



MACHINE LEARNING

Similarity Search, Part 4: Hierarchical Navigable Small World (HNSW)

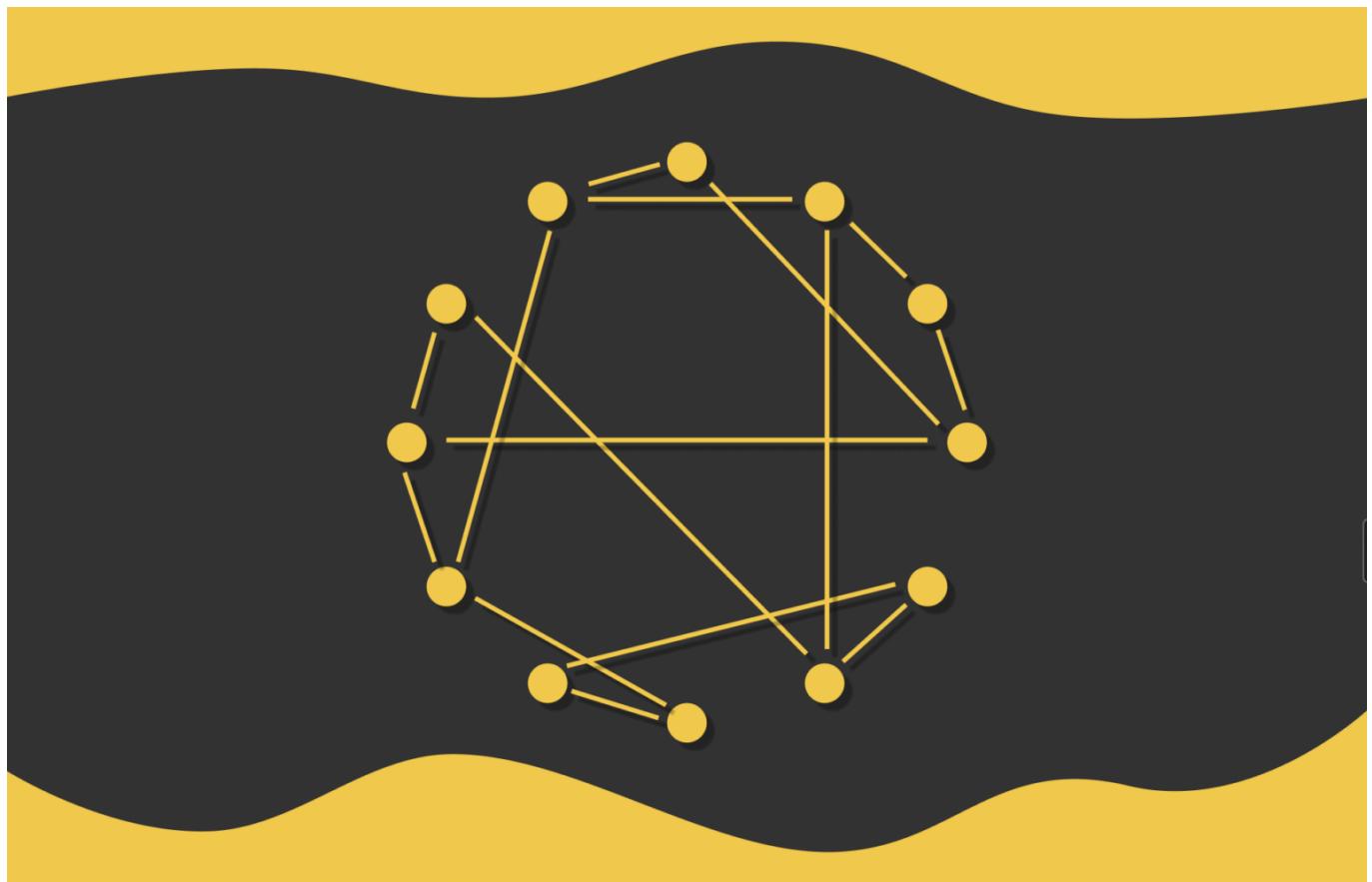
Hierarchical Navigable Small World (HNSW) is a state-of-the-art algorithm used for an approximate search of nearest neighbours. Under the...

Vyacheslav Efimov

Jun 16, 2023 • 17 min read



Discover how to construct efficient multi-layered graphs to boost search speed in massive volumes of data



Similarity search is a problem where given a query the goal is to find the most similar documents to it among all the database documents.

Introduction

In data science, similarity search often appears in the NLP domain, search engines or recommender systems where the most relevant documents or items need to be retrieved for a query. There exists a large variety of different ways to improve search performance in massive volumes of data.

Hierarchical Navigable Small World (HNSW) is a state-of-the-art algorithm used for an approximate search of nearest neighbours. Under the hood, HNSW constructs optimized graph structures making it very different from other approaches that were discussed in previous parts of this article series.

The main idea of HNSW is to construct such a graph where a path between any pair of vertices could be traversed in a small number of steps.

A well-known analogy on the famous six handshakes rule is related to this method:

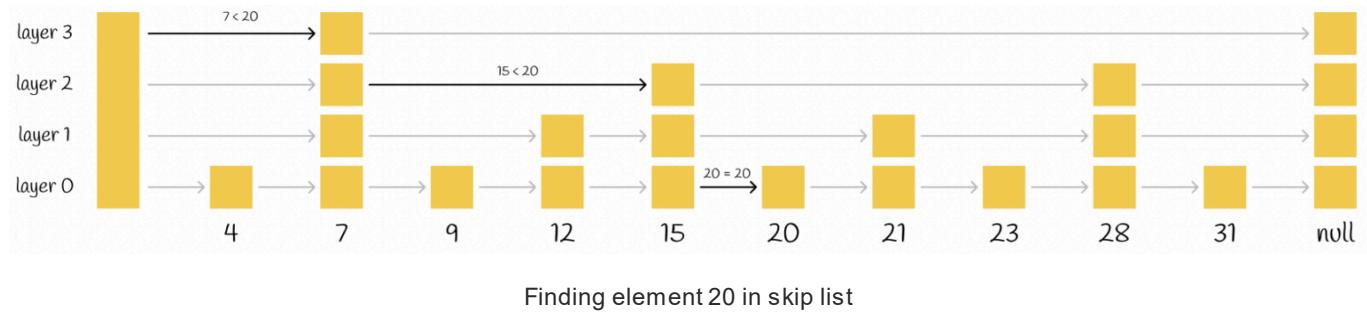
All people are six or fewer social connections away from each other.

Before proceeding to inner workings of HNSW let us first discuss skip lists and navigable small words – crucial data structures used inside the HNSW implementation.

Skip lists

Skip list is a probabilistic data structure that allows inserting and searching elements within a sorted list for $O(\log n)$ on average. A skip list is constructed by several layers of linked lists. The lowest layer has the original linked list

with all the elements in it. When moving to higher levels, the number of skipped elements increases, thus decreasing the number of connections.



The search procedure for a certain value starts from the highest level and compares its next element with the value. If the value is less or equal to the element, then the algorithm proceeds to its next element. Otherwise, the search procedure descends to the lower layer with more connections and repeats the same process. At the end, the algorithm descends to the lowest layer and finds the desired node.

