

Approximate Nearest Neighbours

IN3120

Based on [this](#) and [this](#)

Overview

What we'd like:

- We'd like to do k -NN lookups in high-dimensional spaces
- We'd like to do this efficiently when we have a gazillion data points
- We might be willing to do it approximately, and sacrifice exactness for efficiency

What are some of the strategies we can apply?

- Brute force
- Tree-based algorithms
- Locality-sensitive hashing
- Quantization
- Graph-based algorithms

When should we choose what?

- Rules of thumb that depend on application requirements, dimensionality, and volume

Voronoi Diagrams

Voronoi diagram

[Article](#) [Talk](#)

From Wikipedia, the free encyclopedia

In **mathematics**, a **Voronoi diagram** is a [partition](#) of a [plane](#) into regions close to each of a given set of objects. In the simplest case, these objects are just finitely many points in the plane (called seeds, sites, or generators). For each seed there is a corresponding [region](#), called a **Voronoi cell**, consisting of all points of the plane closer to that seed than to any other. The Voronoi diagram of a set of points is [dual](#) to that set's [Delaunay triangulation](#).

The Voronoi diagram is named after mathematician [Georgy Voronoy](#), and is also called a **Voronoi tessellation**, a **Voronoi decomposition**, a **Voronoi partition**, or a **Dirichlet tessellation** (after [Peter Gustav Lejeune Dirichlet](#)). Voronoi cells are also known as **Thiessen polygons**.^{[1][2][3]} Voronoi diagrams have practical and theoretical applications in many fields, mainly in [science](#) and [technology](#), but also in [visual art](#).^{[4][5]}

Illustration [\[edit \]](#)

As a simple illustration, consider a group of shops in a city. Suppose we want to estimate the number of customers of a given shop. With all else being equal (price, products, quality of service, etc.), it is reasonable to assume that customers choose their preferred shop simply by distance considerations: they will go to the shop located nearest to them. In this case the Voronoi cell R_k of a given shop P_k can be used for giving a rough estimate on the number of potential customers going to this shop (which is modeled by a point in our city).

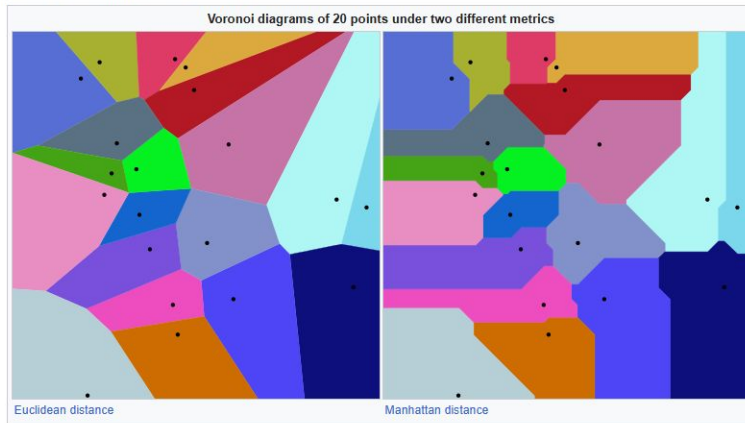
For most cities, the distance between points can be measured using the familiar [Euclidean distance](#):

$$\ell_2 = d[(a_1, a_2), (b_1, b_2)] = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$$

or the [Manhattan distance](#):

$$d[(a_1, a_2), (b_1, b_2)] = |a_1 - b_1| + |a_2 - b_2|.$$

The corresponding Voronoi diagrams look different for different distance metrics.



