DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node

S.J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, H.V. Simhadri (2019)

Presented by Jason Liu

K-Nearest Neighbors (search)

- Given P points and query q, find the nearest k points to q
- Highly useful for recommendation systems and many machine learning applications
- In this paper, points are assumed to be in high dimensional Euclidean space
- Goal: solve kNN with ms level latency on billions of points in hundreds of dimensions on a standard workstation
- In order to facilitate many queries, need to construct a small low latency data structure for solving this problem

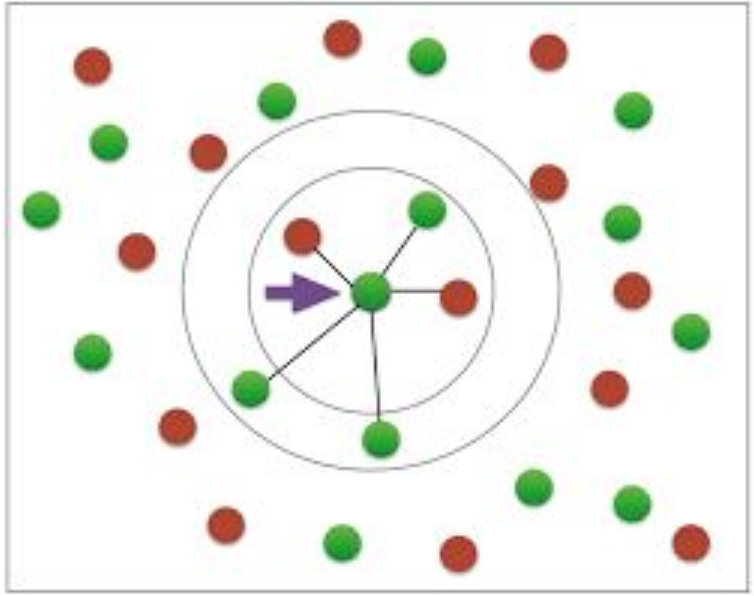


Image credits:
https://towardsdatascience.com/building-a-k-nearest-neighbors-k-nn-model-with-scikit-learn-51209555453a

Approximate KNN

- Approximations are easier than finding exactly the k nearest neighbors
- X-recall@Y measures the fraction of the X nearest neighbors that show up within an output list of Y candidates
- Ideally algorithms perform well for k-recall@k
- Approaches include k-d trees, which perform poorly for high dimensions, and Locality Sensitive Hashing, which is bad at exploiting distributional quirks
- Conventional wisdom: things should usually fit in main memory, as disks are too slow
    - SSDs take several hundred microseconds per random read: only 5 to 10 accesses are possible if we want 5-ms latency

Prior SOTA approaches

- Inverted Index and Data Compression
  - Split data into M partitions and compare query to a few of the nearest partitions
  - Uses compressed form of points so that all points fit in main memory
  - Less than 64 GB of memory needed for 1B points, sub-5 ms results, but also 1-recall@1 is about 0.5
  - Can achieve good 1-recall@100 values
- Shard-based approach
  - Split into many shards, with one index per shard. Send queries to multiple candidate shards in order to find all nearest neighbors
  - Points are uncompressed, so only shards fit in memory, not the whole thing: even 100M points needs 75 GB memory
  - Achieves 0.98 100-recall@100 with sub-5 ms latencies on 2B points

Graph-based indexing algorithms

- Naive distance computations take too long: too many distances to compute
- Graphical techniques form a sparse graph on the points.
- Given a source point, compute a neighbor to a given point by repeatedly adding neighbors from the nearest candidate into the candidate pool
- Converges so long as SNG property holds: for any source s and point p either s and p are adjacent or there is a neighbor of p closer to both s and p.

Algorithm 1: GreedySearch(s, x_q, k, L)

Data: Graph G with start node s, query x_q, result size k, search list size $L \geq k$

Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

```
begin
    initialize sets L ← {s} and V ← ∅
    while L \ V ≠ ∅ do
        let p* ← arg min_{p∈L\V} ||x_p − x_q||
        update L ← L ∪ N_out(p*) and
          V ← V ∪ {p*}
        if |L| > L then
            update L to retain closest L
              points to x_q
    return [closest k points from L; V]
```

RobustPrune technique

- SNG graphs have too many hops: as many as O(n) in a 1d graph.
- RobustPrune with alpha > 1 ensures that if s, p aren't adjacent then there is a closer point adjacent to p whose distance from s is a fraction of the distance from s to p
- Leads to logarithmic hop count
- Construction is too slow when candidate set is large