Based on the information from [Wikipedia](#), a skip list has the main parameter p which defines the probability of an element appearing in several lists. If an element appears in layer i , then the probability that it will appear in layer $i + 1$ is equal to p (p is usually set to 0.5 or 0.25). On average, each element is presented in $1 / (1 - p)$ lists.

As we can see, this process is much faster than the normal linear search in the linked list. In fact, HNSW inherits the same idea but instead of linked lists, it uses graphs.

Navigable Small World

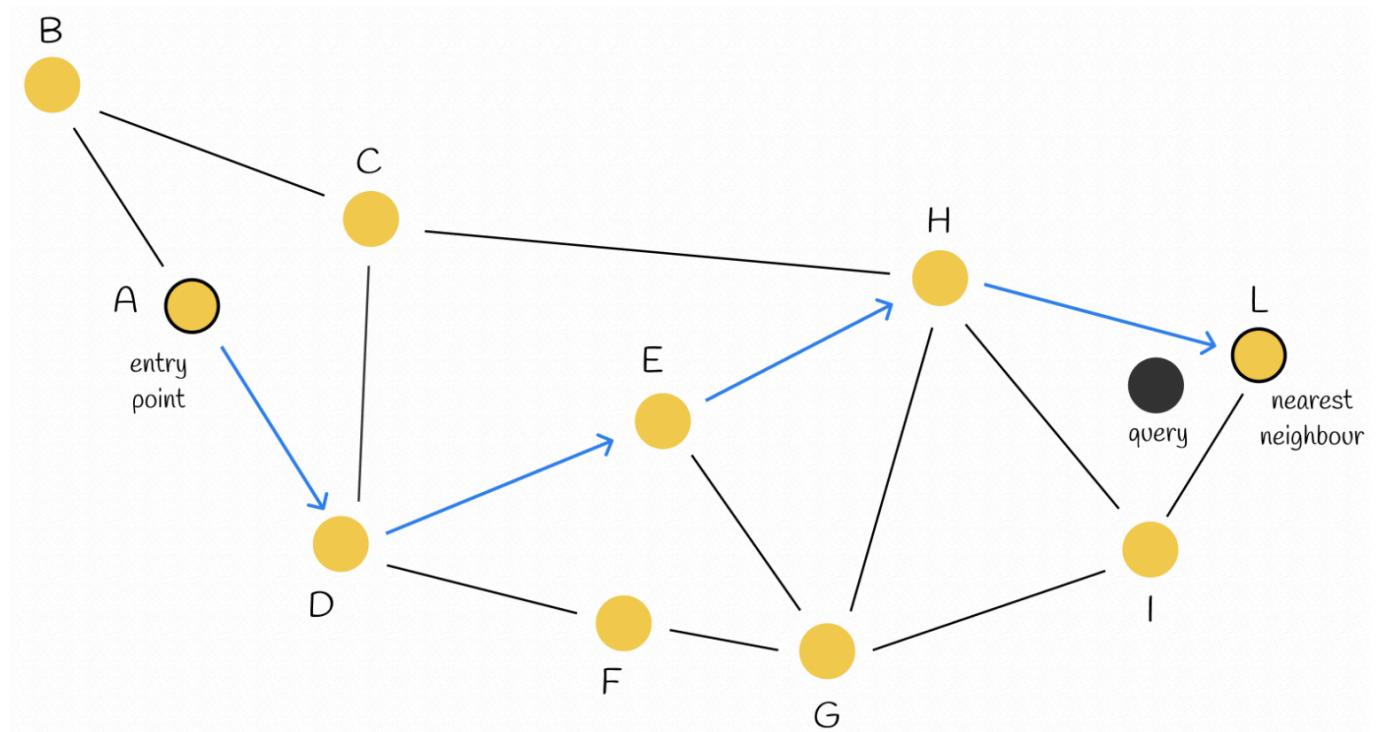
Navigable small world is a graph with polylogarithmic $T = O(\log^k n)$ search complexity which uses greedy routing. **Routing** refers to the process of starting the search process from low-degree vertices and ending with high-degree vertices. Since low-degree vertices have very few connections, the algorithm can rapidly move between them to efficiently navigate to the region

where the nearest neighbour is likely to be located. Then the algorithm gradually zooms in and switches to high-degree vertices to find the nearest neighbour among the vertices in that region.

Vertex is sometimes also referred to as a **node**.

Search

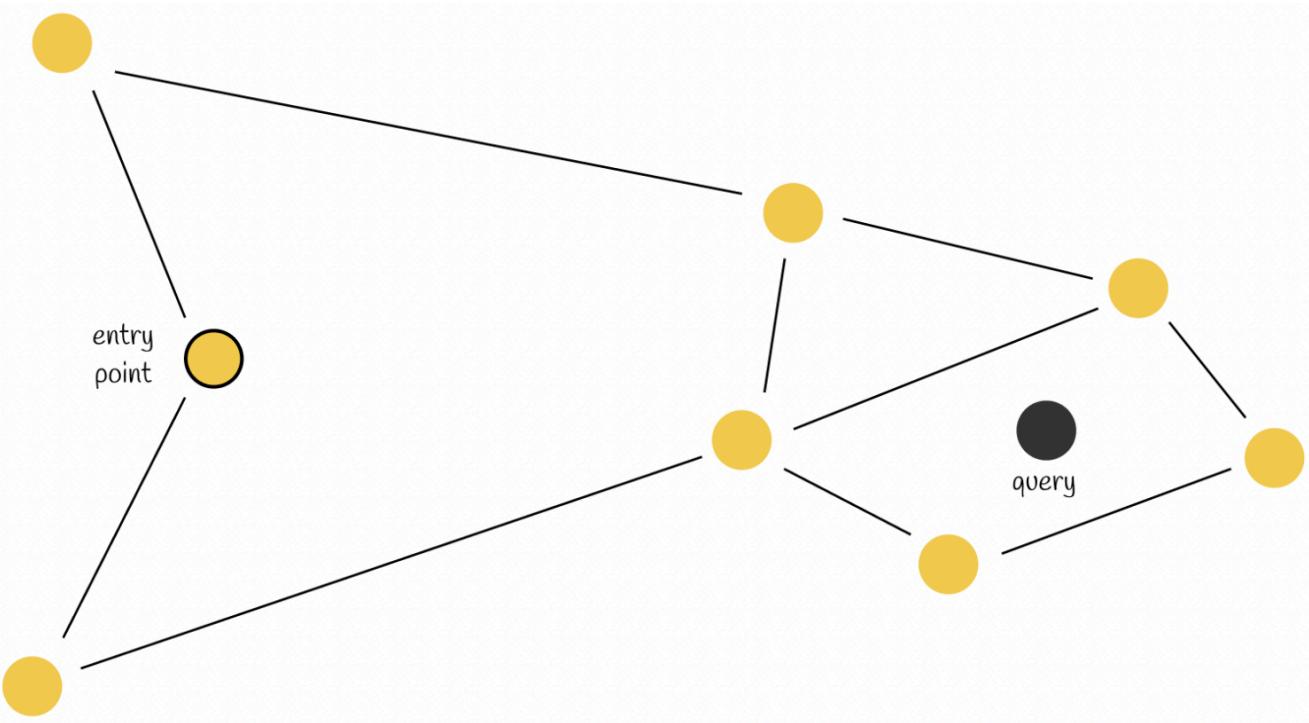
In the first place, search is proceeded by choosing an entry point. To determine the next vertex (or vertices) to which the algorithm makes a move, it calculates the distances from the query vector to the current vertex's neighbours and moves to the closest one. At some point, the algorithm terminates the search procedure when it cannot find a neighbour node that is closer to the query than the current node itself. This node is returned as the response to the query.



