An Empirical Comparison of Exact Nearest Neighbour Algorithms

Ashraf M. Kibriya and Eibe Frank

Department of Computer Science
University of Waikato
Hamilton, New Zealand
{amk14, eibe}@cs.waikato.ac.nz

Abstract. Nearest neighbour search (NNS) is an old problem that is of practical importance in a number of fields. It involves finding, for a given point q, called the query, one or more points from a given set of points that are nearest to the query q. Since the initial inception of the problem a great number of algorithms and techniques have been proposed for its solution. However, it remains the case that many of the proposed algorithms have not been compared against each other on a wide variety of datasets. This research attempts to fill this gap to some extent by presenting a detailed empirical comparison of three prominent data structures for exact NNS: KD-Trees, Metric Trees, and Cover Trees. Our results suggest that there is generally little gain in using Metric Trees or Cover Trees instead of KD-Trees for the standard NNS problem.

1 Introduction

The problem of nearest neighbour search (NNS) is old [1] and comes up in many fields of practical interest. It has been extensively studied and a large number of data structures, algorithms and techniques have been proposed for its solution. Although nearest neighbour search is the most dominant term used, it is also known as the best-match, closest-match, closest-point and the post office problem. The term similarity search is also often used in the information retrieval field and the database community. The problem can be stated as follows:

> Given a set of n points S in some d-dimensional space X and a distance (or dissimilarity) measure M, the task is to preprocess the points in S in such a way that, given a query point $q \in X$, we can quickly find the point in S that is nearest (or most similar) to q.

A natural and straightforward extension of this problem is k-nearest neighbour search (k-NNS), in which we are interested in the k ($\leq |S|$) nearest points to q in the set S. NNS then just becomes a special case of k-NNS with k=1.

Any specialized algorithm for NNS, in order to be effective, must do better than simple linear search (the brute force method). Simple linear search, for n d-dimensional data points, gives $O(dn)$ query time[1] and requires no preprocessing.

[1] Time required to return the nearest neighbour(s) of a given query.

Ideal solutions exist for NNS for $d \leq 2$, that give O($d$log$n$) query time, and take O(dn) space and O(dnlog$n$) preprocessing time. For $d = 1$ it is the binary search on a sorted array, whereas for $d = 2$ it is the use of Voronoi diagrams and a fast planar point location algorithm [2]. For $d > 2$, all the proposed algorithms for NNS are less than ideal. Most of them work well only in the expected case and only for moderate d's (≤ 10). At higher d's all of them suffer from the curse-of-dimensionality [3], and their query time performance no longer improves on simple linear search. Algorithms that give better query time performance at higher d's exist but only for relaxations of NNS, i.e. for approximate NNS [4, 5], near neighbour search [6, 7], and approximate near neighbour search [7].

KD-Trees are among the most popular data structures used for NNS. Metric Trees are newer and more broadly applicable structures, and also used for NNS. Recently a new data structure, the Cover Tree, has been proposed [8], which has been designed to work well even at higher dimensions provided the data has a low intrinsic dimensionality. This paper presents an empirical comparison of these three data structures, as a review of the literature shows that they have not yet been compared against each other. The comparison is performed on synthetic data from a number of different distributions to cover a broad range of possible scenarios, and also on a set of real-world datasets from the UCI repository.

The rest of the paper is structured as follows. Section 2 contains a brief overview of the three data structures that are compared. Section 3 presents the experimental comparison. It outlines the evaluation procedure employed, and also presents the empirical results. The paper concludes with some final remarks in Section 4.

2 Brief overview of the NNS data structures

The following sub-sections give a brief overview of KD-Trees, Metric Trees and Cover Trees. Particular emphasis has been given to Cover Trees, to provide an intuitive description of the technique.

2.1 KD-Trees

KD-Trees, first proposed by Bentley [9], work by partitioning the point-space into mutually exclusive hyper-rectangular regions. The partitioning is achieved by first splitting the point-space into two sub-regions using an axis-parallel hyperplane, and then recursively applying the split process to each of the two sub-regions. For a given query q, only those regions of the partitioned space are then inspected that are likely to contain the k^{th} nearest neighbour. Recursive splitting of the sub-regions stops when the number of data points inside a sub-region falls below a given threshold. To handle the degenerate case of too many collinear data points, in some implementations the splitting also stops when the maximum relative width of a rectangular sub-region (relative to the whole point-space) falls below a given threshold. KD-Trees require points in vector form, and use this representation very efficiently.

Each node of a KD-Tree is associated with a rectangular region of the point-space that it represents. Internal nodes, in addition to their region, are also associated with an axis-parallel hyperplane that splits their region. The hyperplane is represented by a dimension and a value for that dimension, and it conceptually sits orthogonal to that selected dimension at the selected value, dividing the internal node's region.

