High-dimensional visual similarity search: k-d Generalized Randomized Forests

[Extended Abstract] *

Yannis Avrithis
Department of Informatics & Telecommunications
University Of Athens
Athens, Greece
iavr@image.ntua.gr

Ioannis Z. Emiris†
Department of Informatics & Telecommunications
University Of Athens
Athens, Greece
emiris@di.uoa.gr

Georgios Samaras
Department of Informatics & Telecommunications
University Of Athens
Athens, Greece
gsamaras@di.uoa.gr

ABSTRACT

We propose a new data-structure, the generalized randomized k-d forest, or k-d GeRaF, for approximate nearest neighbor searching in high dimensions. In particular, we introduce new randomization techniques to specify a set of independently constructed trees where search is performed simultaneously, hence increasing accuracy. We omit backtracking, and we optimize distance computations. We release public domain software `GeRaF` and we compare it to existing implementations of state-of-the-art methods. Experimental results on SIFT and GIST visual descriptors, indicate that our method is the method of choice in dimensions around 1,000, and probably up to 10,000, and datasets of cardinality up to a few hundred thousands or even one million. For instance, we handle a real dataset of 10^6 GIST images represented in 960 dimensions with a query time of less than 1 sec on average and 90% responses being true nearest neighbors.

CCS Concepts

•Information systems → Nearest-neighbor search; •Computing methodologies → Visual content-based indexing and retrieval;

Keywords

randomized tree, space partition, image search, high dimension, open software, GIST images

*A full version of this paper is available at `http://arxiv.org/pdf/1603.09596v1.pdf`

†Partially supported by a bilateral collaboration with INRIA Sophia-Antipolis (France) funded by the Department of Informatics & Telecoms.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

Nearest Neighbor Search remains a fundamental optimization problem with both theoretical and practical open issues today, in particular for large datasets in dimension well above 100. An exact solution using close to linear space and sublinear query time is impossible, hence the importance of approximate search, abbreviated *NNS*. Given a finite dataset $X \subset \mathbb{R}^d$ and real $\epsilon > 0$, $x^* \in X$ is an ϵ-approximate nearest neighbor of query $q \in \mathbb{R}^d$, if $dist(q, x^*) \leq (1+\epsilon) dist(q, x)$ for all $x \in X$. For $\epsilon = 0$, this reduces to exact NNS.

High-dimensional NNS arises naturally when complex objects are represented by vectors of d scalar features. NNS tends to be one of the most computationally expensive parts of many algorithms in a variety of applications, including computer vision, pattern recognition and classification, multimedia databases, knowledge discovery and data mining, machine learning, document retrieval and statistics [4, 5]. Large scale problems are quite common in such areas, for instance $> 10^7$ points and $> 10^5$ dimensions.

Previous work. There are many efficient approaches to NNS. We focus on the most competitive ones, with emphasis on practical performance, in particular for applications in image similarity search. An important class of methods consists in data-dependent methods, where the decisions taken for space partitioning are based on the given data points. The Balanced Box Decomposition (BBD) tree [3] is a variant of the quadtree. It subdivides space into axis-aligned hyperrectangles. The implementation in library `ANN` [6] seems to be the most competitive method, for roughly $d < 100$.

A successful contribution has been library `FLANN` [7, 8]; the method has been most successful on SIFT image descriptors with $d = 128$. `FLANN` constructs a forest of up to 6 randomized k-d trees and performs simultaneous search in all trees. It chooses the split coordinates adaptively but all leaves contain a single point. The implementation adopts some optimization techniques, such as unrolling the loop of distance computation, but our software goes significantly further in this direction.

In high dimensional space, tree-based data structures are affected by the curse of dimensionality. One option by [1] is to perform dimension reduction and use BBD trees on the reduced space. Another option is Locality Sensitive Hashing (LSH); the idea of LSH is to hash the points of the data set so as to ensure that the probability of collision is much higher

for objects that are close to each other than for those that are far apart. One implementation that we use for comparisons is in library `E2LSH` [2].

