Project number IST-25582

**CGL**

Computational Geometric Learning

# Randomized kd-trees for Approximate Nearest Neighbor Search

**STREP**

**Information Society Technologies**

| | |
|---|---|
| Period covered: | November 1, 2011–October 31, 2012 |
| Date of preparation: | November 28, 2013 |
| Date of revision: | November 28, 2013 |
| Start date of project: | November 1, 2010 |
| Duration: | 3 years |
| Project coordinator name: | Joachim Giesen (FSU) |
| Project coordinator organisation: | Friedrich-Schiller-Universität Jena<br>Jena, Germany |

# Randomized kd-trees for Approximate Nearest Neighbor Search

Ioannis Z. Emiris* Dimitri Nicolopoulos*

November 28, 2013

**Abstract**

We implement randomized kd-trees in CGAL for approximate Nearest-Neighbor Search, based on the corresponding algorithms of the FLANN library. The package supports $k$ nearest point searching. The input points are preprocessed into several tree data structures by modifying standard kd-trees. The trees adapt to the pointset by assigning higher probability in splitting along coordinates of high variance. The implementation is faster than CGAL kd-trees in high dimensions. Our feature extends the 4.3 CGAL dD Spatial Searching functionality and introduces a new splitting rule extending the Splitter base class. This report is a follow up of [EKKNTF12, Sect.2].

**Keywords:** approximate nearest neighbor, kd-tree, randomization

## 1 Introduction

Nearest neighbor search (NNS) is a fundamental question and plays a pivotal role in a set of applications ranging from computer vision, databases, machine learning to data sequencing, and bioinformatics. Such applications can often only be described effectively by high dimensional datasets. As exact NNS methods are almost linear in the input research has turned to approximation methods. This is an area of very active research, where several approaches have been proposed; we focus on tree-based approaches.

The kd-tree was introduced in [Ben75], presented as a generalized method of using binary trees in high dimensions. Later work [FBF77] introduced the optimized kd-tree that proposed $N$ build time and $\log(N)$ search time, for $N$ points in low dimensions. The logarithmic search time has been shown not to apply in high dimensions, where it seems to approach $O(N)$; the bottleneck is typically due to the amount of backtracking. It thus becomes important to exploit probabilistic approaches where the output can be accepted if it lies within a (multiplicative) error factor $\epsilon$ from the true closest neighbor.

For the kd-tree method proposed in [SAH08], the coordinate at which the dataset is split at every tree level is chosen randomly among those for which the set shows greatest variance, thus aiming at exploiting input structure. A constant number of trees are built, using independent random choices; all trees are searched for each query. An implementation provided by Muja and Lowe [ML09] has been shown experimentally to be an improvement to implementations before that and seems to achieve one of the lowest search times in certain applications, especially related to image processing and computer vision, where data points are correlated in unknown ways, although lying in a very high-dimensional ambient space (e.g. beyond 100 dimensions). The method has also been used in [EF13] where the main oracle is reduced to NNS.

Aiming at high dimensions (e.g. 50, 100 or beyond) we provide a C++ implementation of the randomized kd-tree approach presented in [SAH08] using the Computational Geometry Algorithms Library (CGAL). Our implementation extends the currently available dD Spatial

*National and Kapodistrian University of Athens, Department of Informatics and Telecommunications, Athens, Greece. emiris@di.uoa.gr, {dimitri.nicolo}@gmail.com

Searching[1] provided by CGAL, by exploiting structure in the input. The implementation is competitive to FLANN; below, we show it is faster than the CGAL kd-trees in finding the (approximate) nearest neighbor (NN). Moreover, our method effectively solves the problem of finding $k > 1$ NNs approximately.

This report is a follow up of [EKKNTF12, Sect.2].