A number of different strategies have been proposed in the literature for the selection of the dimension and the value used to split a region in KD-Trees. This paper uses the Sliding Midpoint of Widest Side splitting strategy, which produces good quality trees—trees that adapt well to the distribution of the data and give good query time performance. This strategy, given in [10], splits a region along the midpoint of the dimension in which a region's hyper-rectangle is widest. If, after splitting, one sub-region ends up empty, the selected split value is slid towards the non-empty sub-region until there is at least one point in the empty sub-region. For a detailed description, and a comparison of Sliding Midpoint of Widest Side to other splitting strategies, see [11].

The search for the nearest neighbours of a given query q is carried out by recursively going down the branch of the tree that contains the query. On reaching a leaf node, all its data points are inspected and an initial set of k-nearest neighbours is computed and stored in a priority queue. During backtracking only those regions of the tree are then inspected that are closer than the k^{th} nearest neighbour in the queue. The queue is updated each time a closer neighbour is found in some region that is inspected during backtracking. At the start, the queue is initialized with k null elements and their distance to q set to infinity.

2.2 Metric Trees

Metric Trees, also known as Ball Trees, were proposed by Omohundro [12] and Uhlmann [13]. The main difference to KD-Trees is that regions are represented by hyper-spheres instead of hyper-rectangles. These regions are not mutually exclusive and are allowed to overlap. However, the points inside the regions are not allowed to overlap and can only belong to one sub-region after a split. A split is performed by dividing the current set of points into two subsets and forming two new hyper-spheres based on these subsets. As in KD-Trees, splitting stops when for some sub-region the number of data points falls below a given threshold. A query is also processed as in KD-Trees, and only those regions are inspected that can potentially contain the k^{th} nearest neighbour. Metric Trees are more widely applicable than KD-Trees, as they only require a distance function to be known, and do not put any restriction on the representation of the points (i.e. they do not need to be in vector form, as in KD-Trees).

Each node of a Metric Tree is associated with a ball comprising the hyper-spherical region that the node represents. The ball is represented by its centre, which is simply the centroid of the points it contains, and its radius, which is the distance of the point furthest from the centre.

A number of different construction methods for Metric Trees can be found in the literature. This paper uses the Points Closest to Furthest Pair method

proposed by Moore [14]. This method first finds the point that is furthest from the centre of a spherical region (centre of the whole point-space in the beginning), and then finds another point that is furthest from this furthest point. The method, thus, tries to find the two points in a region that are furthest from each other. Then, points that are closest to one of these two points are assigned to one child ball, and the points closest to the other one are assigned to the other child ball. The method produces good quality Metric Trees that adapt well to the distribution of the data. A detailed comparison of this method with other construction methods for Metric Trees can be found in [11].

2.3 Cover Trees

Cover Trees [8] try to exploit the intrinsic dimensionality of a dataset. They are based on the assumption that datasets exhibit certain restricted or bounded growth, regardless of their actual number of dimensions.

Cover Trees are N-ary trees, where each internal node has an outdegree $\leq N$. Each node of the tree contains a single point p, and a ball which is centred at p. The points are arranged in levels, such that each lower level acts as a cover for the previous level, and each lower level has balls half the radius than the ones at the previous level. The top level consists of a single point with a ball centred at it that has radius $2^{i'}$, with an i' big enough to cover the entire set of data points. The next level consists of points with balls of half the radius than the top-most ball ($2^{i'-1}$), which cover the points at a finer level. The bottom-most level consists of points that have balls covering only those single points. A point at any level i in the tree is also explicitly present in all the lower levels.

The structure is built by arbitrarily selecting a point from the list of data points and creating the top-level ball. This same point is then used to build a smaller ball at the next lower level. This creation of smaller balls from the same point is repeated until we reach a level where a ball covers only that single point. Then the procedure backtracks to the last higher-level cover ball that still has unprocessed points, arbitrarily picks the next available point, and then recursively builds cover balls for this point at lower levels. The procedure is illustrated graphically in Figure 1.

When searching for the nearest neighbours of a given query q, we go down the levels of the tree, inspecting nodes at each level. At each level i we add only those nodes for further inspection whose centre points are inside the query ball (i.e. the ball centered at the query). The radius of the query ball is set to the distance of the current best k^{th} nearest neighbour (found from among the centre points of the nodes so far inspected) plus the radius of the balls at the current level i (which is 2^i). This amounts to shrinking the query ball as we go down the levels, and inspecting children of only those nodes whose ball centres are within the query ball. The search stops when at some level the inspected nodes are all leaf nodes with no children. At this stage the k-nearest neighbours in our priority queue are the exact k-nearest neighbours of the query. The procedure is illustrated graphically in Figure 2. Note that the figure shows the final shrunken query ball at each level.