A different hashing approach is to represent points by short binary codes to approximate and accelerate distance computations. There are several recent extensions, and the current state of the art in up to 10^9 points in 128 dimensions is locally optimized product quantization (LOPQ) [5].

Contribution. Our main contribution is to propose a new, randomized data-structure for high dimensional NNS, namely the k-d *Generalized Randomized Forest* (k-d GeRaF[1]), which generalizes the k-d tree. We employ adaptive and randomized algorithms for choosing the split coordinate, and further randomization techniques to build a number of independent k-d trees. We also provide automatic configuration of the parameters The number of trees depends on the input and may go up to the tens or hundreds. We examine alternative ideas, such as random shuffling of the points, random isometries, leaves with several points, and methods for accelerating distance computation. By keeping track of encountered points, we avoid repeated computations [8]. We analyze the theoretical and practical aspects of our approach with emphasis on the experimental analysis for image data.

We have experimented with parameters of all methods and observed the difficulty, in general, to optimize them. Automatic configuration works well for `GeRaF`, which has the fastest building time. `GeRaF` also scales very well, even for $d = 10^4$ or $n = 10^6$, and, at the same accuracy, it is faster than competition for d roughly in the range $(10^3, 10^4)$, and n in the hundreds of thousands or millions.

2. THE k-d GeRaF

The limitations of a single k-d tree for high d are overcome by searching multiple, randomized trees, simultaneously. Overall, m different randomized k-d trees are built, each with a different structure such that search in the different trees is independent; *i.e.*, neighboring points that are split by a hyperplane in one, are not split in another. Search is simultaneous in the m trees, *i.e.*, nodes from all trees are visited in an order determined by a shared priority queue. There is no backtracking, and search terminates when c leaves are visited.

2.1 Randomization

The key insight is to construct substantially different trees, by randomization. Multiple independent searches are subsequently performed, increasing the probability of finding approximate nearest neighbors. Randomization amounts to either generating a different randomly transformed pointset per tree (*e.g.*, rotation or shuffling), or choosing splits at random at each node (*e.g.*, split dimension or value). As discussed below, we investigate four randomization factors.

Rotation. For each k-d tree, we randomly rotate the input pointset or, more generally, apply a different isometry [10]. Each resulting tree is thus based on a different set of dimensions. During search, the query is rotated before descending each tree. However, distances are computed between the original stored points and the original query.

Split dimension. In a conventional k-d tree, the pointset is halved at each node along one dimension; dimensions are examined in order even for high d. Here, we find the t dimensions of highest variance for the input set and then choose uniformly at random one of these t dimensions at each node. Thus, different trees are built from the given pointset.

[1]https://github.com/gsamaras/kd_GeRaF

Algorithm 1: k-d GeRaF: building

```
input  : pointset X, #trees m, #split-dimensions t,
         max #points per leaf p
output: randomized k-d forest F
begin
    V ← ⟨VARIANCE of X in every dimension⟩
    D ← ⟨t dimensions of maximum variance V⟩
    F ← ∅                                          ▷ forest
    for i ← 1 to m do
        f ← ⟨random transformation⟩                ▷ isometry,
        shuffling
        F ← F ∪ (f, BUILD(f(X)))                   ▷ build on
        transformed X, store f
    return F
function BUILD(X)  ▷ recursively build tree (node/leaf)
    if |X| ≤ p then                                ▷ termination reached
        return leaf(X)
    else                                           ▷ split points and recurse
        s ← ⟨one of dimensions D at random⟩
        v ← ⟨MEDIAN of X in dimension s⟩
        (L, R) ← ⟨SPLIT of X in dimension s at value v⟩
        return node(c, v, BUILD(L), BUILD(R))      ▷ build
        children on L, R
```

Split value. The default split value in a conventional k-d tree is the median of the coordinates in the selected split dimension. `FLANN` uses the mean for reasons of speed. Here, we compute the median, which would yield a perfect tree, and then randomly perturb it [9]. In particular, the split value equals the median plus a quantity δ uniformly distributed in $[-3\Delta/\sqrt{d}, 3\Delta/\sqrt{d}]$, where Δ is the diameter of the current pointset; δ is computed at every node during building [11].