## 2 Construction and Search

The *Cut-dimension* is the coordinate where we split the data at any particular level. It lies in $\{1, \ldots, d\}$ where $d$ is the ambient dimension of the input. Let $p(l)$ be the function mapping the tree level $l$ to the *Cut-dimension*, at that level, and let $p(l) = -1$, if $l$ is not a valid tree level. Let $\vec{x} \in \mathbb{R}^d$. We define $\pi : \mathbb{R}^d \times \mathbb{N} \to \mathbb{R}$ as the function that, given a vector (of a point) $\vec{x}$ and level $l$ of the tree, it returns the $p(l)$-th coordinate of $\vec{x}$, namely $\pi(\vec{x}, l) = \vec{x}_{p(l)}$.

```
Data: Set $N$ data points in $\mathbb{R}^d$
Result: Vectors of length $d$ with the mean and variance per coordinate
i=1 ;
while $i \leq d$ do
    Calculate mean $m$ of a 10% sample size of $N$ at dimension $i$ ; vecMean(i) = m ;
    Calculate variance var of a 10% sample size of $N$ at dimension $i$ ; vecVar(i) = var ;
    i++;
end
```
**Algorithm 1**: Mean and variance per coordinate

We build a number of trees independently. For each tree, at each level the projected dimension $p(l)$ is chosen at random from the dimensions displaying the $t$ highest variances, for some $t$. They are picked by the randomized Algorithm 1. These $t$ coordinates will be used to entirely partition the tree. If $d$ is the dimension of the data points then we notice that the $d-t$ coordinates will never be used to partition the tree. The overall algorithm for building the randomized kd-tree is Algorithm 2.

Notice that data-points are stored at leaves, and leaves contain more than one point. This is indispensable when storing points in very high dimension.

Let us turn to searching. Backtracking efficiency provides the total efficiency. In [SAH08], they obtain it by searching multiple trees, thus changing the concept of a search on a single tree. The descent of each tree provides a NN candidate. A queue is maintained containing the ordered neighbors from the $m$ trees. Searching $m$ trees with a limitation of $r$ search nodes is simply searching each tree for $r/m$ nodes.

In a standard kd-tree, the coordinate dividing the data is the one for which the data has greatest variance. The data variance in fact is similar in many of the coordinates the subdivision is made. So, by selecting at random, the authors of [SAH08] argue that the probability of the query node falling on either side of the node in each level remains quite high, thus maintaining backtracking efficiency.

The tree is such that the order of points $x_i$ tested to find the closest match to query $q$ is the order of the distance of $\pi(x_i)$ from $\pi(q)$, where $\pi$ is a projection onto a lower dimensional space. The kd-tree search in high-dimensions is closely related to NN search after the aforementioned projection into the lower-dimensional space. In [SAH08] they observe that the original NN remains an approximate NN after the projection. They clearly show the advantage of searching in several independent projections. Using independent projections into lower dimensional spaces boosts the probability that the closest point will be among the closest points in at least one of the projections. Results are shown in [SAH08, fig.5]. They are based on modeling the probability

[1]http://doc.cgal.org/latest/Spatial_searching/index.html

```
Data: Set $N$ data points in $\mathbb{R}^d$
Result: Randomized kd-tree
initialization;
vecMean = Mean(N);
vecVar = Var(N);
kVec  // k highest variance dimensions ;
foreach level $l$ do
    set $p(l)$ to randomly chosen value from kVec;
end
while tree not built do
    ROOT ← $\mathbf{x}$ with $x_{p(l)}$ equal or closest to vecMean($p(0)$);
    foreach $l$ level do
        foreach children of $x$ in $N$ do
             // Subdivide list of points by plane perpendicular to axis;
            // of 'cutfeat' dimension at 'cutval' position ;
            if $\pi(x, l) <$ vecMean(p(l)) then
                // Place index to left side of array;
            else
                // Place index to right side of array;
            end
        end
    end
end
```

**Algorithm 2**: Building the randomized kd-tree

of failure using $m$ trees as the product of the independent probabilities of failure from searching approximately $r/m$ cells in each of the $m$ trees.