Greedy search process in a navigable small world. Node A is used as an entry point. It has two neighbours B and D. Node D is closer to the query than B. As a result, we move to D. Node D has three neighbours C, E and F. E is the closest neighbour to the query, so we move to E. Finally, the search process will lead to node L. Since all neighbours of L are located further from the query than L itself, we stop the algorithm and return L as the answer to the query.

This greedy strategy does not guarantee that it will find the exact nearest neighbour as the method uses only local information at the current step to

take decisions. **Early stopping** is one of the problems of the algorithm. It occurs especially at the beginning of the search procedure when there are no better neighbour nodes than the current one. For the most part, this might happen when the starting region has too many low-degree vertices.

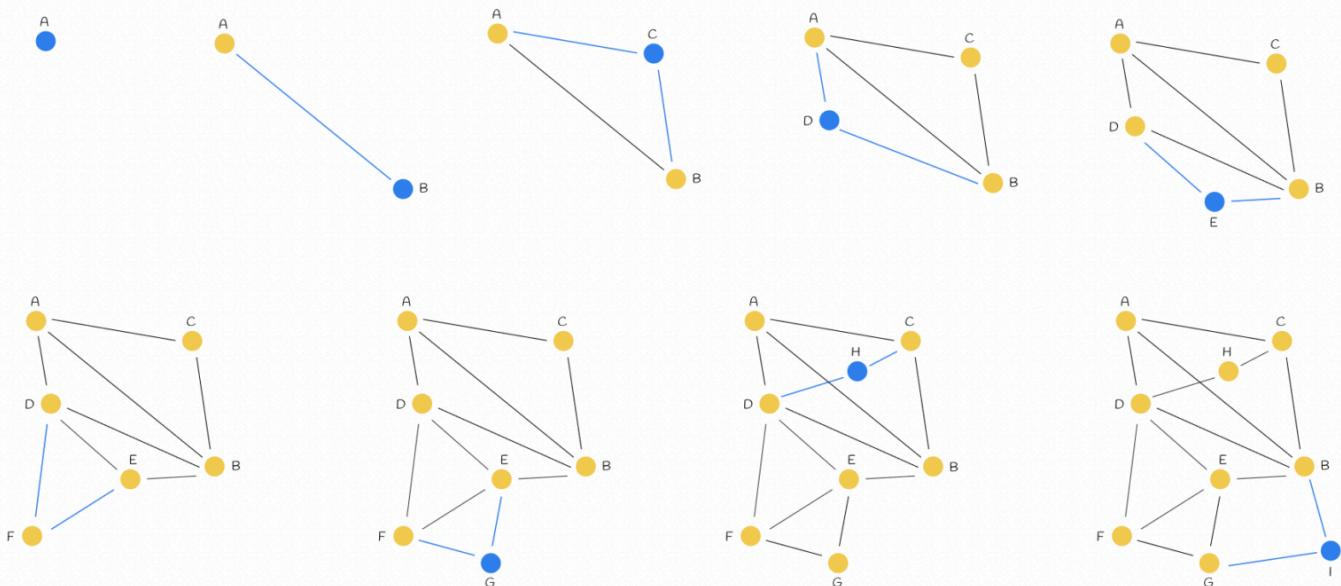


Early stopping. Both neighbours of the current node are further away from the query. Thus, the algorithm returns the current node as the response, though there exist much closer nodes to the query.

The search accuracy can be improved by using several entry points.

Construction

The NSW graph is built by shuffling dataset points and inserting them one by one in the current graph. When a new node is inserted, it is then linked by edges to the M nearest vertices to it.



Sequential insertion of nodes (from left to right) with $M = 2$. At each iteration, a new vertex is added to the graph and linked to its $M = 2$ nearest neighbours. Blue lines represent the connected edges to a newly inserted node.

In most scenarios, long-range edges will likely be created at the beginning phase of the graph construction. They play an important role in graph navigation.

Links to the closest neighbors of the elements inserted in the beginning of the 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. – Yu. A. Malkov, D. A. Yashunin

From the example in the figure above, we can see the importance of the long-range edge AB that was added in the beginning. Imagine a query requiring the traverse of a path from the relatively far-located nodes A and I . Having the edge AB allows doing it rapidly by directly navigating from one side of the graph to the opposite one.

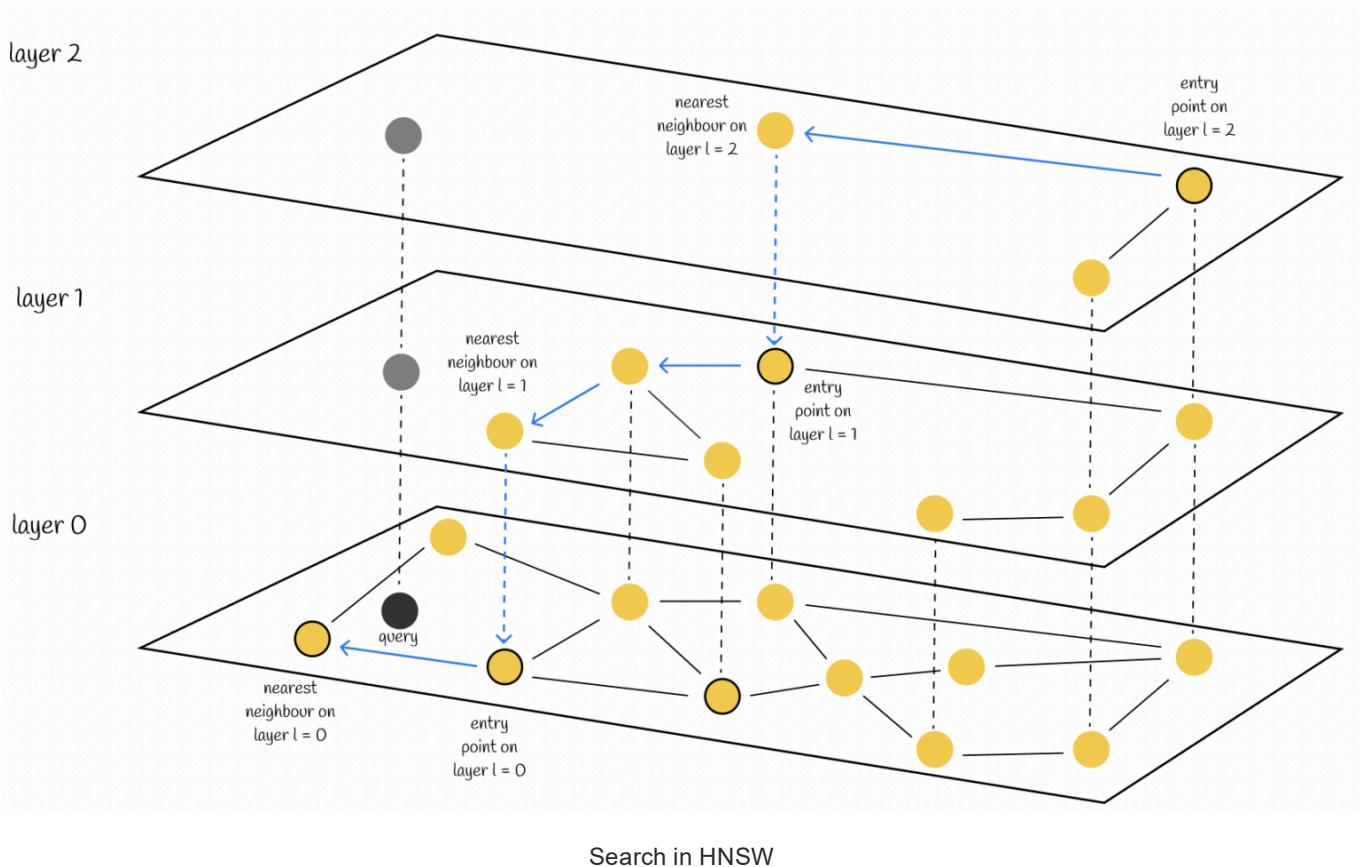
As the number of vertices in the graph increases, it increases the probability that the lengths of newly connected edges to a new node will be smaller.