Algorithm 2: $\mathrm{RobustPrune}(p, \mathcal{V}, \alpha, R)$

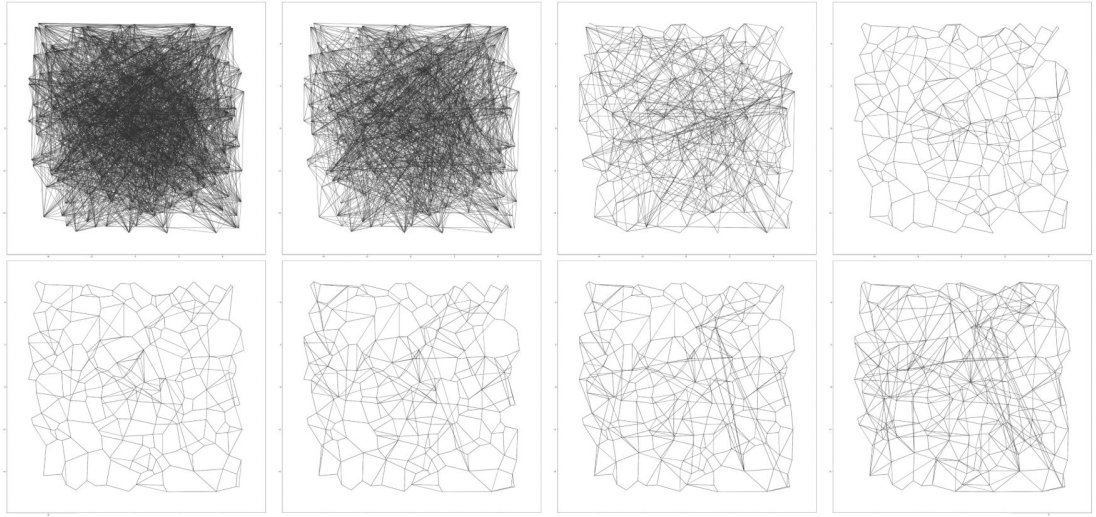
Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R

Result: G is modified by setting at most R new out-neighbors for p

```
begin
    V ← (V ∪ N_out(p)) \ {p}
    N_out(p) ← ∅
    while V ≠ ∅ do
        p* ← arg min_{p'∈V} d(p, p')
        N_out(p) ← N_out(p) ∪ {p*}
        if |N_out(p)| = R then
            break
        for p' ∈ V do
            if α · d(p*, p') ≤ d(p, p') then
                remove p' from V
```

Vamana

- Starts with random regular graph
- Runs RobustPrune only on candidate vertices encountered when trying to reach p from s
  - These vertices are likely to be on a relatively direct s-p path, giving better edges
- Runs two passes, once with alpha=1 and once with alpha=2
  - Former is faster and quickly smooths out the graph, latter adds long-distance connections



Algorithm 3: Vamana Indexing algorithm

Data: Database P with `n` points where `i`-th point has coords $\mathtt{x}_i$, parameters α, `L`, `R`
Result: Directed graph `G` over P with out-degree `<=R`
begin
  `initialize` G `to a random R-regular directed graph`
  `let s denote the medoid of dataset` P
  `let` σ `denote a random permutation of` $1..n$
  for $1 \leq i \leq n$ **do**
    `let` $[\mathcal{L}; \mathcal{V}] \leftarrow$ GreedySearch$(s, \mathtt{x}_{\sigma(i)}, 1, L)$
    `run` RobustPrune$(\sigma(i), \mathcal{V}, \alpha, \mathtt{R})$ `to update out-neighbors of` $\sigma(i)$
    for *all points* `j` *in* $N_{\text{out}}(\sigma(i))$ **do**
      if $|N_{\text{out}}(j) \cup \{\sigma(i)\}| > \mathtt{R}$ **then**
        `run` RobustPrune$(j, N_{\text{out}}(j) \cup \{\sigma(i)\}, \alpha, \mathtt{R})$ `to update out-neighbors of` j
      else
        `update` $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \sigma(i)$

Comparison

- HNSW:
  - Only uses single pass with alpha=1
  - Candidate set for each RobustPrune is only final candidate set: much more local graph
  - Empty graph initialization
  - On optimal parameters, about 50% slower than Vamana at index construction on 1M vertices
  - Average vertex requires about 11 out-of-cache hops for 1M vertices
- NSG:
  - Only uses single pass with alpha=1
  - Complicated initialization of approximation k-NN graph.
  - On optimal parameters, about 200% slower than Vamana at index construction on 1M vertices
  - Average vertex requires about 9 out-of-cache hops for 1M vertices
- Vamana:
  - Average vertex requires between 3 and 7 out-of-cache hops for 1M vertices depending on average degree