## 3 Implementation

The spatial searching package `RKd tree` implements exact and approximate query searching by providing implementations of algorithms supporting

- both nearest and furthest neighbor searching
- both exact and approximate searching
- (approximate) k-nearest and k-furthest searching
- query items representing points (extending to spatial objects)

The input pointset is represented either by Cartesian coordinates or by homogeneous coordinates. The approximate Randomized kd-tree spatial searching package is designed for datasets which are small enough in order to store the search structure in the main memory.

**Parameter description.** Users can perform a query for one or $k$ points. There are numerous parameters that can be set when calling the constructor.

1. Vector of Trees - The vector of trees.
2. Query_item - The query point.
3. $k$ - The number of NNs to search for.

4. $\epsilon$ - The multiplicative error in the distance is $1 + \epsilon \geq 1$.

5. $r$ - The maximum number of leaf nodes that can be checked.

6. Distance - The distance metric (CGAL defined, Euclidean, Manhattan or other).

Before performing a query a 'Point' & 'Index' tuple is instantiated containing the dataset points and the associated indices. Each tree is instantiated as a point index tuple and is pushed into the vector of trees. The query item is a 'Point'. Given constant $\epsilon \geq 0$, the distance of a point returned as a $k$-th NN must be $\leq (1 + \epsilon)\rho$, where $\rho$ denotes the distance to the real $k$-th NN; $\epsilon = 0$ implies the exact search.

**Build.** `Random_kdtree_k_neighbor_search` defines the functionality for approximate NN. The main input is the vector of `RKd_tree` structures of `Point` and index tuples. For each tree the root function is called. The function builds the tree and returns the root node to the `RKd_tree`. For each tree the point set is shuffled and a new random tree is built. The leaf nodes reference a single node. Each node is split in accordance to the splitting definition passed as a template parameter.

`Mean_split` extends the `Splitters` class and defines functionality for splitting a point container by a hyperplane into an upper and lower container. `Point_container` defines the container structures for the point sets associated with each node of the binary tree.

The rule defined by the mean split estimates the mean and variance of each of the $d$ coordinates of the dataset. A hyperplane is used to separate the halfspaces in each node. They are defined by a cutting dimension $cd$ and cutting value $cv$ [2]. The variance of coordinates with the highest values are pooled together out of which one is chosen, at random, as the $cd$ associated with the node. The $cv$ is the mean at coordinate equal to $cd$.

`RKd_Point_container` is stripped down `Point_container` like class.
`Split(RKd_Point_container&, Separator&)` partitions the point set into the upper partition and lower partition. The lower partition is passed into the container of the parameter passed into the `Split` function.

**Search.** The query point traverses each tree down to a leaf containing a single point. The point is passed into a sorted queue if the distance is less than the distance of the longest distance in the queue. During tree traversal, a record of the branches not taken is kept in a heap if the branch possibly contains points nearer than the ones in the queue. A search is performed for all trees, and the nearest points are kept in a single sorted queue which is returned as the output. The size of the queue is set by the user. Once a first traversal is performed on each tree the heap is 'popped' one by one and the branch is traversed down to a leaf node referencing a point. This process is performed until the maximum number of searches is performed or until the error factor specifies that no more searches need to be performed and the resulting queue can be returned as output.

## 4 Experimental Results

We compare search time and precision for two approximation methods, namely the kd-tree of CGAL and ours, using pointsets in 128 dimensions of cardinality of the order of $10^5$.

Since performance may depend on the datasets, we are looking to select datasets that: (1) show the performance extrema in search time between the algorithms, (2) maximize each algorithm's own performance, whereas (3) we also include real-world data. We have thus generated a series of test scenarios using both random data and SIFT data, a key benchmark in computer vision and image processing.

[2] see `http://doc.cgal.org/latest/Spatial_searching/classCGAL_1_1Plane_separator.html`

In all figures, the Search time gain over linear search is plotted against the precision, where Search time gain is defined as the fraction of linear search time over approximate search time. Linear search is the CGAL kd_tree search with zero error, hence exact. Precision is defined as the fraction of queries yielding an exact answer. On the plots, precision ranges in $[0, 1]$, where 1 stands for 100% exactness.