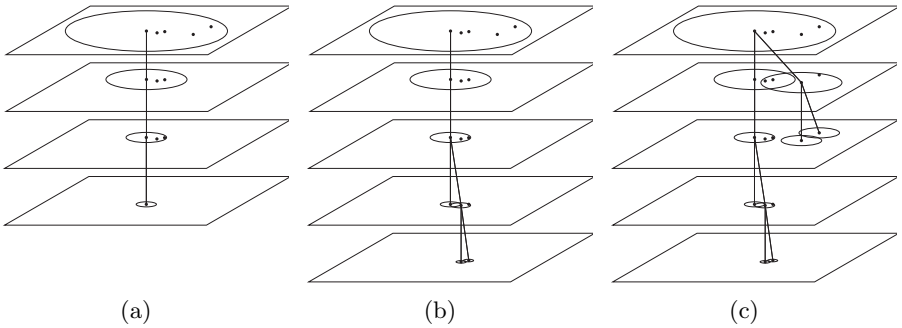

Fig. 1. Illustration of the construction method for Cover Trees. Tree at the end of (a) the first branch of recursion, (b) the second branch of recursion, and (c) the third and final branch of recursion.

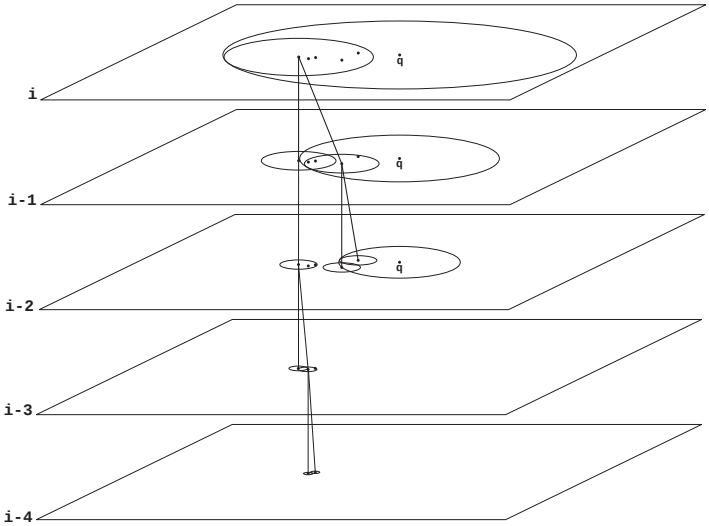

Fig. 2. Illustration of Cover Tree query. The query ball shrinks as the search proceeds.

3 Empirical comparison of the data structures

The comparison of the data structures is performed on synthetic as well as real-world data. Synthetic data was used to experiment in controlled conditions, to assess how they behave for increasing n (no. of data points) and increasing d (no. of dimensions), while keeping the underlying distribution constant.

On synthetic data, the evaluation of the data structures was carried out for $d = 2, 4, 8, 16, 32, 80$ and $n = 1000, 2000, 4000, 8000, 16000, 100000$. For each combination of n and d, data points were generated from the following distributions: uniform, Gaussian, Laplace, correlated Gaussian, correlated Laplace,

clustered Gaussian, clustered ellipsoids, straight line (not parallel to any axis), and noisy straight line. Most of these distribution are provided in the ANN library [10], the rest were added for this research. The correlated Gaussian and correlated Laplacian distributions are designed to model data from speech processing, the line distributions were added to test extreme cases, and the remaining distributions, especially the clustered ones, model data that occurs frequently in real-world scenarios. More details on these distributions can be found in [11].

The data structures were built for each generated set of data points, and were evaluated first on 1000 generated query points that had the same distribution as the data, and then on another 1000 generated query points that did not follow the distribution of the data, but had uniform distribution. In other words, results were obtained for increasing d for a fixed n, and for increasing n for a fixed d, when the query did and when it did not follow the distribution of the data. Moreover, each of these evaluations were repeated 5 times with different random number seeds and the results were averaged. Note that for each dataset the dimensions were normalized to the $[0, 1]$ range.

To obtain results for real-world data, we selected datasets from the UCI repository that had at least 1000 examples. In each case, the class attribute was ignored in the distance calculation. Nominal attributes were treated as integer-valued attributes, and all attributes were normalized. Missing values were replaced by the mean for numeric attributes, and the mode for nominal ones. On each dataset, the data structures were evaluated 5 times using a random 90/10 data/query set split, and the results reported are averages of those 5 runs. Also, the evaluations for both the artificial and the real-world data were repeated for $k = 1, 5$, and 10 neighbours.