HNSW

HNSW is based on the same principles as skip list and navigable small world. Its structure represents a multi-layered graph with fewer connections on the top layers and more dense regions on the bottom layers.

Search

The search starts from the highest layer and proceeds to one level below every time the local nearest neighbour is greedily found among the layer nodes. Ultimately, the found nearest neighbour on the lowest layer is the answer to the query.



Similarly to NSW, the search quality of HNSW can be improved by using several entry points. Instead of finding only one nearest neighbour on each layer, the `efSearch` (a hyperparameter) closest nearest neighbours to the query vector are found and each of these neighbours is used as the entry point on the next layer.

Complexity

The authors of the [original paper](#) claim that the number of operations required to find the nearest neighbour on any layer is bounded by a constant. Taking into consideration that the number of all layers in a graph is logarithmic, we get the total search complexity which is $O(\log n)$.

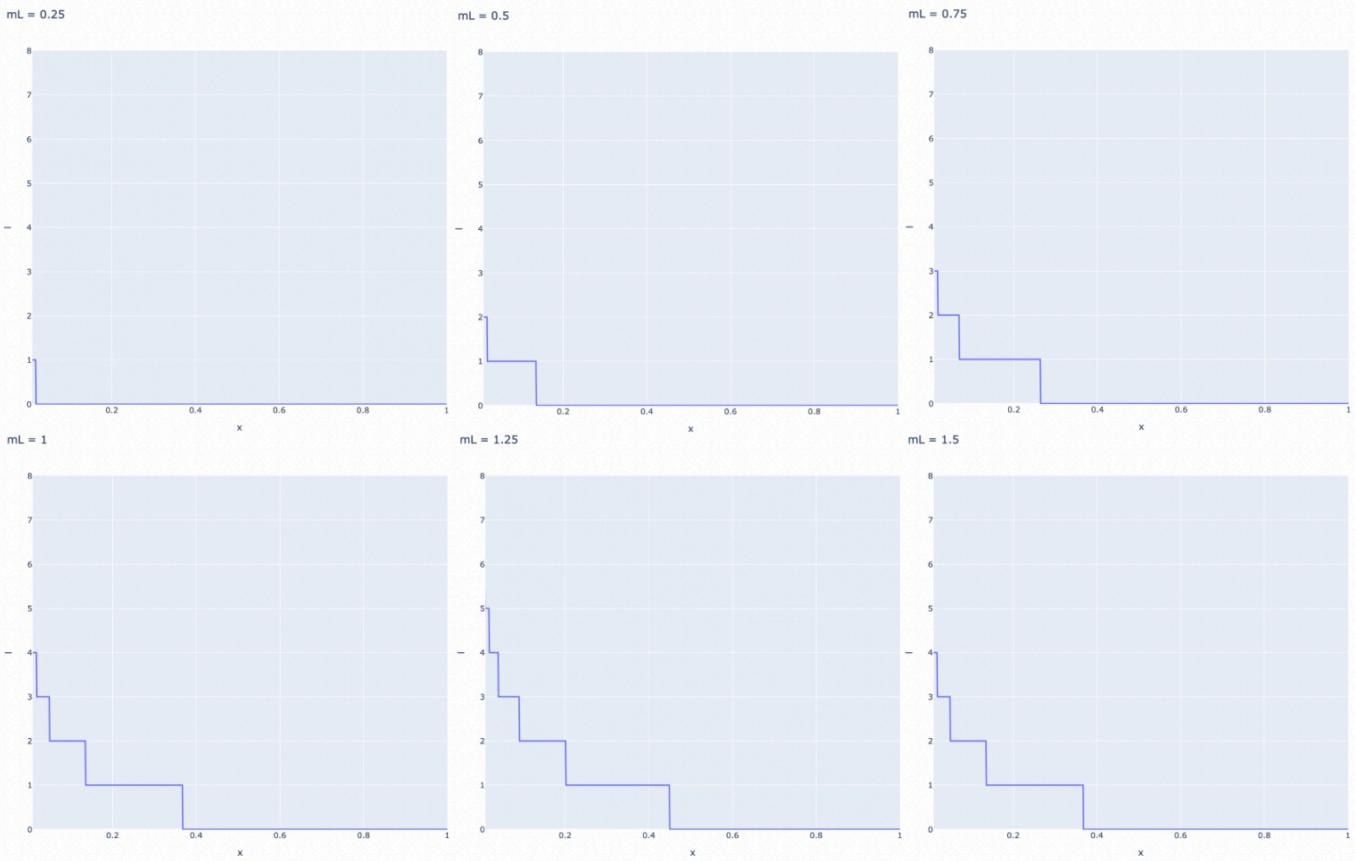
Construction

Choosing the maximum layer

Nodes in HNSW are inserted sequentially one by one. Every node is randomly assigned an integer l indicating the maximum layer at which this node can present in the graph. For example, if $l = 1$, then the node can only be found on layers 0 and 1. The authors select l randomly for each node with an *exponentially decaying probability distribution* normalized by the non-zero multiplier m_L ($m_L = 0$ results in a single layer in HNSW and non-optimized search complexity). Normally, the majority of l values should be equal to 0, so most of the nodes are present only on the lowest level. The larger values of m_L increase the probability of a node appearing on higher layers.

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

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



Distribution of the number of layers based on normalization factor mL . The horizontal axis represents values of the uniform $(0, 1)$ distribution.

To achieve the optimum performance advantage of the controllable hierarchy, the overlap between neighbors on different layers (i.e. percent of element neighbors that are also belong to other layers) has to be small.
– Yu. A. Malkov, D. A. Yashunin.