Error $\epsilon$ is directly related to the obtained precision. An exact search is possible by setting $\epsilon = 0$, in which case it does not payoff to construct more than one trees.

Another parameter affecting search time is the maximum number of leaf-nodes allowed to be checked. Reaching a leaf-node represents a full traversal and a final distance calculation and comparison. It is obvious that more comparisons provide higher precision with a search time increase.

**Poisson distribution.** This is a random number distribution that produces integers according to the following probability distribution:

$$P(i|\mu) = \frac{\mu^i}{i!} e^{-\mu}, \quad i \in \mathbb{Z}.$$

The numbers produced are random integers where each value represents a specific count of independent events occurring within a fixed interval, based on the observed mean rate at which they appear to happen. The distribution parameter is mean $\mu$, passed in through the constructor.

We produced a data set of dimension $d = 128$ and data size $n = 5 \cdot 10^5$. Each dimension displays a Poisson distribution of varying mean. In Figure 1 we find that for a lower precision, the proposed RKd-tree algorithm performs better, but veers closer to its CGAL counterpart as precision increases. Also note that considerable more build time is needed to obtain the faster search time.

**Poisson and Uniform distributions.** When all 128 dimensions display the same distribution there may not be a significant difference in choosing one dimension over the other as a cutting dimension. So if within the Poisson distributed data we place a small number of uniformly distributed coordinates we find that, as a tree is created, it tends to perform higher quality cuts by using the coordinates of uniform distribution.

In practice, this seems to pay off considerably. We keep build times quite low yet obtaining very fast search times producing highly precise results. The results indicate a large gain continuing up to extremely high precision values: an order of magnitude over Kd-tree, even at the 98% precision mark. In fact, it is the only instance in the tests performed that we find the RKd-tree algorithm obtaining over 95% precision with such ease in build and search time. See Figure 2.

**Normal distribution.** This is a very commonly occurring continuous probability distribution, found to represent a large amount of real occurring instance data very well. It captures the probability that an observation in some context will fall between any two real numbers, described by the following:

$$p(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The distribution produces real valued numbers around two parameters: mean $\mu$ and variance $\sigma$. The idea was to provide varying means and variances. A small number of the coordinates, selected at random, are chosen to display a significantly higher variance.

The results are closer to those of the Poisson distribution, ranging in values approaching an order of magnitude difference. One thing to note is that up to 50% precision the number of trees was kept small, but in order to obtain more accurate results the build time increased considerably. See Figure 3.

**Uniform.** The case where we expected the least improvement in search time over the Kd-tree is in using uniform distribution for all coordinates. Both a float and integer data set were produced. The uniform discrete distribution is described by the following:

$$P(i|a,b) = \frac{1}{b-a+1}, \quad a \leq i \leq b, \quad i \in \mathbb{Z}.$$

This distribution produces integers in the range $[a, b]$, where each value has an equal likelihood of being produced. In similar fashion we produce the real valued uniform distribution.

Indeed, Figure 4 on integer data shows that any relevant precision is obtained almost at the cost of linear search; for both algorithms. We did not plot the RKd-tree results past the 10% precision mark due to the fact that high build times are needed.

**SIFT.** Finally, a real case data set was used. SIFT [Low99] stands for Scale Invariant Feature Transform and is a method developed to detect local features in images. The features are invariant to image scaling, rotation, translation, and partially invariant to illumination changes and affine or 3D projection. We use a dataset of dimension $d = 128$ and size $n = 33 \cdot 10^4$, also tested in [ML09] (denoted 330K+). The number of trees did not exceed 64.

Figure 5 illustrates the speedup of approximate search over linear (exact) search, with varying precision. It is apparent that, as precision increases, there is a diminishing gain over linear search. Even for the case of 32 trees built (largest gain in query time), the build time is still smaller than that of CGAL kd-trees yielding largest precision: 82.91 sec (RKd_tree_search) compared to 97.57 sec (Kd_tree_search). For 40% precision approximately, our RKd-tree offers a 7x gain in search time over the CGAL kd-tree search method. Even for 90% precision, we notice about a 10x gain in search time.