DiskANN

- Store the Vamana index on disk (with full precision vertices) and compressed vertices in main memory
- During GreedySearch, simultaneously collect neighbors for several nearby candidates
- Vertices within 3 hops of source vertex are stored in cache
  - Because only 3 out-of-cache hops are needed on average, only 3 disk queries are needed
- To compute index without using too much memory, use k-means clustering to find cluster centers and map each point to 2 nearby clusters
  - Empirically, computing Vamana index for each cluster and taking edge unions performs reasonably well but doesn't need much memory to compute
  - One-shot Vanama on 1B vertices takes 1100GB of RAM to construct, but only runs about 20% faster per query
  - Construction times are reported but are on different machines and therefore unclear how to compare
- Full-precision vertices are retrieved when visiting vertices to get better results while taking advantage of easier compressed calculations when choosing vertices to visit

Comparison with comparable memory footprints

- Recall that single-machine in-memory methods use Inverted Indexing and Data Compression
- Generally good 1-recall@100, but poor 1-recall@1
- 1-recall@1 for IVFOADC+G+P-16 plateaus at 37.04% for 1B vertices
- 1-recall@1 for IVFOADC+G+P-32 plateaus at 62.74% for 1B vertices
- 1-recall@1 for DiskANN can reach 100% for 1B vertices, and achieves 95% within 3.5ms

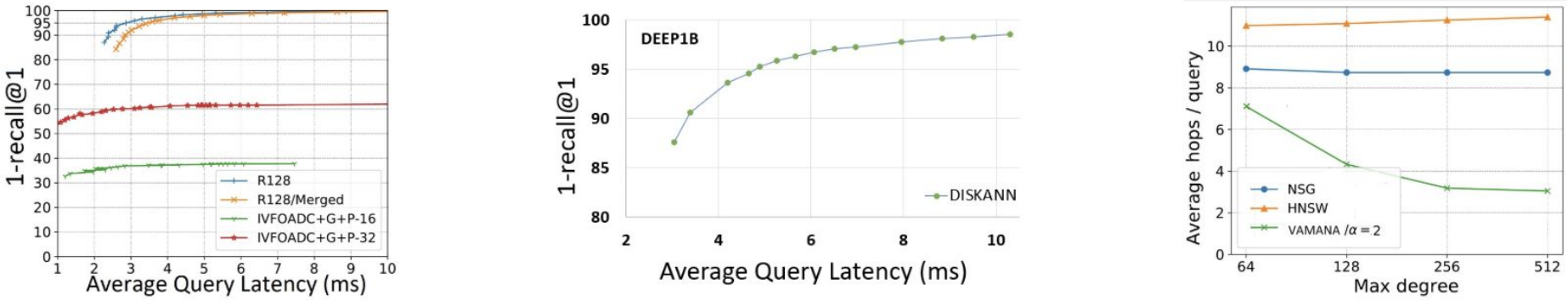


Figure 2: (a)1-recall@1 vs latency on SIFT bigann dataset. The R128 and R128/Merged series represent the one-shot and merged Vamana index constructions, respectively. (b)1-recall@1 vs latency on DEEP1B dataset. (c) Average number of hops vs maximum graph degree for achieving 98% 5-recall@5 on SIFT1M.

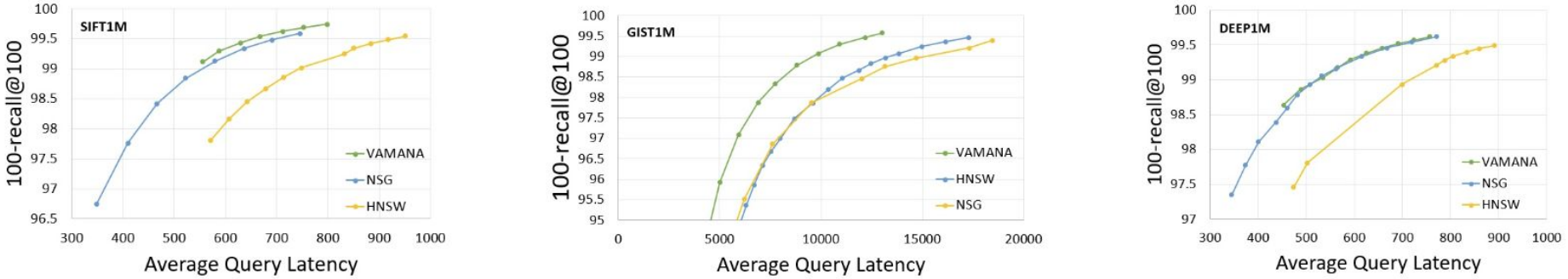


Figure 3: Latency (microseconds) vs recall plots comparing HNSW, NSG and Vamana.

Comments

- Needs more numerical results (tables)
- Data for 1-pass Vamana vs 2-pass Vamana
- Possibly even more passes of Vamana can provide better results?