Brute Force

- Do a full scan and get exact results
- Optimized implementations make heavy use of SIMD instructions and GPUs
- Application requirements, dimensionality and volume dictates if this is feasible

$\|x - y\|_2^2$ by SIMD

```
float fvec_l2sqr(const float * x,
               const float * y,
               size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps(x); x += 8;
        __m256 my = _mm256_loadu_ps(y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

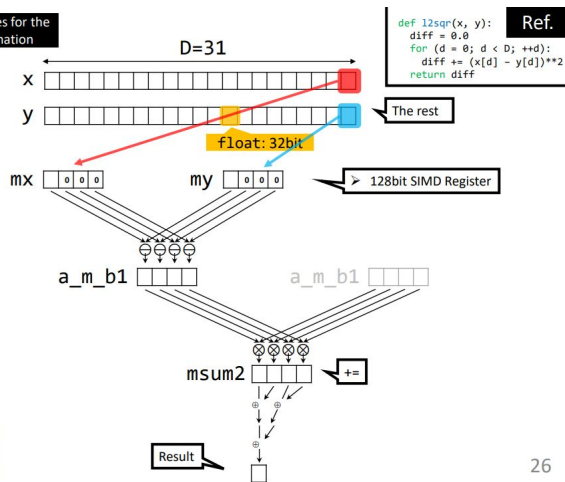
    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps(x); x += 4;
        __m128 my = _mm_loadu_ps(y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = _mm_loadu_ps(x); x += 1;
        __m128 my = _mm_loadu_ps(y); y += 1;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps(msum2, msum2);
    msum2 = _mm_hadd_ps(msum2, msum2);
    return _mm_cvtss_f32(msum2);
}
```

Rename variables for the sake of explanation



```
def l2sqr(x, y):
    diff = 0.0
    for (d = 0; d < D; ++d):
        diff += (x[d] - y[d])**2
    return diff
```

26

NN in GPU (faiss-gpu) is 10x faster than NN in CPU (faiss-cpu)

Benchmark: <https://github.com/facebookresearch/faiss/wiki/Low-level-benchmarks>

➤ NN-GPU always compute $\|q\|_2^2 - 2q^T x + \|x\|_2^2$

➤ k-means for 1M vectors ($D=256$, $K=20000$)

✓ 11 min on CPU

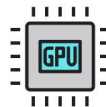
✓ 55 sec on 1 Pascal-class P100 GPU (float32 math)

✓ 34 sec on 1 Pascal-class P100 GPU (float16 math)

✓ 21 sec on 4 Pascal-class P100 GPUs (float32 math)

✓ 16 sec on 4 Pascal-class P100 GPUs (float16 math)

x10 faster



➤ If GPU is available and its memory is enough, try GPU-NN

➤ The behavior is little bit different (e.g., a restriction for top-k)

30

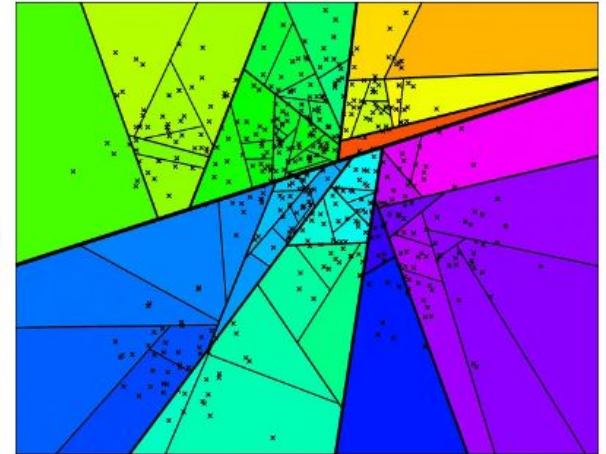
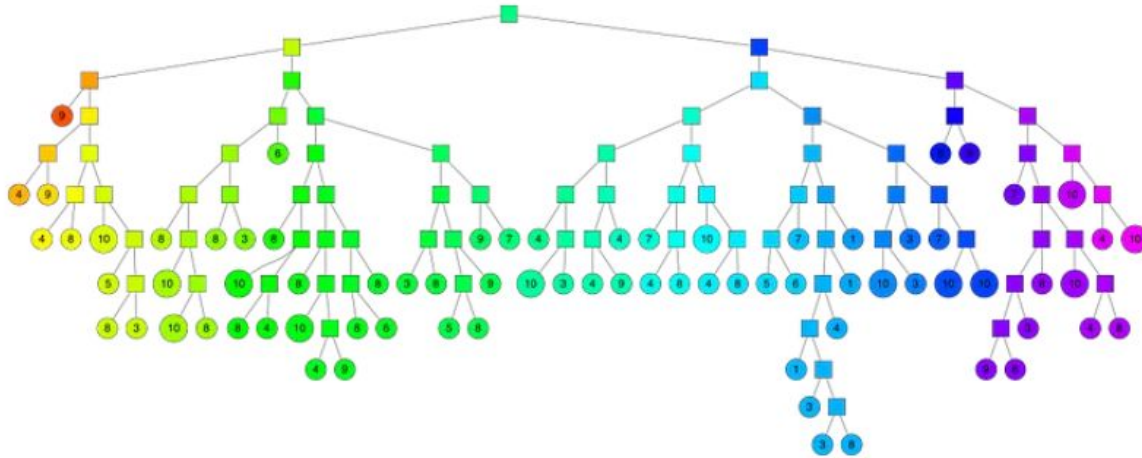
Tree-Based Algorithms