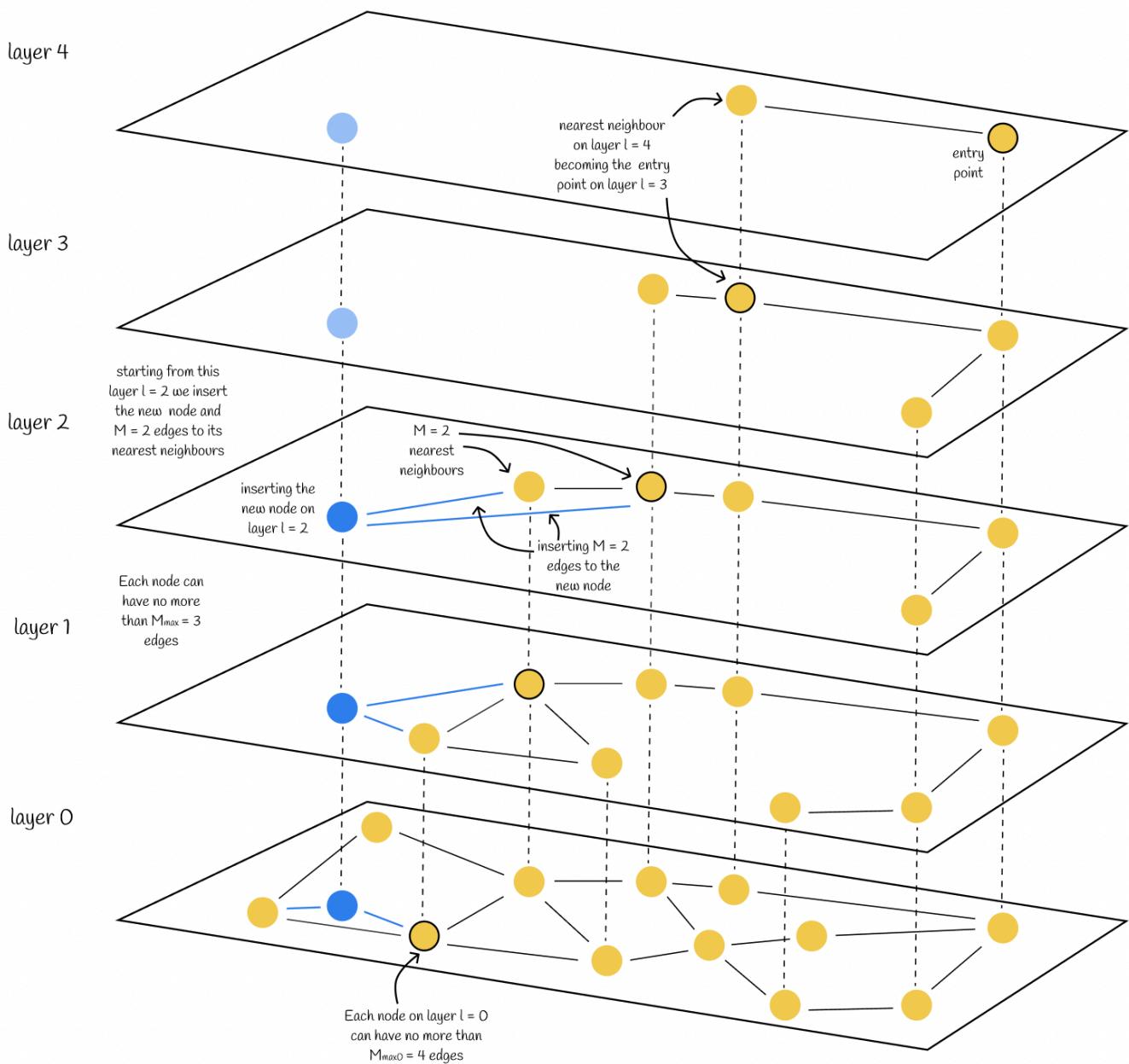
One of the ways to decrease the overlap is to decrease mL . But it is important to keep in mind that reducing mL also leads on average to more traversals during a greedy search on each layer. That is why it is essential to choose such a value of mL that will balance both the overlap and the number of traversals.

The authors of the paper propose choosing the optimal value of mL which is equal to $1 / \ln(M)$. This value corresponds to the parameter $p = 1 / M$ of the skip list being an average single element overlap between the layers.

Insertion

After a node is assigned the value l , there are two phases of its insertion:

1. The algorithm starts from the upper layer and greedily finds the nearest node. The found node is then used as an entry point to the next layer and the search process continues. Once the layer l is reached, the insertion proceeds to the second step.
2. Starting from layer l the algorithm inserts the new node at the current layer. Then it acts the same as before at step 1 but instead of finding only one nearest neighbour, it greedily searches for *efConstruction* (hyperparameter) nearest neighbours. Then M out of *efConstruction* neighbours are chosen and edges from the inserted node to them are built. After that, the algorithm descends to the next layer and each of found *efConstruction* nodes acts as an entry point. The algorithm terminates after the new node and its edges are inserted on the lowest layer 0.



Insertion of a node (in blue) in HNSW. The maximum layer for a new node was randomly chosen as $l = 2$. Therefore, the node will be inserted on layers 2, 1 and 0. On each of these layers, the node will be connected to its $M = 2$ nearest neighbours.

Choosing values for construction parameters

The original paper provides several useful insights on how to choose hyperparameters:

- According to simulations, good values for M lie between 5 and 48. Smaller values of M tend to be better for lower recalls or low-dimensional data while higher values of M are suited better for high recalls or high-dimensional data.