Shuffling. When computing the split value at each node in a conventional k-d tree, the current pointset at the node is used, which is a subset of the original pointset. Even if the split value is randomized, it is still possible that the same point is chosen if the same coordinate value occurs more than once in the selected dimension. This is particularly common when points are quantized; for instance, SIFT vectors are typically represented by one byte per element. We thus randomly shuffle points at each tree. Hence, different splits occur despite ties.

2.2 Building

The overall building algorithm for k-d GeRaF, consisting of m trees, is outlined in Alg. 1. For simplicity, only the random split dimensions are included, while the split value is the standard median. There is a random data transformation f per tree, which may include either an isometry, shuffling, or both; in case of an isometry, it is stored for use during search.

Given a dataset X, the t dimensions of maximum variance, say D, are computed. For each tree, X is transformed according to a different function f and then the tree is built

recursively. At each node, one dimension (coordinate), say s, is chosen uniformly at random from D and X is split at the median in s. The two subsets of X, say L, R, are then recursively given as input datasets to the two children of the node. The split node so constructed contains the split dimension s and the split value v. Splitting terminates when fewer than p points are found in the dataset, in which case the point indices are just stored in a leaf node. When n is much higher than d, the bottleneck of the algorithm is finding the median, which is $O(n)$ on average. Otherwise, the bottleneck is computing the variance per dimension, which is $O(d)$. The space requirement for the entire data structure is $O(nd)$ for the data points and $O(nm)$ for the trees, including both nodes and indices to points, for a total of $O(n(d+m))$.

Each random isometry can be a rotation [11] or reflection, and in general requires the generation of a random orthogonal matrix R. We rather use an elementary Householder reflector P for efficiency [10]. In particular, given unit vector $u \in \mathbb{R}^d$ normal to hyperplane H, the orthogonal projection of a point x onto H is $x - (u^\top x)u$. Its reflection across H is twice as far from x in the same direction, that is, $y = x - 2(u^\top x)u = Px$, where $P = I - 2uu^\top$. Although P is orthogonal, the computation of reflection Px is $O(n)$, involving a dot product and an element-wise multiplication and addition. This is because $uu^\top$ is of rank one. We only need to store vector u for each tree.

2.3 Searching

Searching takes place in parallel in all trees; this does not refer to independent search per tree, but rather that nodes from all trees are visited in a particular order using a shared min-priority queue Q. The idea is that given a bound c on the total leaves to be checked, the query iteratively descends the most promising nodes from all trees, and the criterion is the distance of the query to the hyperplane specified by each node.

A shown in Alg. 2, the query initially descends all trees of forest F while all visited nodes are stored in Q, without checking any leaves. Then, for each node extracted from Q, the query descends again, this time computing distances to all points in the leaf. For each decision made at a node while descending, the other one is stored in Q. In particular, the *signed* distance $d = N.\textsc{dist}(q)$ of query q to the hyperplane specified by node N is

$$N.\textsc{dist}(q) = N.tree.f(q)_{N.c} - N.v \qquad (1)$$

where $N.tree.f$ is the isometry of the tree where N belongs, and $N.c$, $N.v$ are the split dimension (coordinate) and value of N, respectively. We descend to a child of N, chosen according to the sign of d, and the other child is stored in Q with the absolute distance $|d|$ as key.

Results are stored in a min-heap H that holds up to k points, where k is the number of neighbors to be returned. For each leaf visited, the distance between q and all points stored in the leaf is computed. For each point X_i of the dataset X, H is updated dynamically such that it always contains the k nearest neighbors to q. The key used for H is the computed (squared) distance $\|q - X_i\|^2$. A separate array keeps track of points encountered so far, such that no distance is computed twice.

For each tree built under isometry f, the transformed query $f(q)$ is used in all tests at internal nodes, but the initial query q is rather used in all distance computations with points stored at leaves. Similarly, the transformed dataset is used only for building the tree but is not stored. This is possible since the isometry leaves distances unaffected. In practice, unlike (1), the query is transformed according to isometries of all trees prior to descending.