Indexing:

- Construct a tree by recursively partitioning the data, until the leaf nodes hold “few enough” data points
- Construct a forest of trees

Querying:

- Start at the query point
- Select one or more trees, follow the branches
- Consider the points located in the regions you end up



See also [this](#) tutorial.

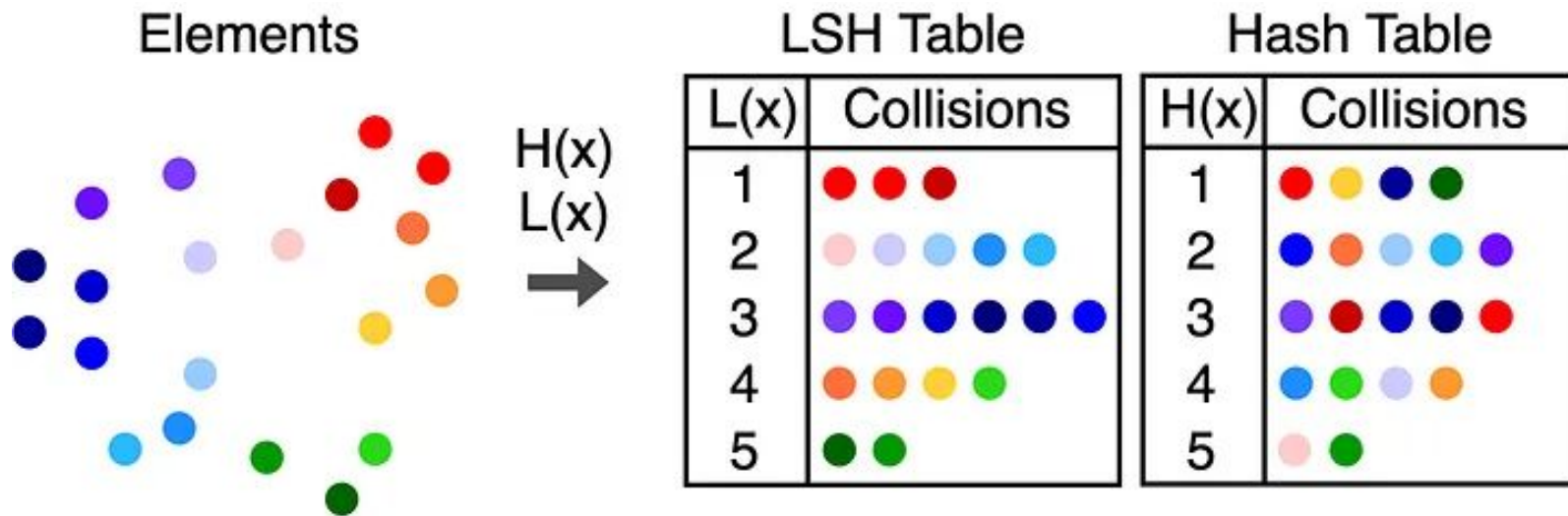
Locality-Sensitive Hashing

Indexing:

- Apply multiple hash functions h to each point to bucket them
- Distance $d(x, y)$ is small $\rightarrow \Pr(h(x) = h(y))$ is high

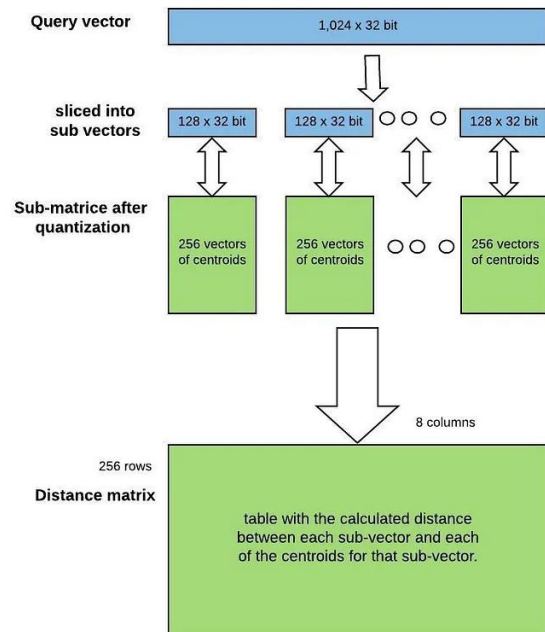
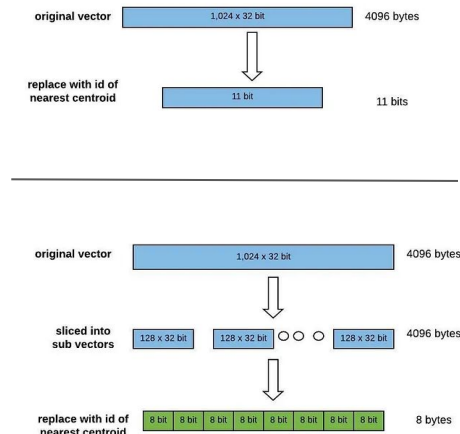
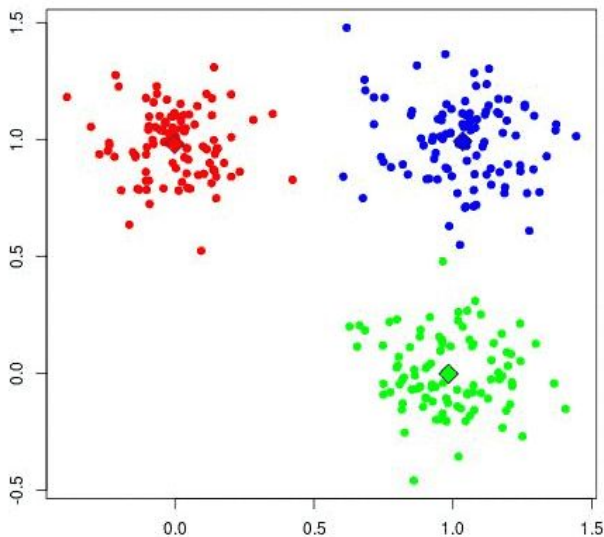
Querying:

- Apply the hashes to the query point
- Consider the points in the buckets we hash to



Quantization

- Recode (cluster) the vectors to reduce the size of the dataset
- Replace each vector with a leaner, approximate and quantized representation (clusters)
- Can be combined with an inverted index



See also [this](#) tutorial.

Graph-Based Methods

Hierarchical Navigable Small World Graphs

The intuition of this method is as follows, in order to reduce the search time on a graph we would want our graph to have an average path.

This is strongly connected to the famous “*six handshake rule*” statement.