All three data structures compared have a space requirement of $O(n)$. For Cover Trees though, the exact space is comparatively higher since it has maximum leaf size 1, but for KD-Trees and Metric Trees it is very similar as they both have maximum leaf size 40. The construction time for Cover Trees is $O(c^6 n \log n)$ [8] (where c is the expansion constant of the dataset [8]), but for KD-Trees and Metric Trees, with their chosen construction methods, it is not guaranteed. However, in the expected case they do construct in $O(n \log n)$ time. The query time of Cover Trees is $O(c^{12} \log n)$ [8], whereas for KD-Trees and Metric Trees it is $O(\log n)$ in the expected case for lower d's. Note that the constant c for Cover Trees is related to the assumption of restricted growth of a dataset, and can sometimes vary largely even within a dataset [8]. Hence, for all the data structures the space is guaranteed, but the construction and query times can best be observed empirically.

For the comparison of query time, linear search is also included in the experiments as a baseline. All compared techniques, including the linear search, were augmented with Partial Distance Calculation [15, 11], which skips the complete distance calculation of a point if at some stage the distance becomes larger than the distance of the best k^{th} nearest neighbour found so far.

For all the experiments the leaf size of KD-Trees and Metric Trees was set to 40. The threshold on a node's maximum relative width in KD-Trees was

set to 0.01. All algorithms were implemented in Java and run under the same experimental conditions.[2] The Cover Tree implementation we used is a faithful port of the original C implementation provided by the authors. Note that the base for the radii of the balls in the algorithm was set to 1.3 instead of 2 in the C implementation, and thus also in ours.

3.1 Results

We present construction times and query times for the synthetic data. For the real-world datasets, only the query times are given, to support the main conclusions observed from the synthetic data.

Figure 3 shows the construction times of the structures on synthetic data for increasing n, for $d = 4$, and also for increasing d, for $n = 16000$. Figures 4 and 5 show the query times, Figure 4 for increasing n for $k = 5$ and $d = 8$, and Figure 5 for increasing d for $k = 5$ and $n = 16000$. All axes are on log scale.

It can be observed from Figure 3 that KD-Trees exhibit the best construction time overall. On all but the line distribution, their construction time grows at the same rate as for the other techniques, but is a constant times faster. The construction time of Cover Trees is very similar to that of Metric Trees on distributions other than the line, but for $d > 16$ it grows exponentially and becomes worst overall.

Considering query time, Figures 4 and 5 show that all three tree methods suffer from the curse-of-dimensionality, and generally become worse than linear search for $d > 16$. At higher d's they are only better than linear search if the points are clustered or lie on a line. KD-Trees are the best method if the query points have the same distribution as the data used to build the trees, otherwise KD-Trees are best for low d's, but for higher d's Cover Trees are best. Metric trees generally perform somewhat better than Cover Trees when the query points have the same distribution as the original data, and somewhat worse otherwise. However, their query times are generally quite close. When the query distribution is not changed to be uniform, KD-Trees, in terms of both construction and query time, are worse than the others only for points lying on a line, a case that is uncommon in practice. Trends like the ones in Figures 3, 4 and 5 were also observed for $k = 1$ and $k = 10$, and other values of d and n.

Table 1 shows the query time of the data structures on the UCI data. All the techniques are compared against KD-Trees, and the symbols $\circ$ and $\bullet$ denote respectively, whether the query time is significantly worse or better compared to KD-Trees, according to the corrected resampled paired t-test [16]. It can be observed that KD-Trees are significantly better than the rest on most datasets. In some cases they are still better than linear search even at higher d's (the dimensions are given in brackets with the name of a dataset). It can also be observed that, in most cases, and in contrast to the results on the artificial data, Cover Trees outperform Metric Trees.

[2] All implementations are included in version 3.5.6 of the Weka machine learning workbench, available from http://www.cs.waikato.ac.nz/ml/weka.