For this data, we also display the cost of build time. The build time increases linearly in the number of trees, see Table 1. As the number of RKd-trees increases, a larger gain over linear search is obtained. Moreover, the lower the accuracy the higher discrepancy we encounter in the expected gain by using more trees. In the experiment performed, search time is so low that lies in the range of 1.0-2.0 msec. These approach the limits of accuracy of the timing function used (std::chrono). The difference between using 4 and 32 trees becomes apparent after 80% precision. In fact, using 4 trees, the number of nodes searched needs to increase significantly in order to surpass 80% precision, given the data size. For the same number of allowed nodes searched, 4 trees yield up to 80% precision whereas 32 trees yield up to 97% precision.

## 5 Future Work

It is interesting to undertake the theoretical study of the method's complexity and error bounds.

Notice the gain of using a larger number of trees, see Figure 5. It is obvious that a parallel build process would provide a considerable gain, and may performed in parallel with the queries, since building each single tree is a disassociated process.

## References

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BL97] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, pages 1000–1006. IEEE Computer Society, 1997.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. Symp. Computational Geometry*, pages 253–262. ACM, 2004.

[EF13] I.Z. Emiris, and V. Fisikopoulos. Algorithms for volume approximation of convex bodies. Technical Report CGL-TR-76, NKUA, 2013.

[EKKNTF12] I.Z. Emiris, A. Konstantinakis-Karmis, D. Nicolopoulos, and A. Thanos-Filis. Data structures for approximate nearest neighbor search. Technical Report CGL-TR-29, NKUA, 2012.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[Lai97] Jim Z. C. Lai. Fast encoding algorithms for tree-structured vector quantization. *Image Vision Comput.*, 15(11):867–871, 1997.

[Low99] D.G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proc. ICCV*, pages 1150–1157, 1999.

[ML09] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In A. Ranchordas and H. Araújo, editors, *Proc. VISAPP: 4th Intern. Conf. Computer Vision Theory & Appl.*, volume 1, pages 331–340, February 2009.

[SAH08] Chanop Silpa-Anan and Richard Hartley. Optimised KD-trees for fast image descriptor matching. In *Proc. IEEE Comp. Vision Patt. Recognition (CVPR)*, 2008.

## Appendix: Experiments

| # Trees | Build Time (sec.) |
|---|---|
| 2 | 5.31 |
| 4 | 10.37 |
| 8 | 22.41 |
| 16 | 49.67 |
| 32 | 82.91 |

Table 1: Build time of Randomized kd-trees.