“There is at most 6 degrees of separation between you and anyone else on Earth.” — [Frigyes Karinthy](#)

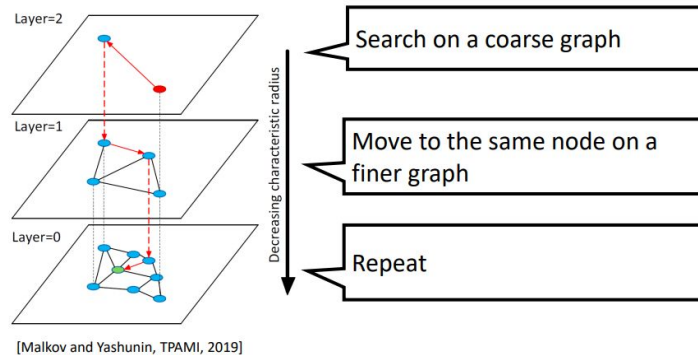
Many real-world graphs on average are highly clustered and tend to have nodes that are close to each other which are formally called small-world graph:

- highly transitive (community structure) it's often hierarchical.
- small average distance $\sim \log(N)$.

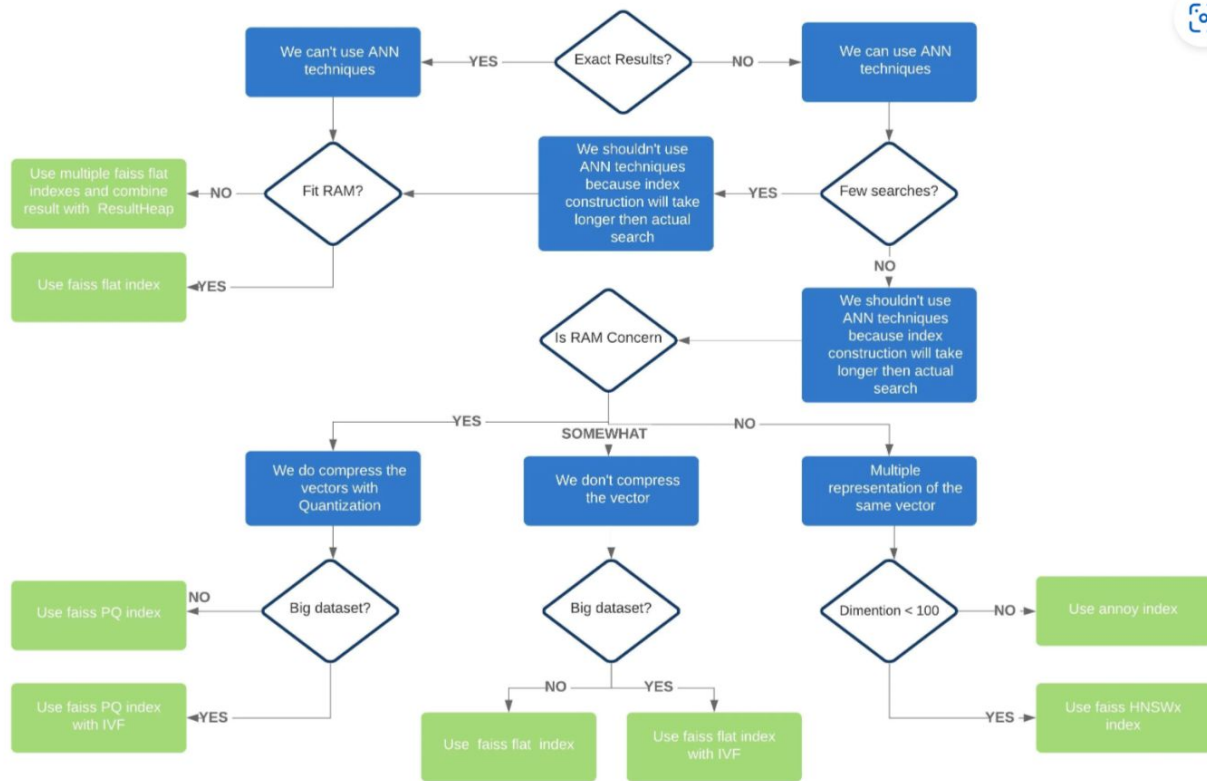
In order to search, we start at some entry point and iteratively traverse the graph. At each step of the traversal, the algorithm examines the distances from a query to the neighbors of a current base node and then selects as the next base node the adjacent node that minimizes the distance, while constantly keeping track of the best-discovered neighbors. The search is terminated when some stopping condition is met.