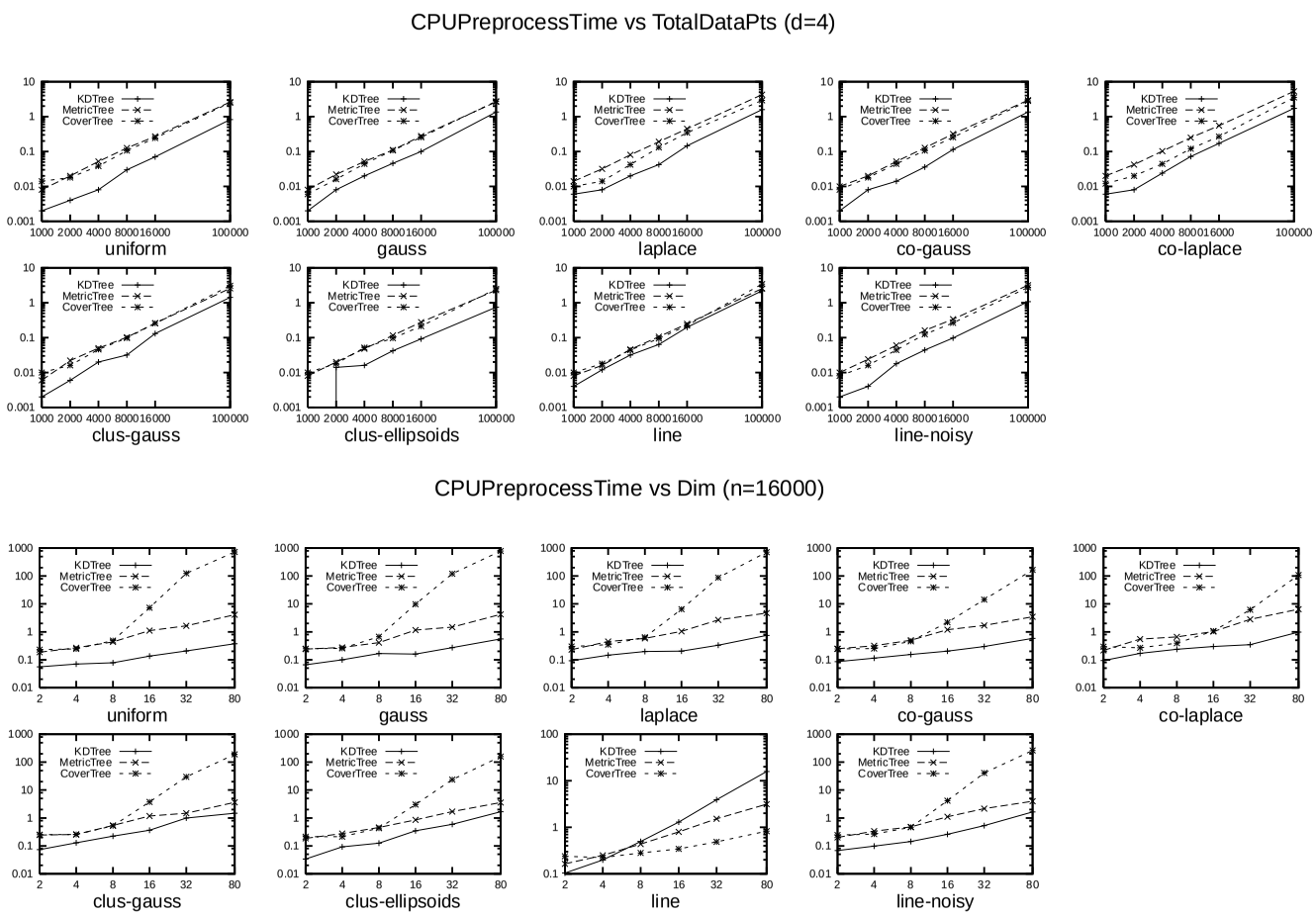

Fig. 3. CPU construction time of the data structures for increasing n, for $d = 4$, and for increasing d, for $n = 16000$.