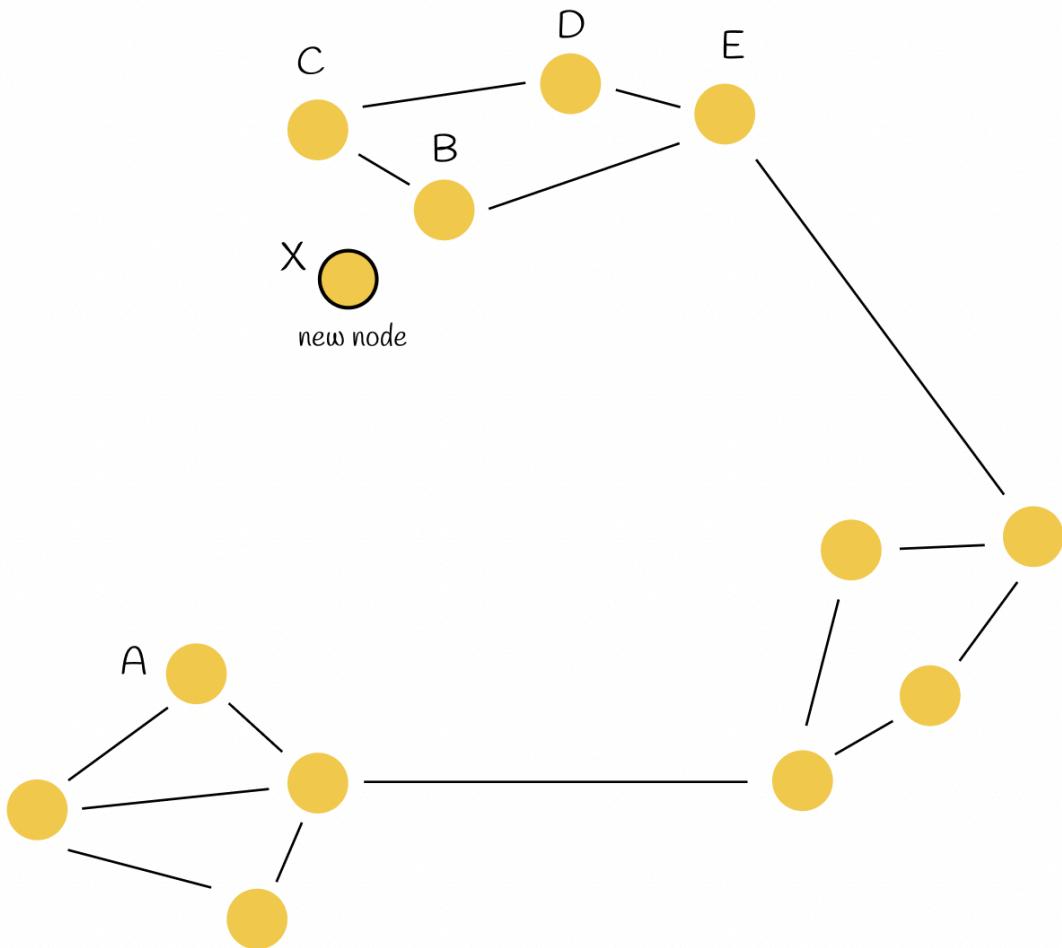
- Higher values of $efConstruction$ imply a more profound search as more candidates are explored. However, it requires more computations. Authors recommend choosing such an $efConstruction$ value that results at recall being close to 0.95–1 during training.
- Additionally, there is another important parameter M_{max} – the maximum number of edges a vertex can have. Apart from it, there exists the same parameter M_{max0} but separately for the lowest layer. It is recommended to choose a value for M_{max} close to $2 * M$. Values greater than $2 * M$ can lead to performance degradation and excessive memory usage. At the same time, $M_{max} = M$ results in poor performance at high recall.

Candidate selection heuristic

It was noted above that during node insertion, M out of $efConstruction$ candidates are chosen to build edges to them. Let us discuss possible ways of choosing these M nodes.

The naïve approach takes M closest candidates. Nevertheless, it is not always the optimal choice. Below is an example demonstrating it.

Imagine a graph with the structure in the figure below. As you can see, there are three regions with two of them not being connected to each other (on the left and on the top). As a result, getting, for example, from point A to B requires a long path through another region. It would be logical to somehow connect these two regions for better navigation.



Then a node X is inserted into the graph and needs to be linked to $M = 2$ other __ vertices.

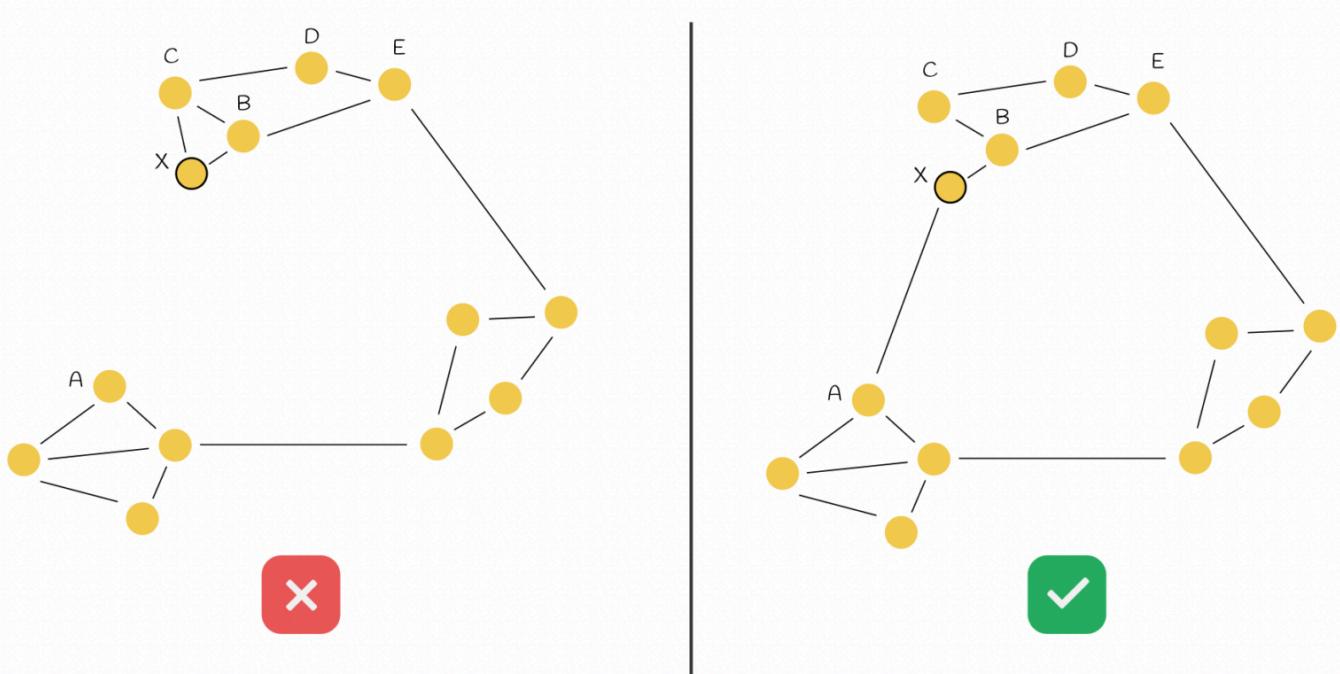
In this case, the naïve approach directly takes the $M = 2$ nearest neighbours (B and C) and connects X to them. Though X is connected to its real nearest neighbours, it does not solve the problem. Let us look at the heuristical approach invented by the authors.

The heuristic considers not only the closest distances between nodes but also the connectivity of different regions on the graph.

The heuristic chooses the first nearest neighbour (B in our case) and connects the inserted node (X) to it. Then the algorithm sequentially takes another most closest nearest neighbour in the sorted order (C) and builds an edge to it only if the distance from this neighbour to the new node (X) is smaller than any distance from this neighbour to all already connected

vertices (B) to the new node (X). After that, the algorithm proceeds to the next closest neighbour until M edges are built.

Getting back to the example, the heuristical procedure is illustrated in the figure below. The heuristic chooses B as the closest nearest neighbour for X and builds the edge BX . Then the algorithm chooses C as the next closest nearest neighbour. However, this time $BC < CX$. This indicates that adding the edge CX to the graph is not optimal because there already exists the edge BX and the nodes B and C are very close to each other. The same analogy proceeds with the nodes D and E . After that, the algorithm examines the node A . This time, it satisfies the condition since $BA > AX$. As a result, the new edge AX and both initial regions become connected to each other.



The example on the left uses the naïve approach. The example on the right uses the selection heuristic which results in two initial disjoint regions being connected to each other.