Algorithm 2: k-d GeRaF: searching.

input : query point q, forest F, #neighbors k, max #leaf-checks c
output: k nearest points

```
1  begin
2      Q.INIT()                          ▷ min-priority queue, initially empty
3      for i ← 1 to m do
4          DESCEND(q, F[i], FALSE)       ▷ descend i-th tree, store path in Q, no checks
5      ℓ ← 0                             ▷ # of leaves checked
6      H.INIT(k)                         ▷ min-heap of size k
7      while ¬Q.EMPTY() ∧ ℓ < c/(1+ε) do
8          (N, d) ← Q.EXTRACT-MIN()      ▷ (node, distance)
9          DESCEND(q, N, TRUE)           ▷ descend again, but check leaves now
10         ℓ ← ℓ + 1                     ▷ increase leaves checked
11     return H

12 function DESCEND(q, node N, check)    ▷ descend node N for query q
13     d ← N.DIST(q)                     ▷ signed distance to boundary
14     if d < 0 then                     ▷ q is in negative half-space
15         Q.INSERT(N.right, |d|)        ▷ remember right child
16         DESCEND(q, N.left, check)     ▷ descend left child
17     else
18         Q.INSERT(N.left, |d|)         ▷ and vice versa
19         DESCEND(q, N.right, check)

20 function DESCEND(q, leaf N, check)    ▷ test query q on leaf N
21     if ¬check then return;
22     for i ∈ N.POINTS do
23         H.INSERT(i, ‖q − X_i‖²)       ▷ distances to points X_i in leaf N
```

3. EXPERIMENTS

This section presents our experimental results and comparisons on a number of synthetic (Klein bottle) and real datasets (SIFT and GIST). Most experiments use the default parameters provided by existing implementations but, on specific inputs, we have optimized the parameters manually. Preprocessing includes building, but for `FLANN` and `GeRaF` it also includes automatic parameter configuration. Build time is related to the required precision as expressed by ϵ. For `LSH`, ϵ is failure probability and its build time is the most sensitive to ϵ. Despite requesting the user to manually determine parameter R, `LSH` performs an automatic parameter configuration as well, which is included in the building process. During search, the miss rate is the percentage of queries where the reported neighbor is *not* the exact one.

Both Table 2 and Figure 1 show that `GeRaF` is typically faster than `LSH` by at least an order of magnitude at the same accuracy. `BBD` and `FLANN` have problems, namely they

	Klein $n = 10^4, d = 10^2$				SIFT $n = 10^6, d = 128$			
ϵ	0	0.1	0.5	0.9	0	0.1	0.5	0.9
BBD	0.13	0.14	0.17	0.14	–	–	–	–
LSH	0.11	0.07	0.03	0.05	–	–	170.1	145.5
FLANN	–	–	–	–	20	19.2	19.8	19.7
GeRaF	0.06	0.06	0.06	0.08	62.6	93.6	90.5	96.0

Table 1: Build time (s) for two representative datasets. FLANN does not finish after 4 hr, which is indicated by '–' on Klein bottle or build times in gray on SIFT, where we have skipped configuration and used default values. BBD runs out of memory on SIFT, as well as LSH for $\epsilon = 0, 0.1$.

ϵ	miss %				search (μsec)			
	BBD	LSH	FLANN	GeRaF	BBD	LSH	FLANN	GeRaF
0.0	0	1	–	2	0.470	2.700	–	0.100
0.1	59	1	–	3	0.043	2.400	–	0.083
0.5	59	20	–	3	0.046	1.900	–	0.083
0.9	59	63	–	5	0.052	0.850	–	0.070

Table 2: Search accuracy and times for Klein $n = 10^4, d = 10^2$. Queries are nearly equidistant to points, which explains high miss rates, especially for BBD and FLANN; '–' indicates preprocessing does not finish after 4 hr.

suffer from either running out of memory or not completing automatic-parameters build (we had to manually input parameters in some cases). Table 3 displays, for all methods, the miss rate and search time as a function of n or d when the other parameter is fixed. In cases where miss rate is not 100%, GeRaF is an order of magnitude faster. The only exception is $d = 100$, where the situation is inversed with FLANN.