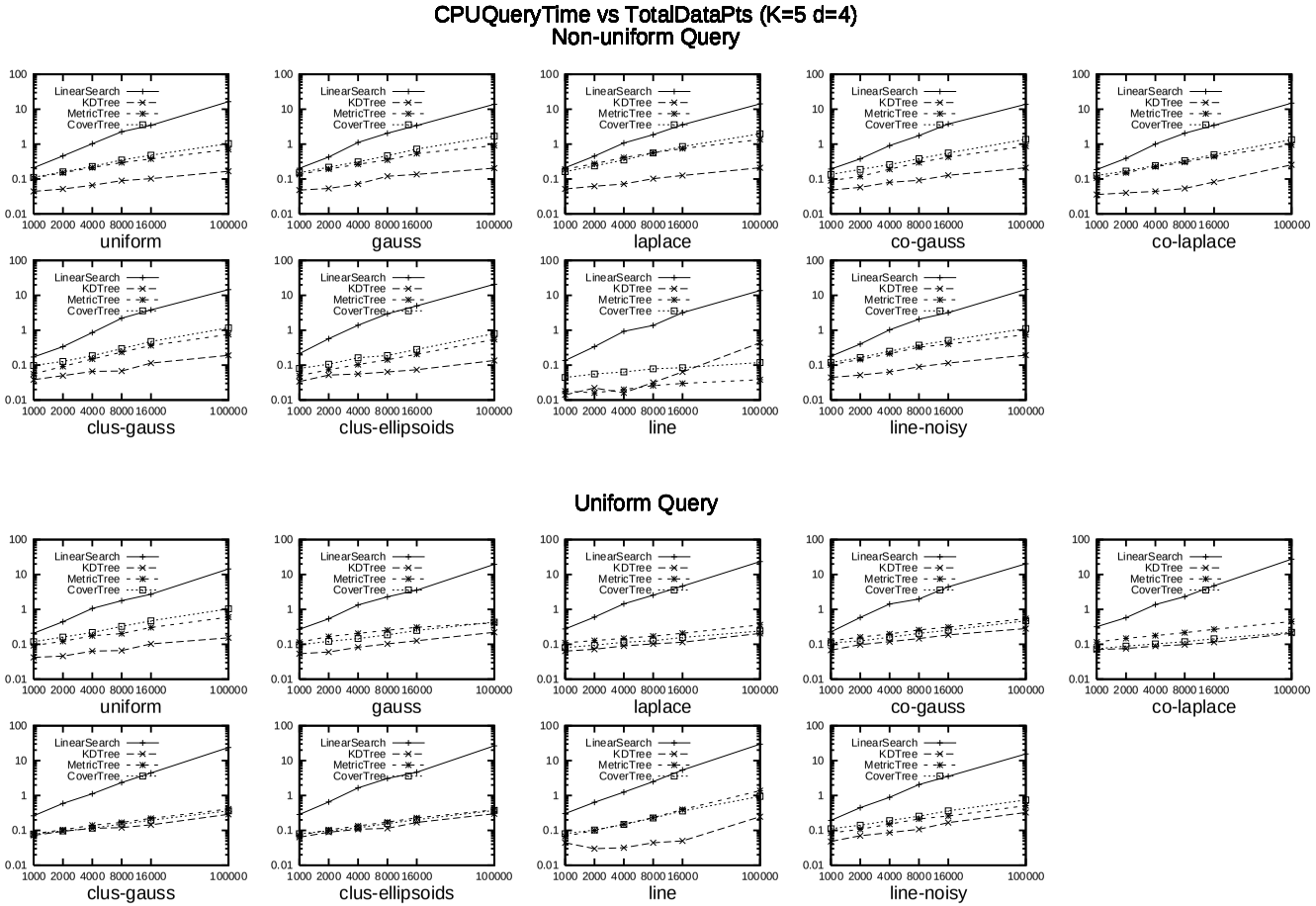

Fig. 4. CPU query time of the data structures for increasing n, for $d = 4$.