Complexity

The insertion process works very similarly, compared to the search procedure, without any significant differences which could require a non-constant number of operations. Thus, the insertion of a single vertex imposes $O(\log n)$ of time. To estimate the total complexity, the number of all inserted

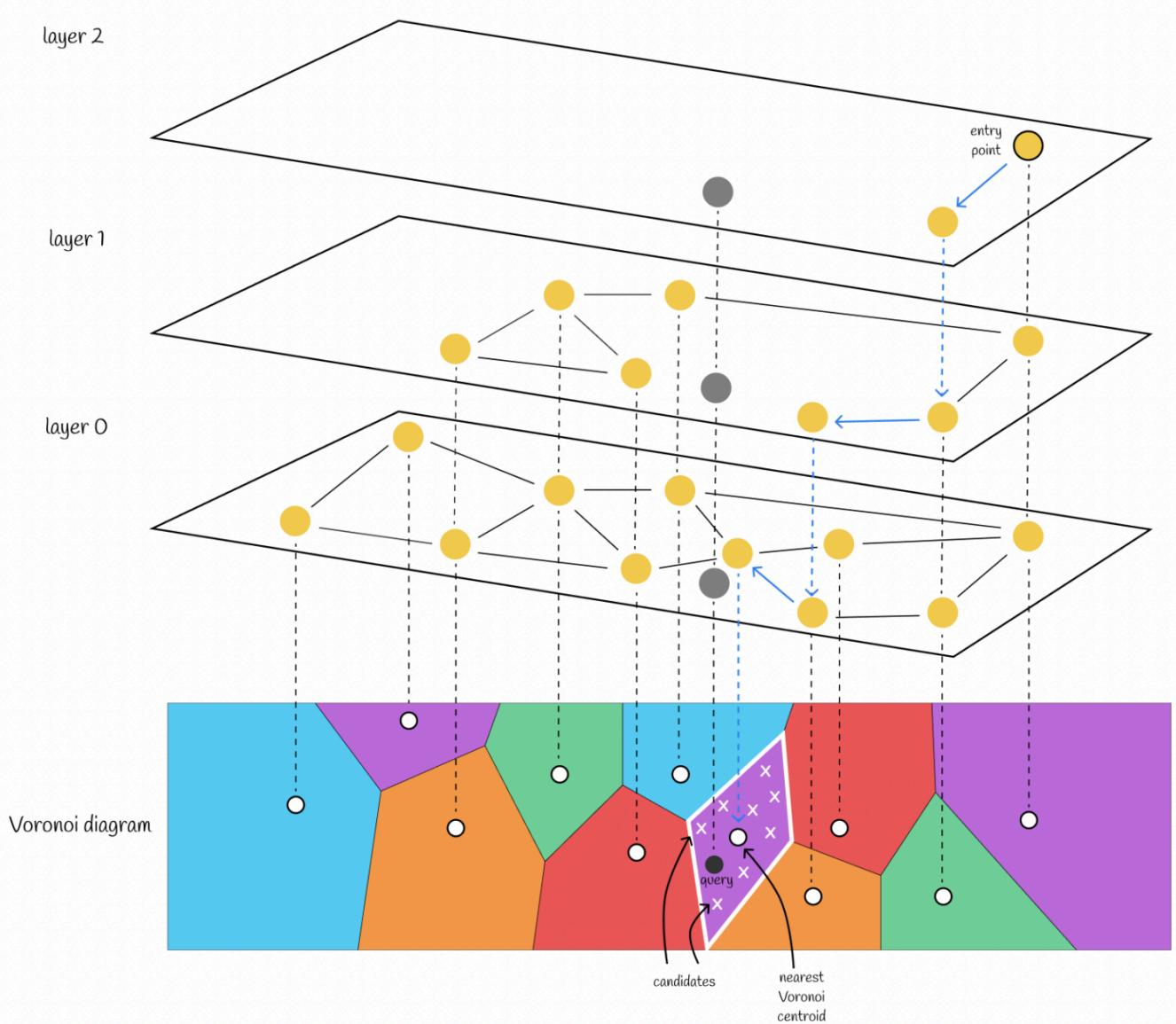
nodes n in a given dataset should be considered. Ultimately, HNSW construction requires $O(n * \log n)$ time.

Combining HNSW with other methods

HNSW can be used together with other similarity search methods to provide better performance. One of the most popular ways to do it is to combine it with an inverted file index and product quantization (*Index/VFPQ*) which were described in other parts of this article series.

Similarity Search, Part 3: Blending Inverted File Index and Product Quantization

Within this paradigm, HNSW plays the role of a **coarse quantizer** for *Index/VFPQ* meaning that it will be responsible for finding the nearest Voronoi partition, so the search scope can be reduced. To do it, an HNSW index has to be built on all Voronoi centroids. When given a query, HNSW is used to find the nearest Voronoi centroid (instead of brute-force search as it was previously by comparing distances to every centroid). After that, the query vector is quantized within a respective Voronoi partition and distances are calculated by using PQ codes.



Choosing the nearest Voronoi centroid by finding the nearest neighbour in HNSW built on top of Voronoi centroids.

When using only an inverted file index, it is better to set the number of Voronoi partitions not too large (256 or 1024, for instance) because brute-force search is performed to find the nearest centroids. By choosing a small number of Voronoi partitions, the number of candidates inside each partition becomes relatively large. Therefore, the algorithm rapidly identifies the nearest centroid for a query and most of its runtime is concentrated on finding the nearest neighbour inside a Voronoi partition.

However, introducing HNSW into the workflow requires an adjustment. Consider running HNSW only on a small number of centroids (256 or 1024): HNSW would not bring any significant benefits because, with a small number of vectors, HNSW performs relatively the same in terms of execution time as

naïve brute-force search. Moreover, HNSW would require more memory to store the graph structure.

That is why when merging HNSW and inverted file index, it is recommended to set the number of Voronoi centroids much bigger than usual. By doing so, the number of candidates inside each Voronoi partition becomes much smaller.

This shift in paradigm results in the following settings:

- HNSW rapidly identifies the nearest Voronoi centroids in logarithmic time.
- After that, an exhaustive search inside respective Voronoi partitions is performed. It should not be a trouble because the number of potential candidates is small.

Faiss implementation

Faiss (Facebook AI Search Similarity) is a Python library written in C++ used for optimised similarity search. This library presents different types of indexes which are data structures used to efficiently store the data and perform queries.

Based on the information from the [Faiss documentation](#), we will see how HNSW can be utilized and merged together with inverted file index and product quantization.

IndexHNSWFlat

Faiss has a class *IndexHNSWFlat* implementing the HNSW structure. As usual, the suffix "*Flat*" indicates that dataset vectors are fully stored in index. The constructor accepts 2 parameters:

- **d**: data dimensionality.

- **M**: the number of edges that need to be added to every new node during insertion.