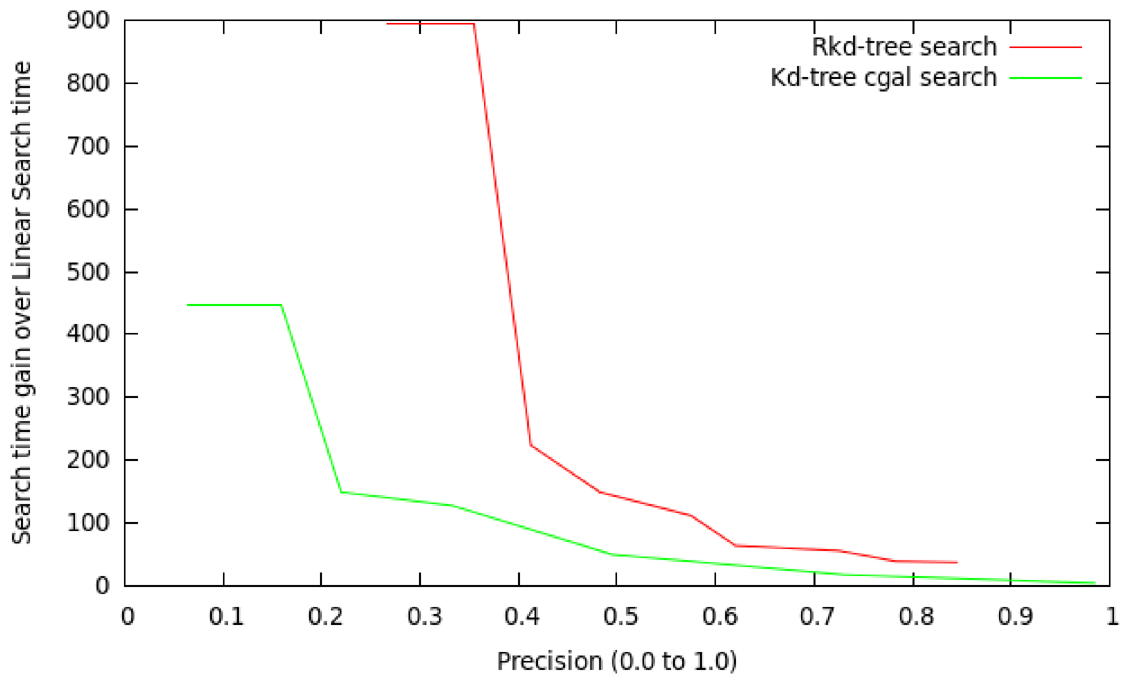

Figure 1: Poisson distributed data.

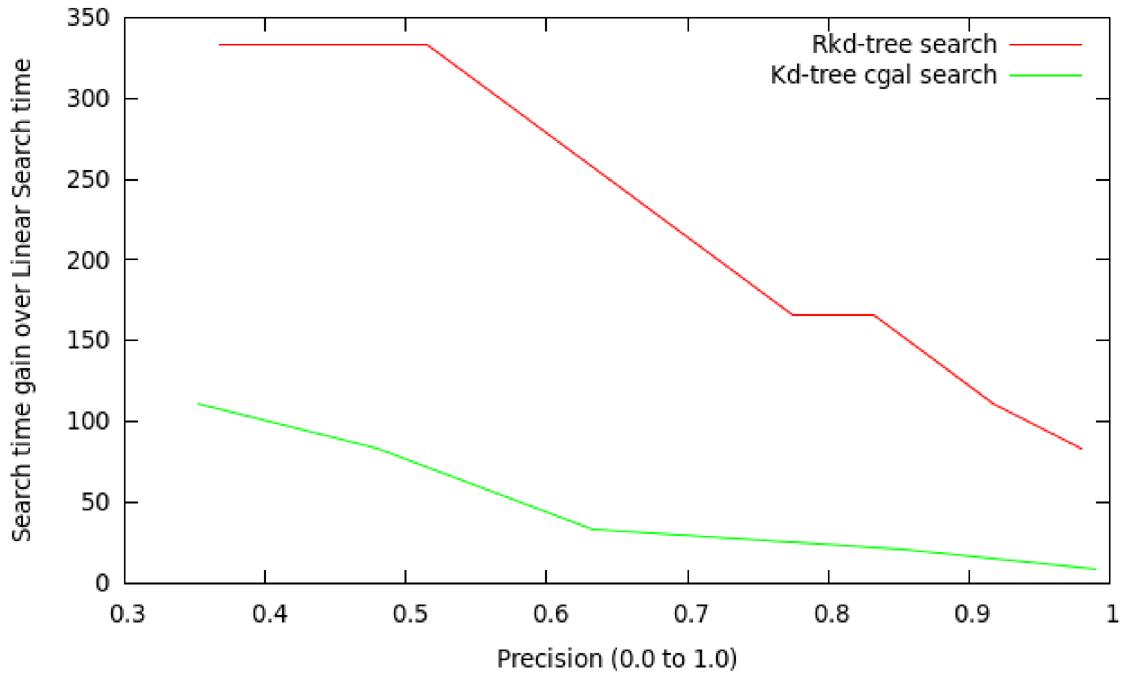

Figure 2: Poisson and uniformly distributed.