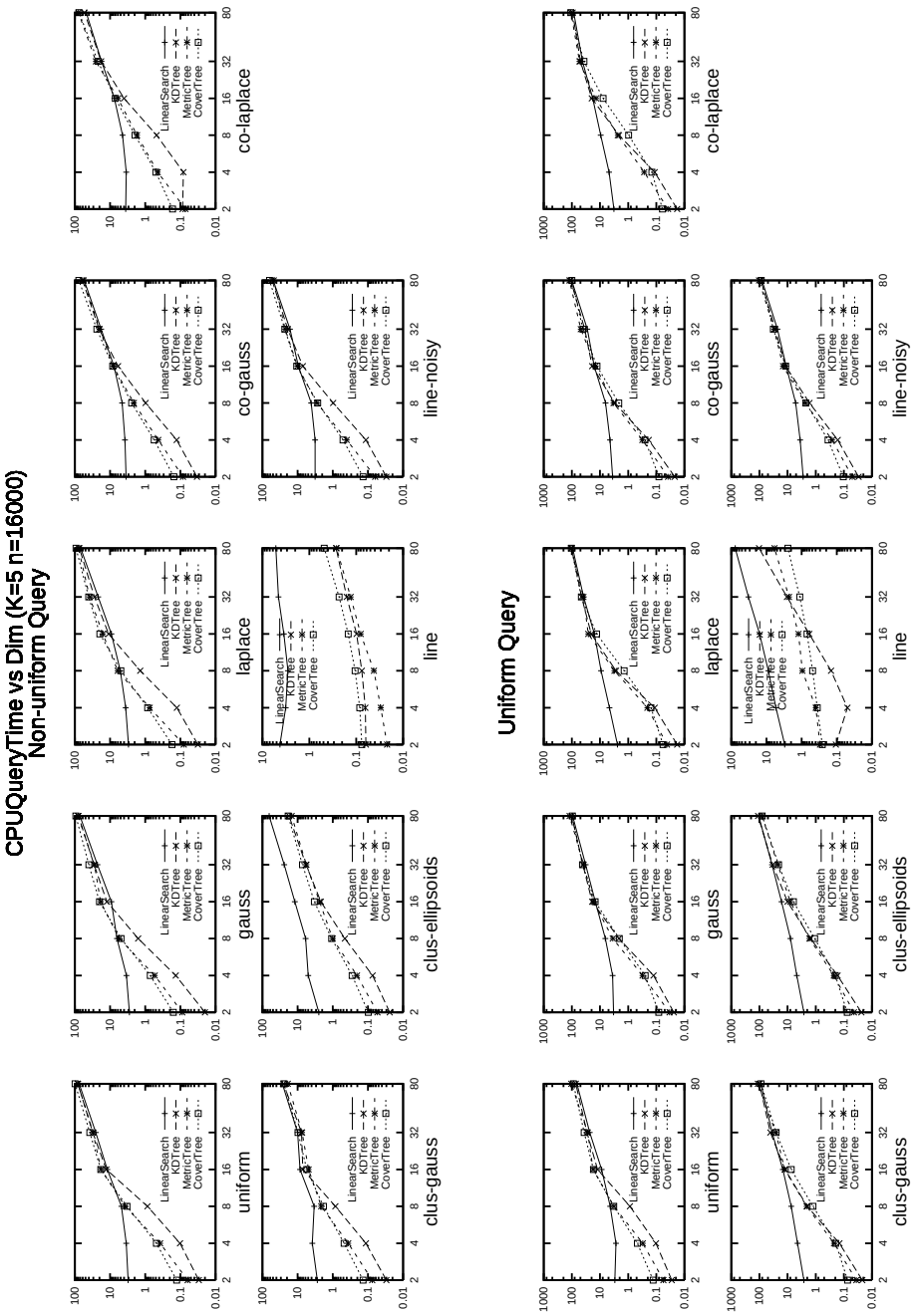

Fig. 5. CPU query time of the data structures for increasing d, for $n = 16000$.

Table 1. Query time of the data structures on UCI data.

Dataset	KD-Trees	Linear Search	Metric Trees	Cover Trees
car(7)	0.03	0.07 ○	0.08 ○	0.07 ○
mfeat(7)	0.02	0.11 ○	0.03	0.04 ○
cmc(10)	0.02	0.05 ○	0.07 ○	0.04 ○
german-credit(21)	0.06	0.06	0.09 ○	0.09 ○
segment(20)	0.03	0.13 ○	0.08 ○	0.08 ○
page-blocks(11)	0.04	0.76 ○	0.17 ○	0.18 ○
sick(30)	0.15	0.60 ○	0.78 ○	0.21 ○
hypothyroid(30)	0.21	0.82 ○	1.06 ○	0.27 ○
kr-vs-kp(37)	0.40	0.57 ○	1.03 ○	0.54 ○
nursery(9)	0.76	2.91 ○	7.31 ○	4.54 ○
mushroom(23)	0.34	2.44 ○	4.05 ○	1.04 ○
pendigits(17)	0.44	3.01 ○	1.22 ○	1.07 ○
splice(62)	2.10	1.93 ●	2.53 ○	2.29 ○
waveform(41)	4.67	4.35 ●	6.05 ○	6.00 ○
letter(17)	4.00	11.42 ○	8.20 ○	6.16 ○
optdigits(65)	4.50	4.79 ○	5.52 ○	4.13 ●
ipums-la-97-small(61)	4.91	4.60 ●	6.27 ○	5.53 ○
ipums-la-98-small(61)	4.48	4.00 ●	5.77 ○	5.25 ○
ipums-la-99-small(61)	6.42	5.60 ●	8.22 ○	7.63 ○
internet-usage(72)	26.90	23.90 ●	35.73 ○	32.45 ○
auslan2(27)	23.71	660.73 ○	100.33 ○	101.62 ○
auslan(14)	28.54	2162.14 ○	297.02 ○	123.70 ○
ipums-la-98(61)	474.78	364.63 ●	602.31 ○	580.48 ○
census-income-test(42)	189.06	556.99 ○	976.03 ○	624.07 ○
ipums-la-99(61)	666.84	513.60 ●	862.59 ○	839.27 ○
abalone(9)	0.06	0.27 ○	0.20 ○	0.12 ○
ailerons(41)	4.35	8.57 ○	11.20 ○	10.47 ○
bank32nh(33)	11.06	9.82 ●	13.84 ○	14.56 ○
2dplanes(11)	12.81	42.56 ○	39.08 ○	23.05 ○
bank8FM(9)	0.68	1.76 ○	1.52 ○	1.51 ○
cal-housing(9)	1.33	7.60 ○	2.60 ○	2.70 ○
cpu-act(22)	0.54	3.32 ○	1.91 ○	1.79 ○
cpu-small(13)	0.23	2.52 ○	1.02 ○	0.92 ○
delta-ailerons(6)	0.10	0.81 ○	0.39 ○	0.40 ○
delta-elevators(7)	0.21	1.48 ○	1.00 ○	0.94 ○
elevators(19)	3.28	7.69 ○	8.55 ○	7.71 ○
fried(11)	16.08	45.07 ○	61.27 ○	47.43 ○
house-16H(17)	3.53	25.79 ○	12.93 ○	10.06 ○
house-8L(9)	1.30	16.79 ○	4.57 ○	3.91 ○
CorelFeatures-ColorHist(33)	16.67	157.90 ○	155.29 ○	75.05 ○
CorelFeatures-ColorMoments(10)	23.64	90.38 ○	54.72 ○	50.14 ○
CorelFeatures-CoocTexture(17)	20.83	110.80 ○	32.76 ○	32.56 ○
CorelFeatures-LayoutHist(33)	35.01	177.49 ○	120.31 ○	104.83 ○
el-nino(12)	173.40	481.58 ○	2056.06 ○	1000.63 ○
kin8nm(9)	0.89	1.93 ○	2.20 ○	1.85 ○
mv(11)	8.56	36.28 ○	21.60 ○	12.11 ○
pol(49)	1.20	14.98 ○	9.61 ○	6.02 ○
puma32H(33)	9.86	8.42 ●	12.21 ○	12.96 ○
puma8NH(9)	0.94	1.97 ○	2.33 ○	2.02 ○
quake(4)	0.01	0.07 ○	0.02	0.02 ○