4. CONCLUSION

We provide a simple but effective automatic parameter configuration that yields the fastest preprocessing, including both configuration and building, as well as a successful trade-off between accuracy and speed. Most competing methods have difficulties, namely they suffer from running out of memory at large scale, slow or non-terminating parameter configuration, or unstable search behavior.

An interesting and relevant feature is that GeRaF appears to exploit intrinsic structure in the input, such as the structure of SIFT image datasets or the Klein bottle. The work in [11] may pave the way for explaining this behavior.

5. REFERENCES

[1] E. Anagnostopoulos, I. Emiris, and I. Psarros. Low-quality dimension reduction and high-dimensional approximate nearest neighbor. In *Proc. Annual Symp. on Computational Geometry*, pages 436–450, 2015.

[2] A. Andoni and P. Indyk. E^{2}LSH 0.1 User Manual, Implementation of LSH: E2LSH, http://www.mit.edu/ ∼andoni/LSH, 2005.

[3] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbors in fixed dimension. *J.ACM*, 45:891–923, 1998.

[4] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Analysis & Machine Intell.*, 33(1):117–128, 2011.

[5] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Comp. Vision & Pattern Recogn.*, 2014.

[6] D. M. Mount and S. Arya. Ann: A library for approximate nearest neighbor searching, https://www.cs.umd.edu/ mount/ann/, 2010.

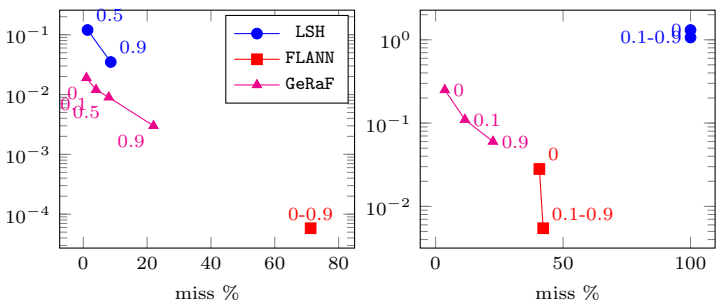
[7] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proc. VISAPP: Intern. Conf. Computer Vision Theory & Appl.*, pages 331–340, 2009.

[8] M. Muja and D. Lowe. Scalable nearest neighbour algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence*, 2014.

[9] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proc. Computer Vision & Pattern Recogn*, 2007.

[10] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Proc. IEEE Computer Vision & Pattern Recognition*, 2008.

[11] S. Vempala. Randomly-oriented kd-trees adapt to intrinsic dimension. In *Proc. Foundations Software Techn. & Theor. Comp. Science*, pages 48–57, 2012.



(c) SIFT $n = 10^6, d = 128$ (d) GIST $n = 10^6, d = 960$

Figure 1: Search accuracy (miss rates) and runtimes (sec) on real datasets. Numbers over points are the values of ϵ. In both cases, BBD is out of memory and FLANN does not preprocess after 4 hr for any ϵ, thus we configured its parameters manually.

n	d	miss %				search (μsec)			
		BBD	LSH	FLANN	GeRaF	BBD	LSH	FLANN	GeRaF
10^3	100	100	0	16	0	1	212	12	199
	1000	100	50	100	50	5	1850	34	14
	5000	100	0	100	0	39	8675	149	122
	10000	100	37	100	2	276	17000	289	520
1000	10^3	100	50	100	50	5	1850	34	14
10000		100	0	100	0	5	1780	–	390
100000		100	8	100	0	276	–	–	10900

Table 3: Klein bottle search for $\epsilon = 0.1$, for varying n or d, where the other parameter is fixed. Search times in gray represent failure cases where miss rate is 100%. Queries are nearly equidistant from the points, which explains high miss rates. '–' indicates preprocessing does not finish after 2 hr.