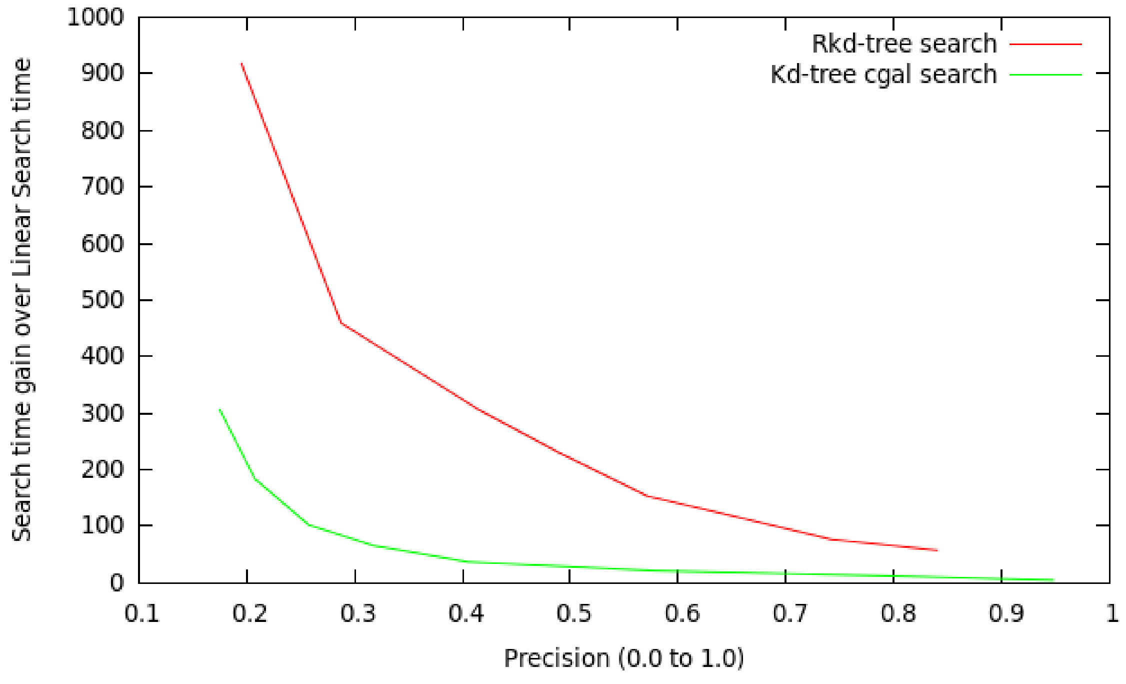

Figure 3: Normal distributed data.