○/● statistically worse/better at 95% confidence level

4 Conclusions

Most of the data structures and techniques proposed since the initial inception of the NNS problem have not been extensively compared with each other, making it hard to gauge their relative performance.

KD-Trees are one of the most popular data structures used for NNS for moderate d's. Metric Trees are more widely applicable, and also designed for

moderate d's. The more recently proposed Cover Trees have been designed to exploit the low intrinsic dimensionality of points embedded in higher dimensions. This paper has presented an extensive empirical comparison of these three techniques on artificial and real-world data. It shows that Metric Trees and Cover Trees do not perform better than KD-Trees in general on the standard NNS problem. On our synthetic data, Cover Trees have similar query time to Metric Trees, but they outperform Metric Trees on real-world data. However, Cover Trees have a higher construction cost than the other two methods when the number of dimensions grows.

References

1. Minsky, M., Papert, S. In: Perceptrons. MIT Press (1969) 222–225
2. Aurenhammer, F.: Voronoi diagrams–A survey of a fundamental geometric data structure. ACM Computing Surveys **23**(3) (1991) 345–405
3. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer (2001)
4. Liu, T., Moore, A.W., Gray, A.G.: Efficient exact k-NN and nonparametric classification in high dimensions. In: Proc of NIPS 2003, MIT Press (2004)
5. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proc 13th Annual ACM symposium on Theory of Computing, New York, NY, ACM Press (1998) 604–613
6. Nene, S.A., Nayar, S.K.: A simple algorithm for nearest neighbor search in high dimensions. IEEE Trans. Pattern Anal. Mach. Intell. **19**(9) (1997) 989–1003
7. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proc 20th Annual Symposium on Computational Geometry, New York, NY, ACM Press (2004) 253–262
8. Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: Proc 23rd International Conference on Machine learning, New York, NY, ACM Press (2006) 97–104
9. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9) (1975) 509–517
10. Mount, D.M., Arya, S.: ANN: A library for approximate nearest neighbor searching. In CGC 2nd Annual Fall Workshop on Computational Geometry (1997) Available from http://www.cs.umd.edu/∼mount/ANN.
11. Kibriya, A.M.: Fast algorithms for nearest neighbour search. Master's thesis, Department of Computer Science, University of Waikato, New Zealand (2007)
12. Omohundro, S.M.: Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute (December 1989)
13. Uhlmann, J.K.: Satisfying general proximity / similarity queries with metric trees. Information Processing Letters **40**(4) (1991) 175–179
14. Moore, A.W.: The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In: Proc 16th Conference on Uncertainty in Artificial Intelligence, San Francisco, CA, Morgan Kaufmann (2000) 397–405
15. Bei, C.D., Gray, R.M.: An improvement of the minimum distortion encoding algorithm for vector quantization. IEEE Transactions on Communications **33**(10) (1985) 1132–1133
16. Nadeau, C., Bengio, Y.: Inference for the generalization error. Machine Learning **52**(3) (2003) 239–281