Additionally, via the **hnsw** field, *IndexHNSWFlat* provides several useful attributes (which can be modified) and methods:

- **hnsw.efConstruction**: number of nearest neighbours to explore during construction.
- **hnsw.efSearch**: number of nearest neighbours to explore during search.
- **hnsw.max_level**: returns the maximum layer.
- **hnsw.entry_point**: returns the entry point.
- **faiss.vector_to_array(index.hnsw.levels)**: returns a list of maximum layers for each vector
- **hnsw.set_default_probas(M: int, level_mult: float)**: allows setting M and mL values respectively. By default, $_levelmult$ is set to $1 / \ln(M)$.

```

● ● ●

1 d = 256 # data dimension
2 M = 32 # number of edges used to connect to each inserted vertex
3
4 index = faiss.IndexHNSWFlat(d, M)
5 index.hnsw.efConstruction = 64 # number of entry points during search
6 index.hnsw.efSearch = 32 # number of entry points during construction
7 index.set_default_probas(32, 0.4) # setting M and normalization factor mL
8
9 index.add(data)
10
11 print(f"Maximum layer: {index.hnsw.max_level}")
12 print(f"Entry point: {index.hnsw.entry_point}")
13
14 levels = faiss.vector_to_array(index.hnsw.levels)
15 print(f"Maximum layer for each vector: {levels}")
16 print(f"Distribution of levels: {np.bincount(levels)}")
17
18 k = 3 # how many nearest neighbours to search for
19 D, I = index.search(query, k)

```

Faiss implementation of IndexHNSWFlat

IndexHNSWFlat sets values for $M_{max} = M$ and $M_{max0} = 2 * M$.

IndexHNSWFlat + IndexIVFPQ

IndexHNSWFlat can be combined with other indexes as well. One of the examples is *IndexIVFPQ* described in the previous part. Creation of this composite index proceeds in two steps:

1. *IndexHNSWFlat* is initialized as a coarse quantizer.
2. The quantizer is passed as a parameter to the constructor of *IndexIVFPQ*.

Training and adding can be done by using different or the same data.

```
● ● ●
1 d = 256 # data dimension
2 M = 64 # number of edges used to connect to each inserted vertex
3 nlist = 8192 # number of Voronoi partitions
4 nsubspaces = 32 # number of subspaces to split each vector
5 nbits = 8 # number of bits required to encode a single cluster ID in each subspace
6
7 quantizer = faiss.IndexHNSWFlat(d, M)
8 index = faiss.IndexIVFPQ(quantizer, d, nlist, nsubspaces, nbites)
9
10 index.train(data)
11 index.add(data)
12
13 index.nprobe = 10 # how many Voronoi partitions to use to search for candidates
14
15 k = 3 # how many nearest neighbours to search for
16 D, I = index.search(query, k)
```

FAISS implementation of IndexHNSWFlat + IndexIVFPQ

Conclusion

In this article, we have studied a robust algorithm which works especially well for large dataset vectors. By using multi-layer graph representations and the candidate selection heuristic its search speed scales efficiently while

maintaining a decent prediction accuracy. It is also worth noting that HNSW can be used in combination with other [Similarity Search](#) algorithms making it very flexible.

Resources

- [Six degrees of separation | Wikipedia](#)
- [Skip List | Wikipedia](#)
- [Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. Yu. A. Malkov, D. A. Yashunin](#)
- [Faiss documentation](#)
- [Faiss repository](#)
- [Summary of Faiss indexes](#)

All images unless otherwise noted are by the author.

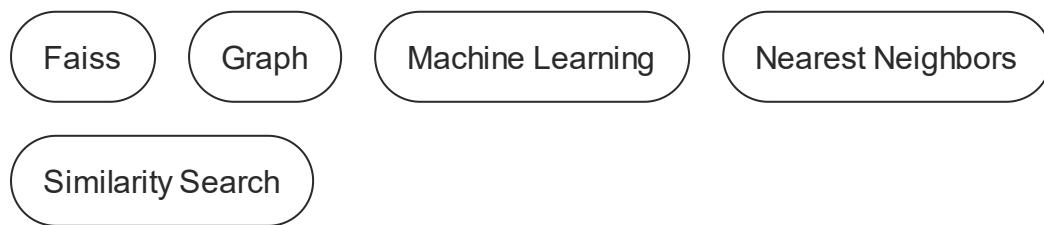
▪ ▪ ▪

WRITTEN BY

Vyacheslav Efimov

[See all from Vyacheslav Efimov >](#)

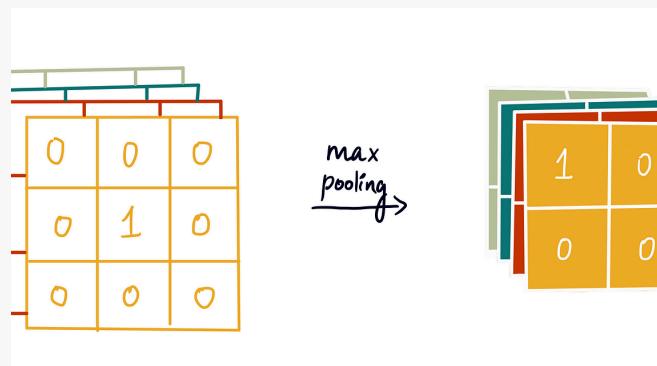
Topics:



Share this article:



Related Articles



ARTIFICIAL INTELLIGENCE

Implementing Convolutional Neural Networks in TensorFlow

Step-by-step code guide to building a Convolutional Neural Network

Shreya Rao

August 20, 2024 • 6 min read



ARTIFICIAL INTELLIGENCE

How to Forecast Hierarchical Time Series

A beginner's guide to forecast reconciliation

Dr. Robert Kübler

August 20, 2024 • 13 min read



DATA SCIENCE

Hands-on Time Series Anomaly Detection using Autoencoders, with Python

Here's how to use Autoencoders to detect signals with anomalies in a few lines of...



MACHINE LEARNING

3 AI Use Cases (That Are Not a Chatbot)

Feature engineering, structuring unstructured data, and lead scoring

Shaw Talebi

Piero Paialunga

August 21, 2024 • 12 min read



DATA SCIENCE

Back To Basics, Part Uno: Linear Regression and Cost Function

An illustrated guide on essential machine learning concepts

Shreya Rao

February 3, 2023 • 6 min read

August 21, 2024 • 7 min read

DATA SCIENCE

Must-Know in Statistics: The Bivariate Normal Projection Explained

Derivation and practical examples of this powerful concept

Luigi Battistoni

August 14, 2024 • 7 min read



DATA SCIENCE

Our Columns

Columns on TDS are carefully curated collections of posts on a particular idea or category...

TDS Editors

November 14, 2020 • 4 min read



DATA SCIENCE

Optimizing Marketing Campaigns with Budgeted Multi-Armed Bandits

With demos, our new solution, and a video

Vadim Arzamasov

August 16, 2024 • 10 min read

DATA SCIENCE

Back to Basics, Part Tres: Logistic Regression

An illustrated guide to everything you need to know about Logistic Regression

Shreya Rao

March 2, 2023 • 8 min read



towards data science

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

© Insight Media Group, LLC 2025

[Subscribe to Our Newsletter >](#)

[ABOUT](#) • [ADVERTISE](#) • [PRIVACY POLICY](#) • [TERMS OF USE](#)

[COOKIES SETTINGS](#)