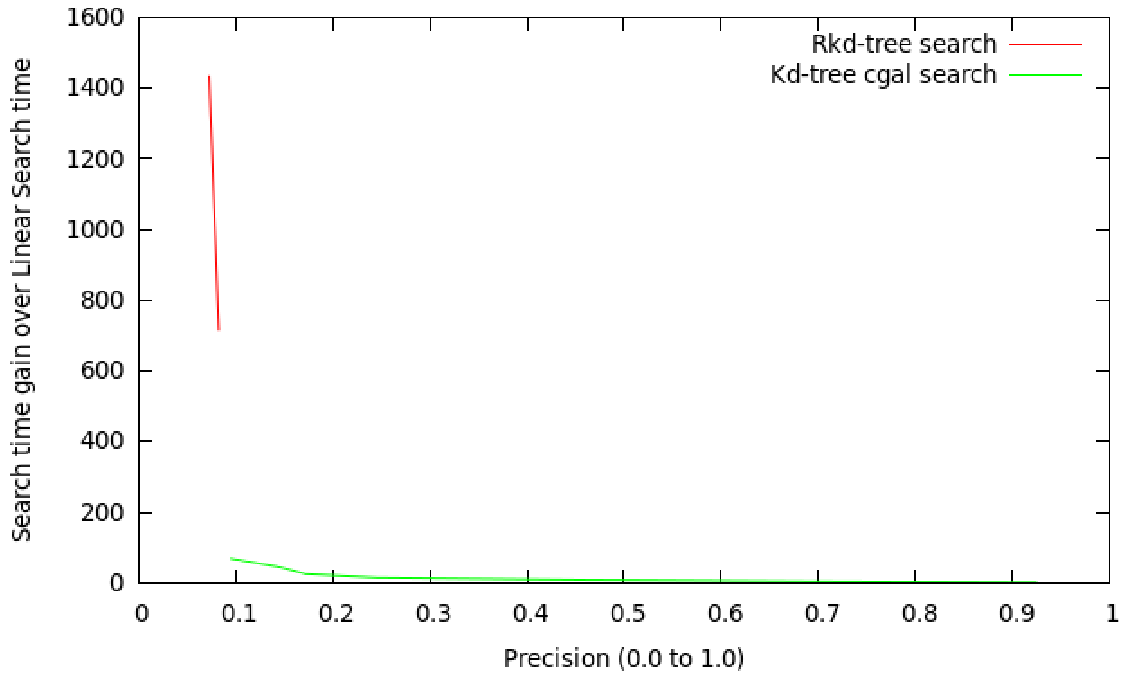

Figure 4: Uniform integer distributed data.

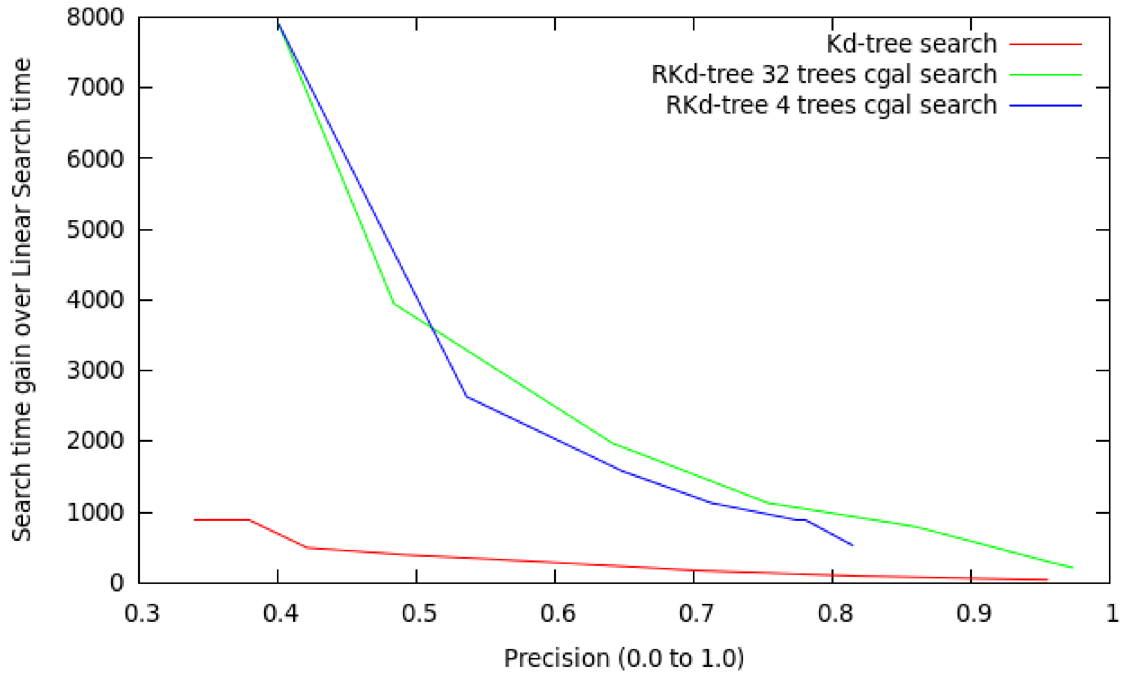

Figure 5: SIFT data.