Shannon (JS) divergence
ImageNet	Signatures extracted from LSVRC-2014 with SQFD (signature quadratic form) distance [59]	1M	272	18.3 s	SQFD
DNA	DNA (deoxyribonucleic acid) dataset sampled from the Human Genome 5 [34].	1M	-	2.4 s	Levenshtein

Evaluation - General Spaces

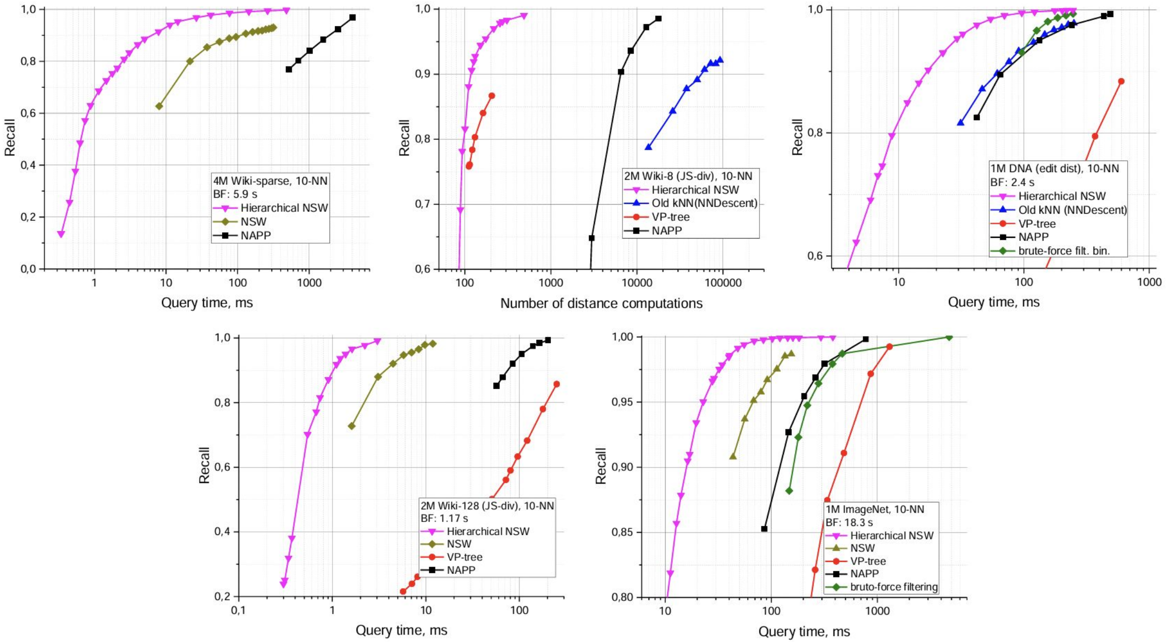


Fig. 14. Results of the comparison of Hierarchical NSW with general space K-ANNS algorithms from the Non Metric Space Library on five datasets for 10-NN searches. The time of a brute-force search is denoted as the BF.

Evaluation - HNSW vs product quantization based algorithms

- PQ-Algorithm: SOTA on billion scale datasets.
- Compare HNSW with SOTA PQ Algorithm in the library: Faiss.

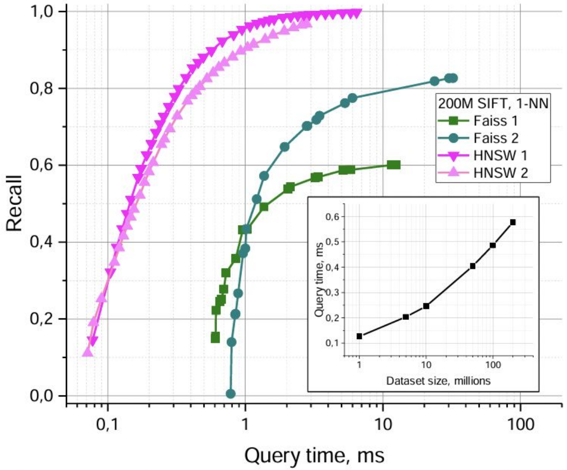


Fig. 15 Results of comparison with Faiss library on the 200M SIFT dataset from [13]. The inset shows the scaling of the query time vs the dataset size for Hierarchical NSW.

TABLE 3.
Parameters for comparison between Hierarchical NSW and Faiss on a 200M subset of 1B SIFT dataset.

Algorithm	Build time	Peak memory (runtime)	Parameters
Hierarchical NSW	5.6 hours	64 Gb	*M*=16, *efConstruction*=500 (1)
Hierarchical NSW	42 minutes	64 Gb	*M*=16, *efConstruction*=40 (2)
Faiss	12 hours	30 Gb	*OPQ64, IMI2x14, PQ64* (1)
Faiss	11 hours	23.5 Gb	*OPQ32, IMI2x14, PQ32* (2)

Conclusion

- HNSW provides a groundbreaking approach to nearest neighbor search, balancing speed and accuracy effectively even in challenging, high-dimensional spaces.

- The HNSW graph demonstrates robustness to various dataset that was not solvable by baseline NSW. It maintains good performance across different types of datasets without significant tradeoffs.

- This method sets a new benchmark for nearest neighbor searches, offering significant implications for machine learning and data retrieval.

Limitations

- Constructing and maintaining the HNSW graph can consume significant memory, especially for large datasets. This can limit the scalability of the method on memory-constrained systems or for applications with extremely large datasets.

- The search in the HNSW structure always starts from the top layer, thus the structure cannot be easily made distributed like baseline NSW.

Future Work

- The number of added connections per layer M can be a meaningful parameter to tune that strongly affects the construction of the index, thus might improve efficiency and effectiveness of HNSW.

- It would also be interesting to compare HNSW on the full 1B SIFT and 1B DEEP datasets and with functionalities such as element updates and removal.

- Design a distributed pipeline for speedup and memory optimization.

Thanks!