Extension: Hierarchical NSW; HNSW

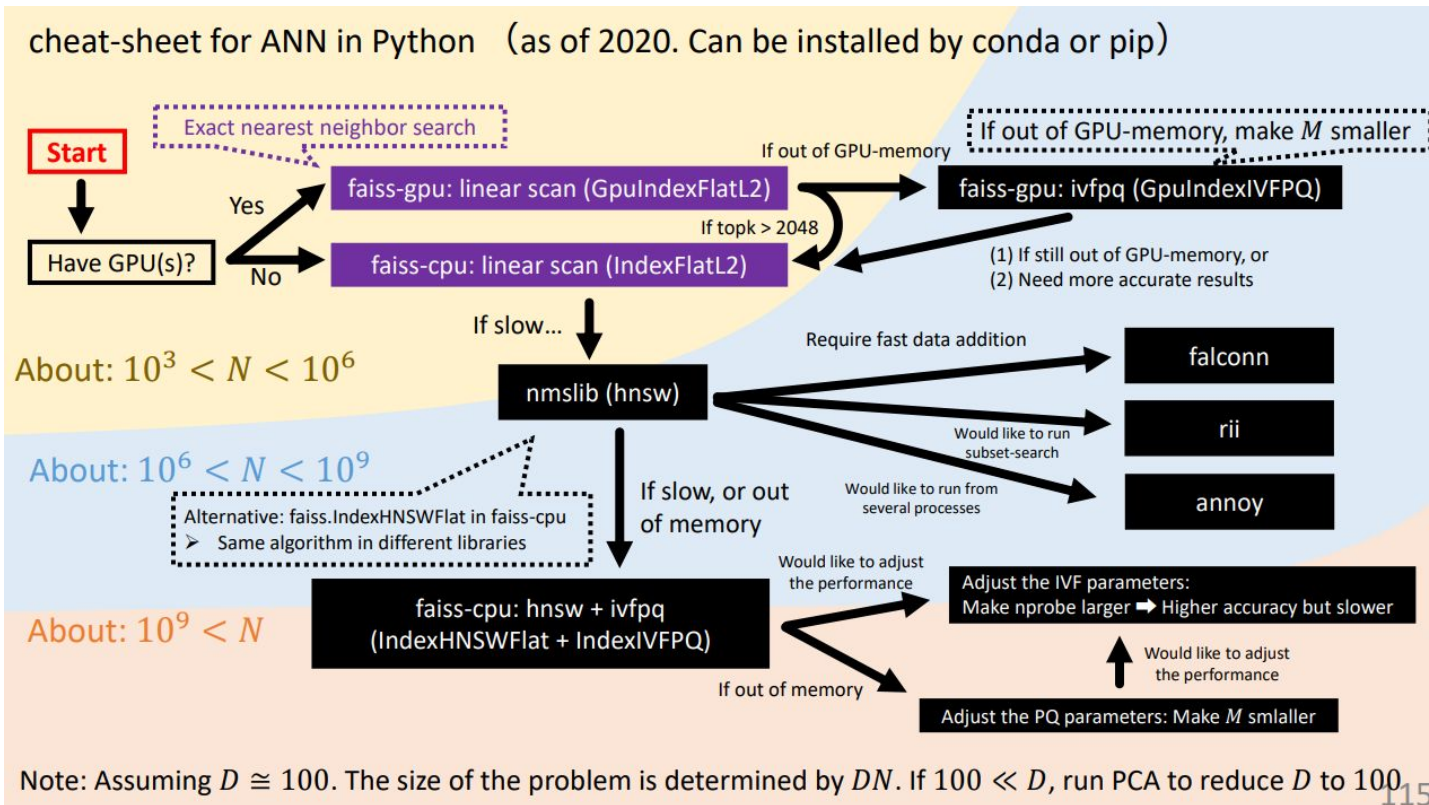
- Construct the graph hierarchically [Malkov and Yashunin, TPAMI, 2019]
- This structure works pretty well for real-world data



Selection



Selection, cont.



Selection, cont.

